# LOCALIZED EVENT-BASED REASONING FOR MULTIAGENT DOMAINS

**Technical Note 423**

January 1988

**By:** Amy L. Lansky
Computer Scientist
Representation and Reasoning Program
Artificial Intelligence Center
and
Center for the Study of Language and Information
Stanford University

A version of this paper appeared in the *Computational Intelligence Journal, Special Issue on Planning* (November 1988). A highly abbreviated version also appeared in *The Proceedings of the Sixth National Conference on Artificial Intelligence* (AAAI-87).

The research described was made possible by the National Science Foundation, under Grant IST-8511167. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representative of the official policies, either expressed or implied, of the National Science Foundation or the United States government.

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

## Abstract

This paper presents the **GEM** concurrency model and **GEMPLAN**, a multiagent planner based on this model. Unlike standard state-based AI representations, GEM is unique in its explicit emphasis on events and domain structure. In particular, a world domain is modeled as a *set of regions composed of interrelated events.* Event-based temporal logic constraints are then associated with each region to delimit legal domain behavior. The GEMPLAN planner directly reflects this emphasis on domain structure and constraints. It can be viewed as a general-purpose constraint satisfaction facility which constructs a network of interrelated events (a "plan") that is subdivided into regions ("subplans"), satisfies all applicable regional constraints, and also achieves some stated goal. Because GEMPLAN is specifically geared towards parallel, multiagent domains, we believe that its natural application areas will include scheduling and other forms of organizational coordination.

GEMPLAN extends and generalizes previous planning architectures in several ways. First of all, it can handle a much broader range of constraint forms than most planners. Second, GEMPLAN's constraint satisfaction search strategy can be flexibly tuned. A third and critical aspect of our work has been an emphasis on *localized* reasoning — techniques that make explicit use of domain structure. For example, GEM localizes the applicability of domain constraints and imposes additional "structural constraints" on the basis of a domain's structural configuration. Together, constraint localization and structural constraints provide semantic information that can be used to alleviate several aspects of the frame problem for multiagent domains. The GEMPLAN planner reflects this use of locality by subdividing its constraint satisfaction search space into *regional planning search spaces.* Utilizing constraint and property localization, GEMPLAN can pinpoint and rectify interactions among these regional search spaces, thus reducing the burden of "interaction analysis" ubiquitous to most planning systems.

## Contents

## 1 Introduction

This paper presents a new representation and planning methodology for parallel domains. Unlike standard AI approaches, our underlying representational framework, *GEM* [19, 20, 21, 22, 23], and our planner, *GEMPLAN*, eschew the primacy of state-based description. Instead, a world domain is modeled as a *set of regions composed of interrelated events or actions*[1]. Each of these regions may be associated constraints on the event behaviors occurring within it. Thus, a domain as a whole is viewed as a myriad of interrelated events that are partitioned into regions and subject to regional constraints. One important focus of this paper is the use of a domain's regional structure (*locality*) to help define its behavioral semantics and to guide reasoning.

Our emphasis on domain *activity* rather than domain *state* is not as radical as it may seem. Many researchers have recognized that event-based descriptions can be more expressive, especially for describing the complex event interrelationships and requirements found in multiagent domains [1, 11, 18, 19]. Unlike other AI representations that utilize events, however, GEM takes events to be the *primary* objects of its underlying domain model.

Also unlike other event-oriented representations, GEM utilizes event-based, first-order-temporal-logic formulas to describe domain constraints. As we shall show, temporal-logic formulas can easily and naturally describe synchronization requirements and other behavioral properties common to parallel domains. GEM's constraint language can also express state-based preconditions, interval constraints, nonatomic event decomposition, and many other constraint forms.

Clearly, our somewhat novel approach to domain representation will necessitate a different approach to planning as well. In particular, the GEMPLAN planner is properly viewed as a *general-purpose constraint satisfaction facility* that can support a broad range of event-based constraints and constraint satisfaction strategies. The planner's task is to construct a network of interrelated events (a "plan") that is subdivided into regions ("subplans"), satisfies all applicable regional constraints, and also achieves some stated goal. Starting with some initial (possibly empty) plan, GEMPLAN repeatedly chooses some constraint, checks to see whether that constraint is satisfied, and modifies the plan (by adding new events and relations) so that it is. The planner continues onward in this fashion, repeatedly testing, extending and modifying its plan, until all applicable constraints (including the stated plan "goal") are met.

[1]While they are sometimes considered distinct, especially in the philosophical literature, in this paper we use the terms *event* and *action* interchangeably.

As stated earlier, a very important aspect of both GEM and GEMPLAN is their use of *locality*. By this we mean the use of domain structure to delimit the applicability of regional constraints, to provide semantic information about potential regional interactions, and to guide reasoning and planning. In GEM, domain regions may be composed so that they overlap, form disjoint sets, or form hierarchies – i.e., they may take on *any* structural configuration that the domain describer wishes. The notion of locality thus generalizes the well-recognized representational technique of hierarchy, and, as a consequence, gains many of the benefits associated with it.

For example, since a GEM domain's structure also implies limitations on its regional interactions, GEMPLAN's regional plans (which satisfy local regional constraints) may be constructed separately and later combined. Such techniques are absolutely critical for dealing with complex parallel domains. Localized reasoning techniques are, of course, beneficial for single-agent domains as well. Single-agent hierarchical planners, for example, have demonstrated the efficacy of divide-and-conquer techniques [31, 38]. Locality requirements, however, impose stronger semantic limitations on regional interactions than are typically found in hierarchical frameworks. This point is discussed in detail in Section 4.

Of course, the utility of locality depends greatly on how a domain is partitioned. This partitioning choice may be influenced by many factors. The domain's physical structure, its functional decomposition, the extent of causal effect within the domain, the kinds of processes that may occur, and the scope of applicability of domain constraints may all play a role. For instance, if a particular set of events is associated with its own collection of constraints, that set would probably be represented as a distinct region of activity. Since many structural factors will be important in any given domain, it is critical that domain decomposition not be restricted to any one particular style (e.g., hierarchies) or to any one particular criterion of decomposition. In GEM, the regional structure of a domain description may reflect many decompositions simultaneously. For example, one set of regions may reflect a physical view of the domain, while another set may offer a causal or process-oriented view. Both may be crucial, and both will cause constraints to be applied only within appropriate regions of activity.

We have found that the use of locality provides many benefits, both in domain representation and in the planning process itself. Probably the most obvious benefit is that locality provides a way of organizing domain descriptions in a coherent way. For example, GEM utilizes domain regions as the building blocks of its domain description language. A complete type facility is available whereby region types and instances may be defined. This has been especially helpful in describing large domains.

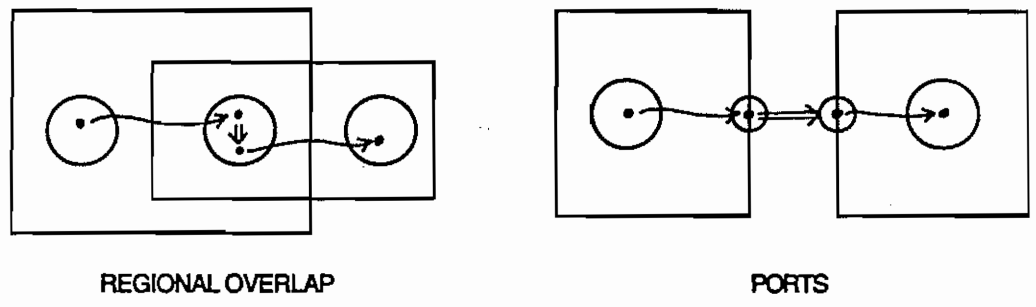


Figure 1: Regional Interactions

Locality can also be used to define domain semantics. For example, the applicability of GEM constraints is confined to particular regions rather than the domain as a whole. Because of this, GEMPLAN is able to limit the scope of constraint satisfaction while planning. Such limitations can greatly ease tractability problems inherent in constraint satisfaction. We call this "scoping" of constraint applicability *constraint localization.*

The very structure of a domain can also be used to limit potential interactions among events — a crucial feature when dealing with parallel domains. In particular, GEM uses a special kind of region, the *group*, for limiting event interactions that cross regional boundaries. These limitations are described formally in GEM by a set of *structural constraints.* Given these constraints, one way of representing *allowable* regional interactions is through *regional overlap*; the subregion corresponding to the intersection of a set of regions represents an arena of potential interaction among them. Another way of representing potential regional interaction is through the use of *input and output port events*; interactions among different regions are then limited to relationships among these ports. The use of ports is common in the design of distributed systems. In GEM, port events may be viewed as a special type of regional overlap; the port events of a region $R$ form a subregion "overlapping" any external region that has access to $R$ (see Figure 1).

Together, constraint localization and structural constraints can be used to infer the scope of effect of domain events and to determine potential interactions among events and properties. In Section 4, we provide structurally based frame axioms that express these effects. The use of domain structure to define frame axioms has been suggested in the AI literature [12, 13], but has not been explored extensively until now.

Locality can also play a major role in problem-solving. In Section 5, we present the GEMPLAN localized planning architecture that partitions both the plan representation and the planning search space according to domain structure. In essence, the GEMPLAN planner explicitly manifests adherence to GEM's locality frame axioms.

Because constraints are localized, there are fewer interactions among GEMPLAN's regional search spaces and, when they do exist (because of regional overlap or the use of ports), they are readily identified. While containment of interaction analysis is a goal of many existing planning systems [31, 35, 36, 38], most do not localize domain descriptions sufficiently. As a result, the task of determining and coping with interactions has always been extremely expensive.

The rest of this paper is organized as follows. Section 2 briefly reviews related work and compares our localized, constraint-based planning method with more traditional planners. Section 3 then presents our underlying domain model. In Section 4 we discuss the use of locality to help solve the frame problem. Finally, in Section 5, we describe GEMPLAN, our localized planner.

## 2 Related Work

Unlike GEM's event-based approach, states are the primary objects of most AI domain models. Typically, the world is viewed as a sequence of "states" or "snapshots." For example, traditional planning systems maintain a description of world state as a set of atomic state formulas. Change from one state to another is brought about through the occurrence of actions – i.e., domain actions or events are viewed as "state transformers." Thus, they are modeled derivatively as relations between world states. Many planners use STRIPS rules [8] to describe the state preconditions and effects of actions. In a STRIPS rule, each action is associated with a set of conditions that must be true of the world state prior to that action, as well as with a set of atomic state formulas that are "added" or "deleted" as a result of its occurrence.

There are well-known difficulties with adapting this kind of framework to parallel domains, foremost being the tendency to require that actions (especially interacting actions) ultimately occur in some sequential order — they cannot occur simultaneously. This is because the underlying semantics of STRIPS action descriptions assume that each action takes place in isolation; if action $a$ is defined to have an effect $Q$ and action $b$ has effect $R$, STRIPS has nothing to say about the effect of executing $a$ and $b$ concurrently. Indeed, if $Q$ and $R$ interact in any way, the result of executing $a$ simultaneously with $b$ would be ill-defined.

Clearly, the sequentiality assumption underlying STRIPS is unsuitable for parallel domains. For instance, one might wish to *require* that two events occur simultaneously (imagine a scenario in which two robots must cooperatively pick up a heavy object together). Similarly, one might wish to describe explicitly the different effects that would entail if two events are executed concurrently rather than sequentially.

STRIPS rules are also very awkward for describing synchronization properties. The control state of a domain must generally be encoded as a set of atomic state formulas; these formulas must then be manipulated by domain actions in fairly complex ways. For example, in order to describe a simple priority property, a state predicate might be used to encode a priority queue. This queue would then be manipulated by the set of relevant action types coordinated by the queue. As a result, synchronization requirements such as priority are not stated as a single explicit rule, but rather, must be stated implicitly in terms of the preconditions and effects of multiple actions. This method of domain description can be quite awkward and error-prone.[2]

Because of these problems, there has been much recent interest in devising formalisms and specification techniques that are better suited to describing parallel domains. For example, Georgeff [12] has recently proposed a way to modify the semantics of the situation calculus (upon which STRIPS was based) to allow for simultaneous action. However, it is not yet obvious how this work can be applied to planning systems. Pednault [29] has also suggested ways to apply traditional models to multiagent planning, by taking advantage of action noninterference. On a more practical note, Wilkins [39] has added limited forms of causal reasoning to SIPE to help infer the effects of events as well as reasoning techniques for certain kinds of resource constraints [38].

Many of these attempts to deal with domain parallelism are marked by a growing emphasis on the use of events – the approach taken in GEM. The work of Drummond [7], Kowalski and Sergot [16], and Hewitt's work on actor-based formalisms [14] all strongly emphasize event-oriented domain description. The interval-based representations of Allen, Koomen, and Pelavin [1, 2, 30], Vilain and Kautz [37], McDermott [27], Ladkin [17], and Shoham and Dean [6, 32] are also much more event-oriented.

GEM differs from most of the aforementioned work in having a *purely* event-based domain model[3] and in its use of temporal logic. Temporal logic is widely used in concurrency theory [28] and has a well-understood semantics. Its incorporation into GEM has thus resulted in a much more coherent and understandable model of concurrent action than any of the logics described above.[4] Finally, GEM differs from all the aforementioned representations in its use of *locality* for structuring domain descriptions and defining multiagent domain semantics.

[2]For a more complete discussion of this point, see my related paper [19].

[3]Events are the *primary* objects of the model. State conditions must then be *derived* from past event behavior.

[4]In fact, GEM has also been used for concurrent program specification and verification [21, 23].

Because of its basis in event-based temporal-logic and its formulation as a localized constraint satisfaction engine, the GEMPLAN planner is quite different from traditional planners. A comparison between GEMPLAN and other planners can be made from at least two standpoints: (1) the variety of domain constraint forms utilized, and (2) the way in which constraints are applied — i.e., the structure of the planning search itself.

