Behavioral Specification and Planning for Multiagent Domains

Technical Note No. 360

November 12, 1985

By: Amy L. Lansky, Computer Scientist
Representation and Reasoning Group
Artificial Intelligence Center
Computing and Engineering Sciences Division

Approved for Public Release; Distribution Unlimited.

This research has been made possible in part by the Office of Naval Research under Contract N00014-85-C-0251, and by the National Science Foundation under Grant IST-8511167.

ABSTRACT

This report discusses a new approach to the specification of properties of multiagent environments and the generation of plans for such domains. The ideas presented elaborate previous work on a formal, behavioral model of concurrent action, called GEM (the **G**roup **E**lement **M**odel). By combining the GEM specification formalism with artificial intelligence techniques for planning, we have devised a framework that seems promising in several respects. First, instead of ad hoc planning techniques, we are utilizing a formal concurrency model as a basis for planning. Secondly, the model encourages the description of domain properties in terms of behavioral constraints, rather than using more traditional state predicate approaches. Behavioral descriptions, which emphasize the causal, temporal, and simultaneity relationships among actions, are particularly suited to describing the complex properties of multiagent domains. Finally, we present an initial proposal for a planner based on behavioral forms of representation. Given a set of constraints describing a problem domain, the proposed planner generates plans through a process of incremental constraint satisfaction.

Contents

1 Introduction **1**

2 Background **2**

3 Domain Description: State-Based versus Behavioral **5**

4 The GEM Model **13**

4.1 Model of Execution 13

4.2 Temporal Assertions on GEM History Sequences 16

4.3 GEM Specification Language 19

4.3.1 Delimiting the Set of Admissible Events and Imposing Constraints Based on Structural Relationships 20

4.3.2 Attaching Additional Constraints to Elements and Groups . . . 22

4.3.3 Element and Group Types 23

4.3.4 Common Constraint Abbreviations 24

5 Proposed Extensions of GEM **27**

5.1 Action Hierarchies 27

5.2 State Descriptions and the Frame Problem 28

5.3 Explicit Time and Temporal Intervals 33

6 Planning Method **34**

7 An Example: The Blocks World **37**

7.1 Blocks World Specification 37

7.2 Generating Plans for the Blocks World 39

8 Conclusions **42**

A GEM Specification Language Syntax **48**

1 Introduction

The real world is full of parallel activity. Groups of people and machines are constantly cooperating and interacting in the performance of complex tasks. Unfortunately, humans cannot understand or plan parallel activities as easily as they might like; it is difficult to forsee the many (potentially harmful) interactions and interleavings that could occur. Computers, however, are uniquely suited to this type of complex reasoning. A representation of concurrency that is intuitively comprehensible to humans as well as computationally tractable could thus greatly enhance our ability to deal with the complexities of multiagent worlds.

This paper is devoted to describing a new approach to representing the properties of concurrent, multiagent domains. The approach is based on GEM (the Group Element Model), a model of concurrency which emphasizes (1) the actions within a domain and (2) the constraints on the temporal, causal, simultaneity, and spatial relationships among those actions [22,23]. We call this approach *behavioral* because it stresses agents and their actions, rather than the state of the environment in which the agents operate. Most specification techniques for planning have been state-based; that is, they describe a domain in terms of a set of state predicates, and the actions of a domain in terms of their effects on these predicates. This paper attempts to show how behavioral descriptions are more suited to expressing many of the complex properties of multiagent domains, while still being able to convey the descriptive power of state predicate approaches.

Because our model is able to capture the causal, temporal, simultaneity, and in some sense, spatial aspects of real-world domains, the work described herein also represents contributions in the area of knowledge representation and naive physics. While the primary emphasis of this paper is specification for purposes of plan synthesis, the close relationship between our work and that of Allen, Hayes, McDermott, Shoham and Dean, and others should not be overlooked [1,2,17,28,37]. We plan to write a follow-up to this report in which we shall compare our specification formalism with other research on the representation of time and action.

Towards the end of this paper we offer some preliminary ideas on how to use behavioral specifications as the basis for a behavioral planning technique. In fact, the ideas presented are themselves the basis of a long-term research program in multiagent planning currently under way. The planning system proposed generates plans by incrementally imposing the set of *behavioral constraints* that constitute the properties and requirements of a given domain.

We expect this planning methodology to break new ground in several respects: the use of a formal and general model of concurrent action as a basis for planning, instead of ad hoc planning techniques; the integration of behavioral and state-based approaches to domain description and plan generation; and the investigation of some new planning techniques and heuristics, including constraint ordering and run-time constraint satisfaction. A long-term goal is to build a graphical user interface for our behavioral planning environment. This interface will serve as a vehicle of explanation and interaction with humans during the planning process, as well as a means of visualizing plan simulation and execution.

2 Background

In recent years, parallelism has been a topic of extensive research in more traditional provinces of computer science: computer architecture, data base systems, and operating systems. Automatic plan generation for multiagent domains has received less attention; moreover, very little of that work has utilized the theoretical results achieved in the foregoing areas. One notable exception is Georgeff's work on integrating and synchronizing multiple single-agent plans [13], which uses Owicki and Gries's technique for assuring freedom from interference [31]. Stuart has implemented Georgeff's scheme [41], by employing Manna and Wolper's algorithms for synthesizing a concurrent program from temporal-logic specifications [25]. While the work described in this paper is also based on previous work in concurrency theory, it differs from Georgeff's and Stuart's in several respects. While their representation does make use of temporal constraints on action orderings, it is still primarily rooted in a state-based framework. GEM's behavioral representation is also much richer – for instance, in its use of causal, simultaneity, and structural relationships, as well as reasoning over *histories*. In addition, our proposed GEM planning technique will generate integrated multiagent plans, rather than construct a synchronized plan from already generated subplans.

In our attempt to apply our GEM-based representation to planning, we hope to utilize many of the techniques developed by past planning efforts. Of the existing planners that influence our work (i.e., planners that produce partially ordered instead of sequential plans), NOAH is probably the most significant [35]. Descendants of NOAH include SIPE [45], NONLIN [42] and [43], and DEVISER [44], each of which improves upon NOAH by including such things as limited backtracking, plan justification or rationalization, and durational reasoning. A significant difference between these planners and our own will be our starting point: a mathematical model of concurrent action

(called *GEM* [22]), rather than a set of ad hoc planning techniques. We expect this will result in a more elegant and conceptually uniform planning method. For example, one technique used by NONLIN emerges naturally. GEM explicitly represents the causal relationships among actions in a plan. If we couple these causal relationships with the constraints that generated them, we obtain the type of information found in NONLIN's "goal structures" [43], namely, a record explicating the causal interrelationship of actions within a plan.

Related to our work, but with a slightly different emphasis, are ongoing investigations into mutual belief: how to represent and manipulate the beliefs of multiple agents about one another [19,29]. Researchers working on commonsense reasoning and natural-language understanding ([4,9,12,17,28,34]) have also found it necessary to represent the knowledge of concurrently operating agents about one another. This is sometimes called *distributed AI*; it is concerned with sets of autonomous agents. Such agents may have incomplete knowledge about one another and may be neither cooperative nor benevolent. Other research in distributed AI is less concerned with modeling mutual belief but focuses on the problem of decentralizing plan synthesis [10,24].

The work described in this paper differs from distributed AI in that it assumes a set of cooperative agents. In the case of full pre-planning, each of these agents will accept a single overall plan and execute its portion therof. Such a plan is generated by a centralized multiagent planner that considers all the possible behaviors of each individual agent. An assumption of agent cooperation is clearly relevant in many domains, especially those in which agents are trying to achieve a common goal. For example, a set of robots trying to assemble a car clearly fits this model. So does the problem of constructing a general, coordinated work plan for a large organization. Although planning under assumptions of agent cooperation is obviously simpler than planning in noncooperative domains, it is a problem that has by no means been solved.

Several other theoretical models and representations for planning have been proposed besides GEM. Most of these models have represented domain actions as atomic events, and the passage of time as a sequence of states. In some cases, the world model employed has allowed only one action at a time to occur [11,27]. In others, actions have been partially ordered but are still considered to be atomic [35,42]. Yet another approach has been taken by Allen and Koomen, Shoham, and Cheeseman [2,3,7,36]. They model actions and/or predicates as intervals and describe domain properties in terms of interval-ordering constraints.

Our formalism follows the more traditional approach of modeling actions as atomic events. This is not to say that we consider action occurrences to be instantaneous, but

rather that they are logically atomic – action duration is not represented explicitly. The advantages of action intervals are not lost however; an action A occurring over an interval can be modeled, for instance, by describing it with two atomic events, representing the beginning and end of A. Because our formalism also allows for the simultaneous occurrence of actions, all of the possible interval relationships among actions (or even predicates) used in the work cited above can be modeled. We discuss the representation of intervals within GEM in Section 5.3.

As mentioned earlier, our work also stands out in its explicit representation of the causal relationships among actions. Causal theories have long been studied by philosophers. They have recently appeared in work on knowledge representation and naive physics [2,17,28,37], as well as in GEM. Our representation of causal relationships is similar to that used by Shoham and Dean [37].

As in previous work [3,36], our model also makes use of temporal logic – a logic that enables reasoning about time. By taking advantage of the descriptive power of temporal logic, we can easily describe goals, preconditions, or any other domain constraint in terms of behavior patterns over time, rather than as single-state expressions. Temporal logic has received much attention in most recent work on multiagent planning and domain representation. This is not surprising; both linear and branching-time temporal logics have long been used for concurrent program specification, verification, and synthesis [8,22,25,32]. A recent workshop on concurrency and planning[1] focused on the potential crossover between traditional concurrency theory and AI. It underscored the prospective benefits to AI if researchers took advantage of temporal logics and concurrency specification techniques. In light of GEM's history as a tool for concurrency specification and verification, the work described in this paper is precisely such an attempt.

[1]The 1984 Concurrency/Planning Workshop at Monterey Dunes, sponsored by the Center for the Study of Language and Information at Stanford and AAAI.

3 Domain Description: State-Based versus Behavioral

In most planning systems, a domain is described by a set of state predicates and robots (agents) are viewed as state transformers – i.e., their actions are described in terms of the way they modify this set of predicates. In this context, the goal of a planning system is to transform an initial state description into a goal state description by concatenating actions that manipulate state predicates in an appropriate fashion.

For example, a blocks world consisting of a table, a set of blocks, and robots, might be described in terms of predicates such as On(Block1, Block2), HandEmpty(Robot1), Holding(Robot2, Block3), Clear(Block1). Actions of the domain are described in terms of the way they affect these predicates. For instance, an action by Robot1 of form PickUp(Block2) might cause the predicate Holding(Robot1, Block2) to be added to the state description, and HandEmpty(Robot1) to be deleted. Given this type of domain description, actions are inserted into a plan because of their effect on world state, rather than because of some explicitly required relationship among actions. Some of the advantages of this approach are its explicit provision of action preconditions, the encoding of world state into predicate form, and its straightforward mapping onto an implementation.

Unfortunately, strictly state-based approaches to domain description can be awkward for describing behavioral properties – i.e., *those that entail complicated causal and temporal relationships among actions.* Priority requirements, for example, fall into this category; they restrict future relationships among actions (for example, the order in which a service is performed) based on past relationships (the order in which requests for service were registered). Other behavioral properties include *simultaneity* and such synchronization properties as *mutual exclusion.*

While most state-based specification frameworks cannot specify simultaneity requirements, they can describe behavioral properties like priority by encoding them in terms of state-predicate manipulations. A key limitation of this approach is that it is *indirect*; imposing a simple behavioral requirement sometimes involves complex manipulation of predicates. Moreover, the description of a property may be distributed among the various action specifications that manipulate relevant predicates, rather than be stated as a single constraint or rule. This leaves the domain describer prone to error and makes system maintenance and debugging more difficult. In an attempt to obtain reasonable performance, successful planners have also had to limit the possible forms of state manipulation allowed. These limitations restrict the kinds of domain properties that can be expressed.

One way to deal with this problem is to adopt a new point of view: although it may be useful to describe actions as state transformers, we should also have the capability of *describing relationships among actions directly, rather than indirectly through state predicates*. Such descriptions take the form of explicit restrictions or *constraints* on the permitted temporal and causal relationships among actions. Given this formulation, planning may be viewed as a process of incremental constraint satisfaction. The goal of such a planning system is to synthesize a plan (a partial ordering of actions) that transforms some initial configuration of actions into some final goal configuration, such that every execution of the plan satisfies all of the domain's constraints.

The basis for the types of constraint descriptions we will be using is GEM, the behavioral model of concurrent action mentioned earlier [22,23]. Behavioral models of execution such as GEM have proved to be natural vehicles for expressing and analyzing properties of computational concurrency [16,18,20,21,23]. GEM, in particular, has been used quite successfully for specifying several concurrency problems and language primitives, as well as verifying concurrent programs. Its utility is due in part to its use of temporal-logic operators over sequences of *histories*, i.e., complete records of past behavior. The GEM specification language also includes a hierarchical, parameterized-type facility, thus making it possible to build new specifications as refinements of old ones. Another useful part of the language is a mechanism for describing structural relationships among actions and the agents that perform them, and for relating structure to constraints on behavior.

