SYNCHRONIZATION OF MULTIAGENT PLANS USING A TEMPORAL LOGIC THEOREM PROVER

Technical Note 350

December 13, 1985

By: Christopher Stuart*
Artificial Intelligence Center
Computer Science and Technology Division

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED

This work was supported in part by the Office of Naval Research, Contract No. N00014-85-C-0251, SRI Project 8342. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of SRI International, the Navy, or the U.S. govenment.

*Currently at the Department of Computer Science, Monash University, Clayton 3168, Victoria, AUSTRALIA .

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

Contents

1 Introduction **2**

2 A theory of action **4**

2.1 Actions 4

2.2 The environment 6

2.2.1 Strings and execution in an environment 10

2.3 Agents 16

2.3.1 Classes of agents 17

2.4 Plan syntax 26

2.5 Plan semantics 27

2.6 Extensions to the syntax 36

3 The interaction problem **38**

3.1 Events and event failure 39

3.1.1 Restricted propositional environments 39

3.2 Synchronizing Plans 44

4 The plan synchronizer **46**

4.1 Interaction analysis 46

4.2 PTL constraint generation 48

4.3 PTL theorem prover 55

4.4 Synchronized plan generation 60

5 An example **61**

6 Conclusion and future work **63**

A Notation conventions **66**

B Propositional Temporal Logic **67**

B.1 Syntax 67

B.2 Semantics 67

References **69**

1 Introduction

Most of us at some time have stopped to watch the construction of a house. It is interesting to watch the successive stages of the project and the activity of workers engaged in the various tasks necessary to reach the desired product. This simple example is useful in considering aspects of the planning problem.

The planning problem is to find some plan that can guide the activity of an agent or agents to achieve a desired goal. The particular subproblem considered here is that of resolving possible conflicts between elements of a plan. There are several ways subplans can interact:

- One subplan may achieve the precondition of another. For example, the sides of a house must be built for support before the roof is laid.
- One may remove the precondition of another. Plumbing is best connected before the wall is finished so that access is easier.
- One may upset a condition that needs to be maintained for a time. The act of building the front steps requires that, while the concrete is drying, others should avoid working on the roof where they may drop things on the new steps.
- They may co-operate. If a truck is sent to pick up timber, it may be efficient to collect bricks on the same trip.

All planning systems operate by combining actions or subplans in some way so that the total plan satisfies some constraint - usually to achieve some goal. They use some means of modeling actions and the changes they induce in the world, and determine the total effect of a plan from its constituent parts and the ways in which they interact.

A very common way of using interaction is to choose actions that achieve the goal, and then plan to achieve the preconditions of those actions. The STRIPS planner[4] uses this principle. However, it does not consider subplans in parallel and look for interactions between parallel branches.

In the planning systems NOAH[13] and SIPE[16], consideration of interactions between subplans is explicit in the planning process. A plan is a partial order of subplans, allowing potential parallelism to be exploited or deferring the choice of an ordering. The planning

technique is to expand subplans into partial orderings of lower level subplans. After the expansion, the system looks for conflicts in the new, more detailed plan and attempts to resolve them. One important way of resolving conflicts is to impose orderings in the plan.

The interaction detection and resolution is performed by *critics*, which look for particular types of interaction and use particular means of resolving them. However, these critics will not necessarily find a correct solution, given that one exists, and the means of resolving conflicts may be unnecessarily restrictive. For example, if it is required that two subplans may not execute at the same time, as will be the case when they use a unique resource, then NOAH will impose an arbitrary ordering on them.

Following an approach suggested by Georgeff [5], this paper considers the use of synchronizing primitives to resolve conflicts and produce a plan that is as unrestrictive as possible. In a sense, the whole planning problem is to find actions that interact in such a way that the plan achieves its goal; in fact, it can be shown that certain general planning problems may be framed and solved as conflict resolution problems. However, the technique developed here is designed specifically for resolving conflicts in existing plans, and is not presented as a means of planning in itself.

We will proceed by considering a theory of action and the world and presenting a simple planning language that can express a wide variety of plans. A formal definition of an agent will be given, and plans will be given a semantics in terms of agents. We will show a pleasing one-to-one correspondence between plans and classes of agents.

A program has been written to synchronize plans in such a way that always allows the plan to complete successfully despite conflicts and interactions between parts of the plan, and the theory behind this program is explored. Plan execution and failure will be discussed, and constraints on plan executions will be found which restrict executions to all and only those that can be guaranteed not to fail. By framing these constraints in a propositional temporal logic and passing them to a theorem prover, we can generate an agent corresponding to the constrained plan. Finally, a revised plan is produced corresponding to the new agent. Related work on using temporal logic constraints on actions as a framework for plan generation is also being done by Lansky [10]. In the final sections of the paper, we consider extensions to the theory and the program, and problems and advantages with the approach.

2 A Theory of Action

In this section of formal definitions, we will adhere to certain conventions. X^* will denote the set of finite sequences over X; $X^\times$ will denote the set of countable sequences over X. Thus $X^* \subset X^\times$. Subscripts will identify elements of a sequence. Multisets will be given as sequences for convenience: a multiset is in fact an equivalence class of sequences which may be obtained from each other by permutation, or alternatively a mapping from elements to integers greater than or equal to zero (we never use infinite multisets).

The standard function *pop* will be used to remove an element from a sequence. If $\sigma = (\sigma_1, \sigma_2, \sigma_3 \ldots)$ is a sequence then $\text{pop}(\sigma) = (\sigma_2, \sigma_3 \ldots)$. $\text{pop}^i(\sigma), i \geq 0$ will correspond to i applications of pop. That is, $\text{pop}^i(\sigma) = (\sigma_{i+1}, \sigma_{i+2} \ldots)$. It is undefined if σ has less than i elements.

The concatenation operator for sequences is $\circ$.

Definition 1 *Let σ and σ' be two sequences. If σ is infinite, then $\sigma \circ \sigma' = \sigma$. Otherwise suppose σ is $(\sigma_1, \sigma_2, \ldots, \sigma_n)$, and σ' is $(\sigma'_1, \sigma'_2, \ldots)$. Then $\sigma \circ \sigma'$ is is $(\sigma_1, \sigma_2, \ldots, \sigma_n, \sigma'_1, \sigma'_2, \ldots)$.*

We will apply set operators to multisets with the following conventions:

- Union is equivalent to concatenation.
- For multiset difference $\sigma \backslash \sigma'$, for each element σ_n of σ, the number of σ_n in the result is the difference of the number in σ and the number in σ' (or 0 if the difference is negative).

As far as possible, we will also be consistent with our use of notation. Where some symbol is used to represent a class of objects, then we will continue to represent the class of objects with that same symbol. The symbol ϕ will always refer to the empty set, sequence, or multiset.

More details on notation are given in an appendix.

2.1 Actions

The world is often considered to be a collection of possible world states, and at a moment one of those states reflects the way the world is. These states can be considered to be interpretations for some logical language, and statements made about the world in that language will be true or false in a particular state.

An agent must have some means of changing the world. This is usually done by having the agent execute actions. The model of an action used by a reasoning agent must induce changes in the world model. In many planning systems, this is done by having actions be modeled as mappings on world states. This is satisfactory for simple sequential plans that do not need to consider the progression of an action over time. Even NOAH and SIPE, which allow partially ordered plans with parallel branches, assume that a plan execution will be a total ordering of those actions, and so ultimately produce a partial order of primitive actions that may be considered to be instantaneous.

When multiple agents are operating in parallel, it may be possible for two actions to be executing simultaneously; therefore, actions must have a beginning and an end, and must occur over a period of time. Much consideration has been given to more complex models of action (for example, see [1,2,6,12]). Many of these decompose actions into more primitive parts that are effectively instantaneous, although Stuart[14] and Hendrix[7] do consider actions inducing continuous functions from time into world states.

For this investigation, it is appropriate to consider an action to be decomposed into discrete transformations of the world, which are called *events*. An event also has an associated correctness condition, which must be true at the moment the event is executed. One might consider that the correctness condition should be true over a period of time; we will show that in our theory the two cases are equivalent. The mapping associated with an event will be deterministic. We could introduce a special *fail* world state, and incorporate the correctness condition as part of the mapping, by mapping all incorrect states to the fail state, but that would be inconvenient when we later restrict the types of mappings allowed for an event.

In this discussion, we will consider the world to be a set of possible states W, and at any moment the real world will correspond to some $\omega \in W$.

Definition 2 *Let W be a set of worlds. Then an* event *for that set of worlds is a mapping and a correctness condition.*

$$\text{events}(W) = W^W \times 2^W$$

Intuitively, at the moment immediately before an event is executed the world must be in one of the states in the correctness condition, and the mapping defines for each world state a unique next state which results from the execution.

An action will be a set of possible finite sequences of events. It is indivisible in the sense that no control can be exercised over it while it is being executed. It is nondeterministic in the sense that any of the sequences of events may occur. The internal structure is used to model progression of the world over time and to define a model of concurrency, which is the standard interleaving model.

Definition 3 *Let W be a set of worlds. Then an* action *for that set of worlds is a set of finite sequences of events.*

$$\text{actions}(W) = 2^{(\text{events}(W)^*)}$$

Several things may be noted about this definition of action. First, there is no control component that guides the choice of sequences of events based on the world state. Since there is no control over the sequence taken, we must insist that all sequences be finite if we wish the action to terminate. Second, since actions are nondeterministic, the fact that events are deterministic is not restrictive. A nondeterministic event can be considered to be a set of deterministic events, and every sequence of such events can be considered to be a set of sequences of deterministic events.

2.2 The Environment

Another aspect of the world relevant to an agent is the means by which its actions are executed. Thus the world state should include the status of actions being executed. The term *environment* is used to mean the world in this larger sense.

The intuition used in developing the following theory of action is that we are interested in agents such as robots, where the agent changes the world through asynchronous devices. The agent will send an output command to some device, and that device responds at some moment after the command is sent. The agent will "know" that the effect of the command is completed, either by waiting for some period of time or by receiving some data from the device. The effect is assumed to have finished at some moment before the time elapsed or the input was received. Note that some commands (such as "start drive motor") may complete with the world in some dynamic state. In general, this paper is not concerned with such low-level commands. The devices through which the agent sends and receives messages is called an *environment*.

An environment at any moment can receive as input some message, or can take the next step in the execution of some *current* action. An action execution may also cause the

environment to send a message as output. All these occasions cause the environment to change state.

More formally, an environment consists of a set of worlds, a set of operators, and a mapping from operators to actions. An operator can be viewed as a command to perform the action associated with the operator.

Definition 4 *An* environment *is a triple* $< W, A, i >$. W *is some set of* world states, A *is some set of* operators. i *is a function* $i : A \mapsto \text{actions}(W)$.

The environment corresponds to a machine that changes state as actions are executed. We make the assumption common in planning systems that the world changes state only as the result of activity of the agent: in this case, by an event execution, which in turn occurs only as the result of an action. This paper is concerned with synchronizing the activities of agents in parallel, and we will show how our formal model of agents can represent multiple real agents.

By considering one real agent in isolation, other agents may be seen to be part of the total environment, which will thus be dynamic. Similarly, certain dynamic aspects of a real environment may be modeled as an agent. Note that we use the term *agent* in two ways. An agent may be a real-world object, which we wish to represent in order to reason about it – a robot or machine or person. An agent also is a mathematical object (yet to be defined) that interacts with an environment. Where there may be confusion, we will refer to *real agents* and *formal agents*.

An environment may execute several instances of the same operator simultaneously, and it may be important for an agent to distinguish them. Thus an agent will have a set ι of *tags* that can be associated with operators, and the environment keeps track of the association. An (operator, tag) pair will be referred to as an *identifier*.

The function i of an environment is extended to identifiers as follows:

$$\forall(\alpha, x) \in A \times \iota \, . \, i(\alpha, x) = i(\alpha)$$

$A \times \iota$ may be regarded as an extended set of operators with the constraint that the interpretation does not depend on the tag. We will use A_ι as shorthand for $A \times \iota$.

At any instant, the environment will be in a certain *environment state*. An environment state has three components:

- The world state
- The state of executing actions
- A failure status reflecting whether or not an event has failed.

Definition 5 *An* environment state *for an environment* $<W, A, \imath>$ *is a 3-tuple* $<\omega, E, \tau>$, *where* $\omega \in W$ *is some world state,* E *is a finite multiset of identifiers with associated sequences of events, and* τ *is either* fail *or* succeed *depending on whether or not the last event failed. That is, for some set* ι

$$\omega \in W$$

$$E \in (A_\iota \times \text{events}(W)^*)^*$$

$$\tau \in \{\text{fail}, \text{succeed}\}$$

An environment state will make sense only in the context of some object (such as an agent) which defines the set ι.

Let the environment be $<W, A, \imath>$, and let the environment state be $<\omega, E, \tau>$.

If the environment receives as input the identifier $\alpha \in A_\iota$, then it arbitrarily selects some sequence of events ς from the set $\imath(\alpha)$, and appends (α, ς) to E.