If one looks closely at STRIPS-based planners, for example, one finds that world properties usually appear in only one form: they must be described within a schema of action preconditions and effects. As a result, only one kind of "constraint" is solved by such planners — the requirement that all action preconditions hold and that goal conditions are satisfied. This can be accomplished with a fixed set of methods that are aptly described by Chapman's truth criterion [5]. Hierarchical planners like NOAH [31] and SIPE [38] broaden the kinds of domain constraints available to include action decomposition. This can be viewed as a constraint of the form: "Every action of type $A$ must be expanded into a network of actions of the form $N$ (or one of set of such networks)." Wilkins has also recently added various kinds of causal constraints to SIPE [39].

Given that traditional planners handle only a few types of constraints, we believe that commonly held views of what "planning is" have become skewed. For example, in a recent paper on planning [15], Korf enumerates the following basic "operators of planning": subgoaling (achieving state-based goals), macro-operations (action decomposition) and abstraction (also manifested as action hierarchies). But these reflect only those planning methods and constraint forms used by current planners. Even the *order* in which these operators and constraints are applied is fixed by many planners. For example, hierarchical planners typically decompose actions first, expanding a plan to its lowest level of detail, and examine interactions among action pre- and postconditions (interaction analysis) last.

The GEMPLAN planner generalizes these approaches by (1) allowing many forms of constraints (ideally, any constraint expressible in first-order temporal logic) and (2) searching the constraint satisfaction search space in very flexible ways. Typical constraint forms expressible in the GEM description language include the following:

- *Event prerequisites*
  For example, one might require that each event instance of some given type be preceded by a particular pattern of behavior. Event preconditions described in terms of state predicates (as in STRIPS) can also be utilized (see Section 3.3.2).

- *Required patterns of behavior expressed as regular expressions*
  This kind of constraint requires that a specific set of events be totally ordered and that it conform to a particular regular expression.

- *Synchronization properties such as priority, mutual exclusion, and simultaneity requirements*
  For example, a typical priority constraint is "first-come-first-served." Mutual exclusion requirements preclude certain activities from occurring at the same time. Simultaneity constraints *require* event instances to co-occur.

- *Nonatomic event decomposition*
  This kind of constraint requires that nonatomic event instances be decomposed into particular patterns of behavior.

- *Interval-based constraints*
  These constraints place limitations on temporal relationships among nonatomic events.

To construct a plan, GEMPLAN must have, for each kind of domain constraint, a set of methods (*fixes*) that can satisfy it. GEMPLAN is currently supplied with constraint satisfaction fixes for a predefined set of constraint forms; in the future, we hope to derive fixes automatically for a subset of temporal logic.

The order in which GEMPLAN applies constraints and their fixes (the structure of its constraint satisfaction search space) is guided both by domain structure and a set of tailorable regional search tables. Each region is associated with a constraint-satisfaction search tree that builds a local plan for that region. Search within a region's tree is then directed by its corresponding search table. Depending on the amount of regional interaction, these regional search trees may be fairly tightly coupled, with search bouncing back and forth between them, or very loosely coupled. The resultant planning search structure is thus extremely flexible. It can be influenced by such factors as domain locality properties, user-supplied heuristics (which may be context-dependent), or default GEMPLAN strategies. It is certainly not confined to fixed phases or strategies.

## 3 GEM Model and Description Language

We now begin with a discussion of GEM's underlying domain model and description language. Much more rigorous and complete definitions of the GEM model, specification language, and underlying temporal logic can be found elsewhere [19, 20, 21, 22, 23]. These descriptions also include more complex examples of GEM in use.

Our primary goal in this section is to bring across the general flavor of the GEM model and specification language, and to briefly illustrate its power and utility. While it may have been desirable to utilize a more familiar notation and formalism (such as the interval logic of Allen), we have found the use of GEM to be critical. This is due to its strictly event-based model, its grounding in first-order temporal logic, its rigorous concurrency semantics, and finally, its incorporation of domain structure. Allen's interval logic, in contrast, does not make use of domain struture in any disciplined way and does not address the concurrency issues that fall within the scope of GEM's modal logic. While an interval logic might be extended to do so, this would be tantamount to using a formalism similar to GEM.

### 3.1 World Model

GEM's underlying world model is constructed in terms of event instances that are clustered into regions of activity. Event instances may be interrelated by three kinds of relations: the temporal order $\Longrightarrow$ (modeling required precedence between event instances), the causal relation $\leadsto$ (modeling a direct causal relationship between event instances), and a simultaneity relation $\rightleftharpoons$ (modeling necessary simultaneity of event instances). The temporal and simultaneity relations have their obvious interpretation. However, the causal relation calls for a bit more explanation.

GEM's causal relation $\leadsto$ is used to model the kinds of one-to-one relationships between event instances that are normally associated with the notion of "process." For example, if a person makes a particular request $r$, the response $s$ that serves that request would be considered causally connected to it ($r \leadsto s$). Although many other responses may follow $r$ in time, only $s$ is causally linked with it. Moreover, our use of causality is distinct from *eventuality*; just because a request $r$ occurs, it need not necessarily find eventual service. Once service $s$ does occur, however, the causal relationship between $r$ and $s$ would come into force. Eventuality requirements, if they are desired, must be explicitly stated.

In GEM, causality also implies temporal flow. Thus, if $r \leadsto s$, we must also have $r \Longrightarrow s$. To model simultaneous forms of "causal correspondence," one must use the simultaneity relation $\rightleftharpoons$. For example, a light switch domain might include a constraint that requires each instance of flipping the light switch to be simultaneous with some instance of the light's going on or off. The simultaneity relation $\rightleftharpoons$ is also commonly used to model required forms of cooperation (e.g., picking a table up at both ends simultaneously).[5]

Throughout this paper we shall use the terms "event" and "event instance" synonymously. Thus, each "event" should be considered an instance of an event type. Whenever we are referring to an event type, we shall explicitly use the term "event type." In addition, event type names are always capitalized; lowercase is used for event instances. As an illustration, we might have an event type **Put(Block,Surface)** and two instances of this type, an event **put(a,table)** and an event **put(b,a)**.

GEM utilizes two kinds of regions of activity: *elements* and *groups*.[6] Elements are the most basic type of region. Every event must belong to some unique element, and all events belonging to the same element must be temporally totally ordered – i.e., elements are regions of sequential activity. Elements (and their constituent events) may then be clustered into groups. Each group is a region of behaviorally encapsulated activity. That is, group boundaries impose limitations on causal flow and required forms of simultaneity within a domain.

## Two-Tiered Model

From a formal standpoint, GEM's world model may be viewed as two-tiered. Each domain description may have several "upper-tier" interpretations or models. These interpretations are called *world plans*; they represent the possible plans that satisfy the domain description. Each of these plans may then have several possible "lower-tier" interpretations. These correspond to the possible executions of the plan.

A *world plan W* is described formally as a structure consisting of a set of events, elements, and groups, plus their interrelationships (see Figure 2). Every event in a world plan models a unique event (instance) that occurs in the world domain, each relation or ordering models relationships between domain events, and each element or group models a logical region of activity:[7]

[5]GEM also allows for *serendipitous* simultaneity (in contrast to the *required* simultaneity modeled by $\rightleftharpoons$). This will be illustrated later.

[6]GEM stands for the *G*roup *E*lement Model.

[7]We assume here that all events are atomic. The model is expanded in Section 3.3.1 to include nonatomic events.

$W = < E, EL, G, \Longrightarrow, \leadsto, \rightleftharpoons, \epsilon >$

- $E =$ A set of event instances
- $EL =$ A set of elements
- $G =$ A set of groups
- $\Longrightarrow$: The temporal ordering
- $\leadsto$: The causal relation
- $\rightleftharpoons$: The simultaneity relation
- $\epsilon$ : A transitive subset relation between events and the regions in which they are contained, as well as between elements and groups and the surrounding regions in which they are contained.

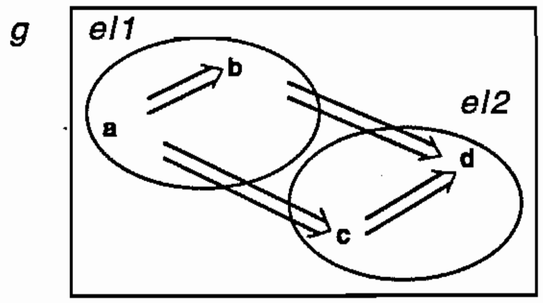
World plans are meant to convey *known* information about domain behavior. This information may be incomplete in the sense that some potential relationships may be left undetermined. For example, for the world plan in Figure 2 we have $a \Longrightarrow b$. Thus, $a$ must always occur before $b$ in every execution of the plan. However, since the world plan contains no temporal relationship between $b$ and $c$, the actual relationship between these two events is unknown — they might occur in either order or even simultaneously. This use of indeterminacy is critical for modeling *parallel* domains. In such domains it is often impossible to know the exact relationships between events. Indeed, humans manifest such indeterminacy themselves. People are often aware of the sequencing and causal relationships of individual processes, but are not able to perceive the actual interleavings and simultaneities that actually exist among many concurrently active processes.

Of course, to be sure that a world plan does satisfy a domain description, we must have some way of modelling its potential executions. This is the role of the lower tier of the GEM world model; the interpretations or executions of a world plan represent its possible "completions." For example, the world plan in Figure 2 can be executed in three possible ways:

| | | | | |
|---|---|---|---|---|
| Execution 1: | 1st $a$ | 2nd $b$ | 3rd $c$ | 4th $d$ |
| Execution 2: | 1st $a$ | 2nd $c$ | 3rd $b$ | 4th $d$ |
| Execution 3: | 1st $a$ | 2nd $b$,$c$ | 3rd $d$ | |

These three executions are the lower-level world models for this world plan. Note that, in the third execution, $b$ and $c$ occur simultaneously. Thus, world plans *need not* be resolved into total orders (an assumption made by many representations and planners); GEM allows for true simultaneity, either required or serendipitous.[8]

[8]In this case, the simultaneity of $b$ and $c$ is serendipitous. If, however, the world plan included the



| | | | |
|---|---|---|---|
| $a \Longrightarrow b$ | $a \Longrightarrow c$ | $b \Longrightarrow d$ | $c \Longrightarrow d$ |
| $a\epsilon el1$ | $b\epsilon el1$ | $c\epsilon el2$ | $d\epsilon el2$ |
| $el1\epsilon g$ | $el2\epsilon g$ | | |

Figure 2: A World Plan

Formally, each possible execution of a world plan is described as a *journal sequence*. Each *journal* $\alpha$ of a world plan $W$ is a possible "state" in the execution of $W$; a journal sequence thus represents a "state sequence." In contrast to most state-based formalisms, however, the state captured by a journal describes not only a single moment, but everything that occurred up to that moment – i.e., it is a record (or journal) of past behavior. For instance, the world plan in Figure 2 has six possible journals, corresponding to the six possible prefixes of the world plan:

$\alpha_0$ : The empty journal.

$\alpha_i$ : The journal containing just event $a$, as well as $a\epsilon el1$ and $el1\epsilon g$.

$\alpha_j$ : The journal containing $a \Longrightarrow b$, as well as $a\epsilon el1$, $b\epsilon el1$, and $el1\epsilon g$.

$\alpha_k$ : The journal containing $a \Longrightarrow c$, as well as $a\epsilon el1$, $c\epsilon el2$, $el1\epsilon g$, and $el2\epsilon g$.

$\alpha_m$ : The journal containing $a \Longrightarrow b$, $a \Longrightarrow c$, $a\epsilon el1$, $b\epsilon el1$, $c\epsilon el2$, $el1\epsilon g$, and $el2\epsilon g$.

$\alpha_n$ : The journal equivalent to the entire world plan in Figure 2.

The GEM journal sequences corresponding to the world plan's three possible executions are the following:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Execution 1: | $\alpha_0$ | $\alpha_i$ | $\alpha_j$ | $\alpha_m$ | $\alpha_n$ | $\equiv$ | 1st $a$ | 2nd $b$ | 3rd $c$ | 4th $d$ |
| Execution 2: | $\alpha_0$ | $\alpha_i$ | $\alpha_k$ | $\alpha_m$ | $\alpha_n$ | $\equiv$ | 1st $a$ | 2nd $c$ | 3rd $b$ | 4th $d$ |
| Execution 3: | $\alpha_0$ | $\alpha_i$ | $\alpha_m$ | $\alpha_n$ | | $\equiv$ | 1st $a$ | 2nd $b$,$c$ | 3rd $d$ | |

relation $b \rightleftharpoons c$, $b$ and $c$ would *have* to occur simultaneously and Execution 3 would be the only possible interpretation of the world plan.

It is interesting to note that the possible journal sequences of a world plan may also be represented as a branching tree. For this example, we would have

$$\begin{tikzcd}
& & \alpha_j \arrow[r] & \alpha_m \arrow[r] & \alpha_n \\
\alpha_0 \arrow[r] & \alpha_i \arrow[ur] \arrow[r] \arrow[dr] & \alpha_k \arrow[r] & \alpha_m \arrow[r] & \alpha_n \\
& & \alpha_m \arrow[r] & \alpha_n &
\end{tikzcd}$$

where $\alpha_a \longrightarrow \alpha_b$ implies that $\alpha_a$ is a subset of $\alpha_b$. This corresponds to the branching tree of states used by McDermott; a chronicle corresponds to a journal sequence [27]. However, by representing this tree as a *world plan*, we can convey information about possible world executions and event interrelationships in a much more compact form.

## 3.2 Description Language

Each GEM domain description $\delta$ is composed of a set of constraints that limit allowable executions or behaviors within a domain. A given world plan $W$ is considered to satisfy (be a valid interpretation of) a description $\delta$ if every one of its journal sequences satisfies every constraint in the description. Given a domain description and a set of "goal constraints," the task of the GEMPLAN planner, then, is to construct a world plan that satisfies them.