While our primary goal is to explore the power and tractability of behavioral methods of specification, it is also our intent to investigate how traditional uses of state predicates can be *integrated* within GEM's behavioral framework. We will describe the GEM model in greater detail in Section 4 and its integration with state-based methods in Section 5.2.

Let's continue to contrast traditional state-based domain descriptions with the behavioral descriptions GEM uses by considering a few examples. Suppose we are specifying a "diving" domain consisting of a diver, a diving board, and water. To perform a certain kind of dive, a diver might go through a sequence of actions such as Jump, Somersault, Twist, StraightenBody. Although divers may not know about, or even be aware of, the complex physical forces that are involved in a dive, they do know that executing a particular sequence of physical movements will produce the desired effect.

How can state predicates be used to describe the diving domain? What are the preconditions for a somersault or a twist? How do these actions change world state?

Finally, how could we use this knowledge to formulate a plan for a particular dive? It would be extremely difficult and essentially useless to describe the physical states of the world during a dive – they are not even perceivable to the diver. In actuality, we only need to describe the states during a dive in terms of "control" predicates. For example, AfterSomersault could be used to indicate that the diver has just finished a somersault. By requiring certain predicates as preconditions of specific actions, this type of representation could be used to force a "dive plan" to take a particular form. But since these predicates do not really reveal any meaningful information about world state, it would be clearer to represent the dive *directly* as a particular set of actions that must occur in a prescribed sequence.

This example illustrates the fact that domains with nonobservable state are often not naturally described by a state predicate description. Indeed, most implemented planners such as NOAH or SIPE provide predefined procedural operators or schemas for behavioral goals such as a dive. Recent work on procedural logic and procedural reasoning systems by Georgeff, Lansky, and Bessiere [14] has also emphasized the representation of procedural forms of knowledge. Procedural descriptions alleviate the tasks of domain description and plan generation by representing preformed plans in a compact form that need not be rederived. Procedural domain properties, along with many other more complex behavioral properties, are easily described in GEM.

While the dive example actually *can* be described fairly easily by most state-based specification techniques, we find that a behavioral style is especially useful for describing the complex interrelationships among actions that often appear in multiagent domains. Behavioral approaches may also be advantageous when state-based methods are adequate for planning purposes but are not fully expressive; for example, they can be used to convey interesting relationships among actions that are useful for plan explanation. To illustrate both of these points, we consider a new domain that utilizes priority relationships among actions.

Suppose there are two types of robots, **red** and **green**, who all share a tool. This tool may only be used by just one robot at a time; furthermore, once a robot has acquired the tool, it may keep it until it has finished a given task. Each robot must request to use the tool, and robots within each color class may use the tool in request order only. In addition, **red** robots have priority over **green** robots. Thus, if a **red** and a **green** robot are both waiting for the tool, the **red** one will get to use it first.

If we tackle this problem with a purely state-predicate approach, we would have to explicitly keep track of the separate request orderings for **red** and **green** robots. We would also have to record how many **red** robots are waiting for the tool at any given

moment. A **green** robot may acquire the tool only if it has successfully requested it, the number of waiting **red** robots is zero, all preceding **green** requests have been served, and the tool is free.

For example, to encode the number of waiting **red** robots, we could use a predicate of the form RedWaiting(i). Predicates of this form would be added to and deleted from the state description by **red** robot requests and tool use actions. Moreover, the RedWaiting predicate added by a given action would be determined not only by the form of the action, but by the RedWaiting predicate that belonged to the state preceding the action. Given this formulation, the state prior to a **green** robot's tool acquisition must satisfy the formula RedWaiting(0).

Now let us consider one of the more complicated domain constraints – the first-come-first-served requirement imposed on each class of robots. We could associate each robot with a "ticket number" in much the same way that lines are handled in bakeries. If each robot can make only one request at a time, a predicate of form Ticket(R,n) could be used to indicate that robot R has ticket number n.[2] When a robot R registers a new request, a predicate Ticket(R,n) is added to the state description, where n is the next ticket number for its class. Once a robot acquires the tool, this predicate is deleted. The state before robot R's acquisition of a tool must then satisfy the following constraint:

$$\text{Ticket}(R.n) \wedge \neg(\exists\ j.\ R')\ [\text{Color}(R')=\text{Color}(R) \wedge \text{Ticket}(R',j) \wedge j<n]\ .$$

Unfortunately, most planners are not equipped to test for or achieve all the possible prerequisite formulas – in fact, perhaps not even a formula such as the one above. Although a general-purpose theorem prover such as QA3 [15] could handle a broader class of prerequisite formulas, theorem provers are not very efficient planners. One approach taken by the SIPE planner is to deal explicitly with specific kinds of resource constraints. Such measures, however, certainly do not represent a general solution to the overall problem of action synchronization.

It is also important to realize that the robot example given above is not overtly complex nor contrived – priority requirements arise quite often in any environment in which resources are shared. The **red/green** robot problem, for example, is a version of the classic readers/writers database problem. As we can see, the manipulation of predicates can become quite complex when priority is being described. In traditional state-based implementations of shared resource problems, data structures are used to

[2] If we allowed each robot to make more than one request at a time, we would have to associate a priority number with each robot-request pair, not just with the robot itself.

help save state information. For example, queue structures are normally used to save requests and preserve information about their ordering. Once a form of priority becomes more unusual, however, a data structure that matches the priority structure directly becomes harder to find, and the priority property becomes harder (and sometimes impossible) to implement [5].

The use of behavioral constraints is a much more elegant approach to priority and other synchronization properties. Domain properties are specified in terms of succinct constraints rather than descriptions that are distributed among the preconditions and postconditions of domain actions. We can describe the entire red/green robot problem with the few simple constraints given below. Each constraint is described both informally and by a constraint formula. Although the reader should not expect to understand these formulas completely at this time, the paragraph below contains a preliminary description of the notation. The GEM formalism will be described fully in Section 4.

Action names beginning with a capital letter (such as Request*), represent an entire class of actions, whereas lowercase action names represent action instances (*request1*). The double arrow* $\Longrightarrow$ *represents the temporal ordering in a plan; the curly arrow* $\rightsquigarrow$ *represents the causal relation. The single arrow* $\longrightarrow$ *(causal prerequisite) indicates a required one-to-one causal relationship between classes of actions. An infix predicate of form "*a cbefore B*" means that action* a *has occurred but has not yet caused an action of type* B *to occur. A temporal expression of form* $\Box(p \supset q)$ *can be read "heneeforth, if* p *is true, then* q *must be true as well." Finally, because we have not yet presented the GEM specification language, we resort below to some nonstandard abbreviations and conventions. When the notation* Robot.<event type> *is utilized within a given constraint, the event instances of the types so denoted must all belong to the same robot.* Color(e) *is used to denote the color of the robot at which event* e *oeeurs.*

RED/GREEN Robot Domain Contraints

1. Each robot tool acquisition must be preceded (caused) by a corresponding tool request. Each such request can be used towards (can cause) only one tool acquisition. Note that this specification allows each robot to make more than one request at a time, something not accommodated by the state-based specification given above (moreover, were it included, it would complicate the specification even more).

   Robot.Request $\longrightarrow$ Robot.UseTool

2. All requests by robots of the same color must be totally ordered, and if two robots of the same color request the tool in a given order, they must be served in that order. The second constraint below may be read as follows: "If at some point in time we find that req1 precedes req2, then, from that moment forward, if req2 is fulfilled, req1 must have been fulfilled as well."

$$(\forall\ \text{req1,req2:Request.}\ \text{req1} \neq \text{req2.}\ \text{color(req1)=color(req2)})$$
$$[\text{req1} \Longrightarrow \text{req2} \vee \text{req2} \Longrightarrow \text{req1}]$$

$$(\forall\ \text{req1,req2:Request.}\ \text{color(req)=color(req2)})$$
$$[\ \text{req1} \Longrightarrow \text{req2} \supset \Box\ (\text{req2} \rightsquigarrow \text{usetool2} \supset \text{req1} \rightsquigarrow \text{usetool1})]$$

3. If a red robot and a green robot are waiting for the tool at the same time, the red robot must get to use it first. Note that we could employ an abbreviation of the form Waiting(a.B) instead of a cbefore B. This would make this constraint definition easier to read.

$$(\forall\ \text{req1,req2:Request.}\ \text{color(req1)=red.}\ \text{color(req2)=green})$$
$$[\ \text{req1 cbefore UseTool} \wedge \text{req2 cbefore UseTool} \supset$$
$$\Box\ (\text{req2} \rightsquigarrow \text{usetool2} \supset \text{req1} \rightsquigarrow \text{usetool1})\ ]$$

4. The tool can be used by only one robot at a time.

$$(\forall\ \text{usetool1,usetool2:UseTool.}\ \text{usetool1} \neq \text{usetool2})$$
$$[\text{usetool1} \Longrightarrow \text{usetool2} \vee \text{usetool2} \Longrightarrow \text{usetool1}]$$

These constraints describe the requirements of the domain clearly and succinctly. Several things about them should be noted. First of all, Constraint 1 emphasizes the relationship between a specific Request action and its corresponding UseTool directly. This type of constraint is easily stated in GEM, which explicitly records causal relationships among actions. The relationship between a specific tool request and the corresponding tool usage would remain hidden in a state-based description unless explicit connection identifiers were associated with each Request and UseTool action.[3]

Secondly, we consider how the notion of "state" manifests itself even in a purely behavioral specification. In the constraint description above, no explicit predicate (such as RedWaiting(i)) is used to encode the number of waiting red robots. Instead, "state" is specified in terms of *derived formulae* about actions. For example, a robot is considered to be "waiting" if some Request action has not yet caused a corresponding

[3]NONLIN's Goal Structures [43] were created for recording this kind of causal relationship.

UseTool action. Such a derived state description is used in Constraint 3: req1 cbefore UseTool. Of course, we could have described the predicate RedWaiting(i) in terms of a more complicated formula, such as

$$||\{\ \text{req:Request} \mid \text{color(req)=red} \wedge \text{req cbefore UseTool}\ \}\ || = i\ .$$

In the behavioral specification above, however, there was no need to keep track of the exact number of waiting **red** robots. It was enough to state the **red/green** priority property in terms of whether *any* specific **red** robot was waiting while a **green** robot was waiting (Constraint 3). In this case, GEM's ability to state properties in terms of a complete history has come into play. A formula of the form [req cbefore UseTool] is applied to the entire history of a plan execution. In contrast, state predicates record only the *current* state; for instance, RedWaiting(i) was needed to *count* the number of waiting **red** robots at any given instant.

While we intend to use derived formulas for describing some state predicates, we also plan on augmenting our underlying formalism so that more traditional predicate descriptions of state can be utilized. In many cases, a state predicate approach is more efficient, can enhance specification modularity, and can make incremental changes in domain specification easier to handle. State predicates are also handy for representing *sensed* information about the real world as well as for interfacing to humans via natural language. There are many cases, however, when state predicate description is unwieldy and the behavioral approach is more intuitive and compact. If both approaches are combined into a single formalism, they can each be used, depending on which is more appropriate. Possible methods of integrating traditional state-based descriptions into GEM are discussed in Section 5.2.

This example has also illustrated a point that will be further elaborated in the next section: GEM's ability to describe priority properties stems largely from its application of temporal operators to sequences of histories or complete state descriptions. Temporal logic, a logic that enables reasoning over sequences of states, will be described further in Section 4. By a *complete state description* or *history*, we mean a description that includes information about past behavior. The state descriptions used by most planners encompass no more than a part of a domain state – usually, the currently observable properties of a domain. In some cases, such as the **red/green** robot domain, a more complete record of execution history is useful. Allen [2] points out that information about the past is also needed in hospital domains; the course of past events in a patient's history is a vital part of medical data, not just the patient's current status. It is clear that a complete formulation of planning should have the capability of recording complete history descriptions, not just current state descriptions.

Simultaneous Activity

Although some existing planners can produce partially ordered plans for multiple agents, their underlying planning model was originally geared towards the production of single-agent plans. Because of this, little emphasis has been placed on the possibility of truly simultaneous activity. Many state-based planners have assumed that actions, even in a concurrent domain, occur one at a time.

In the real world, actions *do* occur simultaneously, especially if domain activity is viewed in the large. To specify and plan multiagent activity in a realistic fashion, we would like to be able to assert that certain actions ***must*** occur simultaneously (for example, two robots picking up a large object at once), or to explicitly ***prohibit*** certain sets of actions from occurring simultaneously (e.g., mutually exclusive access to a resource such as a tool or even physical space). We might also like to admit situations in which the relationship between two actions is truly ***un***specified: the actions could be ordered, occur simultaneously, or even overlap. As we will show, a behavioral approach easily accommodates description of all these types of properties.

4 The GEM Model

This section presents a broad overview of GEM's current specification capabilities. A full description of the model and its application to program specification and verification may be found in my doctoral dissertation [22] as well as in a paper coauthored with Owicki in 1983 [23].