If the multiset E is non-empty, the environment may at any time select an arbitrary element (α, ς) from E for execution. If ς is the empty sequence, then the element (α, ς) is deleted from E and the identifier α is provided as output. Otherwise let ς_1 be the event (δ, γ). δ is a state transformation and γ is the correctness condition. If it is not the case that

$$\omega \in \gamma$$

then the environment is said to have *failed* the correctness condition of the event, and τ becomes *fail*; otherwise, τ becomes *succeed*. In both cases, ς in E is replaced by the sequence pop(ς), and ω becomes $\delta(\omega)$.

The operation of the environment also has a fairness constraint: any sequence in the set E will eventually execute. Since these sequences are finite, it follows that any action will eventually complete. The environment is said to *quiescent* if the set E is empty. In this case it may change state only on receiving some message.

Definition 6 *An environment is* quiescent *iff the environment state* $< \omega, E, \tau >$ *is such that* $E = \phi$.

An agent is the object with which the environment exchanges messages. For this initial formalization, we consider agents for which the exchange of messages is the only means of interaction with the environment. This corresponds to the case where a plan is constructed with no conditional testing except possibly on variables entirely local to the plan. This is often the case for simple plans that assume well-defined outside interference (if any) and for which actions are a high-level abstraction of lower-level routines that perform simple interaction with the world. For example, a planner may consider a *goto* operation to be primitive and have that action implemented as a routine that follows walls in the real world and otherwise guarantees that the total action result is quite predictable. This is the case with the STRIPS planner controlling the SHAKEY robot.

Planning for an agent may be done by modeling the real world and the agent's capabilities as an environment, and finding a plan to exchange messages with the environment in such a way that it never fails the correctness condition of any event and the sequence of world states produced satisfies some constraint, usually a condition on the final state of the world.

An agent is defined formally as a mathematical object that interacts with an environment only by sending and receiving identifiers. Agents in this formal sense are used to define semantics for a class of plans. The execution of a plan corresponds to some sequence of messages between an agent and the environment. Let A_ι be the set of identifiers. Then the sequence of messages will be denoted by a string (which means the same as sequence) over the alphabet $\{\text{begin}, \text{end}\} \times A_\iota$. For $\alpha \in A_\iota$, *(begin α)* corresponds to the agent sending α to the environment to cause the associated action to be executed, and *(end α)* corresponds to the environment sending α to the agent to indicate that the associated action has completed. Where there is no confusion, we refer to ***strings*** and assume them to be over this alphabet. Strings may be infinite.

Note that the agent may receive an *end* message at any time for an action that is currently executing, and that the operation of the environment restricts the possible strings since it will never send an *end* message unless it has previously received a corresponding *begin*.

2.2.1 Strings and Execution in an Environment

In this section, we consider classes of strings and the way in which they correspond to execution in an environment. For any string and an environment, we find associated sequences of string execution states and world states. We define *reasonable* strings, which correspond to possible executions in the environment, and *fair* strings, which correspond to possible executions in the environment under the fairness constraint. We also find equivalence classes for sets of strings that induce the same sets of sequences of world states. This section may be skipped by those who are not interested in that level of detail.

Definition 7 *A string over the alphabet* {begin, end} $\times A_\iota$ *is* reasonable *iff for any* $\alpha \in A_\iota$ *and any finite initial substring, the number of* (begin α) *is greater than or equal to the number of* (end α).

The set of *reasonable* strings corresponds exactly to the set of all possible message sequences an environment could exchange with an agent, without the fairness constraint.

Definition 8 *A reasonable string is* fair *iff any finite initial substring which has more* (begin α) *than* (end α) *is eventually followed by an* (end α).

Fair strings correspond to the possible message sequences under the fairness constraint that any action is eventually completed. Fair strings can also, however, result from unfair executions.

Proposition 1 *For any finite fair string and any* $\alpha \in A_\iota$*, the number of* (begin α) *is equal to the number of* (end α).

Proposition 2 *Any finite reasonable string is the prefix for some fair string.*

Since the agent interacts with the environment only through the sending and receiving of messages, we will consider the possible sequences of environment states for a given message sequence. This also will be a more formal description of the operation of an environment.

Suppose the environment $<W, A, i>$ is given. There is a *successor* relation on environment states. A state is a successor of another if it can be reached by a single event execution for a current action.

Definition 9 *Let* $<\omega_1, E_1, \tau_1>$ *be an environment state. Then a state* $<\omega_2, E_2, \tau_2>$ *is a* successor *of* $<\omega_1, E_1, \tau_1>$ *iff*

$$\begin{aligned} \exists(\alpha,\varsigma) \in E_1 \;.\;\; & \varsigma \neq \phi, \\ & \varsigma_1 = (\delta, \gamma), \\ & \omega_2 = \delta(\omega_1), \\ & E_2 = (E_1 \backslash \{(\alpha,\varsigma)\}) \cup \{(\alpha, \text{pop}(\varsigma))\}, \\ & \text{if } \omega_1 \in \gamma \;\; \text{then } \tau_2 = \text{succeed} \\ & \qquad\qquad\quad \text{else } \tau_2 = \text{fail} \end{aligned}$$

An *end* message can be regarded as a function between environment states. For consistency with the operation of the *begin* message, which involves a nondeterministic choice for an action, we will represent the function as mapping a state to a singleton set of states, or to the null set if the original state is one which could not send the message.

Definition 10 *If* $\exists(\alpha,\varsigma) \in E \;.\; \varsigma = \phi$ *then*

$$(\text{end } \alpha)(<\omega, E, \tau>) = \{<\omega, E \backslash \{(\alpha,\varsigma)\}, \tau>\}$$

otherwise

$$(\text{end } \alpha)(<\omega, E, \tau>) = \phi$$

A *begin* message can be regarded as a function from environment states to sets of environment states.

Definition 11

$$(\text{begin } \alpha)(<\omega, E, \tau>) = \{<\omega, E \cup \{(\alpha,\varsigma)\}, \tau>: \varsigma \in i(\alpha)\}$$

We can define a successor relation between pairs of environment states and strings. For convenience, we will refer to an (environment state, string) pair as a *string execution state.*

Definition 12 *Let* $(<\omega, E, \tau>, \sigma)$ *be given. Then* $(<\omega', E', \tau'>, \sigma')$ *is a* successor *if one of the following cases holds:*

- $\sigma = \sigma'$ *and* $<\omega', E', \tau'>$ *is a successor of* $<\omega, E, \tau>$.
- $\sigma' = \text{pop}(\sigma)$ *and* $<\omega', E', \tau'> \in \sigma_1(<\omega, E, \tau>)$.
  σ_1 *is of the form* $(\text{begin } \alpha)$ *or* $(\text{end } \alpha)$.

Now given a string σ we can define the associated set of sequences of string execution states.

Definition 13 *Let σ be any string. A sequence of string execution states ξ is an* associated sequence *for the string iff the following properties hold:*

- $\xi_1 = (<\omega, \phi, \text{succeed}>, \sigma)$ *for some* $\omega \in W$.
- ξ_n *is a successor of* ξ_{n-1} *for every n from 2 to the end of the sequence.*
- ξ *is infinite, or has a final element* $\xi_n = (<\omega, E, \tau>, \sigma)$ *where* $\sigma = \phi$.

The above definitions define carefully the operation of the environment. It always begins in a quiescent state and it changes state only by executing an event or exchanging a message with an agent. With the operation of the environment well defined, we can prove the association between reasonable strings and possible environment executions.

Proposition 3 *A string is reasonable iff it has some associated sequence of string execution states.*

We also present the simple but useful result that makes the above definition sensible by asserting that all the string is eventually used.

Proposition 4 *For any reasonable string and any associated sequence of string execution states, any finite initial prefix of the original string will eventually be stripped to produce a string in the sequence.*

The fairness constraint on the operation of an environment is that any available event is eventually executed.

Definition 14 *A sequence of string execution states ξ is* fair *iff it has the property*

- *For any* $\xi_n = (<\omega_n, E_n, \tau_n>, \sigma_n)$ *in* ξ, *and for any* (α, ς) *in* E_n, *there is some* $m > n$ *such that*

$$\begin{array}{rcl} \xi_m & = & (<\omega_m, E_m, \tau_m>, \sigma_m) \\ \xi_{m-1} & = & (<\omega_{m-1}, E_{m-1}, \tau_{m-1}>, \sigma_{m-1}) \\ (\alpha, \varsigma) & \in & E_{m-1} \\ E_m & = & \left[\begin{array}{lll} \text{if } \varsigma = \phi & \text{then} & (E_{m-1} \backslash \{(\alpha, \varsigma)\}) \\ & \text{else} & (E_{m-1} \backslash \{(\alpha, \varsigma)\}) \cup \{(\alpha, \text{pop}(\varsigma))\} \end{array} \right] \end{array}$$

The following result asserts that the definition of fair strings is sensible.

Proposition 5 *A string is fair iff some associated sequence of environment states is fair.*

Note that if the environment is always fair, then only fair strings of messages are possible but there still may be an unfair execution of an environment for a fair string.

Proposition 6 *For any finite fair string, the final state in any associated sequence of environment states will be quiescent.*

We need to capture the idea that the set of tags used in a string is arbitrary: they serve only to mark operators. Thus we introduce the concept of string isomorphism.

Definition 15 *Two strings are* isomorphic *iff there is a one to one mapping between the sets of tags used in each string such that transforming all the tags in one string by that mapping will give the other string.*

Two sets of strings are isomorphic *iff there is a one to one mapping between the sets of tags used in each set such that transforming all the tags in a string in one set will give a string in the other set.*

The *raison d'être* of the planning process is the sequence of world states that is induced by a plan. Thus we also consider the associated set of world states without regard to the actual moment of message exchange. This enables us to consider hierarchies of actions, since it reduces the execution of a plan to a sequence of event executions, which is exactly the definition of an action execution.

Since we are interested in the failure of events, we must include τ in the world state. Thus we now consider the set of world states to be the set of tuples

$$W \times \{\text{fail}, \text{succeed}\}$$

Note that an event, which has been given as a (state transition, correctness condition) pair could be given as a single mapping on this extended world state. We choose not to do this because we will later restrict the class of mappings for events, and it is convenient to have the mapping separate from the correctness condition.

Definition 16 *Let a string be given. A sequence of world states is* associated *with the string if it can be obtained from an associated sequence of string execution states ξ by the following algorithm:*

1. *Remove all* ξ_n *from the sequence for which the strings in* ξ_n *and* ξ_{n+1} *are different.*
2. *For every* $\xi_n = (<\omega, E, \tau>, \sigma)$ *in this reduced sequence, take the pair* (ω, τ).

When we come to define agents, we will be interested in the sets of message strings they could exchange with an environment, and the resulting sets of sequences of world states. Thus the following definition will be useful.

Definition 17 *Two sets of strings are* equivalent for an environment *iff every sequence of world states associated with a string in one set is also associated with a string in the other set. They are* equivalent *if they are equivalent for every environment that defines the operators used.*

Two sets of strings are totally equivalent for an environment *iff every sequence of string execution states associated with a string in one set is the same as a sequence of string executions states associated with a string in the other set except for the tags. They are* totally equivalent *if they are totally equivalent for every environment that defines the operators used.*

Proposition 7 *Isomorphism implies total equivalence. Total equivalence implies equivalence.*

We will attempt to characterize these relations.

Definition 18 *Given two strings* σ *and* σ', σ' *is a* refinement *of* σ *iff*

- *If* $m > n$, *and* $\sigma'_n = \sigma'_m =$ (begin α x), *and there are more* (begin α x) *than* (end α x) *in every prefix of the string* $(\sigma'_n, \sigma'_{n+1}, \ldots, \sigma'_m)$, *then there is some* y *such that* $\sigma_n = \sigma_m =$ (begin α y).
- *If* $m > n$, *and* $\sigma'_n =$ (begin α x), *and* $\sigma'_m =$ (end α x), *and there are more* (begin α x) *than* (end α x) *in every prefix of the string* $(\sigma'_n, \sigma'_{n+1}, \ldots, \sigma'_{m-1})$, *then there is some* y *such that* $\sigma_n =$ (begin α y) *and* $\sigma_m =$ (end α y).

Intuitively, a string is a refinement if it is a restriction on the ways operators can overlap.

Definition 19 *A string* σ *is* specific *if it is a refinement of every string that is itself a refinement of* σ.

Intuitively, a specific string defines a unique overlapping of operators. We can use these definitions to find an alternative definition of total equivalence.

Proposition 8 *Two strings are totally equivalent iff every specific string that is a refinement of one is a refinement of the other.*

Two sets of strings are totally equivalent iff every specific string that is a refinement of some string in one set is also a refinement of some string in the other set.

Now we go on to find an alternative definition for equivalence. Given a string, we define a partial order on instances of identifiers. Intuitively, this expresses the necessary order on identifiers when they are identified with arbitrary instants somewhere between matching begin and end messages.