Just as elements and groups model the structural aspects of a domain, they also serve as the structural components of the GEM description language. Each description $\delta$ is composed of a set of element and group declarations. An element declaration must be associated with a set of event types, which describe the kinds of events that may occur within it. A group declaration is describes the elements and subgroups which comprise it. Element and group declarations may also be associated with first-order linear-temporal-logic constraints. These constraints are localized, applying only to those events that occur within the element or group with which they are associated. Every domain description also includes a set of default constraints imposed by the element/group structure of the domain. These *structural constraints* (and their effect on the frame problem) are discussed in Section 4.

Figure 3 illustrates a sample description and a possible world plan for a cooking-class domain. The cooking class is described as a group consisting of a fire alarm element and two kitchen subgroups, both of which share a teacher element. Each kitchen also contains a set of student elements and an oven element. Typical constraints that might be used in this domain include rules regarding individual student behavior, limitations on the use of the oven in each kitchen, requirements for student cooperation on certain tasks, descriptions of appropriate reactions to teacher instructions, and constraints on the behavior of the fire alarm. Figure 3 also illustrates the use of GEM's region-type definition and instantiation mechanism. As mentioned earlier, lowercase names are used to denote event, group, and element instances; types are capitalized. In the full GEM description language, region-type inheritance and refinement may also be employed [19]. Such type hierarchies commonly appear in programming languages as well as in frame-based AI representations.

Perhaps one of the most interesting aspects of the GEM description language is its use of temporal-logic constraints. In a recent paper [19], we described a semantics for these constraints and fully demonstrated their expressive power. Here we shall simply illustrate the constraint language with a few examples.

Linear temporal operators are modal operators that apply formulas to sequences. GEM's temporal operators are applicable to journal sequences that go forward in time (the operators $\Box$ (*henceforth*), $\Diamond$ (*eventually*), $\bigcirc$ (*next*), and $P\ U\ Q$ (*P until Q*)) as well as backwards in time ($\triangle$ (*before*), $\overleftarrow{\Box}$ (*until now*), $P \overset{Q}{\leftarrow}$ (*Q back to P*)). Given a journal sequence of the form $S = \alpha_0, \alpha_1, ....$, we use the notation $S[i]$ to denote the $i^{th}$ tail sequence of $S$ — i.e., $\alpha_i, \alpha_{i+1}, ....$ The semantics of $\Box$, $\Diamond$, and $\triangle$, for example, may then be described as follows:

$$\begin{array}{lll} \textit{Henceforth } P: & S[i] \models \Box P & \equiv \ (\forall j \geq i)\, S[j] \models P \\ \textit{Eventually } P: & S[i] \models \Diamond P & \equiv \ (\exists j \geq i)\, S[j] \models P \\ \textit{Before } P: & S[i] \models \triangle P & \equiv \ \textit{if } i \geq 1 \textit{ then } S[i-1] \models P \textit{ else false} \end{array}$$

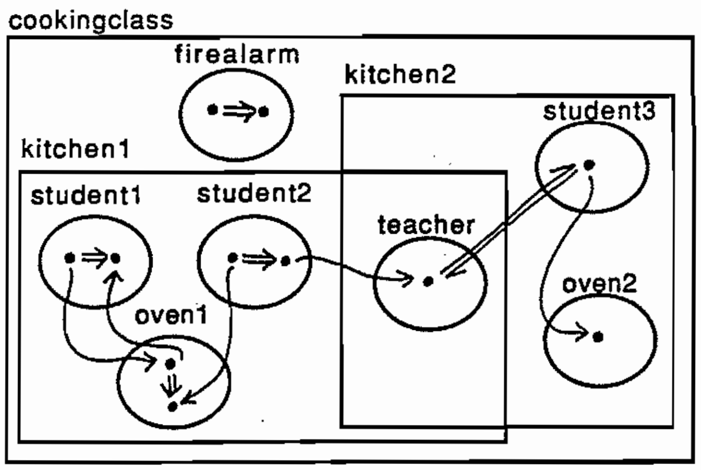
As an example, consider a property that must be true of all journals. Normally, a constraint of the form $\Box P$ would be used. For instance, a simple prerequisite constraint might be the following:

$$\Box\ (\forall\ \mathsf{bake{:}Bake}) \\ [\ \mathsf{occurred(bake)} \supset (\exists\ \mathsf{ovenreq{:}OvenRequest})\ \mathsf{ovenreq} \leadsto \mathsf{bake}\ ]$$

In other words, if a Bake event ever occurs, a correponding request to use the oven must have preceded it and caused it to occur. For any event e, occurred(e) is defined to be true of a journal if and only if e is part of the journal.

Another property that might be imposed on all journals is a requirement of cooperative cake preparation for certain kinds of cakes. For example, we might have the following simultaneity constraint:

$$\Box\ (\forall\ \mathsf{prepare1(upside\text{-}down\text{-}cake){:}Prepare}) \\ [\mathsf{occurred(prepare1(upside\text{-}down\text{-}cake))} \supset (\exists\ \mathsf{prepare2(upside\text{-}down\text{-}cake){:}Prepare}) \\ \mathsf{prepare1(upside\text{-}down\text{-}cake)} \neq \mathsf{prepare2(upside\text{-}down\text{-}cake)} \land \\ \mathsf{prepare1(upside\text{-}down\text{-}cake)} \rightleftharpoons \mathsf{prepare2(upside\text{-}down\text{-}cake)}\ ]$$



```
                    polygon = group     circle = element     dot = event

Student = ELEMENT TYPE          Oven = ELEMENT TYPE         Kitchen = GROUP TYPE
EVENT TYPES                     EVENT TYPES                           (teacher,
   Prepare(cake)                   Bake(cake)                          {s}:SET OF Student
   OvenRequest                  CONSTRAINTS                            o:Oven)
CONSTRAINTS                        :                        CONSTRAINTS
   :                            END Oven                       :
END Student                                                 END Kitchen


student1 = Student ELEMENT
student2 = Student ELEMENT
student3 = Student ELEMENT
oven1 = Oven ELEMENT
oven2 = Oven ELEMENT
kitchen1 = Kitchen GROUP (teacher,{student1,student2},oven1)
kitchen1 = Kitchen GROUP (teacher,{student3},oven2)

teacher = ELEMENT          firealarm = ELEMENT       cookingclass = GROUP(kitchen1,kitchen2,
EVENT TYPES                EVENT TYPES                                    firealarm)
   Instruct                   Ring                   CONSTRAINTS
CONSTRAINTS                CONSTRAINTS                  :
   :                          :                      END cookingclass
END teacher                END firealarm
```

Figure 3: A Cooking Class Domain

When past behavior dictates the course of future behavior, we typically use a constraint of the form: $P \supset \Box Q$. This may be read "if $P$ holds for the events in some journal, then, for every journal that follows in the sequence, $Q$ must hold." This constraint form is commonly used for priority requirements as well as for many other naturally occurring domain properties. For example, a simple first-come-first-served requirement for use of an oven might be written as follows:

$$\begin{array}{l}(\forall\ \text{ovenreq1,ovenreq2:OvenRequest, k:Kitchen, bake2:Bake})\\ \text{ovenreq1}\epsilon\text{k} \wedge \text{ovenreq2}\epsilon\text{k} \wedge \text{ovenreq1} \Longrightarrow \text{ovenreq2} \supset\\ \Box\ [\ \text{ovenreq2}\leadsto\text{bake2} \supset (\exists\ \text{bake1:Bake})[\text{ovenreq1}\leadsto\text{bake1} \wedge \text{bake1}\Longrightarrow\text{bake2}]\ ]\end{array}$$

In other words, if two oven requests in the same kitchen occur in some order, they must be serviced in the order in which the requests were made. Service has occurred once an OvenRequest event has caused a corresponding Bake event. Note that this constraint does not require that a Bake event actually occur, only that Bake events occur in the correct sequence. If we wished also to guarantee that all oven requests will be served, the following *eventuality* constraint could be used:

$$\begin{array}{l}(\forall\ \text{ovenreq:OvenRequest})\\ [\ \text{occurred(ovenreq)} \supset \Diamond\ (\exists\ \text{bake:Bake})\ \text{ovenreq}\leadsto\text{bake}\ ]\end{array}$$

Note that these constraints do not make use of state information, at least not in the form of state predicates. Indeed, encoding the first-come-first-served or eventuality constraint would be quite cumbersome (if not impossible) in STRIPS-based frameworks. For instance, we would have to encode the order in which oven requests occur by using some atomic state formula (e.g. queue([ovenreq1,ovenreq2,....])). This queue would be modified as a result of OvenRequest and Bake actions. The occurrence of Bake actions would also be restricted by the queue's contents. In addition, we would need to encode explicitly the connection between specific baking actions and oven requests. As we can see, things can become quite complex, even for simple requirements, when STRIPS action descriptions are used.

However, sometimes it *is* useful to utilize state descriptions as event preconditions, and backwards temporal operators provide the means to do so. For example, one might utilize a constraint of the form

$$\text{justoccurred(e)} \supset \triangle\ \text{precondition(e)}\ .$$

This may be read "if e has just occurred, then precondition(e) must hold in the preceding journal." The atomic formula justoccurred(e) is defined to be true exactly when event e enters a journal sequence: $\mathsf{justoccurred(e)} \equiv \mathsf{occurred(e)} \wedge \Delta \neg\, \mathsf{occurred(e)}$. Other state predicates (such as $\mathsf{precondition(e)}$) can be defined similarly in terms of event-based formulas. This is discussed at length in Section 3.3.2.

We have also investigated the use of *localized* temporal operators in the GEM constraint formalism — operators that apply to journal sequences composed only of events within a particular region (rather than within the domain as a whole). Localized temporal expressions are necessary for writing truly localized constraints. For example, if we wish to state that something must occur "next," it is important that "next" be qualified with respect to location — what happens next within a particular region is different from what happens next when the domain is viewed as a whole. These localized operators are described in more detail elsewhere [19].

Finally, regular-expression *pattern constraints* may also be employed. These require certain sets of events to be totally ordered and conforming to a particular pattern. For example, by associating a constraint of the form $(\mathsf{Prepare(cake)} \Longrightarrow \mathsf{OvenRequest})^{*\Longrightarrow}$ with each student, we would force students to alternate between cake preparation and oven request actions. A formal description of pattern constraints in terms of temporal logic formulas is given by us elsewhere (see discussion of *threads* in my doctoral thesis [21]).

## 3.3 Representational Issues

Our work on the GEM description language has focused not only on locality and event-based temporal constraints, but also on the task of incorporating many other commonly-used represenational methods into the GEM framework. Among these are the use of nonatomic events, constraints among event intervals, and the definition of state predicates in terms of past event behavior. We shall review this work here briefly.

### 3.3.1 Nonatomic Events and Interval Relationships

Nonatomic events may be integrated into GEM by associating each such event with a set of two or more events — i.e., the internal events that comprise it. In particular, each nonatomic event $a$ must be associated with at least two atomic events (denoted $a'$ and $a''$) representing the initiation and termination of $a$, respectively. Our use of initiation and termination events corresponds to the mapping commonly used between interval- and point-based representations; an interval is viewed as the period of time between two endpoints. In GEM, the interval within a journal sequence in which a nonatomic event occurs is precisely the journal subsequence in which the event's initiation event has occurred but not its corresponding termination event. For example, for a nonatomic event $a$, this would correspond to the journal subsequence in which the following formula is true: $occurred(a') \land \neg occurred(a'')$.

Because GEM is an event-oriented description framework, it is most natural to define a nonatomic event in terms of its subevents, temporally enclosed within its initiation and termination events. However, one might also wish to define a nonatomic event type as corresponding to those journal subsequences (intervals) in which some state description holds true. This latter method more closely resembles the method for defining intervals in Allen's interval calculus [1]. Its use in GEM is discussed elsewhere [19].

Formally, nonatomic events may be included within the GEM model by adding a set of nonatomic events to the underlying structure $W$ for world plans and inserting a composition relation $\kappa$ between a nonatomic event and its composite events. Thus, for each nonatomic event $a$, $\kappa(a', a)$ and $\kappa(a'', a)$ must hold. Various other constraints must also be added that constrain the occurrence of a nonatomic event and its composite events [19]. For example, $a \Longrightarrow b$ is considered to hold between nonatomic events $a$ and $b$ if and only if $a'' \Longrightarrow b'$ holds. Similarly, $a \rightleftharpoons b$ is true if and only if $a' \rightleftharpoons b'$ and $a'' \rightleftharpoons b''$ are true. Indeed, it is obvious that, using initiation and termination events, we can define all of the interval relationships in Allen's interval calculus:[9]

| | | |
|---|---|---|
| a after b | $\equiv$ | $b'' \Longrightarrow a'$ |
| a before b | $\equiv$ | $a'' \Longrightarrow b'$ |
| a equal b | $\equiv$ | $a' \rightleftharpoons b' \land a'' \rightleftharpoons b''$ |
| a meets b | $\equiv$ | $a'' \rightleftharpoons b'$ |
| a overlaps b | $\equiv$ | $a' \Longrightarrow b' \land a'' \Longrightarrow b'' \land b' \Longrightarrow a''$ |
| a during b | $\equiv$ | $b' \Longrightarrow a' \land a'' \Longrightarrow b''$ |
| a starts b | $\equiv$ | $a' \rightleftharpoons b' \land a'' \Longrightarrow b''$ |
| a finishes b | $\equiv$ | $b' \Longrightarrow a' \land b'' \rightleftharpoons a''$ |

[9] An even broader set of interval relationships exists when temporal relationships between nonatomic events are left undetermined. For example, we might know that $a' \Longrightarrow b'$, but the relationship between $a''$ and $b''$ may be unknown. The resulting broadened interval calculus contains thirty possible interval relationhips and corresponds to a subset of the logically possible disjunctive combinations of interval relationships in Allen's calculus.