4.1 Model of Execution

GEM's model of concurrent activity is based on a very behavioral view of the world. At each point in time, we consider the world's state to be completely characterizable in terms of all actions that have occurred up to that instant, as well as the causal, temporal, simultaneity, and spatial relationships that exist among those actions.

To state it more formally, GEM models concurrent activity as *computations*. Each computation C is composed of a set of unique objects called *events* which are related by a partial temporal ordering $\Longrightarrow$, a partial causal relation $\rightsquigarrow$, and a simultaneity relation $\rightleftharpoons$. Events are also grouped into two types of sets: *elements* and *groups*.

$C =< E, EL, G, \Longrightarrow, \rightsquigarrow, \rightleftharpoons, \epsilon >$

- $E =$ A set of unique event objects.
- $EL =$ A set of unique element objects.
- $G =$ A set of unique group objects.
- $\Longrightarrow$: $(E \times E)$ The temporal ordering is a partial, irreflexive, antisymmetric, transitive ordering among events.
- $\rightsquigarrow$: $(E \times E)$ The causal relation is a partial, irreflexive, antisymmetric, nontransitive relation between events.
- $\rightleftharpoons$: $(E \times E)$ The simultaneity relation is a partial, reflexive, symmetric, transitive relation among events.
- $\epsilon : (E \times \{EL, G\})$ A set membership relation between events and elements, or events and groups.

Each event in a computation models a distinct action in the world domain, each relation or ordering relationship models an actual relationship between domain actions, and each element or group, a logical location of activity or grouping of locations in the domain consisting of a set of events. In order to model the world in a meaningful and

coherent fashion, various rules and conventions must be obeyed by all computations. These are described below.

Each event in a computation is distinct; it may be viewed as a unique token. Moreover, events should be regarded as atomic – that is, they occur atomically relative to other events in the computation. This is not to say that events are totally ordered; events *may* happen simultaneously. We just do not represent event duration explicitly. From an intuitive standpoint, it might be useful for the reader to view each event as the end point of some logical action. Actions with duration can be modeled by using two or more events. For example, we might use one event to model the start of the action, another to model its conclusion. Action intervals are discussed further in Section 5.3.

In our specifications, events may have parameters and may also be organized into types, each of which represents some class of actions. For example, *Paint(o:Object, c:Color)* could represent the class of actions that paint an object a certain color. A specific instance of this type might be *paint(ladder,red)*. Lowercase tokens are used to denote specific event instances, with uppercase being used for event types.

Events can be related by three relations: $\Longrightarrow$, $\rightsquigarrow$, and $\rightleftharpoons$. The temporal order $\Longrightarrow$, a partial, irreflexive, antisymmetric, transitive relationship, models event ordering in time. In contrast, the causal relation $\rightsquigarrow$ is partial, irreflexive, antisymmetric and *not* transitive – it represents "direct" causality. The existence of a causal relation between two events also implies a temporal ordering between them ($e1 \rightsquigarrow e2 \supset e1 \Longrightarrow e2$), but the reverse is not true; just because two events may be forced to occur in some sequence does not mean that they are causally related. For example, two noncommunicating humans may be modeled as causally autonomous units, even if they are forced to walk through a door one at a time. Finally, the simultaneity relation $\rightleftharpoons$ is partial, reflexive, symmetric, and transitive, and models the ***necessarily*** simultaneous occurrence of events.[4] Note that events can still potentially occur simultaneously even if they are unrelated by $\rightleftharpoons$.

In addition to being ordered, events are also grouped into two types of sets: *elements* and *groups*. Elements are used to represent locations of sequential activity. Each event must belong to exactly one element and all events belonging to the same element must be totally ordered by the temporal ordering $\Longrightarrow$. The notation e^i indicates that event e is the ith event taking place at its particular element.

[4]It is interesting to note C.G.Jung suggestion that world phenomena be described not only in terms of causal and temporal relationships, but with a simultaneity relation as well. This was proposed as a way of describing unrepeatable psychic phenomena [39].

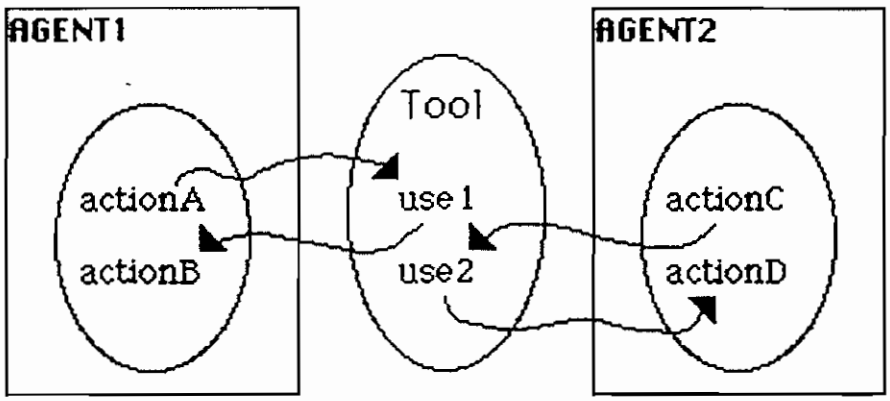


Figure 1: Execution of Agent1 and Agent2

The other type of event set, the *group*, is used for representing higher-level structural relationships among locations of domain activity. Each group consists of the events within a specified list of elements and/or other groups. In Section 4.3 we will show how groups are used for modeling causal limitations (for example, scope rules) among the events occurring at different loci of activity.

The following example illustrates the ideas presented thus far. Suppose we were using GEM to represent two independent agents, *Agent*1 and *Agent*2, which share a tool and perform the following activities:

```
Execution of Agent1:     actionA; Tool.use1; actionB

Execution of Agent2:     actionC; Tool.use2; actionD
```

Let us assume that all uses of the tool must occur sequentially, but no specific order is presumed. Figure 1 depicts one way GEM might represent this domain.[5] Any GEM computation within this domain would have to include the following temporal relationships:

$$\begin{array}{ccccc} Agent1.actionA & \Longrightarrow & Tool.use1 & \Longrightarrow & Agent1.actionB \\ & & \Uparrow \text{ or } \Downarrow & & \\ Agent2.actionC & \Longrightarrow & Tool.use2 & \Longrightarrow & Agent2.actionD \end{array}$$

While we know that some temporal order must exist between $Tool.use1$ and $Tool.use2$, it should be noted that no temporal ordering can be assumed between $actionA$ and $actionC$, or between $actionB$ and $actionD$. In fact, our representation allows for the possibility of these events occurring simultaneously.

[5]Squares represent groups, circles represent elements, and lowercase names represent events.

Finally, for purists among our readers, we should point out that both elements and groups are merely notational devices; the representation of a computation *could* be simplified to include only events and the relations $\Longrightarrow$, $\rightsquigarrow$, and $\rightleftharpoons$. By clustering events into elements and groups, however, we implicitly impose constraints on the temporal and causal relationships among those events, thereby easing the task of specification. For example, because *use1* and *use2* both belong to the same element (*Tool*), they are implicitly compelled to be totally ordered. Similarly, the use of groups within the specification depicted in Figure 1 imposes some constraints on the causal relation that force causal independence between the events belonging to *Agent1* and those belonging to *Agent2*. These implicit constraints will be described in greater detail in section 4.3. All of these element and group constraints, however, could have been left until imposed explicitly by the domain describer.

4.2 Temporal Assertions on GEM History Sequences

The structure of a GEM computation actually allows for many alternative physical executions. For example, a computation of the form

$$\begin{array}{ccccc} & & \Longrightarrow & b & \Longrightarrow & & \\ a & & & & & d & \\ & & \Longrightarrow & c & \Longrightarrow & & \end{array} \qquad (1)$$

could be performed in three possible ways in physical time:

1) 1st a 2nd b 3rd c 4th d
2) 1st a 2nd c 3rd b 4th d
3) 1st a 2nd b,c 3rd d

Note that, in the third execution, b and c occur simultaneously. We know that *one* of these time executions must occur, but cannot assume any one of them actually does.

We shall call the possible executions of a computation its *valid history sequences*. A *history* α of a computation C is simply a set of partially ordered events that is a *prefix* of that computation. It may be pictured as the set of events (and the relationships among them) on one side of a potential time line cutting through a computation. In other words, each history represents a possible point an execution of the computation, plus everything that has happened up until then. For example, computation (1) has six histories, each consisting of a given set of events and their relationships:

α_0 :{ } α_i :{a} α_j :{a,b} α_k :{a,c} α_m :{a,b,c} α_n :{a,b,c,d}

Each *valid history sequence (VHS)* represents an execution that is growing with time. Every history in the sequence (except the first) must be a superset of its pre-

decessor. Moreover, two events may enter a given VHS in the same history only if it is possible for them to occur simultaneously, i.e., they have no temporal ordering relationship between them. For example, if e1$\Longrightarrow$e2, then e1 and e2 would have to enter a VHS in distinct histories. In the same vein, if two events *must* take place simultaneously (for example, events $e1$ and $e2$, where $e1 \rightleftharpoons e2$), they must always enter a given VHS in the same history. A history sequence is said to be *complete* if it starts with the empty history. Such a history represents an execution of a computation that starts at a point in time in which no events have yet occurred. In the example given above, there are three potential complete valid history sequences that contain all events a, b, c, d:

S1: $\alpha_0 \quad \alpha_i \quad \alpha_j \quad \alpha_m \quad \alpha_n$

S2: $\alpha_0 \quad \alpha_i \quad \alpha_k \quad \alpha_m \quad \alpha_n$

S3: $\alpha_0 \quad \alpha_i \quad \alpha_m \quad \alpha_n$

We now define a valid history sequence more formally. A sequence of histories S of the form α_0, α_1, α_2,... is a VHS if and only if it has the following properties:

1. The sequence is monotonically increasing: $\alpha_0 \subset \alpha_1 \subset \alpha_2...$

2. Temporally ordered events must enter the sequence in distinct histories:
   $(\forall\, \alpha_i \epsilon S, i > 0)(\forall\, ej, ek \epsilon \{\alpha_i - \alpha_{i-1}\}) \neg [ej \Longrightarrow ek]$

3. Simultaneous events must enter the sequence in the same history:
   $(\forall\, ej, ek)[ej \rightleftharpoons ek \supset (\forall\, \alpha_i)[ej \epsilon \alpha_i \supset ek \epsilon \alpha_i]]$

We will be using first-order temporal logic formulas (*constraints*) to describe the properties of multiagent domains. Simple *immediate* (nontemporal) formulas Q are applied to single histories, e.g., $\alpha \models Q$. The formulas can utilize the usual connectives and quantifiers with their standard interpretation: $\wedge, \vee, \neg, \supset, \Longleftrightarrow, \forall, \exists, \exists!$ (existence of a unique coustant). Event, element, and group instances are used as constants, over which range event, element and group variables. The predicates that may be evaluated with respect to a particular history include the following:

$occurred(e)$	Event e is in the history.
$e \epsilon EL$	Event e is in the history and belongs to element EL.
$e \epsilon G$	Event e is in the history and belongs to group G.
$e1 \rightsquigarrow e2$	$e1$ and $e2$ are in the history and $e1$ causes $e2$.
$e1 \Longrightarrow e2$	$e1$ and $e2$ are in the history and $e1$ temporally precedes $e2$.
$e1 \rightleftharpoons e2$	$e1$ and $e2$ are in the history and must occur simultaneously.

The linear-time temporal (modal) operators $\Box$ (henceforth), $\Diamond$ (eventually), and $\bigcirc$ (next) are logical operators that apply formulas to sequences.[6] In most temporal logics they apply formulas to sequences of states [32]. GEM follows traditional formulations of linear-time temporal logic, but applies the modal operators to sequences of histories (i.e., VHSs).[7] For valid application (and nesting) of temporal operators, each VHS must (and does) have the *tail closure property*: if $S = \alpha_0, \alpha_1, ..., \alpha_{i-1}, \alpha_i, \alpha_{i+1}, ...$ is a VHS, then $S[i] = \alpha_i, \alpha_{i+1}, ...$ is a VHS. Given a history sequence of the form $S = \alpha_0, \alpha_1, ...$, we may then define the semantics of the temporal operators as follows:

- P is *henceforth* true for a sequence S if it is true of every tail sequence of S.

$$S \models \Box P \equiv (\forall i \geq 0)\, S[i] \models P$$

- P is *eventually* true for a sequence S if it is true for some tail sequence of S.

$$S \models \Diamond P \equiv (\exists i \geq 0)\, S[i] \models P$$

- P is true *next* for a sequence S if it is true of the first tail of S.

$$S \models \bigcirc P \equiv S[1] \models P$$

- An immediate (nontemporal) formula Q is true of a VHS S if it is true of the first history in the sequence:

$$S \models Q \equiv \alpha_0 \models Q$$

First-order temporal logic formulas may be applied to GEM computations by viewing a computation C as the set of all its valid history sequences. A computation satisfies a constraint if and only if all its valid history sequences satisfy that constraint:

$$C \models P \equiv (\forall \text{ VHS } S \text{ of } C)\, S \models P\,.$$

---

[6] This is in contrast to branching-time temporal logics, which regard the future as a hranching tree. In these logics, the modalities can vary according to how they are applied to the various paths through that tree. We have found linear-time temporal logic to be adequate as well as simpler to use.