Definition 20 *Let $\mathcal{N}$ be the set of natural numbers: integers greater than 0. Let A_ι be the set of identifiers used in some reasonable string σ, and θ be some symbol which is never used as an identifier. Let $A'_\iota = A_\iota \cup \{\theta\}$. There is a partial order on $\mathcal{N} \times A'_\iota$ corresponding to σ given as follows:*

If $\alpha, \beta \in A_\iota$, then $(m, \alpha) < (n, \beta)$ iff the mth occurrence of (end α) *in σ comes before the nth occurrence of* (begin β).

If $\alpha \in A_\iota$, and there is an nth occurrence of (begin α), *then $(1, \theta) < (n, \alpha)$.*

If $\alpha \in A_\iota$, and there is an nth occurrence of (end α), *then $(n, \alpha) < (2, \theta)$.*

Such a partial order can be considered to be a set of pairs. Thus there is a subset relation between the orders.

Definition 21 *Let σ_1 and σ_2 be two strings. Then σ_1 is* faster *than σ_2 iff the partial order corresponding to σ_2 is a subset of that corresponding to σ_1.*

Intuitively, a string σ_1 is faster than a string σ_2 if σ_1 can be formed from σ_2 by moving pairs of *(begin α)* and *(end α)* closer together in the sequence. This corresponds to delaying the start of an action and also requiring it to complete sooner. This corresponds to the case where an action may delay a while before commencing, and then execute faster to complete earlier than it did in the first instance.

The following is the key result for finding equivalent sets of strings.

Proposition 9 *σ_1 is faster than σ_2 iff for every environment the set of associated sequences of world states for σ_1 is a subset of the set for σ_2.*

As a corollary to the above we get the required characterization of equivalence.

Definition 22 *The* asynchronous closure *of a set of strings is the set of all strings faster than some string in the original set.*

Proposition 10 *Two sets of strings are equivalent iff their asynchronous closures are totally equivalent.*

We choose the term *asynchronous closure* because it reflects the fact that a faster string may be obtained from a given string by delaying the start of actions to the last possible moment and allowing them to complete arbitrarily quickly, instead of insisting on redundant synchronization of the moments of message exchange.

We will also be interested in sequences of world states that fail at some stage. The following definition is useful.

Definition 23 *A string is* safe *for an environment iff no associated sequence of world states ever contains a world state with a* fail *status.*

2.3 Agents

We have already stated that an agent is some object that exchanges messages with an environment. It could thus be characterized by the set of strings it will exchange with an environment.

It is convenient to give an agent some additional structure, and to characterize the classes of agents able to be represented by plans. The usual means of defining objects that accept sets of strings is by computation theory – automata or Turing machines. We will represent an agent as a nondeterministic finite automaton[9].

Definition 24 *An* agent *is a labeled directed graph defined by a 6-tuple* $<A, \iota, N, I, F, t>$. *A is a set of operators, ι is a set of tags, N is a finite set of nodes, I is some element of N called the* initial node, *F is some subset of N called* final nodes, *and t is a set of arcs between nodes labeled with an identifier and one of the words* begin *or* end. *That is*

$$t \subseteq N \times N \times \{\text{begin}, \text{end}\} \times A_\iota$$

The first node of an arc is called the source; *the second is called the* destination.

An agent is always in some state represented by a node, and interacts with an environment by the exchange of identifiers. The term *agent state* will be used interchangeably with

current node; where there is no confusion, simply *state* is used. As it sends and receives messages, the agent changes state.

We will normally consider an arc in an agent to be a triple, consisting of two nodes and a label from the set $\{begin, end\} \times A_\iota$.

If there is an arc labeled *(begin α)* which has the current state as its source, then the agent may send the identifier α as output and the destination of the arc becomes the current state. If there is an arc labeled *(end α)* which has the current state as its source, and the identifier α is received as input, then the destination of the arc becomes the current state. Where there are several such arcs, one is selected nondeterministically. If an identifier is received as input and there is no arc from the current node labeled *(end α)*, then the agent is said to *block*.

The initial state of the agent is the node I. If the state of the agent is ever in the set F, the agent may *terminate*. Once an agent has terminated, it makes no further changes of state, sends no further messages, and ignores any messages it receives.

We insist that an active agent will not pause indefinitely as long it has the option of changing state. That is:

- An agent will not pause indefinitely at a nonfinal node with an outgoing arc labeled with a *begin* message.
- An agent will not pause indefinitely at a final node without terminating.

2.3.1 Classes of Agents

In this section, we define formally the operation of an agent and find particular interesting classes of agents. An agent will induce sets of sequences of world states, and we characterize classes of agents that induce the same set of world state sequences. We define agent termination and deadlock. We will be interested only in certain types of agent: *complete* agents, which will always accept an *end* message from the environment, and *bounded* agents, which correspond to a finite set of real agents. We define a class of *regular* agents, and claim that any bounded complete agent is effectively equivalent to some regular agent. This section may be skipped by those not interested in that level of detail.

The operation of an agent in conjunction with an environment may be defined more formally with the concept of an *agent execution state* in the same way as we used a string execution state to define the operation of an environment.

Definition 25 *An* agent execution state *for some given environment and agent with a common set of operators is an (environment state, agent state) pair, where the tags used in the environment state are those given for the agent.*

We define a successor relation on agent execution states.

Definition 26 *Let* $< W, A, i >$ *be an environment and* $< A, \iota, N, I, F, t >$ *be an agent. Let the agent execution state* $(< \omega, E, \tau >, \eta)$ *be given. Then* $(< \omega', E', \tau' >, \eta')$ *is a* successor *if one of the following conditions holds:*

- $\eta = \eta'$ *and* $< \omega', E', \tau' >$ *is a successor of* $< \omega, E, \tau >$.
- $\exists \rho \ . \ (\eta, \eta', \rho) \in t$ *and* $< \omega', E', \tau' > \in \rho(< \omega, E, \tau >)$.
  ρ *is of the form* (begin α) *or* (end α), *for some* $\alpha \in A_\iota$.

Definition 27 *Given an environment* $< W, A, i >$ *and an agent* $< A, \iota, N, I, F, t >$, *a sequence of agent execution states* ξ *is an associated sequence for the agent if the following properties hold:*

- $\xi_1 = (< \omega, \phi, \text{succeed} >, I)$ *for some* $\omega \in W$.
- ξ_n *is a successor of* ξ_{n-1} *for every n from 2 to the end of the sequence.*

Each agent execution is associated with some string of communications between agent and environment in the obvious way.

Definition 28 *For any agent execution sequence* ξ, *the associated string is constructed in the following way. Let* $\xi_n = (< \omega_n, E_n, \tau_n >, \eta_n)$. *For every* ξ_n *for which the size of* E_n *is different from that of* E_{n-1}, *add a message to the string. If* E_n *has a new element* (α, ς), *then add* (begin α). *Otherwise* E_{n-1} *has an element* (α, ϕ) *missing from* E_n, *so add* (end α).

An agent is more than simply the set of associated strings, since there may be several different types of execution sequence.

Definition 29 *An agent* accepts *a string iff it corresponds to some finite or infinite path through the graph.*

Definition 30 *Given any agent* $< A, \iota, N, I, F, t >$, *and any environment* $< W, A, i >$

- *An agent* runs *on a string iff it is associated with some agent execution sequence.*
- *An agent* blocks *on a string iff it is associated with a finite agent execution sequence that ends in a state* $(<\omega, E, \tau>, \eta)$ *for which*

$$\exists(\alpha, \phi) \in E \,.\, \forall \eta' \in N \,.\, (\eta, \eta', \text{end}, \alpha) \notin t$$

- *An agent* terminates *on a string iff it is associated with a finite agent execution sequence that ends in a state* $(<\omega, E, \tau>, \eta)$ *for which* $\eta \in F$.
- *An agent* deadlocks *on a string iff it is associated with a finite agent execution sequence that ends in a state* $(<\omega, E, \tau>, \eta)$ *for which* $E = \phi$, *and*

$$\forall \eta' \in N, \alpha \in A_\iota \,.\, (\eta, \eta', \text{begin}, \alpha) \notin t$$

Since an agent is nondeterministic, these cases for a string are not mutually exclusive.

Proposition 11 *An agent* runs *on a string iff the agent accepts the string and it is reasonable.*

An agent terminates *on a string iff the string is reasonable and corresponds to a finite path through the graph from the initial node to some final node.*

An agent deadlocks *on a string iff the string is fair and corresponds to a finite path through the graph from the initial node to some nonfinal node, and there are no arcs from the last node labeled with a* begin *message.*

An agent blocks *on a string iff the string is reasonable and corresponds to a finite path through the graph from the initial node to some arbitrary node, and for some* $\alpha \in A_\iota$ *there are more* (begin α) *arcs than* (end α) *arcs on the path, and there is no arc from the last node labeled with* (end α).

Proposition 12 *Given an agent and environment, every finite agent execution sequence is either the proper prefix of another agent execution sequence, or else the agent blocks, deadlocks, or terminates on the associated string.*

Agents may be characterized by the strings that induce these various types of execution. Note these cases for strings do not depend on the environment chosen since they do not depend on world states. Intuitively, an agent runs on a string if the string corresponds to a possible sequence of messages with an environment. The agent will deadlock if it gets into

a nonfinal state from which it cannot make further transitions; it may terminate if it gets into a final state; and it may block if it gets into a state for which the environment might send a message the agent is not equipped to receive.

Definition 31 *Two agents are* isomorphic *if there is a one to one mapping between the sets of tags used in the reasonable strings accepted by the agents such that if the tags in the strings are replaced according to the mapping, then the sets of strings on which they run/deadlock/terminate/block are the same in each case.*

Note that isomorphism of agents does not correspond to isomorphism of finite automata, since we only consider the reasonable strings and we give weight to nonterminating paths through the agent.

We have already stated that a planner's concern is with the sequences of world states induced by an agent, where the world state includes the success or failure of events. It may also be concerned that the agent terminates cleanly in some sense, so we extend world states to include the termination status of an agent. Thus we will again extend the set of world states to be

$$W \times \{\text{fail}, \text{succeed}, \text{terminated}, \text{deadlocked}, \text{blocked}\}$$

Definition 32 *Let an agent and environment be given. A sequence of world states is* associated *with the agent if it can be obtained from an associated sequence of agent execution states ξ by the following algorithm. Let $\xi_n = (<\omega_n, E_n, \tau_n>, \eta_n)$.*

1. *Remove all ξ_n from the sequence for which the sizes of E_n and E_{n-1} are different.*
2. *For every ξ_n in this reduced sequence, take the pair (ω_n, τ_n).*
3. *If the original sequence ξ was finite and the last state ξ_z was $(<\omega_z, E_z, \tau_z>, \eta_z)$, then one may optionally add one more world state by one of the following rules:*
   - *If $\eta_z \in F$, then $(\omega_z, \text{terminated})$ may be added.*
   - *If $E_z = \phi$ and $\eta_z \notin F$, and there is no arc from η_z labeled* (begin α)*, then $(\omega_z, \text{deadlocked})$ may be added.*
   - *If $\exists(\alpha, \phi) \in E_z$ and there is no arc from η_z labeled* (end α)*, then $(\omega_z, \text{blocked})$ may be added.*

We define equivalence for agents as we did for sets of strings.

Definition 33 *Two agents are* equivalent for an environment *iff in conjunction with that environment they induce the same set of sequences of world states. They are* equivalent *if they are equivalent for any environment.*

Two agents are totally equivalent for an environment *iff in conjunction with that environment they induce the same set of sequences of agent execution states, except for the nodes and tags used. They are* totally equivalent *if they are totally equivalent for any environment.*

Proposition 13 *Two agents are* equivalent *iff the sets of strings on which they run/terminate/deadlock/block are equivalent.*

Two agents are totally equivalent *iff the sets of strings on which they run/terminate/deadlock/block are totally equivalent.*

Proposition 14 *Isomorphic agents are totally equivalent. Totally equivalent agents are equivalent.*

An agent has no control over when the environment will send a message for a completed action, and so should accept an *end* message at any time. In other words, we are not really interested in agents that may block, and agents that terminate before the environment is quiescent are at least suspect.

Definition 34 *An agent is* complete *iff there is no string on which it blocks. It is* fully complete *iff it is complete, and every string on which it terminates is fair.*

Proposition 15 *An agent is complete iff for any finite agent execution sequence ξ for an arbitrary environment for which the last state in the sequence is $\xi_n = (<\omega_n, E_n, \tau_n>, \eta_n)$, and for any $(\alpha_n, \varsigma_n) \in E_n$, there exists an agent execution sequence which is the same as the original with one state ξ_{n+1} concatenated, for which E_{n+1} is the same as E_n except that (α_n, ς_n) is replaced by $(\alpha_n, \mathrm{pop}(\varsigma_n))$, or is deleted if $\varsigma_n = \phi$.*

A complete agent is fully complete iff for any finite agent execution sequence ξ for an arbitrary environment for which the last state in the sequence is $\xi_n = (<\omega_n, E_n, \tau_n>, \eta_n)$, and $\eta_n \in F$, it follows that $E_n = \phi$.