### 3.3.2 State Predicates

Our experience with GEM has shown that many domain properties can easily be described without the use of state predicates and, in many cases, described more naturally. However, state predicates are often useful for encoding or abbreviating aspects of past behavior and for describing the initial state of a domain. Our approach to state emphasizes the duality between state-based and event-based representations; just as state-based descriptions represent events as relations between states, we represent state predicates, dually, as formulas pertaining to events. The truth or falsity of an atomic state formula thus depends on the definition of its predicate.

In many cases, we can provide a formula that represents a complete description of the state predicate. For example, we might define serviced(ovenrequest) as follows:

$$\mathsf{serviced(ovenrequest)} \equiv (\exists\ \mathsf{bake})\ \mathsf{ovenrequest} \rightsquigarrow \mathsf{bake}.$$

If, in some journal, an event ovenrequest has occurred and has caused a corresponding bake event, then ovenrequest has been serviced. If, on the other hand, this formula is false, the request has not been serviced.

We shall call such formulas *complete predicate definitions.* Note that, in this case, there is no ambiguity about the effects of events on the predicate serviced. No matter what event occurs in the domain, its effect on serviced will be determined completely by its effect on the valuation of the defining formula for serviced. In our current implementation of GEMPLAN, all predicates are completely defined.

It is possible, of course, that one might want to use weaker, *incomplete* predicate definitions — e.g., assertions of the form formula $\supset$ P. Similarly, we may wish to utilize constraints on relationships among predicates – for example, $\mathsf{P} \supset \mathsf{Q}$ or $\mathsf{P} \supset \mathsf{Q} \vee \mathsf{R}$. It is clear that the use of incompletely defined predicates will immediately necessitate frame rules. For example, let us suppose that formula $\supset$ P and that formula is true of a particular journal. If formula becomes false in the next journal, should we assume that P becomes false as well? Similarly, the combination of incomplete definitions with rules such as $\mathsf{P} \supset \mathsf{Q} \vee \mathsf{R}$ can lead to effects on P, Q and R that are unknown or ambiguous.[10]

Of course, the frame problems introduced by incomplete predicate definitions are no different from those faced by any state-based formalism, and similar methods can be used to solve them.[11] For example, one appealing approach is to minimize the

[10]If P, Q, and R were completely defined, $\mathsf{P} \supset \mathsf{Q} \vee \mathsf{R}$ would simply expand into an additional constraint on event behavior.

[11]These problems also underline the attractiveness of purely event-based constraints and complete predicate definitions!

effect of events on state formulas. Thus, an event $e$ would have an effect on P only if it were explicitly known (or, perhaps, deducible) to have such an effect. Circumscription is one way of achieving this kind of minimization [26]. Another possibility we have explored is to use the locality properties of a domain explicitly to constrain the effects of events. We shall discuss this approach in the next section.

GEM predicate definitions can also be constructed to mimic the use of add/delete lists in STRIPS frameworks. First, appropriate event types must be classified as either "adders" or "deleters" of a particular state predicate. These classifications might even be made conditional on the situation in which an event occurs. A predicate P could then be defined by an expression of the following form:[12]

$$(\exists\ \mathsf{a{:}Adder(P)})\ [\ \mathsf{occurred(a)} \land (\forall\ \mathsf{d{:}Deleter(P)})\ [\mathsf{d} \Longrightarrow \mathsf{a}]\ ]$$

(P has been made true and cannot have subsequently been made false). Note that this definition is essentially equivalent to Chapman's modal truth criterion [5]. (GEMPLAN actually contains a full implementation of condition attainment and protection based on the modal truth criterion.)

Other definitions could also be employed for P and they might be complete or incomplete, depending on the desired effect. In any case, this approach captures the mechanism used by most planners for evaluating or calculating whether or not a state condition holds at some point in the execution of a plan. It also enables modular definition of states — "adders" and "deleters" may be associated with a description incrementally. We utilize this adder/deleter scheme for forming complete definitions of the predicates clear and on in the blocks world domain described in the appendix.

Unlike formalisms based on STRIPS-like action descriptions, however, the use of predicate definitions (complete or incomplete) suffers from none of the problems created by the possibility of simultaneous action. Because effects upon world state are not prescribed in the context of individual action descriptions, there is no difficulty in dealing with simultaneous activity. If the formula defining the truth-value of an atomic formula P is true of a journal, P itself is true of the journal. Indeed, a formula may be defined *in terms* of simultaneity properties — for example, we might use a definition of the form $\mathsf{P} \equiv (\exists\ \mathsf{e1{:}E1,\ e2{:}E2})\ \mathsf{e1} \rightleftharpoons \mathsf{e2}$ .

[12]While this expression is clearly second-order, it should be regarded as a definition schema that is instantiated, yielding a first-order formula for each P.

## 4 Locality and the Frame Problem

As we have seen, the use of incompletely defined state predicates leads to many of the same frame problems faced by other AI representations. Frame problems also underlie one of the more challenging aspects of planning: the resolution of subplan interactions. In this section we show how the locality properties of a domain can help define regional independence, thereby alleviating many such difficulties.

Recent literature has discussed several aspects of the frame problem [9, 12, 13, 33]. Perhaps the most significant and best understood is what Georgeff [12] has called the *combinatorial problem*, i.e., how to state which properties remain unaffected by the occurrence of domain events. Another aspect of the frame problem is the difficulty of dealing with *qualification* — namely, how to describe the preconditions and effects of an event, while allowing for subsequent qualification. Additional frame problems are the *ramification problem* (determining the extent of influence of an event occurrence) and the tendency towards *overcommitment* — that is, an assumption that all domain events are known.

In a recent paper, Georgeff shows how *independence axioms* can be used to solve the combinatorial and related frame problems. Assuming that one is able to state which events are independent of which properties, he offers a general-purpose law of persistence that guarantees that properties will remain unaffected by the occurrence of independent events. One of the unresolved issues of Georgeff's theory is exactly how to specify independence axioms. However, he suggests domain-structuring techniques as a possible solution. Hayes [13] has also suggested that domain structure be used as a way of delineating frame axioms. We shall now show how GEM's use of locality can serve this very purpose.

As discussed earlier, GEM's use of locality provide two kinds of semantic information about a domain:

1. *Constraint Localization.* The constraints and other definitions or properties associated with an element or group pertain only to activity within that element or group.

2. *Structural Constraints.* The very structure of a domain implies, by definition, additional constraints on domain behavior. In particular, elements impose sequentiality constraints on their composite events, and groups impose restrictions on the causal and simultaneity relationships that cross group boundaries.

For the remainder of this section we shall define these concepts more formally and show how they can be used to address the frame problem. Clearly, this will greatly affect planning as well. Depending on the potential interdependencies among events and properties, the application of constraints during planning can be focused and minimized.

Let us begin with a more formal definition of constraint localization. First, we need a few definitions. We say that a property $\psi$ belongs to a region r if it is explicitly associated with r in the domain description: property-belongs($\psi$,r). We say that an element or group x structurally belongs to a group g if it is explicitly declared as a *direct* component of g within g's declaration: structure-belongs(x,g). For example, in the cooking class, we have structure-belongs(student1,kitchen1), but $\neg$ structure-belongs(student1,cookingclass).

Next, we introduce the use of ports. As stated in the introduction, group boundaries can be "permeated" in two ways: through the use of group overlap, and by means of ports — "holes" in group boundaries. These ports come in two flavors: input and output. Let us assume that world plan structures have been expanded to include the following relations:

inputport(e,g) : Event e is an input port for group g.

outputport(e,g) : Event e is an output port for group g.

port(e,g) : Event e is either an input or output port (or both) for group g.

In the cooking class, for example, we might wish to allow baking events to trigger the fire alarm. This could be accomplished by making all Bake events output ports of their respective kitchens. We use the notation openinput(g) and openoutput(g) to indicate that g is *completely* open (its boundary is "transparent") to input or output, respectively. Thus, for example, openoutput(g) implies that *all* events belonging to g are output ports of g:

$$(\forall\ e,g)\ \text{event-belongs}(e,g) \land \text{openinput}(g) \supset \text{inputport}(e,g)$$
$$(\forall\ e,g)\ \text{event-belongs}(e,g) \land \text{openoutput}(g) \supset \text{outputport}(e,g)$$

If we wished to allow *every* event in kitchen1 to trigger the fire alarm, we would state this as openoutput(kitchen1).

Given the above, we can now define the relation event-belongs. We consider an event to belong directly to a region r if r is an element and the event occurs at r or, if r is a group, the event occurs at an element that structurally belongs to r or is a port of some subgroup of r. Intuitively, event-belongs(e,r) is true if e's containment within r is not blocked by a group boundary.

event-belongs(e,r) $\equiv$ (element(r) $\wedge$ e$\epsilon$r) $\vee$
[ group(r) $\wedge$
[ [ ($\exists$ elem:ELEMENT) e$\epsilon$elem $\wedge$ structure-belongs(elem,r) ] $\vee$
[ ($\exists$ g':GROUP) structure-belongs(g',r) $\wedge$ port(e,g') ] ] ]

For example, while an event of oven1 belongs to kitchen1 (e.g., event-belongs(bake,kitchen1)), that same event does not "belong" to the cooking class ($\neg$ event-belongs(bake,cookingclass)). If, however, Bake events were ports of the Kitchen groups, we would have event-belongs(bake,cookingclass). The reader should be careful not to confuse the relations $\epsilon$ and event-belongs. The relation $\epsilon$ is transitive and encompasses all structural enclosures within a domain. In contrast, event-belongs represents the notion of *direct* membership in an element or group and is restricted by group boundaries.

We are now able to define the concept of constraint or property localization. In particular, we define property-influence as follows:

*Constraint Localization:*

property-influence(e,$\psi$) $\equiv$
($\exists$ g) property-belongs($\psi$,g) $\wedge$ event-belongs(e,g)

In other words, an event e can potentially *influence* a property $\psi$ if and only if e and $\psi$ belong directly to the same region. For example, as currently structured, firealarm events are the only events that can influence properties belonging to the cookingclass group. If Bake events were made ports, as described earlier, they too could potentially influence cookingclass constraints.

The reader has probably already noticed the similarity between the notion of property-influence and that of scoping in programming languages. An event can influence a property if it occurs within the property's region of definition or if it has been "made visible" or "exported" by a subregion through a port. One can also see how property-influence can be associated with Georgeff's notion of *independence*. Namely, if an event has no influence on a property, it can be assumed to be independent of it. We provide a formal frame rule that links property-influence to independence a bit later on.

Of course, events may also influence properties by interacting with other events. For example, if e1 $\leadsto$ e2 and property-influence(e2,$\psi$), one might think of e1 as influencing $\psi$, at least indirectly. We now define *structural constraints*, which constrain this sort of event interaction.

The most important kind of structural constraint deals with the limitations imposed by group boundaries. In particular, *group boundaries impose limitations on the scope* *of causal effect and required simultaneity.* Dynamic restructuring of groups is also allowed in an expanded version of the GEM model [21]. However, we do not use it in this paper or in the current version of GEMPLAN.

In previous descriptions of GEM [19], we used group boundaries to restrict only the inward flow of causality – i.e., events within a group could causally affect events outside the group, whereas the reverse was prohibited.[13] In this paper, we generalize the limitations imposed by a group to form a *two-way* barrier to both causal flow and required forms of simultaneity.

The group structural constraint may be stated as follows:

*Group Structural Constraint:*

$\mathsf{e1} \rightsquigarrow \mathsf{e2} \vee \mathsf{e1} \rightleftharpoons \mathsf{e2} \supset \mathsf{access(e1,e2)}$

Thus, an event e1 may cause or be necessarily simultaneous with an event e2 only if e1 has *access* to e2. The notion of event access may be described informally as follows: An event $e1$ has access to another event $e2$ if and only if

1. $e1$ has direct access to $e2$ (they both belong directly to the same group – i.e., they are siblings) or
2. $e1$ has indirect access to $e2$ because of the way groups are nested, overlapped, or associated with input and output ports.

Below we provide a precise definition of this concept.[14,15] Several illustrations of *access* are presented in Figure 4.

$\mathsf{access(e1,e2)} \equiv \mathsf{direct\text{-}event\text{-}access(e1,e2)} \vee \mathsf{indirect\text{-}event\text{-}access(e1,e2)}$

$\mathsf{direct\text{-}event\text{-}access(e1,e2)} \equiv (\exists\ \mathsf{elem1, elem2{:}ELEMENT})$
$\quad \mathsf{e1} \in \mathsf{elem1} \wedge \mathsf{e2} \in \mathsf{elem2} \wedge \mathsf{sibling(elem1,elem2)}$

[13]Thus, the groups utilized in our previous reports on GEM could be characterized as having *openoutput*.

[14]Intuitively, event-group-access(e,g) implies that event e can access the outside of g's boundary. Similarly, group-event-access(e,g) implies that, from outside g, event e can be accessed. Finally, group-group-access(g1,g2) is true if the region outside of g2's boundary can be accessed from outside g1's boundary.