[7] In a way, each history may be viewed simply as a very "large" state.

4.3 GEM Specification Language

How do we define the constraints that delimit the properties of a domain and the legal plans that may be executed within it? The GEM specification language serves this purpose: it is a set of abbreviations and notational devices for writing first-order temporal logic constraints on history sequences. Semantically, any GEM specification σ is equivalent to (can be expanded into) a set of ordinary first-order temporal logic formulas about events, elements, and groups and the relations $\Longrightarrow$, $\rightsquigarrow$, $\rightleftharpoons$, and ϵ. Constraints define a class of executions by stating explicitly:

- What types of events may occur
- How those events must be clustered into elements and groups
- What ordering relationships must exist among the events
- What parameters are associated with events and how the relationships between events affect their parameter values

Every plan generated by the proposed GEM-based planner will be represented by a set of partially ordered events – i.e., by a computation. A given domain specification σ will "admit" only a subset of the possible plans imaginable. The planning algorithms must guarantee that every physical execution (i.e., every valid history sequence) of a generated plan will adhere to the rules of the domain (the constraints in specification σ) and achieve the desired goal.

Just as elements and groups model the structural aspects of a domain, they also serve as the structural components of our specification language. Each specification σ consists of a set of element and group declarations, along with a set of explicit constraints on the events that belong to those elements and groups. Element and group types may be declared and instantiated, and constraints are "scoped"; they are imposed only on the events that belong to the element or group with which they are associated.

For the sake of brevity, we will not describe the entire GEM specification language here. Nor can we possibly integrate the full language into the planner; GEM was designed as a formalism, not as an executable entity. Part of our research effort involves identifying a useful but implementable subset of the full model. We describe below a set of initial constraint types and other language mechanisms that we hope to include in our domain description language and planning system. Appendix A contains a syntax for the portion of the GEM specification language presented in this report.

4.3.1 Delimiting the Set of Admissible Events and Imposing Constraints Based on Structural Relationships

In GEM, the set of events allowed within a plan is delimited by

1. declaring locations of activity (i.e., *elements*) and

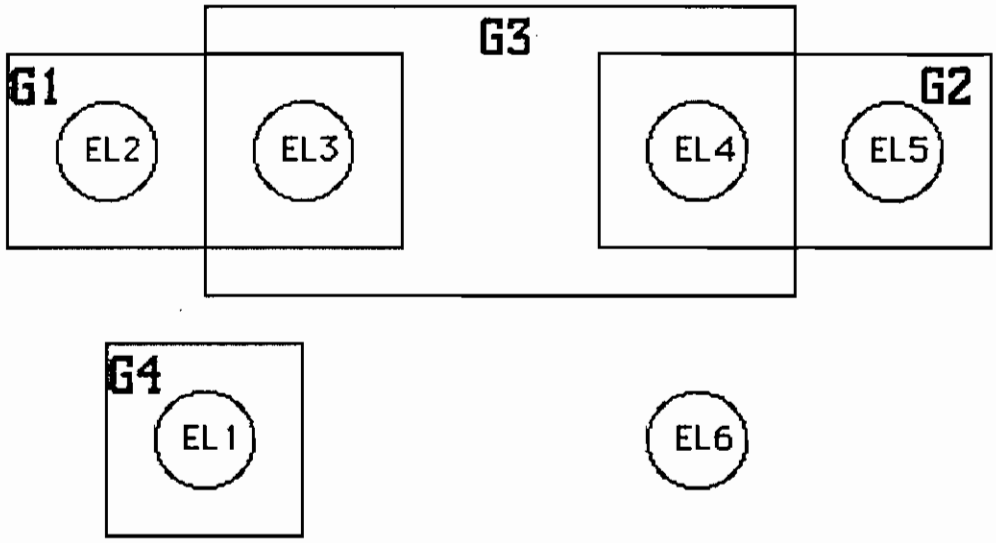
2. explicitly listing the types of events that may occur at each location.

Intuitively, we may think of a location as an agent, and of the events at a location as the legal actions of that agent. Only events occurring at declared locations may be part of a valid plan. Moreover, events belonging to the same element or location are forced to occur sequentially – i.e., they must be totally ordered within the temporal ordering $\Longrightarrow$.

For example, the following element specification models an object called Obj1 associated with two types of events, Move (the action of moving the object to a specified location) and GetCurrentPosition (an action that yields information about the object's current position). Because they belong to the same element, all Move and GetCurrentPosition events belonging to Obj1 are constrained implicitly to be totally ordered within the temporal order $\Longrightarrow$.

```
Obj1 = ELEMENT
EVENTS
  Move(x:INTEGER, y:INTEGER)
  GetCurrentPosition(lastx:INTEGER, lasty:INTEGER)
END Obj1
```

Just as clustering events into elements is a way of constraining their temporal ordering, a way to limit the permitted causal relationships among events is to cluster them into *groups*. Each group definition serves as an implicit constraint on the causal relation between events. Groups can be used to model the causal limitations imposed by physical proximity, programming-language scope rules, and other causal limitations related to structure. In fact, the use of structure as an implicit constraint on the causal effects of actions is one way of handling the frame problem.




```
ELEMENTS   EL1,EL2,EL3,EL4,EL5,EL6
G1 = GROUP(EL2,EL3)
G2 = GROUP(EL4,EL5)
G3 = GROUP(EL3,EL4)
G4 = GROUP(EL1)
```

Figure 2: A Group Structure

As an example of the use of group structure, consider the group/element structure depicted in Figure 2. The possible causal relationships among the events belonging to the elements in the figure are limited as follows:

An event in	*may only cause an event in*
$EL1$	$EL1, EL6$
$EL2$	$EL2, EL3, EL6$
$EL3$	$EL2, EL3, EL4, EL6$
$EL4$	$EL3, EL4, EL5, EL6$
$EL5$	$EL4, EL5, EL6$
$EL6$	$EL6$

Note how $G1$, $G2$, and $G4$ could be representations of agents, with $G3$ modeling a communication channel between $G1$ and $G2$, and with $EL6$ representing a resource shared by all three agents.

Certain events may also be designated as *ports*. If so, they serve as "access holes" to their groups [22]. More specifically and formally, we can define the constraint on the causal relation imposed by a group structure as follows. Given that $e1 \epsilon EL1$ and $e2 \epsilon EL2$, $e1 \rightsquigarrow e2$ is allowed only if

$$access(EL1, EL2) \vee [port(e2, G) \wedge access(EL1, G)].$$

We define $access(x, y)$ to be true if either (1) x and y belong to the same group or (2) y is global to x. If we assume that all elements and groups within a specification are surrounded by a single group (an assumption that does not further constrain the permitted causal relationships among events), we can define $access(x, y)$ as follows:

$$access(x, y) \equiv (\exists G)[y \epsilon G \wedge contained(x, G)] ,$$

where $y \epsilon G$ denotes direct membership and

$$contained(x, G) \equiv x \epsilon G \vee (\exists G')[x \epsilon G' \wedge groupcontained(G', G)]$$
$$groupcontained(G1, G2) \equiv (\forall x)[x \epsilon G1 \supset x \epsilon G2] .$$

4.3.2 Attaching Additional Constraints to Elements and Groups

As we have indicated, each GEM specification consists of a set of element and group declarations. In addition to the constraints imposed implicitly by group/element structure, constraints may also be associated with each element or group. These constraints pertain only to the events belonging to the particular element or group in which the constraint is defined. For example, we could augment our definition of Obj1 as follows:

```
Obj1 = ELEMENT
EVENTS
  Move(x:INTEGER, y:INTEGER)
  GetCurrentPosition(lastx:INTEGER, lasty:INTEGER)
CONSTRAINTS
1) (∀ move:Move, currpos:GetCurrentPosition)
  [(move⟹currpos) ∧ ¬(∃ anothermove:Move) [move⟹anothermove⟹currpos]]
  ⊃ move.x = currpos.lastx ∧ move.y = currpos.lasty
END Obj1
```

Constraint 1 states that each CurrentPosition event belonging to Obj1 must truly yield the most current position of object Obj1.

4.3.3 Element and Group Types

To alleviate the specification task further, GEM also includes a mechanism for describing element and group *types*. Each instance of a defined type is a unique element or group with a structure identical to that of its type description. From the standpont of semantics, the use of types and instances may be viewed as a simple text substitution facility; each type instance is shorthand for a separate but identical element or group declaration. The GEM type facility is extremely flexible and expressive; types may not only be parameterized, but may also be defined as refinements of other previously defined types.

As an illustration, we return once again to our description of Obj1. We might form a more generic Object element type description, as follows:

```
Object = ELEMENT TYPE
EVENTS
  Move(x:INTEGER, y:INTEGER)
  GetCurrentPosition(lastx:INTEGER, lasty:INTEGER)
CONSTRAINTS
  :
END Object
```

Obj1, as well as any other such object, may then be described as an instance of type Object:

```
Obj1 = Object ELEMENT
```

We could also form a new element type that is a refinement of Object. For example, a LimitedObject is a type of Object with a limited range of movement.

```
LimitedObject(xlim:INTEGER, ylim:INTEGER) = Object ELEMENT TYPE
ADD CONSTRAINT:
(∀ move(x,y):Move) [ |x| ≤ xlim ∧ |y| ≤ ylim ]
```

We have found both the scoping of constraints (i.e., applying constraints only to the groups and elements in which they are defined) and the use of group and element type-hierarchies to be invaluable specification tools. Without them, writing complex specifications would become almost impossible.

4.3.4 Common Constraint Abbreviations

In principle, the constraints associated with elements and groups may be any first-order temporal logic formulas. However, certain constraints arise so frequently that abbreviations for them are particularly useful. We list some of the most commonly employed ones below. Because of the computational infeasibility of enforcing *all* first-order constraints, it is likely that a certain subset of constraint types or *constraint paradigms* will be chosen as targets for implementation in our proposed planner. We hope to include all of the constraint forms listed below.

- **Prerequisite Constraints**
  Prerequisite constraints are used for restricting the causal relation $\rightsquigarrow$ among events. These constraints form the backbone of most specifications and we believe them to be easily enforceable by a planner. In general, prerequisite constraints serve a *generative* function in forming plans – they result in the insertion of new events into the partial ordering.

  - *Simple Prerequisite:* $E1 \longrightarrow E2$
    Each event of type $E2$ must be caused by exactly one event of type $E1$, and each event of type $E1$ can cause at most one event of type $E2$. In essence, this is a one-to-one causal requirement. For example, in the preceding section we had Robot.Request $\longrightarrow$ Robot.Usetool.

    $$E1 \longrightarrow E2 \equiv$$
    $$(\forall e2 : E2)(\exists! e1 : E1)[e1 \rightsquigarrow e2]] \wedge$$
    $$(\forall e1 : E1)(\exists \textit{ at most one } e2 : E2)[e1 \rightsquigarrow e2]$$

  - *Nondeterministic Prerequisite:* $\{E1, ...En\} \longrightarrow +E$
    Each event of type E must be caused by exactly one event of type $E1$ or $E2$ or... En, and an event of type $E1$ or... En can cause at most one event of type E.

    $$\{E1, ..., En\} \longrightarrow +E \equiv$$
    $$(\forall e : E)(\exists! ei : \{E1, ..., En\})[ei \rightsquigarrow e] \wedge$$
    $$(\forall ei : \{E1, ..., En\})(\exists \textit{ at most one } e : E)[ei \rightsquigarrow e]$$

  - *Fork:* $E \longrightarrow \{E1, ...En\} \equiv (\forall i, 1 \leq i \leq n)\, E \longrightarrow Ei$
  - *Join:* $\{E1, ...En\} \longrightarrow E \equiv (\forall i, 1 \leq i \leq n)\, Ei \longrightarrow E$

- **Simultaneous Activity**
  A required simultaneous occurrence of, say, each event of type $E1$ with an event of type $E2$ could be described by a constraint of the following form:

$$(\forall e1 : E1)(\exists e2 : E2)[e1 \rightleftharpoons e2] \wedge (\forall e2 : E2)(\exists e1 : E1)[e1 \rightleftharpoons e2]$$

- **Priority and Mutual Exclusion**
  Suppose that we define $e1$ *cbefore* $E2$ (causal before) as follows:

$$e1 \; cbefore \; E2 \equiv occurred(e1) \wedge \neg(\exists e2 : E2)[e1 \rightsquigarrow e2]$$

  ($e1$ has occurred but has not yet caused an event of type $E2$). We can then express priority and mutual exclusion properties by using the following kinds of constraints:

  - *Priority of transitions of the form* $e1 \rightsquigarrow e2$ *over those of the form* $e3 \rightsquigarrow e4$ *:*

$$(e1 \; cbefore \; E2 \wedge e3 \; cbefore \; E4) \supset \Box \, [e3 \rightsquigarrow e4 \supset e1 \rightsquigarrow e2]$$

  - *Mutual exclusion between any events occurring between* $e1$ *and a corresponding event of type* $E2$*, and those between* $e3$ *and a corresponding event of type* $E4$*:*

$$\neg(e1 \; cbefore \; E2 \wedge e3 \; cbefore \; E4)$$

    For example, if two robots cannot wait for a tool at the same time, we would have

$$(\forall request1, request2 : Robot.Request, request1 \neq request2)$$
$$\neg(request1 \; cbefore \; UseTool \wedge request2 \; cbefore \; UseTool)$$

    If we define a similar *tbefore* abbreviation as follows:

$$e1 \; tbefore \; E2 \equiv occurred(e1) \wedge \neg(\exists e2 : E2)[e1 \Longrightarrow e2]$$

    then more strictly temporal forms of mutual exclusion (or priority) can be expressed, i.e., constraints of the form

$$\neg(e1 \; tbefore \; E2 \wedge e3 \; tbefore \; E4).$$

    Finally, simple mutual exclusion between two events would reduce to a restriction of the form $e1 \Longrightarrow e2 \vee e2 \Longrightarrow e1$.