Given any agent, we can find a *corresponding complete* agent that operates in the same way as the original, except that the new agent ignores messages that the first cannot accept.

Definition 35 *Given any agent, the* corresponding complete *agent is made by adding arcs as follows: take any node for which there is a reasonable path from the initial node to that node with more* (begin α) *than* (end α) *for some* $\alpha \in A_i$, *and for which there is no outgoing arc labeled* (end α). *Add an arc that is labeled* (end α) *and has that node as source and destination.*

Proposition 16 *Given any agent, the corresponding complete agent accepts all strings accepted by the original, blocks on no strings, deadlocks on all the strings deadlocked on by the original, and terminates on all the strings terminated on by the original.*

The following result gives an extended formal definition of the operation of an agent that makes an agent effectively the same as its corresponding complete agent; hence we need not consider any further agents that are not complete.

Proposition 17 *If we add a new rule to the definition of the successor relation on agent execution states:*

- $(<\omega', E', \tau'>, \eta')$ *is a successor of* $(<\omega, E, \tau>, \eta)$ *if*

$$\begin{aligned} &\eta = \eta',\ \omega = \omega',\ \tau = \tau' \text{ and} \\ &\exists \alpha \in A_i \ .\ \forall \eta'' \in N \ .\ (\eta, \eta'', \text{end}, \alpha) \notin t \text{ and} \\ &\qquad (<\omega', E', \tau'>, \eta') \in (\text{end } \alpha)(<\omega, E, \tau>, \eta) \end{aligned}$$

then the sets of associated agent execution states for an agent and its corresponding complete agent are the same.

In general, there is no way of making a *corresponding fully complete* agent. However, we can change the definition of agent execution so that an agent may not terminate unless the environment is quiescent. Formally, we would change the definition of *sequences of world states associated with an agent* by allowing $(\omega_z, \text{terminated})$ to be added to the end of a sequence iff the last state $(<\omega_z, E_z, \tau_z>, \eta_z)$ in the sequence of agent execution states was such that $\eta_z \in F$ and $E_z = \phi$.

The strings accepted by an agent are not necessarily reasonable, so we make the following definition.

Definition 36 *An agent is* consistent *iff every string it accepts is reasonable.*

A consistent agent is more clearly representative of agents in the real world in that every path through the agent corresponds to a string on which the agent will run.

The main result we will be interested in is finding the class of agents that can be represented by plans. Note that an agent makes no use of sensory input from the world model. This is a common assumption in planning systems. Agents do, however, "know" when an action ends, since they may change state on receiving a message that the action has terminated. However, the exact moment of completion is not known, since an environment does not return a message until the corresponding sequence has been completed at some earlier moment. Similarly, the environment may wait an arbitrary time after receiving a message before executing the first event. This lag corresponds to the intuition that, for any information-processing agent such as a person or a robot, there is a small but finite time between the decision to do some act and the moment when it begins to change the real world. Also, any input from the real world takes a moment to be recognized. This effect is clearly seen for a computer program that interacts with the world through communication channels to some I/O device. These lags will have some bearing in finding classes of agents that are equivalent.

Proposition 18 *Two complete agents are equivalent iff the asynchronous closures of the sets of strings on which they run/deadlock/terminate are totally equivalent.*

Definition 37 *A complete agent is* asynchronously closed *if the sets of strings on which it runs/deadlocks/terminates are all asynchronously closed.*

Proposition 19 *Any complete agent is equivalent to some asynchronously closed agent.*

The class of asynchronously closed agents is still too broad. We will be using agents to model the activity of some finite number of real agents in the real world. Thus there will be an upper bound on the number of actions the environment will be executing at any time.

Definition 38 *An agent is* bounded *iff there is a finite upper bound on the difference between the number of* (begin α) *arcs and the number of* (end α) *arcs on an arbitrary (finite or infinite) reasonable path through the graph and for an arbitrary* $\alpha \in A_t$.

Proposition 20 *Any complete consistent agent is bounded.*

Proposition 21 *A complete bounded agent is isomorphic to a consistent agent.*

Proposition 22 *An agent is bounded iff for any environment* $< P, A, i >$ *such that* $i(\alpha)$ *is finite for every* $\alpha \in A_\iota$*, the agent/environment machine is finite state.*

The definition of an agent allows it to terminate before the environment is quiescent. This implies that the terminated agent could receive messages from the environment that it will ignore. We have shown how to construct a corresponding complete agent which ignores messages that would otherwise cause it to block. If an agent is bounded, one can also construct a *corresponding fully complete* agent, which is exactly like the original, except that where the first may terminate on some unfair string, the corresponding fully complete agent will terminate only on any fair string that is formed from the first by adding only *end* messages.

Definition 39 *Given a complete agent a, an agent a' is a* corresponding fully complete *agent iff the sets of reasonable accepted strings are equal, the set of strings on which a and a' deadlock are equal, and the set of strings on which a' terminates is the set of all fair strings that can be formed by adding* end *messages to the strings on which a terminates.*

Proposition 23 *For any complete agent we can find a* corresponding fully complete *agent iff it is bounded.*

We present a class of agents which will be shown to correspond to plans, and then show the classes of agents equivalent and totally equivalent to such agents.

Definition 40 *An agent is* specific *if every reasonable string it accepts is specific.*

Definition 41 *An agent is* regular *iff*

- *it is fully complete.*
- *it is consistent.*
- *it is asynchronously closed.*
- *it is specific.*

Proposition 24 *An agent is asynchronously closed, bounded, and fully complete iff it is totally equivalent to some regular agent.*

Proposition 25 *An agent is complete and bounded iff it is equivalent to some regular agent.*

Deadlock will be associated with problems in a synchronization skeleton; therefore, we will be interested in agents that do not deadlock.

Definition 42 *An agent is* deadlock-free *iff there are no strings on which it deadlocks.*

Proposition 26 *A consistent agent is deadlock-free iff no nonfinal node with no outgoing arcs is reachable from the initial node.*

All the properties of agents discussed so far are independent of the environment, as long as all the necessary operators have interpretations. However, since the planner is concerned with sequences of world states and event failure, the interpretation of operators will be important.

Definition 43 *An agent is* safe *for an environment iff for every associated sequence of world states, the status is never* fail*: equivalently, iff every reasonable string it accepts is safe.*

When we modify plans so that they become safe and deadlock-free, there must be some relationship between the original and modified agents generated by the plans. We require that the new agent allow some subset of the sequences of world states allowed by the original.

Definition 44 *A complete agent a is* included *in another complete agent a' if the sets of strings on which a runs, terminates, and deadlocks are all subsets of the sets of strings on which a' runs, terminates, and deadlocks, respectively.*

Given an environment and a complete agent a, the maximal safe deadlock-free *agent a' is a complete safe deadlock-free agent included in a which includes every other complete safe deadlock-free agent included in a.*

Proposition 27 *Given an environment, every complete agent has a maximal safe deadlock-free agent that is unique up to isomorphism, or it includes no complete safe deadlock-free agent.*

In a similar way to that in which ϵ transitions are used in automata theory, an agent is permitted to be defined with arcs labeled with ϵ. This is a convenience. An agent with ϵ transitions is regarded as exactly the same as the agent without such transitions given by the following algorithm:

1. For any nodes n_1, n_2, n_3 such that there is a finite sequence of ϵ transitions leading from n_1 to n_2 and a single arc labeled ρ from n_2 to n_3, where ρ is not ϵ, add an arc labeled ρ from n_1 to n_3.

2. For any node such that there is a finite sequence of ϵ transitions leading to a final node, add that node to the set of final nodes.

3. Delete all ϵ transitions, and delete all nodes that are not reachable from the initial node.

An agent with ϵ transitions may be considered to operate in the following manner.

If there is a path from the current node consisting of some initial (possibly empty) sequence of ϵ arcs followed by a *(begin α)* arc, then the agent may send α as output and the end of the path becomes the current node. If the agent receives α as input, and there is a path from the current node consisting of some initial (possibly empty) sequence of ϵ arcs followed by an *(end α)* arc, then the end of one of those paths becomes the current node. Otherwise if an input of α is received, the agent blocks.

If the agent is ever in a state where there is some (possibly empty) sequence of ϵ arcs to a final node, then it may terminate.

2.4 Plan syntax

A *plan* is some description of an agent. The planning problem becomes that of finding a plan that denotes an agent that will induce some suitable sequence of changes in the world model, given information about the environment.

Given three symbol sets A, M and S being operators, memory states, and signals, respectively; plans can be defined recursively.

- For any $\alpha \in A$, α is a plan.
- For any $m \in M$, $s \in S$, (set m), (send s), and (guard m s) are plans.
- Λ is a plan. Λ is a unique symbol not in A.
- If p_1 and p_2 are two plans, then $p_1;p_2$ is a plan.
- If p_1 and p_2 are two plans, then $p_1 \parallel p_2$ is a plan.

- If p_1 and p_2 are two plans, then $p_1 \mid p_2$ is a plan.
- If p_1 is a plan, then $p_1^\times$ is a plan.

It is assumed that devices exist for defining a unique parse of a plan, such as brackets or operator precedence rules. The actual technique is not important.

The intended meanings of the various plan components are:

- ; is the sequence construct.
- $\|$ is the parallel construct.
- $\mid$ is for nondeterministic selection.
- $*$ is for loops.
- (send s), (set m), and (guard m s) are synchronizing primitives.
- α where $\alpha \in A$ is the command to execute an action.
- Λ is the null plan.

2.5 Plan Semantics

The semantics for plans is given as a mapping from plans to *agents*. The agent is said to be an *interpretation* for the plan. A plan may have several interpretations, all of which are isomorphic to each other. Thus the mapping is actually a mapping from a plan to a class of isomorphic agents.

We present two alternative definitions of this mapping, and then show that they are equivalent.

The first definition of the mapping proceeds in two stages. First, a mapping to a graph is defined that includes arcs labeled with memory states and signals as well as identifiers. Then a mapping is given from this graph to a graph that conforms to the definition of an agent.

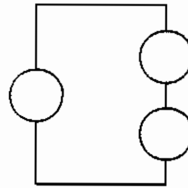
Let the sets A, M, and S be given. Let ι be an arbitrary set of tags. Then p is a function from the set of plans to the set of graphs defined by a 4-tuple $<N, I, F, t>$, where N is a finite set of nodes, I is an initial node, F is a nonempty set of final nodes, and t is a set of labeled arcs between nodes.

The labels may be

- (begin α) or (end α) where $\alpha \in A_\iota$.
- (send s), (set m), or (guard m s) where $m \in M, s \in S$.
- the symbol ϵ.

p is defined recursively.

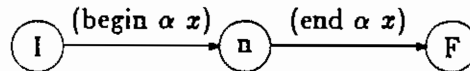
We now list formal definitions and intuitive pictorial representations of p for the syntactic elements. In the pictorial representation for combinations of subplans, the agent for a subplan is drawn as



The node on the left is the initial node, the nodes on the right are the final nodes. The initial and final nodes of the total plan are labeled I and F, respectively.

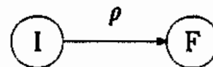
- For the plan α where $\alpha \in A$, choose an arbitrary $x \in \iota$. $p(\alpha)$ is the graph

$$< \{I, n, F\}, I, \{F\}, \{(I, n, (\text{begin } \alpha\ x)), (n, F, (\text{end } \alpha\ x))\} >$$



- For the plan ρ, where ρ is (send s), (set m), or (guard m s) for $m \in M, s \in S$, $p(\rho)$ is the graph

$$< \{I, F\}, I, \{F\}, \{(I, F, \rho)\} >$$



- For the plan Λ, $p(\Lambda)$ is the graph

$$< \{n\}, n, \{n\}, \phi >$$

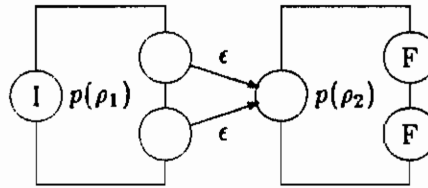
IF

- Suppose graphs for two plans are given as

$$p(\rho_1) =< N_1, I_1, F_1, t_1 >$$
$$p(\rho_2) =< N_2, I_2, F_2, t_2 >$$

Assume that the sets N_1, N_2 are disjoint. Then the interpretation for $\rho_1;\rho_2$ is

$$\begin{aligned} p(\rho_1;\rho_2) =< & N_1 \cup N_2, \\ & I_1, \\ & F_2, \\ & t_1 \cup t_2 \cup \{(f, I_2, \epsilon) : f \in F_1\} > \end{aligned}$$



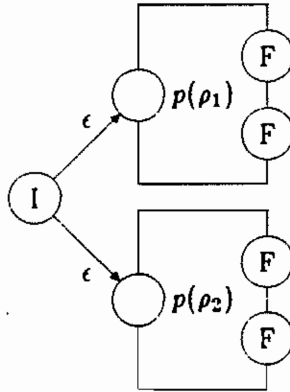
- Suppose interpretations for two plans are given as

$$p(\rho_1) =< N_1, I_1, F_1, t_1 >$$
$$p(\rho_2) =< N_2, I_2, F_2, t_2 >$$

Assume that the sets N_1, N_2 are disjoint. Then the interpretation for $\rho_1 \mid \rho_2$ is

$$\begin{aligned} p(\rho_1 \mid \rho_2) =< & N_1 \cup N_2 \cup n, \\ & n, \\ & F_1 \cup F_2, \\ & t_1 \cup t_2 \cup \{(n, I_1, \epsilon), (n, I_2, \epsilon)\} > \end{aligned}$$

where n is a new node not in N_1 or N_2.