[15]Note that, if all ports were bidirectional, access would no longer be a directional concept. In this case, its definition would reduce to the following:

$\mathsf{access(e1,e2)} \equiv (\exists\ \mathsf{g{:}GROUP})\ \mathsf{event\text{-}belongs(e1,g)} \wedge \mathsf{event\text{-}belongs(e2,g)}.$

indirect-event-access(e1,e2) $\equiv$ ($\exists$ g1,g2,...gn:GROUP)
event-group-access(e1,g1) $\wedge$ group-group-access(g1,g2) $\wedge$ ...
group-group-access(gn-1,gn) $\wedge$ group-event-access(gn,e2)

sibling(x,y) $\equiv$ ($\exists$ g:GROUP) structure-belongs(x,g) $\wedge$ structure-belongs(y,g)

event-group-access(e,g) $\equiv$ ($\exists$ elem:ELEMENT) e$\epsilon$elem $\wedge$
[ sibling(elem,g) $\vee$
[ structure-belongs(elem,g) $\wedge$ (openoutput(g) $\vee$ outputport(e,g)) ] ]

group-event-access(g,e) $\equiv$ ($\exists$ elem:ELEMENT) e$\epsilon$elem $\wedge$
[ sibling(elem,g) $\vee$
[ structure-belongs(elem,g) $\wedge$ (openinput(g) $\vee$ inputport(e,g)) ] ]

group-group-access(g1,g2) $\equiv$
sibling(g1,g2) $\vee$
[ structure-belongs(g1,g2) $\wedge$ openoutput(g2) ] $\vee$
[ structure-belongs(g2,g1) $\wedge$ openinput(g1) ]

For example, our description of the cooking class domain is constructed so as to bar access between students that belong to different kitchens. However, because the teacher belongs to *every* kitchen (it is a region of overlap among kitchens), a student event may access a teacher event, which may then access a student event in some other kitchen.[16] As another example, consider the case in which all kitchens are declared to have *openoutput*. This would enable all student, teacher, and oven events to access events at the fire alarm; for example, there would be an access path of the form event-group-access(studentevent,kitchen) $\wedge$ group-event-access(kitchen,firealarmevent). Notice, too, that the predicate access captures purely structural information about a domain. Thus, we have defined the behavioral notion of allowed effect strictly in terms of domain structure.

The second GEM structural constraint is much easier to describe. It states that all events belonging to the same element must be totally ordered:

*Element Structural Constraint:*

($\forall$ elem,e1,e2) e1$\epsilon$elem $\wedge$ e2$\epsilon$elem $\supset$ e1$\Longrightarrow$e2 $\vee$ e2$\Longrightarrow$e1

Elements are thus natural vehicles for representing resources that, by their very nature, are limited to sequential forms of behavior.

[16]Note that access is *not* transitively closed – access(x,y) $\wedge$ access(y,z) does not necessarily imply access(x,z). For example, while a student event has access to the teacher and the teacher has access to a second student, the two students may not have access to each other.

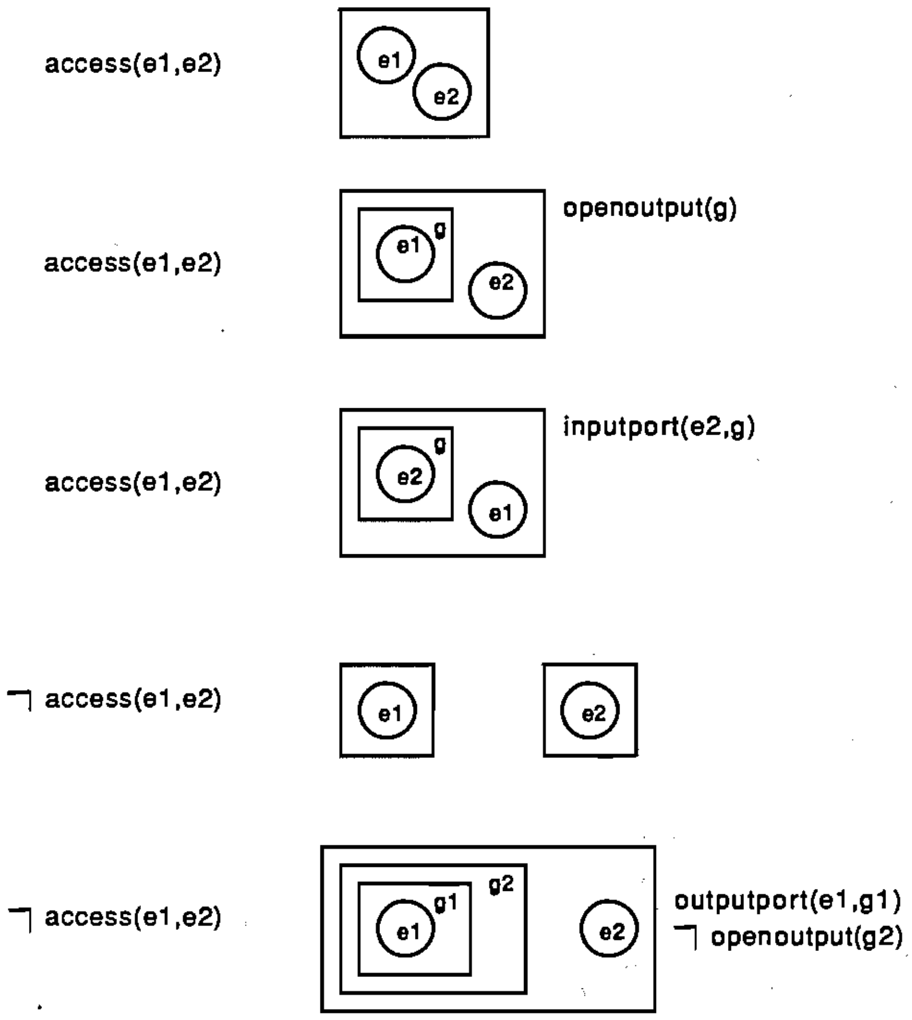


Figure 4: Event Access

At this point, our approach to the frame problem should be fairly clear; by utilizing constraint localization and the notion of event access, event/property independence can be determined. We begin with Georgeff's rule of persistence [12]:

$$holds(\phi, s) \wedge holds(\psi, s) \wedge (\forall e)[occurs(e, s, w) \supset independent(\phi, e, \psi)] \supset holds(\psi, succ(s, w))$$

To quote Georgeff, "this rule states that, if we are in a state $s$ in which both $\phi$ and $\psi$ hold, and if all events that occur in state $s$ of [execution sequence] $w$ are independent of $\psi$ under conditions $\phi$, then $\psi$ will also hold in the next (resulting) state." In GEM notation, this would be written as follows:

*Rule of Persistence:*

$$\triangle(\phi \wedge \psi) \wedge (\forall e)[justoccurred(e) \supset independent(\phi, e, \psi)] \supset \psi.$$

Obviously, the difficulty now lies in specifying the notion of independence. We use the following rule, which enables the derivation of independence from property-influence, and hence, from constraint localization.

*Independence Rule:*

$\neg$ property-influence(e,$\psi$) $\supset$ ($\forall$ $\phi$) independent($\phi$,e,$\psi$)

That is, if $e$ cannot influence $\psi$ (because of the way the domain is structured), $\psi$ is independent of the occurrence of $e$.

Using the structural constraints imposed by group structure, we can strengthen this notion of independence even further. Specifically, if there can be no possible chain of effect between $e$ and any event that pertains to $\psi$, then event $e$ cannot influence $\psi$, either directly or indirectly. This is clearly a stronger form of independence; it implies independence from both the immediate occurrence of an event as well as its effects. It also suggests the notion of *degree of independence.* For instance, we might wish to say that $e$ is independent of $\psi$ with respect to any chain of effect with length less than or equal to $n$. We now define this more formally.

We begin by defining the notion of event-influence — i.e., whether or not events may influence each other. To be completely thorough, we recognize that interactions among events may occur not only through direct event access but also as a result of mutual interference with the same property.

event-influence(e1,e2) $\equiv$
  access(e1,e2) $\vee$
  ($\exists$ $\psi$) property-influence(e1,$\psi$) $\wedge$ property-influence(e2,$\psi$)

Then we utilize the following definitions:

$\text{influence}(e,0,\psi) \equiv \text{property-influence}(e,\psi)$

For $n \geq 1$,

$\text{influence}(e,n,\psi) \equiv$

$(\exists\ e1...ei,\ 1 \leq i \leq n)$

$\text{event-influence}(e,e1) \land ... \land \text{event-influence}(ei\text{-}1,ei) \land \text{property-influence}(ei,\psi)$

This definition can then be used to derive degree of independence:

*Extended Independence Rule:*

$\neg\ \text{influence}(e,n,\psi) \supset (\forall\ \phi)\ \text{degree-of-independence}(\phi,e,\psi,n)$

Note that $\text{influence}(e,n,\psi) \supset \text{influence}(e,m,\psi)$ for all $m \geq n$. Thus, if n is the length of the ***shortest*** path between e and some event that influences $\psi$, then e will be independent of $\psi$ with degree n-1 or less. It should also be noted that **influence** is still completely determinable from the structure of a domain.

As we have now demonstrated, locality helps to solve the combinatorial problem by delineating event/property independence. The notion of **influence** also helps to address the *ramification problem*; the possible ramifications of an event e and their extent are clearly defined by the degree to which e influences other events and properties. Constraint localization, coupled with GEM's implicit structural constraints, can also help to solve problems of *overcommitment*. For example, if a set of properties and constraints pertains only to activity within a region $r$ and $r$ has no ports, then no events that occur outside $r$ can affect them. Thus, there is no need to overcommit to any assumption about the occurrence (or lack thereof) of events external to $r$.

The element structural constraint also implies some kinds of independence. Because they inherently limit potential forms of parallelism, elements guarantee that certain properties will be free from interference. In the cooking class, for example, the oven's internal sequentiality implies that it can have only one thing baking in it at a time. Thus, we are assured that different uses of the same oven will never interfere directly with one another.

Finally, because the ports of a group $g$ serve as "zones of influence" from the world outside $g$, they can be used as vehicles for addressing the *qualification problem*. Suppose an event $e$ belongs to $g$ and $g$ has no ports. Certainly, the preconditions and effects of $e$ will be limited to the properties and constraints belonging to $g$. One reason to add ports to $g$ would be to serve as placeholders for the introduction of qualifications or abnormalities regarding these properties. A natural strategy is to assume that the given set of preconditions and effects for $e$ is complete and that no port events occur unless otherwise specified or deducible. In essence, this amounts to minimizing the occurrence of port events. If these events do indeed take place, they evoke qualifications or abnormal situations. These abnormalities may then once again be circumscribed. For example, to describe the "potato in the tailpipe" problem, we might assume that all event types that could affect a car internally are known and belong to a group *car*. We could then use *car* ports to represent potential external influences upon the car. These ports would be categorized into various classes of abnormal events (*exhaust-obstruction, vandalism,* etc.) that might occur. Thus, while the use of port events does not *solve* the qualification problem, it provides a well-defined framework in which circumscription or some other form of minimization can be utilized.

Of course, the usefulness of group/element structure in addressing any of these frame problems is predicated on the assumption that a domain *can* be described in terms of loci of sequential activity clustered into relatively independent groups. Ultimately, the benefits of domain locality will depend on the skill of the domain expert in subdividing a domain description and on the nature of the domain itself. If a domain expert is able to structure knowledge so that regional interactions are limited, domain planning will much cheaper. Tradeoffs must obviously be made between allowing possibly relevant, but costly, interactions and the ultimate tractability of domain reasoning.

As far as the description-writing process goes, we have tried to demonstrate how GEM aids a domain expert by providing a well-defined, structured description-writing framework. Optimally, GEM users should adhere to a methodology that utilizes this framework to maximum effect. As far as the inherent partitionability of domains, we, admittedly, have demonstrated this only for the domains described in this and previous papers. However, we believe that such partitionability will be possible for many real-world domains and certainly for almost all domains currently modeled by AI systems.

For example, most physical resources in the world are subject to sequentiality constraints; such physical regions may thus suitably be represented as elements.[17] Localized descriptions will certainly be attainable for factory automation domains or other domains involving the coordination of established organizational entities. Such domains are usually hierarchically as well as physically structured. Thus, they are suited to description in terms of resources clustered into physical regions, with well-defined ports of interaction (the "visible" events at the top level of a given hierarchy).

[17]In the worst case, one would simply associate each event with its own element.

Moreover, the fact that humans are capable of dealing with the complexities of the world is evidence that they utilize some form of reasoning facility based on regional independence (and thus that the world *can* be so partitioned). As people enter certain contexts, the characteristics of those contexts come into greater focus and awareness, while those of other contexts fade into the background. In GEM, these foci of attention correspond to the localization of activity and constraints. In the GEMPLAN planner, the concept of shifting focus corresponds to the flow of the reasoning process itself. When reasoning about a particular region, GEMPLAN manipulates only that region's local plan, satisfying only local regional constraints. By knowing which regions and constraints may or may not interfere with one another, the planner can determine which regional constraints to check, thus focusing the flow of planning search.

## 5 Localized Planning

In this section we describe the **GEMPLAN** multiagent planning system. Written in Prolog on a Sun 3, it has already been used to generate multiagent solutions to blocks-world and Tower of Hanoi problems. The appendix contains an annotated GEMPLAN planning session that illustrates the coordination of multiple robot arms in the blocks-world domain.

Formally, GEMPLAN's task may be characterized as the construction of a world plan, all of whose executions satisfy a given set of domain constraints and achieve some stated set of goals. Input to the planning system includes a structural description of the domain, the constraints associated with each element and group, a set of initial conditions (which might be expressed as a set of initial events that have taken place or as a set of atomic state formulas assumed to be initially true), and the set of goal constraints to be achieved. Given the initial world plan characterized by user input data (possibly empty), the planner repeatedly chooses a constraint (either a domain or goal constraint), checks to see whether the constraint is satisfied and, if it is not, either backtracks to an earlier decision point in the planning process or continues on, modifying the world plan so that the constraint is satisfied.

This constraint satisfaction process may be seen as the search of a set of trees, one for each region. (See the search tree for **student1** in Figure 5 on page 37. This figure will be described in more detail later on.) At each tree node is stored a representation of the currently constructed world plan for that region. When a node is reached during the planning search, a constraint is checked. To satisfy it, the search space branches for each of the possible ways of repairing or "fixing" the world plan. These *fixes* may involve the addition of new events, elements, groups, or event interrelationhips.