Priority and mutual exclusion constraints will be harder for a planner to enforce than prerequisite constraints because all possible executions of a given plan must often be considered. We discuss alternative approaches to this problem in Section 6.

- **Event Preconditions**
  Using the modal operator $\bigcirc$ (next), we can easily express the notion of an event *precondition*. If the precondition for an event e is defined by a formula P, we simply require that P be true in the history just preceding the history in which e enters a valid history sequence. Specifically, we could define the abbreviation $Precondition(e, P)$ as follows:

$$Precondition(e, P) \equiv (\bigcirc occurred(e) \wedge \neg occurred(e)) \supset P$$

  If we use the abbreviation

$$occursnext(e) \equiv \bigcirc occurred(e) \wedge \neg occurred(e)\ ,$$

  then $Precondition(e, P)$ reduces to $occursnext(e) \supset P$.

5 Proposed Extensions of GEM

While GEM, as it is currently defined, is an extremely powerful specification tool, there are definitely many potential areas in which it could be enhanced as a specification framework for multiagent planning. Below we consider three of these areas and discuss various related issues and prospective approaches.

5.1 Action Hierarchies

One extension of GEM that we would like to implement is the capability of modeling composite actions – that is, actions whose descriptions can be decomposed into sets of partially ordered events. This extension would be relatively easy to do if "composite events" were regarded merely as abbreviations of more complex networks of events.

For example, we could define an event type A that would always expand into a temporal network of the form:

$$\begin{array}{ccccccc} & & & \Longrightarrow & C & \Longrightarrow & \\ \mathit{BeginA} & \Longrightarrow & B & & & & E \Longrightarrow \mathit{EndA} \\ & & & \Longrightarrow & D & \Longrightarrow & \end{array}$$

Each instance a of type A would be considered as beginning with an event instance of type BeginA, containing event instances of type B, C, D, E with the prescribed temporal relationships, and ending with one of type EndA. The event types BeginA and EndA serve as a sort of interface between A and the rest of the specification. For instance, if we had a constraint of the form $F \longrightarrow A$, it would simply be expanded into $F \longrightarrow \mathit{BeginA}$. This type of hierarchy was utilized in my dissertation on GEM [22] and was referred to as *event abbreviation.*

If we want the hierarchical specification of actions to preserve certain semantic properties, however, the problem becomes greatly complicated. For example, suppose we write a specification that uses a [noncomposite] event type A, and then go on to generate a plan that utilizes events of type A – namely, a plan that conforms to all of the constraints in our initial domain specification. Now, suppose that we wish to alter our specification and decompose A further, say, into the form described above. (This would be a desirable method of constructing a domain specification because it conforms to the general principles of top-down design.) The question is, is our previous plan still valid? Unfortunately, it may not be.

For example, we could have a constraint about the event types composing A that was not violated in the original plan, but is now violated in the plan in which event A

has been expanded. In hierarchical planners such as NOAH, this sort of problem was handled by requiring that system constraints be rechecked after every event expansion. We could of course utilize this same "brute force" approach. A more desirable solution, however, would be to find a set of restrictions on domain constraints that permits hierarchical expansion without subsequent rechecking after expansion.

One such restriction, for example, might require that the set of event types composing an expanded event *not* be part of the original specification. Under this restriction, event expansion would necessarily be accompanied by the addition of new event types to the specification itself (as well as new pertinent constraints). In essence, we would be requiring that there be no "interaction" between the the events composing an event and the other events in the original plan [22]. This methodology conforms to the notion of event expansion as the process of mapping a high level plan onto one that is at a lower level of detail. Even when using a top-down method of development, however, there will still be cases where such prohibitive restrictions could not be obeyed. In these situations rechecking would have to be done.

5.2 State Descriptions and the Frame Problem

One of our more important goals in extending GEM is to allow for different ways of describing state. Despite the emphasis in this paper on describing domain properties strictly in terms of action interrelationships rather than in terms of the state of their environment, some properties really do lend themselves better to state-based description. For instance, this is true for properties that specifically deal with the environment in which actions occur (one example might be a requirement for the maintenance of the integrity of a data base).

State-based descriptions can also be more modular. In other words, the ways of achieving a particular state can easily be varied and expanded without significantly altering the domain description as a whole. For example, suppose P is prerequisite to an action A. There may be many equally good ways of achieving P, any one of which could be inserted into a plan before A. In addition, new ways of achieving P may arise in future. Traditional state-predicate approaches make it easy to add these new methods to the domain description without altering it significantly. One need only assert which actions make P true (or untrue); a state-based planner will then be able to easily use any of them to achieve P. In contrast, a ***strictly*** behavioral approach has to describe explicitly all the action patterns that enable A to occur. This type of description might be more difficult to modify or amend.

Obviously, there is a trade-off between using strictly behavioral-style descriptions and using those that are based on state predicates. A property that really has to do with specific action relationships should be described as such; a property that can more easily be described in terms of a state abstraction should be described by using state. Luckily, there is no difficulty in employing both types of description within the GEM specification framework.

Probably the most straightforward way of integrating state descriptions into GEM is by using *derived state predicates*. Such predicates are really formulas about actions and their relationships. We have already seen some uses of derived state predicates – for example, the definitions of the infix predicates *cbefore* and *tbefore*. As another example, we could use the following derived formula on execution history to define the predicate HandEmpty(R) (robot R's hand is empty):

$$\begin{aligned}\mathsf{HandEmpty(R)} \equiv \neg\ &(\exists\ \mathsf{pickup{:}R.PickUp})\ [\mathsf{occurred(pickup)} \wedge \\ &\neg\ (\exists\ \mathsf{putdown{:}R.PutDown})\ [\mathsf{pickup} \rightsquigarrow \mathsf{putdown}]]\end{aligned}$$

or, equivalently (using the *cbefore* predicate),

$$\mathsf{HandEmpty(R)} \equiv \neg\ (\exists\ \mathsf{pickup{:}R.PickUp})\ [\mathsf{pickup\ cbefore\ R.PutDown}]\ .$$

These definitions state that a robot's hand is empty if it has not executed any PickUp action that has not been followed by a corresponding PutDown action.

The derived-state-predicate approach, however, does suffer from some of the same modularity problems described above. In order to describe a derived predicate, all action patterns that make that predicate true have to be forseen. However, we make this method of state description more modular and extensible by defining a derived state predicate P as the disjunct of all action patterns that are supplied as a means of achieving P over the course of a specification. With this methodology, new ways of achieving P could easily be added.

One alternative "twist" on the derived predicate approach is to cast traditional state-predicate add/delete lists in terms of the GEM formalism. Specifically, for each state predicate P, we could define two generic event classes: $Add(P)$ and $Delete(P)$. Any event type considered to be one that makes P true would be classified as an $Add(P)$ event type, while any event type that makes P false would be a $Delete(P)$ event type. For example, in the robot scenario described above, we might classify R.PickUp as a Delete(HandEmpty(R)) event type and R.PutDown as an Add(HandEmpty(R)) event type. Any predicate P for which $Add(P)$ and $Delete(P)$ event types have been declared could then be described by the derived formula

$$P \equiv (\exists\ \text{adder:Add(P)})[\text{occurred(adder)} \wedge (\forall\ \text{deleter:Delete(P)})[\text{occurred(deleter)} \supset \text{deleter} \Longrightarrow \text{adder}]]$$

Although this derived formula seems to be a second-order definition, the $Add(P)$ and $Delete(P)$ definitions should just be regarded as abbreviations that are expanded into sets of event types. The generic derived formulas for all predicates P are then instantiated for each predicate P in the specification. Both of these steps preserve the first-order nature of the specification.

It is also interesting to note that this method of defining state bears great similarity to the definitions of what it means "for a property to hold" that were given in recent theses by Pednault [33] and Chapman [6]. Both definitions have the flavor of: "something has made the property hold, and nothing could have possibly occurred afterwards that makes it not hold." We reversed the notion of "nothing bad has occurred afterwards" to "every bad thing must have occurred before" to handle the possible case of simultaneous occurrence of an *Add* and a *Delete* event. Because Pednault investigated only sequential plans, this case was not considered. Chapman's definition for partially ordered plans, however, is equivalent to ours.

Actually, the definition given above does not fully capture the conventional uses of add/delete lists. To do so, we would also have to specify the *conditions* under which an event qualifies as an "adder" or "deleter" of a predicate. Traditionally, this has been done by specifying the effects of an action in the same context that action preconditions are supplied. Axioms of the form "When action A occurs, predicates $P_1 ... P_n$ must have been in the preceding state, and afterwards, predicates $Q_1 ... Q_m$ are deleted and predicates $R_1 ... R_k$ are added" are utilized. The precondition predicates in these axioms are often used in a way that constrains the added and deleted predicates. For example, we might say "In order for a robot R to put down a block, predicate $Holding(R, n)$ (i.e., R is holding n blocks) must have been true in the preceding state, and afterwards, predicate $Holding(R, n)$ is deleted and $Holding(R, n-1)$ is added."

To fully mimic this traditional way of specifying the effects of events, we must *qualify* our delineation of *Add* and *Delete* events.[8] Each event type that is supplied as a member of $Add(P)$ or $Delete(P)$ would also have to be associated with a derived predicate (event precondition) that must be true in order for an event of that type to qualify as an "adder" or "deleter" of P.

[8] Of course, in a behavioral framework, state can be encoded in several ways, thus making the qualification of "adder" and "deleter" events unnecessary anyway. For example, to encode *Holding(R,n)* we could use a derived formula that counts the number of PickUp and PutDown events executed by robot R. To fully mimic the traditional add/delete approach, however, event qualification is needed.

More formally, we could use the function *add-pre(e,P)* to denote the precondition that must be true before event e is executed in order for it to qualify as an "adder" of P. Similarly, we would use the notation *delete-pre(e,P)* for a "deleter" of P. Then, assuming that we want *E1,... En* to be the event types which potentially make P true, and *F1,... Fm* those that make P false, we would use the following meta-level definition to qualify the *Add(P)* and *Delete(P)* event classes with respect to a given VHS S:

$$\mathsf{Add(P) = \{\ e{:}\{E1....\ En\} \mid (\forall\ \alpha \epsilon\ S)\ \alpha \models PreCondition(e,add\text{-}pre(e,P))\ \}}$$

$$\mathsf{Delete(P) = \{\ f{:}\{F1....\ Fm\} \mid (\forall\ \alpha \epsilon\ S)\ \alpha \models PreCondition(f,delete\text{-}pre(f,P))\ \}}$$

In other words, the "adders" of P are all those events of type *E1* or ... *En* whose associated "adder precondition" is true in the history just before they enter VHS S. The analogous description is then used for the "deleters" of P. We are also currently investigating the use of *backwards* temporal operators (operators that apply formulas to past histories) that will obviate the need for a meta-level definition as used above. Such backwards temporal operators will also allow even richer derived state formulas to be written.

As an extension to the *Add/Delete* approach we might also want to allow for the possibility of using "state recognition" or "state testing" events. In real world situations, an environmental state may not have been deliberately achieved via a planned action, but rather, the state has simply been observed. For example, we might notice that a box is red, although we never painted it that color. In such cases, we could use a special purpose axiom of the form

$$(\forall P)[Observe(P) \epsilon Add(P)].$$

Namely, if an agent somehow observes that state predicate P is true, then as far as our representation is concerned, P is true; the observation serves as an unqualified *Add(P)* event. Similarly, we might use an axiom of form

$$(\forall P)[Observe(\neg P) \epsilon Delete(P)].$$

Despite its similarity to the add/delete lists used by planners, the method of describing state discussed here is still *derived* – that is, the values of state predicates are still derived or computed from the action structure of a given plan or computation history. Whereas events are definitely "first class" objects in our model, state predicates are not. This standpoint is, of course, natural within an event-based framework such as GEM. It also has an interesting advantage – states are always *calculated.* This

encourages domain describers to avoid some of the ambiguous situations that arise when a purely state-based approach is in use.