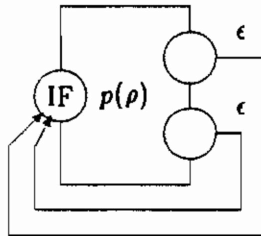


- Suppose an interpretation for a plan is given as

$$p(\rho) = < N_1, I_1, F_1, t_1 >$$

Then the interpretation for $\rho^\times$ is

$$\begin{aligned} p(\rho^\times) = <&N_1, \\ &I_1, \\ &\{I_1\}, \\ &t \cup \{(f, I_1, \epsilon) : f \in F_1\} > \end{aligned}$$



- Suppose interpretations for two plans are given as

$$p(\rho_1) = < N_1, I_1, F_1, t_1 >$$
$$p(\rho_2) = < N_2, I_2, F_2, t_2 >$$

with no identifier common to t_1 and t_2. Then the interpretation for $\rho_1 \parallel \rho_2$ is

$$\begin{aligned} p(\rho_1 \parallel \rho_2) = <&N_1 \times N_2, \\ &(I_1, I_2), \\ &F_1 \times F_2 \\ &\ \{((n_{11}, n_2), (n_{12}, n_2), \rho) : n_2 \in N_2, (n_{11}, n_{12}, \rho) \in t_1\} \\ &\cup \{((n_1, n_{21}), (n_1, n_{22}), \rho) : n_1 \in N_1, (n_{21}, n_{22}, \rho) \in t_2\} > \end{aligned}$$

This is a standard interleaving model of concurrency, and cannot be represented pictorially in the same way as the previous constructs.

Now let q be a mapping from the range of p to the set of agents for the operators A. q will be defined by an algorithm for removing arcs labeled with anything that is not ϵ, or *(begin α)* or *(end α)* for $\alpha \in A_i$.

A new memory state is introduced, $\lambda \notin M$. The set of memory states augmented with λ is referred to as $M' = M \uplus \{\lambda\}$. Let *memarc* be a predicate on the labels of arcs which is true iff the label is of the form *(set m)* or *(guard m s)*.

First, given a graph $< N, I, F, t >$, expand it into the graph

$$
\begin{aligned}
&< N \times M', \\
&(I, \lambda), \\
&\{(f, m) : f \in F, m \in M'\}, \\
&\quad \{((n_1, m), (n_2, m), \rho) : (n_1, n_2, \rho) \in t, m \in M', \neg \text{memarc}(\rho)\} \\
&\cup \{((n_1, m), (n_2, m), (\text{guard } m\ s)) : (n_1, n_2, (\text{guard } m\ s) \in t\} \\
&\cup \{((n_1, m_1), (n_2, m_2), \epsilon) : (n_1, n_2, (\text{set } m_2)) \in t, m_1 \in M', m_2 \in M\} >
\end{aligned}
$$

Any subgraph of the whole that is not connected to the initial node by some sequence of arcs may be deleted.

This first step models the memory as part of the agent state. The memory remains unaltered by transitions that do not *set* the state. A *guard* arc may be taken only if the memory is in the appropriate state. The information contained in the *set* arc is now contained in the new memory state, and so the distinguishing label is thrown away.

The *guard* arcs are also allowed to be taken only in conjunction with an appropriate *send* arc, thus providing a means for synchronization. This is ensured in the second stage of the algorithm.

Let the new graph produced by this first stage of the algorithm be given. For any quadruple of arcs of the form

$$
\begin{aligned}
&((n_1, m), (n_2, m), (\text{send } s)) \\
&((n_2, m), (n_4, m), (\text{guard } m\ s)) \\
&((n_1, m), (n_3, m), (\text{guard } m\ s)) \\
&((n_3, m), (n_4, m), (\text{send } s))
\end{aligned}
$$

where $s \in S$ and $m \in M$ is common to all four, add an arc of the form

$$((n_1, m), (n_4, m), \epsilon)$$

Then delete all arcs labeled with a *guard* or *send*.

This operation ensures that a send is immediately followed by an appropriate guard. Again, sections of the graph not connected to the initial node by some sequence of arcs may be deleted. Let the resulting graph be $< N, I, F, t >$.

Now choose an arbitrary ι that contains all the tags used in t, and $< A, \iota, N, I, F, t >$ corresponds to the definition of an agent. This is the result of the mapping q. The complete function mapping plans to agents, and thus defining the semantics of a plan, is the composition of p and q.

Proposition 28 *Any agent that is given as the semantics of a plan is regular.*

This definition of a plan semantics is suitable for showing how plans are combined into larger plans, but is a little unintuitive. An equivalent definition is given which corresponds more closely to the idea that the semantics of a plan is defined by some agent that *executes* the plan like a program, and that the states of the agent correspond to some kind of *program location.*

Let A, M, and S be given as before, and let ρ be some plan. Again, we augment M with an initial memory state λ.

The set of states of the agent incorporate a *program location* and a memory state. The program location is an assignment of one of *todo, doing, done,* and *quiet* to every subplan of the plan ρ. More formally, if P is the set of all subplans of ρ, then the set of nodes used in the agent is a subset of

$$(\{\text{todo}, \text{doing}, \text{done}, \text{quiet}\}^{P}) \times M'$$

The intended meaning of the assignment to subplans is that a subplan may either be 'about to be executed', 'in the process of execution', 'just completed', or 'not considered for the moment'. The initial state is that in which *todo* is assigned to the plan as a whole, *quiet* is assigned to every proper subplan, and the memory state is λ. The final states are those in which *done* is assigned to the plan as a whole, and *quiet* is assigned to every proper subplan. The memory state is arbitrary.

The nodes in the agent are described using a simple syntax for the program location.

Definition 45 *Let Z be the set* {todo, doing, done, quiet}. *A* program location *is defined recursively by the following rules:*

- *$e(\rho)$ is a program location for any $e \in Z$, and for any synchronization operation ρ.*
- *$e(\alpha, x)$ is a program location for any operator α and any tag x selected from an arbitrary set ι.*
- *$e(\Lambda)$ is a program location for any $e \in Z$.*
- *$e(\varrho_1; \varrho_2)$, $e(\varrho_1 \mid \varrho_2)$, and $e(\varrho_1^{\times})$ are program locations if $e \in Z$ and ϱ_1, ϱ_2 are program locations.*
- *$e(\varrho_1 \parallel \varrho_2)$ is a program location if $e \in Z$ and ϱ_1, ϱ_2 are program locations, and there are no identifiers common to ϱ_1 and ϱ_2.*

The interpretation for a sentence in this language is a mapping from Z to all the subplans of some plan, and should be intuitive. Nodes may be considered to be marked with such a syntactic program location and a memory state from M'.

Let a plan ρ be given. Then find a corresponding program location ϱ that defines a mapping from Z to all the subplans of ρ. Let ι be some set of tags containing all the tags used in ϱ. By implication, ϱ also replaces all operators in the plan with identifiers in such a way that no identifier is common to distinct branches of a parallel subplan.

For the following table, the first column gives a possible subprogram location that may appear anywhere within the program location at a node. ϱ_1 and ϱ_2 are arbitrary program locations. Where such a subprogram location exists, add an arc labeled ϵ to the node with the subprogram location replaced by the program location in the second column, and with the memory state unchanged.

todo(quiet(ϱ_1); quiet(ϱ_2))	doing(todo(ϱ_1); quiet(ϱ_2))
doing(done(ϱ_1); quiet(ϱ_2))	doing(quiet(ϱ_1); todo(ϱ_2))
doing(quiet(ϱ_1); done(ϱ_2))	done(quiet(ϱ_1); quiet(ϱ_2))
todo(quiet(ϱ_1) $\mid$ quiet(ϱ_2))	doing(todo(ϱ_1) $\mid$ quiet(ϱ_2))
todo(quiet(ϱ_1) $\mid$ quiet(ϱ_2))	doing(quiet(ϱ_1) $\mid$ todo(ϱ_2))
doing(done(ϱ_1) $\mid$ quiet(ϱ_2))	done(quiet(ϱ_1) $\mid$ quiet(ϱ_2))
doing(quiet(ϱ_1) $\mid$ done(ϱ_2))	done(quiet(ϱ_1) $\mid$ quiet(ϱ_2))
todo(quiet(ϱ_1)$^\times$)	done(quiet(ϱ_1)$^\times$)
todo(quiet(ϱ_1)$^\times$)	doing(todo(ϱ_1)$^\times$)
doing(done(ϱ_1)$^\times$)	todo(quiet(ϱ_1)$^\times$)
todo(quiet(ϱ_1) $\|$ quiet(ϱ_2))	doing(todo(ϱ_1) $\|$ todo(ϱ_2))
doing(done(ϱ_1) $\|$ done(ϱ_2))	done(quiet(ϱ_1) $\|$ quiet(ϱ_2))
todo(Λ)	done(Λ)

The intuitive meaning of these arcs is as follows:

- To commence two subplans in sequence, begin the first subplan.
- When the first has finished, it becomes quiet and the second begins.
- When the second has finished, the whole is completed.
- To commence the nondeterministic selection of two subplans, either begin the first
- or begin the second.
- In either case, when one has finished, the whole is completed.
- To commence a loop, either do nothing and consider the loop complete
- or commence the subplan encompassed by the loop.
- When the subplan encompassed by a loop is complete, recommence the loop.
- To commence two subplans in parallel, commence them both at once.
- When both are complete, the whole is finished.
- The null plan can be considered done with no effort.

To handle operators from A:

If a node has a subprogram location of the form *todo(α)* for some $\alpha \in A_\iota$, add an arc labeled *(begin α)* to the node with that subprogram location replaced by *doing(α)* and with the memory state unchanged.

If a node has a subprogram location of the form *doing(α)* for some $\alpha \in A_\iota$, add an arc labeled *(end α)* to the node with that subprogram location replaced by *done(α)* and with the memory state unchanged.

To handle the synchronization operations:

If a node has a subprogram location of the form *todo((set m))* where $m \in M$, and an arbitrary memory state, then add an arc labeled ϵ to the node where the subprogram location is *done((set m))*, and the memory state is m.

If a node with memory state $m \in M$ has a program location with, in one place, a subprogram location of the form *todo((send s))* and, in another place, one of the form *todo((guard m s))* where $s \in S$, then add an arc labeled ϵ to the node where the memory state is unchanged, the first subprogram location is replaced by *done((send s))*, and the second is replaced by *done((guard m s))*.

This shows more clearly the intended purpose of the synchronizing primitives. A *guard* or *send* can be executed only simultaneously with a *send* or *guard*, respectively, and then only when the program is in the correct state.

This particular form of primitive is derived from the parallel programming language CSP[8], which uses a form of guarded selection, where a guard may be a combination of an input/output operation and a normal conditional. Input/output operations are constrained to occur simultaneously.

Proposition 29 *Given any plan, any two agents formed by either of these two different ways are isomorphic.*

Proposition 30 *For any arbitrary regular agent, there is a plan that has an interpretation isomorphic to that agent.*

This last theorem, together with some results from the previous section, asserts that there is a one to one correspondence between the equivalence classes (under isomorphism) of regular agents and the equivalence classes of agents that are interpretations for plans. Also, there is a one to one correspondence between the equivalence classes (under total

equivalence) of bounded complete agents and the equivalence classes of agents that are interpretations for plans.

Note that this semantics for a plan ultimately replaces synchronization operations with ϵ arcs. Intuitively, this allows a plan to backtrack over synchronization operations but not action operators. Although there is an alternative definition that prevents this backtrack, there are three reasons why we don't use it:

- It is more complex.
- Both interpretations cover the same classes of agents.
- The effect of the second interpretation is to allow the agent to deadlock on more strings - other sets are unchanged. But if the first interpretation is deadlock-free, so is the second.

2.6 Extensions to the Syntax

Certain useful syntactic constructs may be added, which have an intended meaning that does not extend the class of agents representing plans. The meaning may be defined either by extending the definition of an interpretation, or by giving a transformation on plans that reduces an extended plan to the original syntax. This section is peripheral to the main thrust of the paper.

One particular extension that we will find useful is the concept of a *plan variable.* We have represented memory as a single state. It would be more usual to have memory as a set of variables.

The definition of plan syntax is extended as follows: replace the set of memory states M by a set of variables V and a set of variable states D. The definition of the primitives *guard* and *set* is extended to be *(guard v d s) and (set v d)* for $v \in V, d \in D, s \in S$.