It is important to note that, unlike many planners, the order in which fixes are applied by GEMPLAN has nothing to do with the ultimate ordering of events in a plan. In essence, a fix massages an ever-growing partial order; the placement of events within this partial order will depend on the nature of the constraint and fix. STRIPS, in contrast, links the order in which events are *added* to a plan with the order in which those events must ultimately occur. This can result in anomalous phenomena, such as the *Sussman Anomaly*, that do not arise in GEMPLAN.

Of course, the order in which constraints and fixes are applied *will* have a definite effect on the speed with which a correct plan can be found, and, depending on the extent of backtracking that occurs, on whether any solution is found at all. Indeed, GEMPLAN is not *guaranteed* to halt (unless some built-in domain heuristic guarantees it to halt). For this reason, GEMPLAN tries to provide for as much flexibility as possible in the ultimate structure of planning search. The order in which constraint checks and fixes are tried, as well as deciding when and where to backtrack or halt, are all guided by user-modifiable *regional search tables.*

As discussed in Section 2, this approach to planning is quite different and perhaps subsumes other planning methods. While most planners can handle only one or two forms of domain constraints (attainment of state conditions, nonatomic-event expansion), GEMPLAN can handle any form of constraint, as long as it is furnished a plan-fixing method for that form. Thus, planning "operators" or methods that have been given distinctive status by traditional planners become unified under the general heading of constraints and constraint satisfaction methods. Indeed, goals to be attained by a plan are also viewed as constraints to be solved. Second, GEMPLAN's constraint satisfaction mechanism is much more flexible than those found in most planners. Because constraints and fixes can be applied in *any* order (determined by the regional search tables), the planning process need not be rigidly broken up into distinct phases.

GEMPLAN is also unique in its use of locality to partition the representation of plans and the planning search space. Each domain region is associated with its own search tree that deals exclusively with constraints that belong to that region. The frame rules described in Section 4 can then be used to guide the planning search. For example, whenever plan modifications are made during planning, only those constraints that could possibly be affected need be rechecked.

This represents a significant advance over previous planners, which do not take advantage of any localizing discipline. Planning complexity is directly related to the amount of regional interaction — which is as it should be. For instance, while planning for tightly coupled regions may involve considerable interaction among their regional search spaces, planning for domains in which there is little regional interaction will be loosely coupled and might even be performed in parallel. Indeed, while our current planning system runs as a single process, with control flow bouncing among regional search trees, there is no reason why GEMPLAN's planning process could not be truly distributed.

## 5.1 Constraint Satisfaction

We turn now to the problem of constraint satisfaction – i.e., how to modify or "fix" a plan so that a particular constraint will be satisfied. In principle, a GEM constraint can be *any* first-order temporal-logic formula. However, because of the intractability of solving arbitrary first-order temporal-logic constraints, we decided that a good initial approach to the constraint-satisfaction problem would be to use *predefined* fixes for common constraint forms. This approach is similar to Chapman's idea of *cognitive cliches* [4] – i.e., utilizing a set of specialized theories that are common to many domains, rather than trying to solve for the most general theory.

The current GEMPLAN system can satisfy the constraint forms used in the blocks-world and Tower of Hanoi domains: event prerequisites, constraints based on regular-expression patterns of events, nonatomic-event expansion, and the attainment and maintenance of state-based conditions. (Thus, GEMPLAN already possesses the functionality of a traditional state-based planner, with the added capability of satisfying some kinds of event-based constraints.)

Our experiences with generating these constraint fixes have been surprisingly straightforward. Perhaps this is due to the event-based nature of GEM constraints; they translate more or less directly into manipulations of the events within a plan. For example, consider the following simple prerequisite constraint for the cooking class domain:

$$(\forall\ \mathsf{bake(cake)}:\mathsf{Bake})(\exists\ \mathsf{prepare(cake)}:\mathsf{Prepare})$$
$$\mathsf{prepare(cake)} \leadsto \mathsf{bake(cake)}$$

In other words, each baking event must be enabled or caused by a student event that prepares a cake. It is clear that there will be two possible fixes for this constraint:

1. A suitable Prepare event is available. This event may then be linked causally to the "lone" Bake event.

2. A new Prepare event is generated to cause the Bake event.

In general, there will always be two possible ways of fixing a constraint of the form $(\exists e : E)P(e)$: use an existing event $e$ that can be made to satisfy $P(e)$ or create a new such event..

Our derivation of fixes for state-based preconditions (defined in terms of "adders" and "deleters" – see Section 3.3.2) also flowed easily from our event-based definition of what it means for a state condition to hold. Indeed, the fixes we have used correspond directly to an implementation of Chapman's *modal truth criterion* [5].

Our fixes for regular-expression pattern constraints were also straightforward to derive, although they are somewhat weaker. Given a pattern over a set of event types, GEMPLAN will first gather together all events of those types. It will then check to see if *any* possible total ordering of those events satisfies the given pattern. The various possible fixes for the constraint correspond to the possible total orders that are in conformance. Thus, these fixes do not add *new* events to achieve a pattern; they only constrain existing event orderings. We felt that this approach was adequate and it is certainly more tractable.

Of course, the use of locality also alleviates tractability problems. For example, finding all total orders for a given partial order is generally very expensive. However, because GEM constraints are applicable only over limited sets of events, such operations tend to become more manageable.

In addition to satisfying constraints, GEMPLAN includes a facility for *accumulating* constraints on the values of unbound event-parameter variables. For example, we might wish to create an event bake(X), where the value of X is not yet determined. As constraints are added on X, its possible values become increasingly more limited. This facility is quite similar in function to the constraint accumulation facility in Wilkins' SIPE system [38] and is implemented utilizing Mackworth's constraint satisfaction techniques [24]. It allows least-commitment planning techniques to be utilized; the events in a plan can be *constrained* rather than modified. GEMPLAN's use of parameter constraint accumulation is illustrated in the sample session found in the appendix.

In the future, we intend to expand the kinds of constraints GEMPLAN can handle. Critical will be the handling of various forms of priority, mutual exclusion, simultaneity, and interval constraints; most of these have fairly obvious or known fixes. Because of our success thus far in deriving constraint fixes, we also have hopes of eventually generating them automatically, at least for some well-defined subset of first-order temporal logic. A starting point will be to incorporate the algorithms conceived by Manna and Wolper [25] and implemented by Stuart [34] for solving propositional temporal-logic constraints. Such automatic techniques guarantee fix *coverage* — i.e. the completeness of a set of fixes for a given constraint is assured. In our current system, completeness is not guaranteed — for example, our fixes for pattern constraints are incomplete. The fixes based on Chapman's modal truth criterion, however, are complete.

GEMPLAN assumes that a constraint has been satisfied once a fix has been successfully applied. One way of adding the notion of *constraint relaxation* into the current system would be the inclusion of additional, albeit less desirable, constraint fixes that approximate a constraint solution. Such fixes could be applied if exact constraint solutions fail.

GEMPLAN has already demonstrated a fair amount of sophistication. The blocks-world application, for instance, can generate maximally parallel solutions in a localized fashion. The Tower of Hanoi application generates optimal solutions to single or multiagent version of the three-disk problem, *without* hints about solving subproblems —

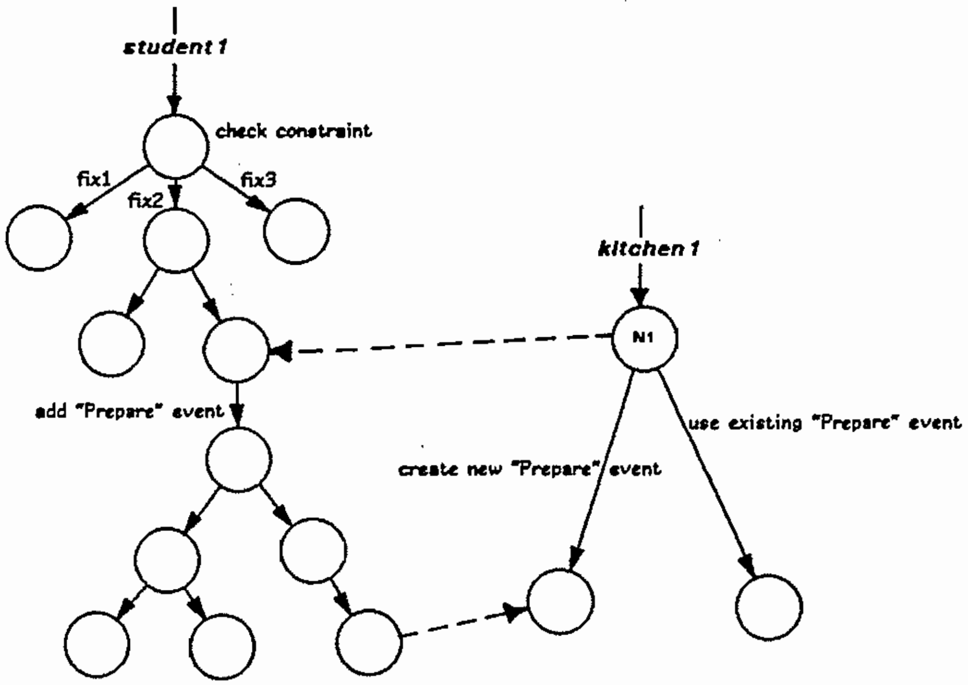


Figure 5: Localized Search Trees for the Cooking Class Domain

only the basic constraints on disk placement are supplied. To our knowledge, thes capabilities are not matched by other planning systems. With the additional capabilities we have outlined, GEMPLAN will certainly be able to handle much more complicated forms of synchronization than other existing planners. Future target applications include scheduling problems, such as those found in factory automation domains. We are hopeful that the use of locality will help alleviate tractability problems normally associated with such domains.

### 5.2 Localized Plan Representation and Search

For the remainder of this paper, we shall devote our attention to the GEMPLAN search architecture. The most important feature of this architecture is its partitioning of the planning search space and plan representation in a way that reflects the group/element structure of a domain. For each element and group there exists a local search tree and plan representation. Each local search tree is focused on the creation of a local plan that satisfies all local constraints. The overall planning system may thus be viewed as a set of mini-planning systems, one for each region. These regional search spaces are tied to one another in accordance with the structure of a domain – global regions have access to their subregions' plans and search spaces. It is these "parent-child" connections that form the glue tying the entire planning system together. For example, kitchen1 constraints will be applicable to (can be influenced by) events at all students, the oven, and the teacher within it. Thus, the **kitchen1** search space will have access to the subplans belonging to all these elements. The GEMPLAN session in the appendix also provides a demonstration of our localized search technique.

Let us begin by first considering GEMPLAN's localized plan representation. Each node in a GEMPLAN search tree for region **r** is associated with a plan descriptor **r-plan-i** that describes a plan within **r** that has undergone some set of fixes to satisfy **r**'s constraints. The plans denoted by plan descriptors are defined by using data *inheritance*; each plan inherits a set of previously defined plan information, supplemented by a set of new events, elements, groups, and relationships explicitly associated with the new descriptor. For example, consider the search tree for region **student1** in Figure 5. The plan associated with each node of this tree would be composed of the plan at its parent node plus events and relations that were added to that parent node plan in order to satisfy some local **student1** constraint. The entire plan at each tree node may thus be derived by following the plan inheritance chain, accumulating plan modifications along the way.

This inheritance-based representation is not only compact, but can also be used for consolidating local plan information to form more global plans. Each GEMPLAN group plan is the union of its subelements' plans (plus port events belonging to its subgroups) along with relations among these that are added by virtue of group constraints. For example, a plan descriptor for group **kitchen1** would have the form

**[kitchen1-plan-a,student1-plan-b,student2-plan-c,teacher-plan-d,oven1-plan-e].**

**student1-plan-b** would denote a plan consisting of events occurring at **student1** that was generated to satisfy **student1**'s constraints, **student2-plan-c** would contain events belonging to **student2** generated to satisfy **student2**'s constraints, and so on. **kitchen1-plan-a** would contain relations added to those found in the rest of **kitchen1**'s subcomponents' plans in order to satisfy **kitchen1**'s constraints.

GEMPLAN stores explicit links between each event and relation within a plan and the search tree node (hence, which constraint and fix) that was responsible for the addition of that event or relation. Obviously, this information can be quite useful for explaining how and why a plan was formed. It can also be used to drive dependency-directed search. For example, if the violation of a particular constraint is attributable to a specific relation or event, it might be useful to backtrack immediately to the tree node where that event or relation was added. Using the link between the offending event or relation and the node where it was created, search can backtrack directly to that node. Although we have not yet utilized this kind of backtracking in our current applications, we hope to investigate its use more thoroughly in the future.

This brings us to the final aspect of GEMPLAN's localized architecture — the localization of the planning search process itself. Just as the overall plan representation is partitioned regionally, so is the overall planning search. Each region's local search tree deals with the satisfaction of local constraints.

Within each regional search tree, the order in which constraints and fixes are applied is determined by the region's *search table.* Regional tables can be set up by a user to define quite flexible kinds of search. They provide four types of information: (1) the order in which to apply constraints, (2) the order in which to try constraint fixes, (3) when and where to backtrack, and (4) when to halt. Whenever backtracking occurs, the node left behind may be retained for possible later exploration. The search may thus utilize a mixture of depth- and breadth-oriented exploration, depending on the strategy determined by the table. Shifts between regional search spaces can be enacted by virtue of table information or the particular nature of a fix. For example, if a fix needs to add an event to another region, search will be pursued within that other region. This phenomenon is illustrated in Figure 5 and discussed below.