For example, suppose we have predicates of the form *Distance(R1, R2, dist)*, *Location(R1, loc1)* and *Location(R2, loc2)* describing the distance between two robots and their respective locations. Descriptions of domain actions *Move(R1, loc3)*, *Move(R2, loc4)* might specify that these events add predicates of the form *Location(R1, loc3)*, *Location(R2, loc4)* to the domain description and delete the predicates *Location(R1, loc1)*, *Location(R2, loc2)*. But how is *Distance(R1, R2, dist)* affected over time? This is dependent on the relative ordering (or even possible simultaneity) of the two actions. It is also influenced by the peculiarities of the domain itself. For example, suppose that the distance between two robots is always fixed – they might be attached to one another by a rigid bar – and that the movement of one robot automatically changes the location of the other. In this case, the *Location* and *Distance* predicates must be computed differently than they would be in other domains. Traditional state-based approaches of manipulating state (i.e. action add/delete lists) do not deal effectively with these problems. But in a framework where state is always derived or calculated, the value of the three predicates is always defined – by derived predicate formulas which have been set up to reflect the properties relevant to a given domain.

Our "calculated" view of state predicates has some other interesting consequences. For example, suppose we asserted that $P \supset Q$ – i.e., if state predicate P is true, then predicate Q is true. How can this assertion be interpreted within our action-based framework? Several approaches might be taken. For example, given that P has been described by some action pattern and Q has not, this assertion might be intepreted as stating that P's action pattern can serve as a definition of Q. If Q has already been defined by its own derived action formula, however, an assertion such as $P \supset Q$ might be interpreted as an action contraint: i.e., that the action formula describing Q must be true whenever the formula for P is true. How to interpret assertions on state predicates is a topic of ongoing research.

How does all of this affect the *frame problem* – i.e., the problem of determining how actions influence the truth or falsity of state predicates? Strictly speaking, purely behavioral descriptions do not really need to address conventional manifestations of the frame problem. The state of the world is considered to change only as events occur; i.e., the "state" *is* the set of event occurrences, and the only type of changes in that state are the occurrences of events.

When using a state predicate described in terms of action occurrences, the frame axiom in force is quite simple: an atomic formula of form $P(a, b)$ changes value only

when events occur that alter the value of P's defining formula. This axiom is also used when we cast a state description in terms of *Add* and *Delete* event classes; at base, this form of description expands into a derived formula. However, by using the *Add/Delete* mechanism we do implicitly assume that the value of a predicate P changes only as it is explicitly manipulated by the $Add(P)$ and $Delete(P)$ actions. While this resembles the "STRIPS" assumption on the surface, it differs in that no assumption is made about actions occurring in isolation. Whether actions occur sequentially or simultaneously, a predicate P's defining formula will yield a truth value for P.

A problem related to the frame problem is the difficulty of specifying all possible circumstances that could affect an action – the ***qualification problem.*** This is exemplified by the "potato in the tailpipe" puzzle – i.e., how can one specify *all* actions that could possibly affect a driver's ability to start a car? One approach might be to *classify* event types, much as we did above for $Add(P)$ and $Delete(P)$. For example, we could have a generic BreakCar event class, and assert that all events of this class prevent one from starting a car. In the course of building a specification, we could classify certain specific event types as BreakCar events. This use of the BreakCar event class is similar to an abnormality condition in McCarthy's work on circumscription [26].

Finally, as we suggested earlier, GEM's ability to structure events into elements and groups is another way of modeling limitations of effect. For instance, if an agent is modeled as having no causal access to a car, then it could not possibly perform any action that could prevent it from being started. This technique is actually opposite to the circumscription approach.

5.3 Explicit Time and Temporal Intervals

The current formulation of GEM includes no explicit use of real time. GEM's use of *relative* time (i.e., representing the order of actions rather than the exact times they occur), however, is probably the most reasonable representation of time for most multiagent applications. After all, when dealing with problems involving concurrency, one must consider potential race conditions affecting action orderings[9] as well as other synchronization concerns. Reliance on real-time estimates for purposes of synchronization can be quite risky.

It would be relatively easy, though, to add the notion of real time to GEM. One approach would be to associate each event with a time parameter. For example, an

[9] A *race condition* occurs when the outcome of a computation depends on the relative speeds of its component processes.

event instance denoted by $e(..., t)$ would be regarded as occurring at time t. Given this formulation, we would impose the following additional constraint upon all computations:

$$(\forall\ e1(t1).\ e2(t2)\text{: EVENT})\ [\ t1 < t2 \supset e1(t1) \Longrightarrow e2(t2)\ ]$$

In other words, given an event $e1$ occurring at time $t1$ and another event $e2$ occurring at time $t2$, if $t1 < t2$, then $e1$ must occur before $e2$ in the temporal order. This constraint would force all valid history sequences of a computation containing timed events to properly reflect the required ordering of those events in time.

Our current formulation of GEM also does not explictly address the problem of representing time *intervals* or events with duration. As we indicated earlier, however, these concepts could he integrated into GEM by means of a few notational additions.

Suppose we had actions a and b that are considered to occur over periods of time. To represent the duration of these events and the possible relationships of the intervals over which they occur, we could begin by representing each single action with two actions: begin(a). end(a). begin(b). end(b). We could then go on to define the possible interval relationships between a and b as follows (these are the same interval relationships used by Allen [2]):

a before b	$\equiv$	$\text{end(a)} \Longrightarrow \text{begin(a)}$
a equal b	$\equiv$	$\text{begin(a)} \rightleftharpoons \text{begin(b)} \wedge \text{end(a)} \rightleftharpoons \text{end(b)}$
a meets b	$\equiv$	$\text{occursnext(end(a))} \supset \bigcirc\ \text{occursnext(begin(b))}$
a overlaps b	$\equiv$	$\text{begin(a)} \Longrightarrow \text{begin(b)} \wedge \text{end(a)} \Longrightarrow \text{end(b)}$
a during b	$\equiv$	$\text{begin(b)} \Longrightarrow \text{begin(a)} \wedge \text{end(a)} \Longrightarrow \text{end(b)}$
a starts b	$\equiv$	$\text{begin(a)} \rightleftharpoons \text{begin(b)} \wedge \text{end(a)} \Longrightarrow \text{end(b)}$
a finishes b	$\equiv$	$\text{begin(b)} \Longrightarrow \text{begin(a)} \wedge \text{end(b)} \rightleftharpoons \text{end(a)}$

A predicate P considered to hold over a specific interval (say, the interval between an event of type Begin(P) and a corresponding event of type End(P)) could be defined by the following derived formula: $\text{P} \equiv (\exists\ \text{begin(p):Begin(P)})\ [\ \text{begin(p) tbefore End(P)}\ ]$ Future work with GEM will establish the actual utility of these proposed extensions.

6 Planning Method

How can we utilize the GEM specification framework as the basis of a planning method? In this section we describe some initial ideas concerning "behavioral" planning. In the next section we shall illustrate these ideas with an example from the blocks world domain.

Like most planners, the input to our system will be a set of domain constraints as well as descriptions of initial and goal configurations. These initial and goal configurations, or "states," might be described by partially ordered sets of events. For example, the initial state in a blocks world might be a partially ordered set of block initialization events. The goal state might be described by a set of final PutDown events; for example, final actions of the form PutDown(A,B) and PutDown(B,C) could be used to describe a final stack of blocks A on B on C.

Alternatively, we could describe the initial state in terms of a set of domain-specific predicates that are defined to be initially true. For example, suppose that we have a [nontemporal] predicate P and want to assert that P is initially true. In essence, we would like the following meta-level formula to be true:

$$(\forall \text{ complete VHS S})[\ S \models P\ ]$$

or equivalently,

$$(\forall \text{ complete VHS S})[\ \alpha_0 \models P\ ]\ .$$

We will abbreviate this formula as INIT P. Initialization conditions can then be expressed as a set of meta-level axioms of this form.

Goal conditions can also be described in terms of state predicates. For instance, in the blocks world domain, we could define On(A,B) as

$$[\ \text{INIT On(A,B)} \wedge \neg\ (\exists\ \text{pickup:PickUp(A)})[\text{occurred(pickup)}]\ ] \vee$$
$$(\exists\ \text{putdown:PutDown(A,B)})$$
$$[\ \text{occurred(putdown)} \wedge \neg\ (\exists\ \text{pickup:PickUp(A)})\ [\text{putdown} \Longrightarrow \text{pickup}]\ ]$$

In other words, A is *on* B if A has been initialized to be on B or as been put down on B and if it has not yet been picked up. A goal of the form On(A,B) could be satisfied by a plan that is defined to satisfy INIT On(A,B) and contains no PickUp(A) event, *or* contains a PutDown(A,B) event that has not been followed by a PickUp(A) event.[10]

Whatever their form, the initialization and goal descriptions present the planner with an initial partial ordering of events – the "beginning" and "end" of a net of actions – with the possible addition of some initialization axioms. The domain constraints are then used to fill in that net. An important planning heuristic is the *order* in which those constraints are applied.

[10]Of course, a conjunctive goal would simply be satisfied by finding events that satisfy the conjunction of its component subgoals. Usually this will be the *union* of a set of events or event patterns.

One good rule of thumb is to impose prerequisite constraints first. These constraints automatically insert new required events (or event alternatives) into the ordering. Other constraints can then be used to constrain this partial ordering further. To choose among event alternatives (for example, we might know that event *a* must be caused by *either* an event *b* or an event *c*), we might make a random or weighted selection and allow for later backtracking. Although we may use backtracking, we would also like to adhere to a least-commitment philosophy. That is, many decisions (for example, event parameter binding) can be put off until they are forced. SIPE, NONLIN, DEVISER, and other planners also combine backtracking with least-commitment.

Another planning heuristic is to use the constraints themselves to help make plan choices. For example, suppose that one of the domain constraints describes the preconditions for event *a*. Given the choice described above, we could decide between *b* and *c* according to which event makes *a*'s preconditions true (or untrue). A graphic user interface to the planner could also be utilized for resolving choices. As a plan is constructed, it could be presented graphically to the user. At this time, the system could request user guidance by offering specific plan choices. Such an interface could also be used to create visual simulations of possible plan executions.

Notice how a planner based on behavioral constraints is really quite distinctive in its approach. The traditional state-based pattern is usually: goal subdivision, separate plan construction, and finally, plan integration and critique. In contrast, a behavioral planner builds plans through direct imposition of constraints on action orderings. Because these constraints are completely domain-specific and can express a wide range of properties, they may force earlier and more intelligent pruning of the planning space. However, such a planning system must also be stronger: it must know how to impose a constraint automatically, given its definition. We are thus moving much of the tedious and error-prone work of the domain describer onto the planner itself. One of the major tasks in our work will be to find a compromise between constraint expressiveness and efficient implementation. We hope to gain from recent work being done on efficient planning techniques by Simmons [38] and by Chapman [6].

One promising area for investigation is the question of whether some forms of constraints can be satisfied better by performing checks (e.g., certain kinds of synchronization) during plan execution. This option would involve development of an enhanced plan interpreter or executor. For example, if the interpreter included primitives for guarding critical sections (a monitor, say), mutually exclusive regions of a domain could be protected by having plans invoke synchronization primitives within the interpreter. One possibility might be to use Manna and Wolper's [25] algorithm

(and Stuart's implementation [41]) for automatically generating a synchronizer from temporal logic constraints.

A behavioral planning technique may also facilitate plan explanation. In a constraint-based planner, each relationship imposed between two actions can be attributed to a specific constraint or set of constraints. In addition, the causal relationships among actions are embedded in the plan itself. By associating each relation between events with the set of constraints that caused that relation to exist, we will effect a method of explaining the rationale behind a plan.

One of the problems we will confront in developing constraint adherence algorithms is how to reason about the potentially explosive number of executions of a given partial order. This problem becomes especially grave when dealing with the possible "pasts" of a given event, or with the temporal operators $\Box$ ("henceforth") and $\Diamond$ ("eventually"). For example, if we assert $\Box\ p$ (henceforth p must be true), then we must check that p is true of every state of every possible execution of the plan. There are several ways of getting around this problem: retaining global information about a plan between histories, having knowledge about the invariance properties of certain predicates, and limiting the form of p to something that can be easily checked. For example, if we have $\Box\ e1\ cbefore\ E2$, we can simply make sure that $e1$ never causes an event of type $E2$. In general, we feel that a judicious choice of constraint paradigms, combined with the maintenance of global information, will alleviate many implementation problems. The above-cited work by Simmons and Chapman also attempts to address some of these problems by applying similar methods.

7 An Example: The Blocks World

In this section we present a GEM specification of a blocks world domain, and show how the domain constraints can be used to generate a plan for moving blocks.