The intuitive meaning of these plans should be clear. *Set* sets a single variable to a single state, and *guard* receives a signal only if a particular variable is in a particular state. The meaning is defined formally by conversion of syntax to the original form.

Let ρ be a plan in the extended form, with operators A, signals S, variables V, and variable states D. V and D, of course, are finite. The semantics for ρ is given by a plan ρ' in the original form. The set M used in ρ' is the set of mappings from variables to variable

states, D^V. The set S' used in ρ' is $S \cup \{x, x'\}$, where x and x' are unique signals not in S. The set of operators is unchanged.

In a similar way that a special initial memory state is assumed for plans, we will assume a special initial variable state. Thus let λ be some variable state not in D. Let $D' = D \cup \{\lambda\}$.

We note that the "|" nondeterministic selection operator is reflexive and associative; therefore, we will allow it to be applied to a set of subplans, with the obvious interpretation.

Every *(guard v d s)* in ρ is replaced by the nondeterministic selection of all primitives *(guard m s)*, where s is unchanged, and m is taken from the set

$$\{m \in D'^V : m(v) = d\}$$

Every *(set v d)* in ρ is replaced by the nondeterministic selection of all subplans of the form

$$\text{(guard } m\ x\text{); (set } m'\text{); (guard } m'\ x'\text{)}$$

where x and x' are the special signals, and (m, m') is taken from the set

$$\{(m, m') \in D'^V \times D'^V : m'(v) = d, \forall v' \in V \backslash \{v\} \ .\ m'(v') = m(v')\}$$

Finally, the whole plan is run in parallel with the subplan

$$((\text{send } x); (\text{send } x'))^{\times}$$

and the subplan *(set m)* is put before the entire plan with the sequence operator, where m is the mapping $\forall v \in V \ .\ m(v) = \lambda$. The two special signals ensure that variables do not interfere with each other.

The semantics for this plan captures the intuitive meaning of variables.

An alternative definition of semantics would be to give a mapping from extended plans directly to agents. The mappings given for plans in the previous section are easily extended.

Let A be a set of operators. Let a plan ρ be given in the extended syntax, with S, V, and D being the signals, variables and variable states respectively. Let D' be D augmented with λ.

Consider the first definition of semantics for plans in the original syntax. This is given as the composition of two mappings $p \circ q$. The mapping p is essentially unchanged, if M is regarded as D'^V. Suppose that $p(\rho)$ is the graph $< N, f, F, t >$.

The second mapping q is modified slightly. No new memory state is needed; therefore, M', the set of memory states in $q(p(\rho))$, is the same as in $p(\rho)$. Thus the set of nodes used is $N \times D'^V$. The initial node is (I, λ'), where I is the initial node from the mapping p, and λ' is the mapping that maps all variables onto the special state λ. Thus q begins by transforming $p(\rho)$ into the graph

$$
\begin{aligned}
&<N \times D'^V, \\
&(I, \lambda'), \\
&\{(f, m) : f \in F, m \in D'^V\}, \\
&\quad \{((n_1, m), (n_2, m), \rho) : (n_1, n_2, \rho) \in t, m \in D'^V, \neg\text{memarc}(\rho)\} \\
&\cup \{((n_1, m), (n_2, m), (\text{guard } v\ d\ s)) : m(v) = d \text{ and} \\
&\qquad\qquad\qquad\qquad (n_1, n_2, (\text{guard } v\ d\ s) \in t\} \\
&\cup \{((n_1, m_1), (n_2, m_2), \epsilon) : (n_1, n_2, (\text{set } v\ d)) \in t, \\
&\qquad\qquad\qquad m_1, m_2 \in D'^V, m_2(v) = d \\
&\qquad\qquad\qquad \forall x \in V \backslash \{v\} \,.\, m_2(x) = m_1(x)\} >
\end{aligned}
$$

The remainder of the mapping q remains essentially the same.

The second definition of plan semantics may be modified in the same way.

Proposition 31 *The interpretations of an extended plan obtained by taking the extended definition of semantics, and by taking the interpretation of the plan obtained by transforming the extended plan, are isomorphic.*

3 The Interaction Problem

Given the theory of plans and action, the problem addressed here is that of ensuring that a plan does not induce deadlock or allow any event to fail. We will identify a plan with its associated agent, and thus use terms previously associated with agents. Thus a plan is *safe* if its associated agent is *safe*, and so on. Given a plan and an environment, we are interested in finding the *maximal safe deadlock-free* plan, if it exists. This is a plan that allows all and only the execution sequences of the original plan for which no event can ever fail. A program has been written which does exactly that for a restricted class of plans in a restricted class of environments, and a second version is under development which will handle any plan and a larger class of environments. This section provides some more

background to the plans and environments used in the second version of the program. The program is called a *plan synchronizer*, for reasons that will become apparent.

3.1 Events and Event Failure

So far, the world has been considered to be some set of world states. We have said that world states could be described by having a world state be an interpretation for some logical language. As we come to provide descriptions of an environment to the plan synchronizer, we will need to specify the language used to describe world states. We use a simple propositional language. There will be some finite set of ***propositions***, and a world state corresponds to the set of propositions true in that state. Sets of world states will be represented as formulas made up of propositions and boolean connectives. The connectives will be $\wedge, \vee, \neg, \supset, \Leftrightarrow$, with the usual interpretations. We will make common use of atomic formulas.

Definition 46 *Given a set of propositions P, $P^{\pm}$ is the set of* atomic formulas *for P.*

$$P^{\pm} = \{p, \neg p : p \in P\}$$

For this program we use a very simple form of event, corresponding to the operators of the STRIPS planner[4]. Events are constrained to add or delete propositions from the world model without reference to the current world state. Also, the correctness condition is a conjunction of propositions or negated propositions. Thus an event is four sets of propositions: an *add* set, a *delete* set, a *require true* set, and a *require false* set.

We will refer to an environment with this type of event as a *restricted propositional environment.*

3.1.1 Restricted Propositional Environments

In this section we formally define restricted propositional environments and claim that all previous results still hold if these are the only environments considered. We show that actions may be characterized by five sets of atomic formulas, which are the formulas that the action asserts, retracts, conflicts, requires, or maintains, respectively, in order to find safe agents. We find simple rules that are necessary and sufficient for an agent to be safe. This section may be skipped by those not interested in that level of detail.

Definition 47 *Given a finite set P of propositions, a* restricted propositional event *is an event for the set of worlds 2^P which can be represented as a 4-tuple of sets of propositions*

$$\text{rpevents}(P) = (2^P)^4$$

For such a 4-tuple $(\pi_1, \pi_2, \pi_3, \pi_4)$, the corresponding (mapping, correctness condition) pair (δ, γ) is defined to be

$$\delta(\pi) = (\pi \cup \pi_1)\backslash\pi_2$$

$$\gamma = \{\pi : \pi_3 \subseteq \pi \text{ and } \pi \cap \pi_4 = \phi\}$$

Definition 48 *An environment $< W, A, i >$ is a* restricted propositional environment *if there is a finite set of propositions P such that $W = 2^P$ and every event used in the range of i is a restricted propositional event for that set of propositions.*

We may assume that for an event $(\pi_1, \pi_2, \pi_3, \pi_4)$, the set pairs (π_1, π_2) and (π_3, π_4) are disjoint, since if there is a proposition in π_3 and π_4 the event will always fail, and any proposition in π_2 is redundant in π_1. An event will also be represented as a pair of sets of atomic formulas (ν_1, ν_2), where

$$\nu_1 = \{p : p \in \pi_1\} \cup \{\neg p : p \in \pi_2\}$$
$$\nu_2 = \{p : p \in \pi_3\} \cup \{\neg p : p \in \pi_4\}$$

Where there is no confusion, we will refer to restricted propositional environments simply as environments, and present them as a 3-tuple $< P, A, i >$, where P is a finite set of propositions, A is a finite set of operators, and i maps each operator onto a set of finite sequences of pairs of sets of atomic formulas.

$$i : A \mapsto 2^{(((2^{P^{\pm}})^2)^*)}$$

There is one very useful attribute of this class of environments. None of the proofs of the theorems so far presented in this paper rely on the fact that environments are not restricted to this class.

Proposition 32 *If the original definition of environment were changed to be restricted propositional environments, all the results so far presented still would hold true.*

There is a class of actions that will invariably allow an agent to fail: those actions that require conflicting conditions to be true without asserting the second condition after the first is required, and those that deny some condition and then subsequently require it to be true. Formally:

Definition 49 *Given a restricted propositional environment, an action is* unsafe *iff every agent which accepts a reasonable string incorporating that action is not safe.*

Proposition 33 *An action (set of finite event sequences) is* unsafe *if one of the following conditions holds for any sequence ς in the set. Let $\varsigma_n = (\varsigma_{n1}, \varsigma_{n2})$.*

- $\exists n, m > n \,.\, \exists \nu \in P^{\pm} \,.\quad \nu \in \varsigma_{n1}$ and
  $\neg\nu \in \varsigma_{m2}$ and
  $\forall i \,.\, n < i < m, \neg\nu \notin \varsigma_{i1}$
- $\exists n, m \geq n \,.\, \exists \nu \in P^{\pm} \,.\quad \nu \in \varsigma_{n2}$ and
  $\neg\nu \in \varsigma_{m2}$ and
  $\forall i \,.\, n \leq i < m, \neg\nu \notin \varsigma_{i1}$

It turns out that for restricted propositional environments, it is not necessary to give a complete description of the environment in which a plan will operate to determine safe agents. To prevent event failure, we need to know whether actions are safe, and to do so we only need five sets of atomic formulas:

- Formulas that will inevitably become true at termination of the action executed with nothing in parallel. In this case, the formula is *asserted* by the action.
- Formulas that could possibly become false at termination of the action executed with nothing in parallel. In this case, the formula is *retracted* by the action.
- Formulas that could become false at some stage during execution of the action. In the case, the formula is *conflicted* by the action.
- Formulas that must be true immediately before the action begins to ensure that nothing will fail if the action is executed with nothing in parallel. In this case, the formula is a *precondition* of the action (is *required* by the action).
- Formulas that must be true for some event in the action. In this case, the formula is a *during condition* of the action (is *maintained* by the action).

More formally:

Definition 50 *Assume for a sequence of events ς, $\varsigma_n = (\varsigma_{n1}, \varsigma_{n2})$. Given an environment $<P, A, i>$, an atomic formula $\nu \in P^{\pm}$, and an operator $\alpha \in A$:*

- α asserts ν *if for all sequences $\varsigma \in i(\alpha)$*

$$\exists n \, . \, \nu \in \varsigma_{n1}, \text{and } \forall m > n \, . \, \neg\nu \notin \varsigma_{m1}$$

- α retracts ν *if for some sequence $\varsigma \in i(\alpha)$*

$$\exists n \, . \, \neg\nu \in \varsigma_{n1}, \text{and } \forall m > n \, . \, \nu \notin \varsigma_{m1}$$

- α conflicts ν *if for some sequence $\varsigma \in i(\alpha)$*

$$\exists n \, . \, \neg\nu \in \varsigma_{n1}$$

- α requires ν *if for some sequence $\varsigma \in i(\alpha)$*

$$\exists n \, . \, \nu \in \varsigma_{n2}, \text{and } \forall m < n \, . \, \nu \notin \varsigma_{m1}$$

- α maintains ν *if for some sequence $\varsigma \in i(\alpha)$*

$$\exists n \, . \, \nu \in \varsigma_{n2}$$

Beware of slightly misleading terminology: a *maintained* formula is not actively maintained true, but the action could fail if it becomes false.

Note that every required formula is maintained; that if an atomic formula is asserted, then its negation is retracted; that an asserted formula may be conflicted as well but not retracted; and that anything retracted is also conflicted.

Proposition 34 *For any action and proposition p, there are ten distinct possible cases:*

1. *p is asserted but not conflicted, and $\neg p$ is retracted.*
2. *$\neg p$ is asserted but not conflicted, and p is retracted.*
3. *p is asserted and conflicted, and $\neg p$ is retracted.*
4. *$\neg p$ is asserted and conflicted, and p is retracted.*
5. *p and $\neg p$ are both retracted.*

6. *p is retracted, and $\neg p$ is conflicted only.*

7. *$\neg p$ is retracted, and p is conflicted only.*

8. *p is retracted, and $\neg p$ is not asserted, retracted, or conflicted.*

9. *$\neg p$ is retracted, and p is not asserted, retracted, or conflicted.*

10. *Neither p nor $\neg p$ is asserted, retracted, or conflicted.*

Given any sequence of communications, the interpretations of actions may be changed arbitrarily without changing the possibility of an event failing, as long as the five sets remain the same.

Definition 51 *Two actions are* equivalent *if the sets of atomic formula they assert/retract/conflict/require/maintain are the same.*

Proposition 35 *Given two environments with the same set of operators, and for which all actions are safe, and for which the interpretations for a given operator are equivalent, an agent is safe in one iff it is safe in the other.*

The rules for ensuring no event failure are:

- An action that has a during condition may not run in parallel with an action that conflicts that during condition.
- An action α_1 that has a precondition may not be begun until some action α_2, which asserts the precondition, has completed execution, and also no action α_3 that retracts that condition may be running at any time from the time α_2 begins until the time α_1 ends.