If a user does not supply domain-specific search information, a default depth-first search strategy is used. The constraints and fixes within each region are chosen in the order in which they are supplied within the domain description. Once constraint checking in a specific region is complete, other regions that may be affected are rechecked. Chronological backtracking occurs when a plan cannot be fixed. Planning halts when either no new options can be explored or all constraints have been checked successfully.

As an illustration, consider the search trees depicted in Figure 5 – one for the kitchen1 group of the cooking class, the other for its student subelement, student1. Let us assume that we have reached the node labeled N1 for kitchen1, and that a set of Bake events has already been inserted into the plan. At this point, we imagine that the following kitchen1 constraint is checked:

$$(\forall\ \mathsf{bake(cake):Bake})(\exists\ \mathsf{prepare(cake):Prepare})\\ \mathsf{prepare(cake)} \rightsquigarrow \mathsf{bake(cake)}\ .$$

As stated earlier, this requires that each baking event be caused by a student event that prepares a cake. If this is already the case (the constraint is satisfied), the planner will move on to the next kitchen1 constraint determined by kitchen1's search table. If this is not the case, however, the two fixes suggested earlier may be tried. First, there may be an existing Prepare event that could be used. This fix corresponds to the right branch emanating from N1. It would result in a causal relationship being added between the existing Prepare event and the lone Bake event.

The alternative fix would generate a new **Prepare** event involving one of the students. This choice is illustrated in Figure 5 as the solid left branch emanating from **N1**. It is actually a "pseudo branch" corresponding to the dashed branch to the **student1** search space, search within that space, and a return from **student1** (also dashed) to the resulting **kitchen1** node. What occurs at this point is as follows. First, the search space for **student1** is resumed where it had left off (in a state where all its internal constraints had been satisfied). This state corresponds precisely to the **student1** plan descriptor found within the **kitchen1** plan descriptor at **N1**. Next, a new **Prepare** event is added to the **student1** plan, and the student's local constraints are rechecked. After **student1**'s constraints have been satisfied, control returns to the **kitchen1** search space.

Notice that no rechecking of the local constraints for any other student, the teacher, or the oven is necessary, since these could not possibly be affected by a **student1** event. However, some global **kitchen1** constraints may have to be rechecked as a result of this change. This clear delineation of which constraints need to be rechecked after a plan modification represents a significant advance over other planners. Even hierarchical planners, which partition the *expansion* of subplans, never really cope with the potential interactions among those subplans in any disciplined manner. As a result, all event interactions within a plan must be checked during a costly global interaction analysis. Indeed, this global interaction analysis, which is based on an examination of event pre- and postconditions, is usually done in a *pairwise* fashion. Thus, if three or more events combine to create ill effects (but pairwise combinations create no violations), these effects will typically never be discovered. In GEM's event-based framework, where constraint satisfaction is conducted with respect to *all* events within a region, methods for dealing with interference can be much more general.

Besides adding new events and relations, some constraint fixes may warrant the addition of entirely new elements or groups to a domain. (This is used in the blocks world domain illustrated in the appendix.) Thus, if a domain's group/element structure is left underconstrained by the domain description (a group may be described as consisting of a *set* of subelements, but the exact number of subelements is left unspecified), the planner can generate domain structure dynamically to accommodate the needs of the application. As new regions are added, new plan descriptors and search trees for these regions are generated.

The use of the GEMPLAN search table has proved to be a quite powerful and flexible means of guiding the planning process. Search can be guided by a domain's locality properties; when fixes modify a plan, rechecking for interference with other constraints can be limited to those regions and constraints that could be affected. In addition, GEMPLAN planning search can be flexibly tuned and focused. Researchers developing the ISIS scheduling system [10] have found resource- and agent-focused search to be useful for job shop scheduling. In GEMPLAN, the search can be focused according to any criterion – the user must simply specify domain structure, regional constraints, and the search table appropriately.

## 6 Conclusions

This paper has presented an event-based formalism, GEM, for representing parallel, multiagent domains. GEM can describe arbitrary forms of domain structure as well as complex synchronization constraints on events and their interrelationships. We have shown how the localizing effects associated with GEM's structuring mechanisms can be used to constrain potential relationships among events, constraints, and properties, thereby solving aspects of the frame problem.

We have also presented the GEMPLAN planning architecture, which directly partitions plan representation and the planning search space according to the structure of a domain. By employing a table-driven search mechanism, the planning process can be guided to take advantage of a domain's locality properties – i.e., precisely where regional interactions may or may not occur. We have used this system to construct multiagent solutions to blocks-world problems; it is our intention to extend its application to more complicated scheduling domains in the near future.

### Acknowledgments

David Fogelsong was an important contributor to the construction of GEMPLAN, especially the localized search mechanism. We would also like to thank those readers and critics who have helped to improve the quality of this report: Michael Georgeff, Martha Pollack, David Wilkins, Mark Drummond, Steven Rubin, Marc Vilain, and Savel Kliachko.

## References

[1] Allen, J.F. "Towards a General Theory of Action and Time," *Artificial Intelligence*, Vol. 23, No. 2, pp. 123-154 (1984).

[2] Allen, J.F. and J.A.Koomen, "Planning Using a Temporal World Model," *IJCAI-83, Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany, pp. 741-747 (August 1983).

[3] Chaitin, G.J. "Register Allocation and Spilling Via Graph Coloring," in *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, SIGPLAN Notices, Volume 17, Number 6, pp. 98-105 (June 1982).

[4] Chapman,D. "Cognitive Cliches," AI Working Paper 286, MIT Laboratory for Artificial Intelligence, Cambridge, Massachusetts (April 1986).

[5] Chapman, D. "Planning for Conjunctive Goals," Masters Thesis, Technical Report MIT-AI-TR-802, MIT Laboratory for Artificial Intelligence, Cambridge, Massachusetts (1985).

[6] Dean, T.L. and D.V. McDermott, "Temporal Data Base Management," *Artificial Intelligence*, Volume 32, Number 1, pp. 1-55 (April 1987).

[7] Drummond, M.E. "A Representation of Action and Belief for Automatic Planning Systems," in *Reasoning About Actions and Plans, Proceedings of the 1986 Workshop at Timberline, Oregon*, M.P. Georgeff and A.L. Lansky (editors), Morgan Kaufman Publishers, Los Altos, California, pp. 189-211 (1987).

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

[9] Finger, J.J. "Exploiting Constraints in Design Synthesis," Ph.D. Thesis, Department of Computer Science, Stanford University, Stanford, California (1986).

[10] Fox, M.S. and Smith, S.F. "ISIS – A Knowledge-Based System for Factory Scheduling," *Expert Systems, the International Journal of Knowledge Engineering*, Volume 1, Number 1, pp. 25-49 (July 1984).

[11] Georgeff, M. P. "Actions, Processes, and Causality," in *Reasoning About Actions and Plans, Proceedings of the 1986 Workshop at Timberline, Oregon*, M.P. Georgeff and A.L. Lansky (editors), Morgan Kaufman Publishers, Los Altos, California, pp. 99-122 (1987).

[12] Georgeff, M. P. "Many Agents are Better Than One," in *The Frame Problem in Artificial Intelligence, Proceedings of the 1987 Workshop*, F. Brown (editor), Morgan Kaufman Publishers, Los Altos, California (1987).

[13] Hayes, P.J. "The Frame Problem and Related Problems in Artificial Intelligence," from *Artificial Intelligence and Human Thinking*, pp. 45-59, A. Elithorn and D. Jones (editors), Jossey-Bass, Inc. and Elsevier Scientific Publishing Company (1973).

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

[15] Korf, R. E. "Planning as Search: A Quantitative Approach." *Artificial Intelligence*, Vol. 33, No. 1, pp. 65-88 (1987).

[16] Kowalski, R.A. and M.J. Sergot, "A Logic-based Calculus of Events," Department of Computing, Imperial College, London, November 1984, revised February 1985, to appear in *New Generation Computing* February 1986.

[17] Ladkin, P. "Time Representation: A Taxonomy of Interval Relations," in *Proceedings of the Fifth National Conference on Artificial Intelligence*, Philadelphia, Pennsylvania, pp. 360-366 (August 1986).

[18] Lamport, L. "Times, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM*, Volume 21, Number 7 (July 1978).

[19] Lansky, A.L. "A Representation of Parallel Activity Based on Events, Structure, and Causality," Technical Note 401, Artificial Intelligence Center, SRI International, Menlo Park, California (1986), also appearing in *Reasoning About Actions and Plans, Proceedings of the 1986 Workshop at Timberline, Oregon*, M.P. Georgeff and A.L. Lansky (editors), Morgan Kaufman Publishers, Los Altos, California, pp. 123-160 (1987).

[20] Lansky, A.L. "A 'Behavioral' Approach to Multiagent Domains," *in Proceedings of 1985 Workshop on Distributed Artificial Intelligence*, Sea Ranch, California, pp. 159-183 (1985).

[21] 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).

[22] Lansky, A.L. and D.S. Fogelsong, "Localized Representation and Planning Methods for Parallel Domains," in *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, Seattle, Washington (July 1987).

[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] Mackworth, A.K. "Consistency in Networks of Relations," *Artificial Intelligence*, Vol. 8, pp. 99-118 (1977).

[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*, Vol. 13, No. 1-2, pp.27-39 (1980).

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

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

[29] Pednault, E.P.D. "Formulating Multiagent, Dynamic-World Problems in the Classical Planning Framework," in *Reasoning About Actions and Plans, Proceedings of the 1986 Workshop at Timberline, Oregon*, M.P. Georgeff and A.L. Lansky (editors), Morgan Kaufman Publishers, Los Altos, California, pp. 47-82 (1987).

[30] Pelavin, R. and J.F. Allen, "A Formal Logic of Plans in Temporally Rich Domains," *Proceedings of the IEEE, Special Issue on Knowledge Representation*, Volume 74, No. 10, pp. 1364-1382 (October 1986).

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

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

[33] Shoham, Y., "What is the Frame Problem," in *Reasoning About Actions and Plans, Proceedings of the 1986 Workshop at Timberline, Oregon*, M.P. Georgeff and A.L. Lansky (editors), Morgan Kaufman Publishers, Los Altos, California, pp. 83-98 (1987).

[34] Stuart, C. "An Implementation of a Multi-Agent Plan Synchronizer Using a Temporal Logic Theorem Prover," *IJCAI-85, Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Los Angeles, California (August 1985).

[35] Tate, A. "Goal Structure, Holding Periods, and 'Clouds'," in *Reasoning About Actions and Plans, Proceedings of the 1986 Workshop at Timberline, Oregon*, M.P. Georgeff and A.L. Lansky (editors), Morgan Kaufman Publishers, Los Altos, California, pp. 267-277 (1987).

[36] 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).

[37] Vilian, M. and H. Kautz. "Constraint Propogation Algorithms for Temporal Reasoning," in *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, Seattle, Washington, pp. 376-382 (August 1986).

[38] Wilkins, D. "Domain-independent Planning: Representation and Plan Generation," *Artificial Intelligence*, Vol. 22, No. 3, pp. 269-301 (April 1984).

[39] Wilkins, D.E. "Using Causal Rules in Planning," Technical Note 410, Artificial Intelligence Center, SRI International, Menlo Park, California (1987).

# APPENDIX

## Annotated GEMPLAN Session

We now present an annotated session with the GEMPLAN planning system. The output includes a trace of the planning search process. The results of all checks for constraint violations are shown. After each successful constraint fix, the newly modified plan is printed. For ease of reading, we have annotated the actual output with explanatory comments and figures, and have eliminated some trace output that could prove confusing to the reader. The current GEMPLAN system also includes a user interface that depicts plans graphically as they are being constructed.

The domain being handled in this session is the blocks world. A user may specify any number of robot arms that may be used (1,2,... or "m" for "maximal"). The planner will then generate a *parallel blocks-world solution commensurate with number of robots at its disposal.*[18]

Each robot can execute two types of events: Pick(Block) and Put(Block,Surface). (In the sample session output, the reader will note that each event is also associated with an integer parameter – for example, put(1,b,c). This parameter serves as a unique identifier for the event.) For each blocks-world problem to be solved, the planner is provided with information about the desired initial and final state. This state is described in terms of the predicate on(X,Y). Thus, an initial configuration might be described as {on(a,b), on(b,c)} and a final state as {on(c,b), on(a,c)}. As we will show, the predicates on and clear used in this domain are completely defined in terms of the domain's initial configuration and all subsequent Pick and Put actions.

The blocks-world domain specification is partitioned into a set of RobotArm elements, all belonging to a global group, globalgroup. Each RobotArm is associated with three local constraints:

- **RobotArm Constraint 1**
  This constraint makes sure that every event of type Put is caused by a corresponding event of type Pick (i.e., a robot cannot put a block down unless it has already picked it up).

[18]We have also implemented an algorithm that can convert any N-robot solution into an M-robot solution, where $1 \leq M \leq N$. Based on a graph-coloring algorithm used for register allocation [3], the algorithm performs an analysis of the thirty possible classes of block usage interference.

- **RobotArm Constraint 2**
  This constraint makes sure that every event of type Pick eventually enables exactly one corresponding event of type Put (i.e., a robot must eventually deposit every block it picks up).

- **RobotArm Constraint 3**
  This pattern constraint makes sure that the sequence of Pick and Put events at the robot alternates — that is, it must form a pattern of the form: pick1 $\rightsquigarrow$ put1 $\Longrightarrow$ pick2 $\rightsquigarrow$ put2 .... This ensures that a robot can be holding at most one block at a time.

The blocks-world domain is also associated with three global constraints pertaining to activity at all robots:

- **globalgroup Constraint 1**
  This pattern constraint makes sure that uses of a block do not interfere with one another. In particular, guarantees that:

  1. All uses of the same block (i.e., for block b, an event pair of the form pick(b) $\rightsquigarrow$ put(b,X)) are sequenced. This assures mutually exclusive access to a block.
  2. Every time an object is put down on top of a block (i.e., for block b, an event of type Put(Y,b)), that block must be stationary — it cannot be held by a robot arm.