7.1 Blocks World Specification

The blocks world is made up of a table and two types of objects: passive agents called *blocks* and active agents called *robots* that manipulate blocks. Because they are passive, no real actions are performed by blocks. In contrast, each robot may perform two types of actions. An action of the form PickUp(b1) models a robot picking up a block b1. PutDown(b1,b2) models the action of placing a block b1 down on top of b2, where b2 is either another block or TABLE. Robot constraint 1 states that each time a

robot puts down a block b1, it must have previously picked up that same block. Robot constraint 2 states that each PickUp action (except the first) must be preceded by some corresponding PutDown action. The combined effect of these two constraints is that each robot's actions must alternate between PickUp and PutDown actions, starting with a PickUp action. Thus, a robot cannot pick up two blocks at once, nor can it deposit an object twice, unless there is an intervening PickUp action.

TABLE = ELEMENT
END TABLE

Block = ELEMENT TYPE
END Block

Robot = ELEMENT TYPE
EVENTS
  PickUp(b1:Block)
  PutDown(b1:Block, b2:{Block, TABLE})
CONSTRAINTS
1) PickUp(b1) $\longrightarrow$ PutDown(b1,b2)
2) $i > 1 \supset$ PutDown $\longrightarrow$ PickUp^i
END Robot

Finally, we model the entire domain as a group called BlocksWorld, consisting of TABLE, a set of Block elements, and Robot elements. The first constraint precludes a block from being manipulated by more than one robot at a time. It does this by imposing a mutual exclusion constraint on uses of the block, wherein block usage encompasses the period between the time a block is picked up and the time it is deposited. In single-agent worlds, this constraint would be unnecessary.

BlocksWorld = GROUP TYPE (TABLE, {b}:SET OF Block, {r}:SET OF Robot)
CONSTRAINTS
1) $(\forall$ b:Block) $\Box \neg (\exists$ pickup1,pickup2:PickUp(b), pickup1$\neq$pickup2)
[pickup1 cbefore PutDown $\wedge$ pickup2 cbefore PutDown]
2) $(\forall$ pickup(b):PickUp) [PreCondition(pickup(b), Clear(b))] $\wedge$
$(\forall$ putdown(b2,b):PutDown) [PreCondition(putdown(b2,b), Clear(b))]
END BlocksWorld

The second blocks world constraint requires that, if a robot picks up a block b or puts down another block on top of a block b, then block b must have been "clear" in the preceding history. In other words, there can be no obstructions that would prevent

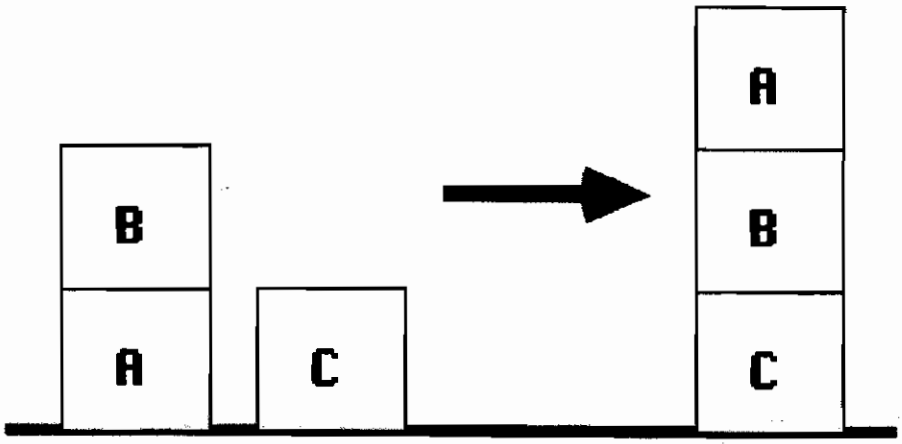


Figure 3: Initial and Goal Configuration of Blocks A, B and C

a robot from carrying out a PickUp or PutDown action. This constraint is actually a perfect example of one that might be best defined using state-based methods – that is, it could be defined by delineating a set of Add(Clear(b)) and Delete(Clear(b)) event types. For now however, we opt for the derived-formula definition of the Clear predicate given below. This definition states that b is "clear" if it is the table (the table is always considered to be clear) or, if b is a block, it is not currently in a robot's hand (all actions of form PickUp(b,b1) have been followed by a corresponding PutDown action) and there is no block on top of b (all actions of form PutDown(b2,b) or initializations of form INIT On(b2,b) have been followed by a corresponding PickUp(b) action).

$$
\begin{aligned}
&\mathsf{Clear}(b) \equiv \\
&(b = \mathsf{TABLE}) \vee \\
&[\ (\forall\ \mathsf{pickup{:}PickUp}(b))[\mathsf{occurred}(\mathsf{pickup}) \supset \\
&\qquad\qquad (\exists\ \mathsf{putdown{:}PutDown})[\mathsf{pickup} \rightsquigarrow \mathsf{putdown}]] \wedge \\
&\ (\exists\ b2)[\mathsf{INIT\ On}(b2,b)] \supset (\exists\ \mathsf{pickup{:}PickUp}(b))[\mathsf{occurred}(\mathsf{pickup})] \wedge \\
&\ (\forall\ \mathsf{putdown{:}PutDown}(b2,b))[\mathsf{occurred}(\mathsf{putdown}) \supset \\
&\qquad\qquad (\exists\ \mathsf{pickup{:}PickUp}(b2))[\mathsf{putdown} \Longrightarrow \mathsf{pickup}]]\ ]
\end{aligned}
$$

7.2 Generating Plans for the Blocks World

Given this description of the blocks world, we now consider the problem of generating a plan for moving a set of blocks from the initial configuration to the goal configuration shown in Figure 3. We begin by constructing a plan for a single robot **R**.

Suppose we describe the initial configuration by a set of three initialization assertions:

INIT On(C,TABLE)
INIT On(B,A)
INIT On(A,TABLE)

The goal is described by three history formulas, **On(A,B)**, **On(B,C)**, and **On(C,TABLE)**, where **On(x,y)** is defined as in Section 6. These initial and goal configurations yield a preliminary plan network of the following form:

INIT On(C,TABLE)
INIT On(B,A) **R.putdown(B,C)**
INIT On(A,TABLE) **R.putdown(A,B)**

By applying Robot Constraint 1 we get a plan of form:

INIT On(C,TABLE)
INIT On(B,A) **R.pickup(B)** $\rightsquigarrow$ **R.putdown(B,C)**
INIT On(A,TABLE) **R.pickup(A)** $\rightsquigarrow$ **R.putdown(A,B)**

Next, we apply Robot Constraint 2. Since **R.pickup(B)** must precede **R.pickup(A)** or vice versa, at least one of these events cannot be the first action of robot **R**. Thus, one of them must be caused by a **Putdown** event. We could add a new **Putdown** event for this purpose or use an existing one. The latter alternative is also supported by BlocksWorld Constraint 1, which affirms that the robot cannot be holding both **B** and **A** in the same history. We thus should have either

R.putdown(A,B) $\rightsquigarrow$ **R.pickup(B)** or
R.putdown(B,C) $\rightsquigarrow$ **R.pickup(A)** .

If we check BlocksWorld Constraint 2, the first choice violates the clear precondition of **R.pickup(B)**. We thus choose the second alternative, yielding

INIT On(C,TABLE)

INIT On(B,A) **R.pickup(B)** $\rightsquigarrow$ **R.putdown(B,C)**

INIT On(A,TABLE) **R.pickup(A)** $\rightsquigarrow$ **R.putdown(A,B)**

which is a plan that satisfies all the constraints.

Now let's consider the option of using multiple robots to accomplish our task. Our initial plan network may utilize two robots, **R1** and **R2**, to satisfy the goals **On(B,C)** and **On(A,B)**.

INIT On(C,TABLE)	
INIT On(B,A)	**R1.putdown(B,C)**
INIT On(A,TABLE)	**R2.putdown(A,B)**

As before, we apply Robot Constraint 1, yielding

INIT On(C,TABLE)			
INIT On(B,A)	**R1.pickup(B)**	$\rightsquigarrow$	**R1.putdown(B,C)**
INIT On(A,TABLE)	**R2.pickup(A)**	$\rightsquigarrow$	**R2.putdown(A,B)**

Robot Constraint 2 is not violated this time; the events **R1.pickup(B)** and **R2.pickup(A)** could both be initial **Pickup** events for their respective robots. Nor are we violating BlocksWorld Constraint 1. We are, however, still violating BlocksWorld Constraint 2. To satisfy the latter, we must be sure that **A**, **B**, and **C** are clear before they are picked up or covered by another block. To satisfy the clearing of **A**, **B** must be picked off **A** before **A** is picked up – i.e., **R1.pickup(B)** $\Longrightarrow$ **R2.pickup(A)**. If this is satisfied, it will also automatically assure that **A** is put down on **B** *after* **B** is picked up (i.e., **R1.pickup(B)** $\Longrightarrow$ **R2.putdown(A,B)**) and thus **B** will definitely be clear before the **R1.pickup(B)** action is executed.

INIT On(C,TABLE)			
INIT On(B,A)	**R1.pickup(B)**	$\rightsquigarrow$	**R1.putdown(B,C)**
	$\Downarrow$		
INIT On(A,TABLE)	**R2.pickup(A)**	$\rightsquigarrow$	**R2.putdown(A,B)**

Next, we must assure that **B** is clear when **A** is put down on it. This clear precondition is violated when **B** is in robot **R1**'s hand. Thus, **B** must have already been deposited by robot **R1** *before* robot **R2** puts **A** down on **B**. This yields a final plan of the form:

INIT On(C,TABLE)

INIT On(B,A)	**R1.pickup(B)**	$\rightsquigarrow$	**R1.putdown(B,C)**
	$\Downarrow$		$\Downarrow$
INIT On(A,TABLE)	**R2.pickup(A)**	$\rightsquigarrow$	**R2.putdown(A,B)**

Note that block **A** may be picked up at the same time that block **B** is being placed on block **C**.

8 Conclusions

This paper has presented the GEM model and specification language and discussed how it may be used for the specification of multiagent domains, as well as serve as a framework for generating plans in those domains. We feel that GEM has much to offer as a specification tool and that it is significant as a representational meduim, quite apart from its application to planning. Not only was it specifically designed to model the complex temporal properties of concurrent environments, but it also deals explicitly with the representation of causality, simultaneity, and logical/spatial structure. Much of the power of GEM's representational capabilities is due to its use of temporal operators over sequences of *histories*, which are complete records of past domain activity and interrelationships.

One of the unique aspects of our model is its emphasis on specifying domain properties in terms of actions and action interrelationships (i.e. *behavioral* specification). Most standard representational schemes describe a domain in terms of a set of state predicates, while the actions of the domain are viewed as "state transformers." One of our goals is to explore the utility and expressiveness of behavioral specification techniques; indeed, past experiences with GEM have been quite positive in this respect [?,?]. However, despite our emphasis on behavioral specification, we also recognize that state-based specification is often better for describing some kinds of properties. Thus, we have also presented various options for integrating the use of state predicates within GEM.

This paper has also offered several initial ideas for the construction of a planner based on GEM domain descriptions. Given a GEM-based description of a multiagent domain and a set of initial conditions and goals, plan construction would entail building a network of partially ordered actions through a process of incremental constraint

satisfaction. Many standard planning techniques can be used, including those of least-commitment, backtracking, and protection. Other techniques will also be explored, including experimentation with constraint ordering and run-time constraint satisfaction (in particular, run-time synchronization). Plan explanation could be facilitated by associating each action relationship within a plan with the constraint that caused that relation to be there. Since the causal relationships among actions are already a natural part of the GEM domain representation, they too can easily be used for plan explanation purposes.

As regards the computational problems faced in constructing such a planner, we hope to isolate a powerful but implementable subset of the GEM specification language. We anticipate that a set of *constraint paradigms* will be identified and that our implementation will be geared towards making these particular types of constraints efficient to enforce.

Finally, a very long-term goal of this research is the utilization of computer graphics as a medium for user interaction with a planning system. Pictures are an invaluable way of explaining complex activity to humans – especially concurrent activity. One of the advantages of a behavioral approach to planning is that behavioral patterns are easily visualized. There are several ways in which a graphic user interface to a planning system could be used. Potential applications include visualizing and helping to guide the planning process, visually simulating and testing plan executions, and plan-execution monitoring. A graphic user interface could also be used for building the domain specification itself. Indeed, many of the GEM specifications constructed by us in the past began as graphic depictions of elements, groups, and events.

Acknowledgments

I would like to thank Michael Georgeff for many enlightening discussions on specification and planning, for reading versions of this paper, and for introducing me to artificial intelligence and the potential for application of work on concurrency theory to knowledge representation and planning. Others who helped by reading an early version of this paper include Barbara Grosz, Ed Pednault, David Wilkins, Peter Cheeseman, and Steven Rubin.

References

[1] Allen, J.F. "Maintaining Knowledge about Temporal Intervals," *Communications of the ACM*, 26 (11), pp. 832-843, (November 1983).

[2] Allen, J.F. "Towards a General Theory of Action and Time," *Artificial Intelligence*, 23, pp. 123-154 (1984).

[3] Allen, J.F. and J.A.Koomen, "Planning Using a Temporal World Model," *IJCAI-83*, Karlsruhe, West Germany, pp. 741-747 (August 1983).