These rules are necessary and sufficient for ensuring that event failure cannot occur. They may also be expressed formally as conditions corresponding to safe strings, which must consequently be satisfied by every reasonable string accepted by a safe agent.

Proposition 36 *Given an environment $< P, A, i >$ with only safe actions, a reasonable string σ is safe iff the following two conditions hold:*

1. - *Given an arbitrary $\nu \in P^{\pm}$*
   - *and an arbitrary n such that $\sigma_n =$ (begin α) and $i(\alpha)$ maintains ν*

- *and an arbitrary $m \geq n$ such that there are more* (begin α) *than* (end α) *in any prefix of the string* $(\sigma_n, \sigma_{n+1}, \ldots, \sigma_m)$
- *there is no $i < m, i \neq n$, such that $\sigma_i =$* (begin β) *and $i(\beta)$ conflicts ν*
- *and for which every prefix of the string $(\sigma_i, \sigma_{i+1}, \ldots, \sigma_m)$ has more* (begin β) *than* (end β).

2. - *Given an arbitrary $\nu \in P^{\pm}$*
   - *and an arbitrary n such that $\sigma_n =$* (begin α) *and $i(\alpha)$ requires ν*
   - *and an arbitrary $m > n$ such that there are more* (begin α) *than* (end α) *in any prefix of the string* $(\sigma_n, \sigma_{n+1}, \ldots, \sigma_m)$
   - *let i be the smallest number (possibly 0) such that for all $\beta \in A$ that retract ν, there are equal numbers of* (begin β) *and* (end β) *in the string $(\sigma_1, \sigma_2, \ldots, \sigma_i)$ and no* (begin β) *in the string* $(\sigma_i, \sigma_{i+1}, \ldots, \sigma_{n-1})$
   - *there must be some $j : i < j < n$ such that $\sigma_j =$* (begin β') *and β' asserts ν and some prefix of the string $(\sigma_j, \sigma_{j+1}, \ldots, \sigma_n)$ has exactly enough* (end β') *to make up the difference between the numbers of* (begin β') *and* (end β') *in the string* $(\sigma_1, \sigma_2, \ldots, \sigma_j)$.

Proposition 37 *Assume that every action is safe. Then an agent is safe iff all the reasonable strings it accepts satisfy the two conditions of the previous theorem, which is the case iff for any atomic formula ν and for any associated sequence ξ of agent execution states the following two conditions hold (Let $\xi_n = (<\omega_n, E_n, \tau_n>, \eta_n)$):*

1. *There is no E_n such that $(\alpha, \varsigma) \in E_n$ and $(\alpha', \varsigma') \in E_n$ and α maintains ν and α' conflicts ν.*
2. *For any E_n such that $(\alpha, \varsigma) \in E_n$ and α requires ν, there is some E_m with $m < n$, where $E_m = E_{m-1} \cup \{(\alpha', \varsigma')\}$, and α' asserts ν, and for every i for which* $\text{pop}^i(\varsigma')$ *is defined, there is a $j : m \leq j \leq n$ and $(\alpha, \text{pop}^i(\varsigma')) \in E_j$, and for every $i : m < i < n$, E_i has no element of the form (α'', ς'') where α'' retracts ν.*

3.2 Synchronizing Plans

The *synchronization skeleton* of a plan is an abstraction of the plan in which detail irrelevant to synchronization is suppressed[3]. Given an arbitrary complete bounded agent, there is

always a way of constructing a plan that has an interpretation totally equivalent to that agent, and that has a clearly separated parallel component corresponding to the synchronization skeleton containing only *(set m)* and *(guard m s)* operations. The rest of the plan contains only operators and *send* operations.

We will use the extended plan syntax. Note that any plan in the original syntax may be put into the extended syntax by defining a single variable for use in all *guard* and *set* commands and having the set of variable states be the same as the original memory states.

A synchronization skeleton may be added to any plan by inserting *send* operations in the plan and running the synchronization skeleton in parallel. The set of possible sequences of communication acts for the resulting agent is a subset of those for the original agent.

The power of a synchronization skeleton is demonstrated by the following result.

Proposition 38 *Let a and a' be arbitrary regular agents, and let ρ be a plan in the extended syntax that has an interpretation isomorphic to a. Then a includes a' iff there is a plan ρ' in the extended variable syntax*

- *that has an interpretation isomorphic to a',*
- *and is of the form $\rho'_1 \parallel \rho'_2$, with a set of variable states that is a superset of the set of variable states used in ρ,*
- *and the only primitive plans in ρ'_2 are* guard *and* set *operations using variables and messages not used in ρ,*
- *and ρ'_1 is obtained from ρ by optionally replacing any primitive operator with sequence pairs incorporating the primitive operator and a* send *operation for a message used in ρ'_2 but not in ρ.*

Manna and Wolper[11] describe an algorithm for generating such a skeleton from propositional temporal logic (PTL) formulas used to express constraints on execution sequences of a plan. PTL has the finite model property, and is thus particularly suited to reasoning about the sequences of input output operations an agent may perform. PTL is described in an appendix.

Given a plan, and for each primitive action the five sets of formulas mentioned above, it is possible to generate PTL formulas that are true for all and only those sequences of communication operations for which the environment cannot fail. These formulas correspond to the two correctness rules given above.

The propositions used in the PTL formula are *not* those manipulated by the agent, but instead correspond to the communication acts of the agent.

4 The Plan Synchronizer

A program has been written in LISP that takes as input a plan and operator descriptions and produces as output a synchronized plan that is the corresponding maximal safe deadlock-free plan, or fails if none exists. The current version of the program handles only deterministic actions, which have only one possible sequence of events, and input plans that have no loops or nondeterministic selection and have no synchronization (all primitive subplans are operators). A second version under development will handle arbitrary input plans and arbitrary restricted propositional environments. This section describes the operation of the second version.

The program proceeds in several stages, each of which will be described in turn. The approach is based on that proposed by Georgeff [5], but is more general.

4.1 Interaction Analysis

The first stage simply lists the operators that assert/retract/conflict/require/maintain each atomic formula. The input to the program is a plan represented as a list of subplans to be executed in sequence, where a subplan is a list expression of one of the following formats:

- A list, where the first element is not a reserved word. This corresponds to an operator.
- A list, where the the first element is SEND, SET, or GUARD, followed by one or two arbitrary list expressions. GUARD is followed by two expressions, the others by one. These correspond to synchronization primitives.
- A list, where the first element is LOOP and remaining elements are subplans. This corresponds to a sequence of subplans in a loop.
- A list, where the first element is PARALLEL and remaining elements are nonempty lists of subplans. This corresponds to sequences of plans executed in parallel.
- A list, where the first element is SELECT and remaining elements are (possibly empty) lists of subplans. This corresponds to the nondeterministic selection of lists of subplans.

In the first version, SET, GUARD, SEND, LOOP, and SELECT were not allowed in the input plan.

The list expressions corresponding to operators are EXECuted to return a list of effects. An effect is one of the words {ASSERT, RETRACT, CONFLICT, REQUIRE, MAINTAIN}, followed by a list of atomic formulas. In the first version, RETRACT was not allowed for an effect. An atomic formula is either a proposition or a list of NOT and a proposition. A proposition is an arbitrary list expression not beginning with NOT.

A subplan is identified in the program as a sequence of numbers. The null sequence identifies the whole plan. Otherwise, each number in turn identifies a subplan within the subplan identified by a prefix of the sequence, according to the following algorithm:

1. Set the current subplan to the whole plan.

2. If the identifying sequence is empty, return the current subplan.

3. Pop the next number n from the identifying sequence.

4. Pop any reserved word from the current subplan if it exists.

5. Set the current subplan to the nth element of the current subplan.

6. Go to step 2.

These sequences will be used as *tags*.

The output of the first stage is a list associating five lists of tags with each atomic formula, and is obtained by simply scanning the plan. For each operator with a formula in the X list (where X is one of RETRACT, ASSERT, CONFLICT, REQUIRE, or MAINTAIN), add the corresponding tag to the X list for that formula. If a tag is added to the ASSERT list for a formula, it is also added to the RETRACT list for the formula's negation; if a tag is added to the RETRACT list for a formula, it is also added to the CONFLICT list; and if a tag is added to the REQUIRE list for a formula, it is also added to the MAINTAIN list.

An error is returned if one operator ASSERTS a formula and its negation, or if one operator CONFLICTS a formula and its negation but RETRACTS neither. Thus the program ensures that the transition associated with an operator corresponds to one of the ten possible cases for a restricted propositional action and a proposition.

All operators are assumed to be safe, so an error is returned if one operator REQUIRES a formula and its negation simultaneously.

4.2 PTL Constraint Generation

An agent is characterized by the strings of messages it may exchange with an environment. If we regard messages as propositions, then strings are interpretations for PTL formulas, where only one proposition is true at a given moment.

Interpretations for PTL formulas are infinite strings, so finite strings are made into interpretations by appending an infinite number of states in which no proposition is true.

Given a restricted propositional environment, it is possible to find a PTL formula that is true for all safe reasonable strings and false for all unsafe reasonable strings. This PTL formula corresponds to the two rules for safe strings given in an earlier section. It is called the *safety constraint.*

Proposition 39 *Assume a restricted propositional environment for which every action is safe. Then there is a PTL formula that is true for all safe reasonable strings and false for all unsafe reasonable strings. This formula is the conjunction of all formulas that can be formed by the following rules:*

- *If ν is an atomic formula, and c is an identifier that conflicts it, and m is an identifier that maintains it, then add the formula*

$$\Box((\text{begin } c) \supset (\neg(\text{begin } m) \cup (\text{end } c))) \\ \wedge \Box((\text{begin } m) \supset (\neg(\text{begin } c) \cup (\text{end } m)))$$

- *If ν is an atomic formula, and d is an identifier that retracts it, and r is an identifier that requires it, and $\{a_i\}_{i=1}^{n}$ is the set of all identifiers that assert it, then add the formula*

$$\Box(\ (\text{begin } d) \supset \\ \neg(\text{begin } r) \cup (\ (\text{end } d) \wedge \\ \neg(\text{begin } r) \cup \bigvee_{i=1}^{n}(\ (\text{begin } a_i) \wedge \\ (\neg(\text{begin } r) \cup (\text{end } a_i))))$$

- *If ν is an atomic formula, and $\{a_i\}_{i=1}^{n}$ is the set of all identifiers that assert it, then add the formula*

$$\neg(\text{begin } r) \cup \bigvee_{i=1}^{n}((\text{begin } a_i) \wedge (\neg(\text{begin } r) \cup (\text{end } a_i)))$$

Given a specific agent, we can simplify this formula in several ways while maintaining the required property. First, we need some relations between identifiers in a plan or agent.

Definition 52 *Let an agent be given.*

- *An identifier* α *is* in parallel with *an identifier* α' *if there is an associated sequence of agent execution states with some environment state that contains both* α *and* α'.
- *An identifier* α may precede *an identifier* α' *if there is an associated sequence of agent execution states* ς *(let* $\varsigma_n = (<\omega_z, E_z, \tau_z>, \nu_z)$*) such that*

$$\exists i, j \;.\; i < j, E_i \text{ contains } \alpha, E_j \text{ contains } \alpha'$$

- *For a set of identifiers* I, *an agent* may be I-free *if there is an associated sequence of agent execution states* ς *(let* $\varsigma_n = (<\omega_z, E_z, \tau_z>, \nu_z)$*) such that*

$$\forall i \;.\; \forall \alpha \in I \;.\; \neg(E_i \text{ contains } \alpha)$$

- *Let* α *be an identifier and* I' *be a set of identifiers. Then* α may precede all I' *if there is an associated sequence of agent execution states* ς *(let* $\varsigma_n = (<\omega_z, E_z, \tau_z>, \nu_z)$*) such that*

$$\exists i \;.\; (E_i \text{ contains } \alpha \wedge \forall \alpha' \in I' \;.\; \forall j < i \;.\; \neg(E_j \text{ contains } \alpha'))$$

- *For identifiers* α *and* α', *and a set of identifiers* I'', "α may precede α' without any I''" *if there is an associated sequence of agent execution states* ς *(let* $\varsigma_n = (<\omega_z, E_z, \tau_z>, \nu_z)$*) such that*

$$\begin{aligned} \exists i, j \;.\; i < j, \quad & E_i \text{ contains } \alpha, \\ & E_j \text{ contains } \alpha', \\ & \forall k, i < k < j \;.\; \forall \alpha'' \in I'' \;.\; E_k \text{ does not contain } \alpha'' \end{aligned}$$

- *For identifiers* α, α', *and* α'', "α may separate α' and α''" *if there is an associated sequence of agent execution states* ς *(let* $\varsigma_n = (<\omega_z, E_z, \tau_z>, \nu_z)$*) such that*

$$\begin{aligned} \exists i, j, k \;.\; i < j < k, \quad & E_i \text{ contains } \alpha, \\ & E_j \text{ contains } \alpha', \\ & E_k \text{ contains } \alpha'', \\ & E_k \text{ does not contain } \alpha \text{ or } \alpha'' \end{aligned}$$

The above relations are also defined for a plan if we take the plan to be equivalent to an agent that is its interpretation. We can relate the above relations to the syntax of a plan without synchronization.