  Note that this constraint would be unnecessary in a single-agent domain. The actual pattern tested has the following form: $((\text{Pick(b)} \rightsquigarrow \text{Put(b,X)})^{*\Longrightarrow} \Longrightarrow (\text{Put(Y,b)})^{*\Longrightarrow})^{*\Longrightarrow}$.

- **globalgroup Constraint 2**
  This precondition constraint makes sure that, in any journal preceding the occurrence of an event of type Pick(b) or Put(X,b), block b is clear:

  $$(\forall\ \text{blockuse(b)}{:}\{\text{Pick(b)},\text{Put(X,b)}\})\ \text{justoccurred(blockuse(b))} \supset \triangle\ \text{clear(b)}$$

  The definition of the predicate clear is defined in terms of adder/deleter events. In particular, given that a block b' is on top of a block b (on(b',b)), an event of the form pick(b') "adds" clear(b). An event of the form Put(X,b) "deletes" clear(b). A block is defined to be initially clear if nothing is defined to be on it.

- **globalgroup Constraint 3**
  This constraint makes sure that, in the final journal for every execution of the plan, all goals of the form on(X,Y) are satisfied. The definition of the predicate on is also defined in terms of adder/deleter events. In particular, on(X,Y) is defined to be true if it was true in the initial state and was not subsequently deleted by an event of type Pick(X) or any movement of Y (Pick(Y), Put(Y,Z)). Or, on(X,Y) is true if an event of type Put(X,Y) has occurred and it cannot have been subsequently followed by a "deleter" event of type Pick(X) or a movement of Y.

We now proceed with our sample session. Note that it also illustrates the use of constraint *accumulation* for determining values of unbound event parameters as well as protection of conditions once they are attained.

```
| ?- [startgemplan].
Loading GEMPLAN....

Welcome to GEMPLAN.

File containing domain table --
(Type in quotes and end with . followed by CR):  'unifiedbwTable.pl'.
File containing domain constraints --
(Type in quotes and end with . followed by CR):  'unifiedbw.pl'.

File unifiedbwTable.pl has been consulted by GEMPLAN.
File unifiedbw.pl has been consulted by GEMPLAN.

Do you want a full trace? (y or n, followed by a .):  y.

How many robots in this domain (1,2,... or m (maximal))?:  m.

File containing initial conditions: amyAnomaly.
Working on problem defined in file amyAnomaly.

INITIAL PLAN:
INIT:on(a,b)
INIT:on(b,table)
INIT:on(c,table)
FINAL:on(a,b)
FINAL:on(b,c)
FINAL:on(c,table)

==========================================================================
```

The initial conditions file has set up the actual problem to be solved. In this case, the planner will create a maximally parallel solution to the problem depicted below:

```
     INITIAL CONDITIONS                 :         GOAL
                                        :
                                        :        -----
                                        :        | a |
                                        :        -----
   -----                                :        | b |
   | a |                                :        -----
   -----         -----                  :        | c |
   | b |         | c |                  :        -----
   -----         -----                  :
=========================================================================
```

```
...Now checking for constraint satisfaction in location globalgroup
Failed constraint #3 of location globalgroup
    in plan [globalgroup-noplan].
    Encountered bugs: [on(b,c)]

Putting event put(1,b,c) into newly created element robotArm1

=========================================================================
```

At this point the planner realizes it must create a new Put event and a new robot arm in order to place block b on top of block c. The search will now focus on satisfying the local constraints of the newly created robot arm.

```
=========================================================================

...Now checking for constraint satisfaction in location robotArm1
Failed constraint #1 of location robotArm1
    in plan [robotArm1-plan1].
    Encountered bugs: [put(1,b,c)]

=========================================================================
```

At this point the planner realizes that, in order to put b down on top of c, block b must first be picked up.

```
=========================================================================

Just FIXED constraint #1 in location robotArm1 using fix #2
    resulting in plan [robotArm1-plan2]
Working plan at location robotArm1 is:

Plan #[robotArm1-plan2]
PLAN INITIALIZATIONS:
INIT:on(a,b)
INIT:on(b,table)
INIT:on(c,table)
SET OF EVENTS AND THEIR RELATIONS:
EVENT pick(2,b) at element robotArm1
pick(2,b) CAUSES put(1,b,c)
```

```
EVENT put(1,b,c) at element robotArm1

----End of events and relations
 for plan [robotArm1-plan2].

========================================================================
```

This plan may be depicted as follows:

$\textsf{robotArm1:}\quad \textsf{pick(b)} \;\rightsquigarrow\; \textsf{put(b,c)}$

```
========================================================================

Plan [robotArm1-plan2]
 in location robotArm1 satisfies constraint #2
Plan [robotArm1-plan2]
 in location robotArm1 satisfies constraint #3
Found a workable plan for location robotArm1
    It is plan [robotArm1-plan2].
Now popping to higher level planning...

Just FIXED constraint #3 in location globalgroup using fix #2
    resulting in plan [globalgroup-plan1,robotArm1-plan2]

Working plan at location globalgroup is:

Plan #[globalgroup-plan1,robotArm1-plan2]
PLAN INITIALIZATIONS:
INIT:on(a,b)
INIT:on(b,table)
INIT:on(c,table)
SET OF EVENTS AND THEIR RELATIONS:
EVENT pick(2,b) at element robotArm1
pick(2,b) CAUSES put(1,b,c)

EVENT put(1,b,c) at element robotArm1

----End of events and relations
 for plan [globalgroup-plan1,robotArm1-plan2].

========================================================================
```

This is the same plan depicted earlier.

```
========================================================================

Plan [globalgroup-plan1,robotArm1-plan2]
 in location globalgroup satisfies constraint #1
Failed constraint #2 of location globalgroup
    in plan [globalgroup-plan1,robotArm1-plan2].
    Encountered bugs: [[pick(2,b),initevent,a,b]]
```

```
======================================================================
```

At this point the planner realizes that block b is not clear and that block a must be picked off it. It generates a new robot arm to do this. (Remember, we are generating the maximally parallel solution. If we had allowed only one robot to be used, this new event would be inserted into robotArm1.)

```
======================================================================

Putting event pick(3,a) into newly created robot robotArm2

...Now checking for constraint satisfaction in location robotArm2
Plan [robotArm2-plan1]
 in location robotArm2 satisfies constraint #1
Failed constraint #2 of location robotArm2
    in plan [robotArm2-plan1].
    Encountered bugs: [pick(3,a)]

======================================================================
```

At this point the planner realizes that block a must be put somewhere, but it does not yet decide where it is placed. It does know, however, that it cannot be positioned on itself.

```
======================================================================

Just FIXED constraint #2 in location robotArm2 using fix #2
    resulting in plan [robotArm2-plan2]
Working plan at location robotArm2 is:

Plan #[robotArm2-plan2]
PLAN INITIALIZATIONS:
INIT:on(a,b)
INIT:on(b,table)
INIT:on(c,table)
SET OF EVENTS AND THEIR RELATIONS:
EVENT pick(3,a) at element robotArm2
pick(3,a) CAUSES put(4,a,unbound)

EVENT put(4,a,unbound) at element robotArm2
Parameter 3 of put(4,a,unbound) has a constrained domain of [b,c,table]
   out of an unconstrained domain of [a,b,c,table].
The constraints that applied are the following:
   not(parameter=a)

----End of events and relations
 for plan [robotArm2-plan2].

======================================================================
```

This plan may be depicted as follows:

$$\text{robotArm2:}\quad \text{pick}(a) \rightsquigarrow \text{put}(a,?\neq a)$$

```
=======================================================================

Plan [robotArm2-plan2]
 in location robotArm2 satisfies constraint #3
Plan [robotArm2-plan2]
 in location robotArm2 satisfies constraint #1
Found a workable plan for location robotArm2
    It is plan [robotArm2-plan2].
Now popping to higher level planning...

Just FIXED constraint #2 in location globalgroup using fix #4
    resulting in plan [globalgroup-plan4,robotArm1-plan2,robotArm2-plan2]
Working plan at location globalgroup is:

Plan #[globalgroup-plan4,robotArm1-plan2,robotArm2-plan2]
PLAN INITIALIZATIONS:
INIT:on(a,b)
INIT:on(b,table)
INIT:on(c,table)
SET OF EVENTS AND THEIR RELATIONS:
EVENT pick(2,b) at element robotArm1
pick(2,b) CAUSES put(1,b,c)

EVENT pick(3,a) at element robotArm2
pick(3,a) CAUSES put(4,a,unbound)
pick(3,a) BEFORE pick(2,b)

EVENT put(1,b,c) at element robotArm1

EVENT put(4,a,unbound) at element robotArm2
Parameter 3 of put(4,a,unbound) has a constrained domain of [b,c,table]
   out of an unconstrained domain of [a,b,c,table].
The constraints that applied are the following:
   not(parameter=a)

----End of events and relations
 for plan [globalgroup-plan4,robotArm1-plan2,robotArm2-plan2].

=======================================================================
```

This consolidated global plan may be depicted as follows:

$$\begin{tikzcd}
\text{robotArm1:} & \text{pick}(b) \arrow[r, rightsquigarrow] & \text{put}(b,c) \\
\text{robotArm2:} & \text{pick}(a) \arrow[r, rightsquigarrow] \arrow[u, Rightarrow] & \text{put}(a,?\neq a)
\end{tikzcd}$$

```
=======================================================================

Failed constraint #3 of location globalgroup
```

```
    in plan [globalgroup-plan4,robotArm1-plan2,robotArm2-plan2].
    Encountered bugs: [on(a,b)]

=========================================================================
```

At this point the planner realizes that block a is no longer on block b at the end of plan execution. The fix chosen is to bind the unbound parameter in the event put(a,?) to be equal to b. In addition, GEMPLAN's protection facility adds the relation put(b,c) $\Longrightarrow$ put(a,b). This is done to protect the condition on(a,b) – obviously, b cannot be moved once a has been placed upon it. The resulting plan is actually the final plan that satisfies all constraints.

```
=========================================================================

Just FIXED constraint #3 in location globalgroup using fix #1
    resulting in plan [globalgroup-plan5,robotArm1-plan2,robotArm2-plan2]

Working plan at location globalgroup is:

Plan #[globalgroup-plan5,robotArm1-plan2,robotArm2-plan2]
PLAN INITIALIZATIONS:
INIT:on(a,b)
INIT:on(b,table)
INIT:on(c,table)
SET OF EVENTS AND THEIR RELATIONS:
EVENT pick(2,b) at element robotArm1
pick(2,b) CAUSES put(1,b,c)

EVENT pick(3,a) at element robotArm2
pick(3,a) CAUSES put(4,a,unbound)
pick(3,a) BEFORE pick(2,b)

EVENT put(1,b,c) at element robotArm1
put(1,b,c) BEFORE put(4,a,unbound)

EVENT put(4,a,unbound) at element robotArm2
Parameter 3 of put(4,a,unbound) has a constrained domain of [b]
   out of an unconstrained domain of [a,b,c,table].
The constraints that applied are the following:
   not(parameter=a)
   parameter=b

----End of events and relations
 for plan [globalgroup-plan5,robotArm1-plan2,robotArm2-plan2].

=========================================================================
```

This plan may be depicted as follows:

$$\begin{tikzcd}
\text{robotArm1:} & \text{pick(b)} \arrow[r, rightsquigarrow] & \text{put(b,c)} \arrow[d, Rightarrow] \\
\text{robotArm2:} & \text{pick(a)} \arrow[r, rightsquigarrow] \arrow[u, Rightarrow] & \text{put(a,?=b)}
\end{tikzcd}$$

```
==========================================================================
Plan [globalgroup-plan5,robotArm1-plan2,robotArm2-plan2]
 in location globalgroup satisfies constraint #1
Plan [globalgroup-plan5,robotArm1-plan2,robotArm2-plan2]
 in location globalgroup satisfies constraint #2
Plan [globalgroup-plan5,robotArm1-plan2,robotArm2-plan2]
 in location globalgroup satisfies constraint #3
Found a workable plan for location globalgroup
    It is plan [globalgroup-plan5,robotArm1-plan2,robotArm2-plan2].
Now popping to higher level planning...
Now finalizing parameter bindings...
The FINAL plan is:

Plan #[globalgroup-plan5,robotArm1-plan2,robotArm2-plan2]
PLAN INITIALIZATIONS:
INIT:on(a,b)
INIT:on(b,table)
INIT:on(c,table)
SET OF EVENTS AND THEIR RELATIONS:
EVENT pick(2,b) at element robotArm1
pick(2,b) CAUSES put(1,b,c)

EVENT pick(3,a) at element robotArm2
pick(3,a) CAUSES put(4,a,unbound)
pick(3,a) BEFORE pick(2,b)

EVENT put(1,b,c) at element robotArm1
put(1,b,c) BEFORE put(4,a,unbound)

EVENT put(4,a,unbound) at element robotArm2
Parameter 3 of put(4,a,unbound) has a constrained domain of [b]
   out of an unconstrained domain of [a,b,c,table].
The constraints that applied are the following:
   not(parameter=a)
   parameter=b

----End of events and relations
 for plan [globalgroup-plan5,robotArm1-plan2,robotArm2-plan2].

yes

==========================================================================
```

Note that, although this particular multiagent plan is actually sequential, it is the most efficient plan for this problem. A single-agent plan would require two more events to clear block a off block b. The current blocks-world application has also been tested on several other problems, including the Sussman Anomaly (which can be generated to use one, two or three robots), a simulated "register switch" problem, where two blocks resting on two other blocks are swapped, and a four-block problem from Sacerdoti's NOAH paper [31].