[4] Appelt, D.E. "Planning Natural-Language Utterances to Satisfy Multiple Goals," Technical Note 259, Artificial Intelligence Center, SRI International, Menlo Park, California (1982).

[5] Bloom, T. "Synchronization Mechanisms for Modular Programming Languages," Technical Report TR-211, MIT Laboratory for Computer Science, Cambridge, Massachusetts (January 1979).

[6] Chapman, D. "Planning for Conjunctive Goals," Master's thesis, Technical Report MIT-AI-TR-802, MIT Laboratory for Artificial Intelligence, Cambridge, Massachusetts (1985).

[7] Cheeseman, P. "A Representation of Time for Automatic Planning," *Proceedings of the IEEE International Conference on Robotics*, Atlanta, Georgia (March 1984).

[8] Clarke, E.M, and E.A.Emerson, "Synthesis of Synchronization Skeletons from Branching Time Temporal Logic," *Proceedings of the Workshop on Logics of Programs - Yorktown Heights, New York*, Lecture Notes in Computer Science, Springer-Verlag, New York, New York, pp. 52-71 (1981).

[9] Cohen, P.R. and H.J.Levesque, "Speech Acts and Rationality," Working Paper, Artificial Intelligence Center, SRI International, Menlo Park, California (February 1985).

[10] Davis, R. and R.G.Smith, "Negotiation as a Metaphor for Distributed Problem Solving," *Artificial Intelligence* 20 (1), pp. 63-109 (1983).

[11] Fikes, R.E, P.E.Hart, and N.J.Nilsson, "Learning and Executing Generalized Robot Plans," *Artificial Intelligence*, 3 (4), pp. 251-288 (1972).

[12] Genesereth, M.R., M.L.Ginsberg, and J.S.Rosenschein, "Cooperation without Communication," Technical Report HPP-84-36, Stanford Heuristic Programming Project, Stanford University, Stanford, California (September 1984).

[13] Georgeff, M. "A Theory of Action for Multiagent Planning," *AAAI-84, Proceedings of the National Conference on Artificial Intellgenee*, pp.121-125 (August 1984).

[14] Georgeff, M., A.Lansky, and P.Bessiere, "A Procedural Logic," International Joint Conference on Artificial Intelligence (IJCAI-85), Los Angeles, California (August 1985).

[15] Green, C. "Application of Theorem Proving to Problem Solving," *Readings in Artificial Intelligence*, B.L.Webber and N.J.Nilsson, eds., pp. 202-222, Tioga Publishing Company, Palo Alto, California (1981).

[16] Greif, I. "Semantics of Communicating Parallel Processes," Technical Report TR-154, MIT Project MAC, Cambridge, Massachusetts (September 1975).

[17] Hayes, P.J. "The Second Naive Physics Manifesto," Cognitive Science Technical Report URCS-10, Department of Philosophy, University of Rochester, Rochester, New York (October 1983).

[18] Hewitt, C. and H.Baker Jr. "Laws for Communicating Parallel Processes," *IFIP 77*, B.Gilchrist,ed., pp. 987-992, North-Holland, Amsterdam, Holland (1977).

[19] Konolige, K. "A Deduction Model of Belief," Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, California (1984).

[20] Lamport, L. "Times, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM 21*, 7, pp. 558-565 (July 1978).

[21] Lamport, L. "A New Approach to Proving the Correctness of Multiprocess Programs," *ACM TOPLAS 1*, 1, pp. 84-97 (July 1979).

[22] Lansky, A.L. "Specification and Analysis of Concurrency," Ph.D. thesis, Technical Report STAN-CS-83-993, Department of Computer Science, Stanford University, Stanford, California (December 1983).

[23] Lansky, A.L. and S.S.Owicki, "GEM: A Tool for Concurrency Specification and Verification," *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pp.198-212 (August 1983).

[24] Malone, T.W., R.E.Fikes, and M.T.Howard, "Enterprise: A Market-Like Task Scheduler for Distributed Computing Environments," Working paper, Cognitive and Instructional Sciences Group, Xerox PARC, Palo Alto, California (1983).

[25] Manna, Z. and P.Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, 6 (1), pp.68-93 (January 1984).

[26] McCarthy, J. "Circumscription – A Form of Non-Monotonic Reasoning," *Artificial Intelligence*, 13 (1,2), pp.27-39 (1980).

[27] McCarthy, J. and P.Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," *Machine Intelligence*, Vol.4, B.Meltzer and D.Michie, eds., pp. 463-502, American Elsevier, New York, New York (1969).

[28] McDermott, D. "A Temporal Logic for Reasoning About Processes and Plans," *Cognitive Science* 6, pp.101-155 (1982).

[29] Moore, R.C. "A Formal Theory of Knowledge and Action," Technical Note 320, SRI International, Menlo Park, California (1984). Also to appear in *Formal Theories of the Commonsense World*, J.R.Hobbs and R.C.Moore, eds., Ablex Publishing Company (1984).

[30] Nilsson, N. Principles of Artificial Intelligence, Tioga Publishing Company, Palo Alto, California (1980).

[31] Owicki, S. and D.Gries, "An Axiomatic Proof Technique for Parallel Programs," *Acta Informatica* (6) (1976).

[32] Owicki, S. and L.Lamport, "Proving Liveness Properties of Concurrent Programs," *ACM TOPLAS 4*, 3, pp.455-492 (July 1982).

[33] Pednault, E.P.D. "A Theory of Classical Planning and Its Application to Plan Synthesis: A Study in Minimizing Search," Ph.D. thesis, Department of Electrical Engineering, Stanford University, Stanford, California (forthcoming).

[34] Rosenschein, J.S. and M.R.Genesereth, "Communication and Cooperation," Technical Report HPP-84-5, Stanford Heuristic Programming Project, Stanford University, Stanford, California (March 1984).

[35] Sacerdoti, E.D. A Structure for Plans and Behavior, Elsevier North-Holland, Inc., New York, New York (1977).

[36] Shoham, Y. "A Logic of Events," Working Paper, Department of Computer Science, Yale University, New Haven, Connecticut (1985).

[37] Shoham, Y. and T.Dean, "Temporal Notation and Causal Terminology," Working Paper, Department of Computer Science, Yale University, New Haven, Connecticut (1985).

[38] Simmons, R. personal communication.

[39] Smith, A. *Powers of Mind,* Ballantine Books, New York, New York, p. 362 (1975).

[40] Stefik, M. "Planning with Constraints (MOLGEN: Part 1)," *Artificial Intelligence,* 16 (2), pp. 111-139 (1981).

[41] Stuart, C. "An Implementation of a Multi-Agent Plan Synchronizer Using a Temporal Logic Theorem Prover," *IJCAI-85,* Los Angeles, California (August 1985).

[42] Tate, A. "Generating Project Networks," *IJCAI-77, 5th International Joint Conference on Artificial Intelligence,* Cambridge, Massachusetts, pp. 888-893 (August 1977).

[43] Tate, A. "Goal Structure – Capturing the Intent of Plans," *ECAI-84: Advances in Artificial Intelligence, Proceedings of the Sixth European Conference on Artificial Intelligence,* T.O'Shea, ed., Elsevier Science Publishers, North Holland (1984).

[44] Vere, S.A. "Planning in Time: Windows and Durations for Activities and Goals," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. PAMI-5, No.3, pp. 246-267 (May 1983).

[45] Wilkins, D. "Domain-independent Planning: Representation and Plan Generation," *Artificial Intelligence,* 22, pp. 269-301 (April 1984).

A GEM Specification Language Syntax

The following syntax describes a subset of the GEM specification language, with the omission of the actual syntax for constraints. (Such constraints simply follow the usual syntax for well-formed first-order temporal logic formulas composed of the predicates on histories described in Section 4.) In the syntax description below, square brackets $[x]$ are used to indicate the optional inclusion of x, the Kleene star applied to parenthesized expressions $(x)*$ indicates zero or more repetitions of x, and $(x)+$ indicates one or more repetitions of x. Use of the vertical bar $(x|y|z)$ indicates the choice of one of a set of alternatives, in this case x or y or z. Quotation marks are used to distinguish uses of the delimiters (,), [,], +, * as part of the GEM specification language from uses of these delimiters as BNF notation (BNF delimiters are unquoted).

```
<specification> ::=  (<element specification> | <group specification>)*

<element specification> ::= <element>
                        ::= <element type>
                        ::= <refined element>
                        ::= <refined element type>

<group specification> ::= <group>
                      ::= <group type>
                      ::= <refined group>
                      ::= <refined group type>

<element> ::= <element name> = ELEMENT
              [ EVENTS
                   (<event type>)+ ]
              [ CONSTRAINTS
                   (<constraint>)+ ]
                END <element name>

          ::= <element name> = <base element type name> ELEMENT

<element type> ::=
   <element type name>[ "(" <param description list> ")" ] = ELEMENT TYPE
 [ EVENTS
     (<event type>)+ ]
 [ CONSTRAINTS
     (<constraint>)+ ]
   END <element type name>

<event type> ::= <event type name>[ "(" <param description list> ")" ]

<param description list> ::= <param description>(, <param description>)*
```

```
<param description> ::= <param name>:<GEM type>

<GEM type> ::= <specification defined type> | <basic type>

<specification defined type> ::= <event type name>
                             ::= <element type name>
                             ::= <group type name>

<basic type> ::= GROUP | ELEMENT | EVENT | VALUE | <conventional type>

<conventional type> ::= INTEGER | BOOLEAN | REAL | STRING


<group> ::= <group name> = GROUP "(" <element group list> ")"
                         [ PORTS "(" <event class list> ")" ]
           [ CONSTRAINTS
               (<constraint>)+ ]
             END <group name>

        ::= <group name> =
            <base group type name> GROUP "(" <element group list> ")"

<element group list> ::= <element or group>(, <element or group>)*
<element or group> ::= <element name> | <group name>


<group type> ::=
  <group type name>["(" <param description list> ")"] =
                            GROUP TYPE "(" <group param list> ")"
 [ PORTS "(" <event class list> ")" ]
 [ CONSTRAINTS
     (<constraint>)+ ]
   END <group type name>

<group param list> ::= <group param>(, <group param>)*

<group param> ::= <param name>:<group param type>
              ::= {<param name>}: SET OF <group param type>

<group param type> ::= <single group param type>
                   ::= {<single group param type>
                          (, <single group param type>)* }

<single group param type> ::= <element type name> | <group type name>
```

```
<event class list> ::= <event class>(, <event class>)*

<event class> ::= <simple event class>
              ::= <element name>.<simple event class>
              ::= <element type name>.<simple event class>
              ::= <group name>.<element name>.<simple event class>
              ::= <group name>.<element type name>.<simple event class>
              ::= <group type name>.<element type name>.<simple event class>

<simple event class> ::= <event type name>
                         [ <occurrence number> ]
                         ["(" <parameter value list> ")" ]

<occurrence number> ::= <constant>

<param value list> ::= <param value>(, <param value>)*

<param value> ::= <string> | <boolean> | <integer> | <real> | <const> |
                  <group name> | <element name> | <event instance name> |
                  <event instance name>.<param name>


<refined element type> ::=
   <refined element type name>["("<param description list>")"] =
                                    <base element type name> ELEMENT TYPE
   <element differences>

<refined element> ::=
   <refined element name> = <base element type name> ELEMENT
   <element differences>

<refined group type> ::=
   <refined group type name>["("<param description list>")"] =
                                    <base group type name> GROUP TYPE
   <group differences>

<refined group> ::=
  <refined group name> = <base group type name> GROUP <element group list>
  <group differences>
```

```
<element differences> ::= (<constraint difference> |
                           <name difference> |
                           <element event difference> )*

<group differences> ::= ( <constraint difference> |
                          <name difference> |
                          <group structure difference> )*

<constraint difference> ::= ADD CONSTRAINT: <constraint>
                        ::= DELETE CONSTRAINT: <constraint>
                        ::= REPLACE CONSTRAINT: <constraint> WITH <constraint>

<name difference> ::= RENAME: <name> WITH <name>

<element event difference> ::= ADD EVENT: <event type>
                           ::= DELETE EVENT: <event type>
                           ::= REPLACE EVENT: <event type> WITH <event type>

<group structure difference>::= ADD TO GROUP: <group param>
                            ::= DELETE FROM GROUP: <group param>
                            ::= REPLACE INSIDE GROUP: <group param> WITH
                                                      <group param>
                            ::= ADD PORT: "(" <event class list> ")"
                            ::= DELETE PORT: "(" <event class list> ")"
                            ::= REPLACE PORT: "(" <event class list> ")"
                                        WITH "(" <event class list> ")"

<element name> ::= <string>
<element type name> ::= <string>
<refined element type name> ::= <string>
<refined element name> ::= <string>
<group name> ::= <string>
<group type name> ::= <string>
<refined group type name> ::= <string>
<refined group name> ::= <string>
<event type name> ::= <string>
<param name> ::= <string>
<constant> ::= <string>
<name> ::= <string>
<event instance name> ::= <string>
<base element type name> ::= <element type name>["(" <param value list> ")"]
<base group type name> ::= <group type name>["(" <param value list> ")"]
```