Proposition 40 *Given a plan ρ containing no synchronization primitives, where every operator is replaced by a unique identifier:*

- *An identifier α is* in parallel with *an identifier α' iff there is a subplan $\rho' = (\rho_1 \parallel \rho_2)$ or $\rho' = (\rho_2 \parallel \rho_1)$ such that α is a subplan of ρ_1 and α' is a subplan of ρ_2.*
- *An identifier α* may precede *an identifier α', iff*
  - *α is in parallel with α'; or*
  - *there is a subplan $\rho' = (\rho_1; \rho_2)$ such that α is a subplan of ρ_1 and α' is a subplan of ρ_2; or*
  - *there is a subplan $\rho' = (\rho_1)^\times$ such that α and α' are both subplans of ρ_1.*
- *For a set of identifiers I, the plan* may be I-free *iff one of the following cases holds:*
  - *The plan is Λ.*
  - *The plan is α for some identifier α not in I.*
  - *The plan is of the form $(\rho')^\times$.*
  - *The plan is of the form $\rho_1; \rho_2$ or $\rho_1 \parallel \rho_2$, and both ρ_1 and ρ_2 are I-free.*
  - *The plan is of the form $\rho_1 \mid \rho_2$, and one of ρ_1 and ρ_2 is I-free.*
- *For identifiers α and α', and the set of identifiers I'', "α* may precede *α'* without any *I''" iff*
  - *alpha is in parallel with α'; or*
  - *there is a subplan $\rho' = (\rho_1; \rho_2)$ such that*
    * *α is a subplan of ρ_1 and α' is a subplan of ρ_2; and*
    * *for every subplan of ρ_1 of the form $(\rho_1'; \rho_1'')$, either α is not a subplan of ρ_1' or ρ_1'' is I''-free; and*
    * *for every subplan of ρ_2 of the form $(\rho_2'; \rho_2'')$ either, α' is not a subplan of ρ_2'' or ρ_2' is I''-free; or*
  - *there is a subplan $\rho' = (\rho_1)^\times$ such that*
    * *α and α' are both subplans of ρ_1; and*
    * *for every subplan of ρ_1 of the form $(\rho_1'; \rho_1'')$, either α is not a subplan of ρ_1' or ρ_1'' is I''-free; and*

- * *for every subplan of* ρ_1 *of the form* $(\rho_1'; \rho_1'')$, *either* α' *is not a subplan of* ρ_1'' *or* ρ_1' *is* I''*-free.*

- *For an identifier* α *and a set of identifiers* I', α may precede all I' *iff for every subplan* $\rho_1; \rho_2$ *either* ρ_1 *is* I'*-free or* α *is not a subplan of* ρ_2.

- *For identifiers* α, α', *and* α'', "α may separate α' and α''" *iff*
  - α' *may precede* α''; *and*
  - α *may precede* α''; *and*
  - α' *may precede* α.

The above result translates relations between identifiers to restrictions on the syntax of a plan without synchronization. For a plan with synchronization, we cannot get the full correspondence, but we can get the following result.

Proposition 41 *For a general plan, the previous theorem will hold true if all "iff" are replaced by "only if".*

Now we go on to find a set of PTL formulas.

Proposition 42 *Assume a restricted propositional environment for which every action is safe. Let some plan be given. Then there is a PTL formula that is true for all safe reasonable strings accepted by the plan and false for all unsafe reasonable strings accepted by the plan. This formula is the conjunction of all formulas that can be formed by the following rules, where relations between identifiers are defined using the plan formed by replacing all synchronization operations in the original with the null plan. (This allows us to use plan syntax to find identifiers satisfying particular relations.)*

- *If* ν *is an atomic formula, and* c *is an identifier that conflicts it but does not retract it, and* $\{m_i\}_{i=1}^n$ *is the nonempty set of identifiers* m *that maintain* ν *and that are in parallel with* c, *then add the formula*

$$\Box((\text{begin } c) \supset (\neg(\bigvee_{i=1}^n(\text{begin } m_i)) \cup (\text{end } c)))$$

- *If* ν *is an atomic formula, and* m *is an identifier that maintains it, and* $\{c_i\}_{i=1}^n$ *is the non-empty set of identifiers* c *that conflict* ν *and that are in parallel with* c, *then add the formula*

$$\Box((\text{begin } m) \supset (\neg(\bigvee_{i=1}^{n}(\text{begin } c_i)) \cup (\text{end } m)))$$

- *Find an atomic formula ν and an identifier d that retracts it. Let U be the set of all identifiers that require, retract, or assert ν. Let R be the set of identifiers:*

$$\{r : r \text{ requires } \nu, d \text{ may precede } r \text{ without any } U\}$$

Let B be the formula

$$\bigvee_{r \in R}(\text{begin } r)$$

Let A be the set of identifiers:

$$\{a : a \text{ asserts } \nu, \exists r \in R \,.\, a \text{ may separate } d \text{ and } r \text{ without any } U\}$$

For each $a \in A$, F_a is one of the following formulas:

  - *If a is in parallel with d, and $\exists r \in R \,.\, a$ is in parallel with r, take the formula*

$$(\text{begin } a) \wedge (B \cup (\text{end } a))$$

  - *Otherwise, if $\exists r \in R \,.\, a$ is in parallel with r, take the formula*

$$(\text{end } a)$$

  - *Otherwise take the formula*

$$(\text{begin } a)$$

Let F be the formula

$$B \cup \bigvee_{a \in A} F_a$$

Now generate one of the following four formulas:

  - *If $\exists r \in R \,.\, d$ is in parallel with r, and $\exists a \in A \,.\, d$ is in parallel with a*

$$\Box((\text{begin } d) \supset (B \cup ((\text{end } d) \wedge F)))$$

  - *Otherwise, if $\exists r \in R \,.\, d$ is in parallel with r*

$$\Box((\text{begin } d) \supset F)$$

  - *Otherwise, if $\exists r \in R \,.\, r$ may precede d, or if $\exists a \in A \,.\, a$ may precede d*

$$\Box((\text{end } d) \supset F)$$

– *Otherwise,*

$$F$$

- *Find an atomic formula ν. Let U be the set of all identifiers that require, retract, or assert ν. Let R be the set of identifiers:*

  $$\{r : r \text{ requires } \nu, r \text{ may precede all } U\}$$

  Let A be the set of identifiers:

  $$\{a : a \text{ asserts } \nu, \exists r \in R \,.\, a \text{ may precede } r \text{ without any } U\}$$

  Then take the formula

  $$\neg \bigvee_{r \in R} (\text{begin } r) \cup \bigvee_{a \in A} (\text{end } a)$$

There are many other ways to get appropriate sets of formulas; this is the particular set found by the program. There is a trade-off between the difficulty of generating the formulas, the time taken to process the resulting formulas in the next section, and the number of distinct propositions used (fewer propositions implies faster processing in the final section as well as faster theorem proving).

We have a PTL formula true for safe strings, but the theorem prover must generate a model corresponding to safe accepted strings. Thus, another constraint is needed: the *ordering constraint*, which is true for all accepted strings.

In general, it is possible to add new propositions, so that in a state one or none of the message propositions are true and some arbitrary subset of the added propositions are true. We can then find a PTL formula that has a model being a set of sequences of sets of propositions, such that if all added propositions are deleted the sequences correspond exactly to the strings accepted by an agent. Rather than develop this method, we augment PTL with *program* formulas, which can be used to represent ordering constraints very efficiently.

The PTL formula that is the conjunction of the ordering constraint and the safety constraint is called the *total constraint*. It characterizes the agent we want as output.

Proposition 43 *Given a plan and environment description, the interpretations of the total constraint generated as above correspond exactly to the reasonable strings accepted by the corresponding maximal safe deadlock-free agent.*

So now we can use temporal logic theorem-proving techniques to solve the interaction problem.

First, we define and give an interpretation for the extended PTL syntax. We define the syntax for an ordering constraint to be a (plan, memory state) pair, where the set of primitive operators used in the plan correspond to messages. Considered as a PTL formula, it is true for all and only the sequences corresponding to possible plan executions if the initial memory state is as given in the constraint. We define this more formally.

Definition 53 *Given a set of memory states M and a set of propositions P, a* program formula *is a pair (m, ρ), where $m \in M$ and ρ is a sentence given by the syntax for plans in which the set of operators is P, the set of memory states is M, and the set of signals is arbitrary.*

The semantics of PTL is given by finding the sequences of states for which a PTL formula is *true*. To define the semantics of a program formula, we start by defining the *arc* set of a program formula.

Definition 54 *Given a plan ρ, a subplan ρ_1 is* current *if there is no subplan of the form $(\rho'_1; \rho'_2)$, ${\rho'_1}^{\times}$ or $(\rho'_1 \mid \rho''_1)$, where ρ_1 is a subplan of ρ'_1 or ρ''_1.*

Definition 55 *The* arc set *of a program formula (m, ρ) is the set of triples (α, m', ρ'), such that α is a proposition and (m', ρ') is a program formula that can be formed by the following rules:*

1. *Make the following transformations on ρ, treating it like a plan, until no more are possible:*
   - *For any current subplan of the form ${\rho'}^{\times}$, replace with either $(\rho'; {\rho'}^{\times})$ or Λ.*
   - *For any current subplan of the form $(\rho_1 \mid \rho_2)$, replace with either ρ_1 or ρ_2.*
   - *Replace $(\rho'; \Lambda)$, $(\Lambda; \rho')$, $(\rho' \parallel \Lambda)$, or $(\Lambda \parallel \rho')$ with ρ'.*
   - *For a pair of matching synchronization operations* (send s) *and* (guard m s), *where both are current and m is the memory state for the program formula given, replace both operations with Λ.*
   - *For any* (set m') *which is current, replace with Λ and change the memory state associated with the program formula from m to m'.*

2. *Choose any current subplan α, where α is a proposition. Replace α with Λ. Then if (m', ρ') is the program formula obtained from all the transformations made, the triple (α, m', ρ') is in the arc set.*

Definition 56 *Given a program formula, construct a graph where the nodes are program formulas and the arcs are labeled with propositions from the set of propositions in the plan formula. Start with a single node labeled with the initial program formula. For any node, take the arc set as defined above. For each (α, m, ρ) in the set, add an arc labeled α from that node to a node labeled (m, ρ).*

By definition, a program formula is true *for all and only those infinite sequences of proposition sets formed by appending states where no proposition is true to finite sequences of singleton sets corresponding to paths from a node labeled with the program formula to a node labeled with (m, Λ) for some arbitrary memory state m.*

A program formula is true for possible terminating execution sequences of the plan given some initial memory state.

It turns out that we only need *order* the propositions used in the safety constraint. Thus, the ordering constraint returned by this stage of the program is found by taking the plan, replacing every identifier α with ((begin α); (end α)), and then replacing every proposition that is not used in the safety constraint with Λ.

The memory state used in the program formula is λ, the extra memory state that is not used in the plan. The output of this section is the conjunction of the safety constraint and the modified version of the ordering constraint using only propositions used in the safety constraint.

4.3 PTL Theorem Prover

An interpretation for a PTL formula is a sequence of sets of propositions for which the formula is true. A model for a formula is the set of all interpretations.

It would be useful if we could represent a model as a finite automaton (over the alphabet of sets of propositions) that accepts all and only the interpretations for a given formula. We cannot quite do that, but we can find a close approximation.

Proposition 44 *For any PTL formula, we can find a labeled graph $< W, N, I, t >$, where W is the set of sets of propositions and is the alphabet of the automaton, N is a finite*

set of nodes, I is an initial node, and t is a set of arcs between nodes labeled with sets of propositions:

$$
\begin{array}{lcl}
W & = & 2^P \\
N & \text{is} & \text{arbitrary and finite} \\
I & \in & N \\
t & \subseteq & N \times N \times W
\end{array}
$$

with the properties:

- *Every interpretation corresponds to an infinite path through the graph.*
- *Every finite path through the graph is the prefix of some interpretation.*

An algorithm for finding a suitable graph is given by Manna and Wolper[11]. We use a similar algorithm, with some simplifications that apply particularly to the problem we are dealing with.

Definition 57 *A PTL formula is* eventuality free *iff every infinite sequence of sets of propositions for which every prefix is also the prefix of some interpretation is itself an interpretation.*

Proposition 45 *The safety constraint generated by the previous stage of the program is eventuality free.*

Proposition 46 *A PTL formula is eventuality free iff there is some graph as given above such that the set of infinite paths through the graph corresponds exactly to the set of interpretations for the formula.*

The algorithm we use relies on the fact that the PTL formulas of the safety constraint are eventuality free, and that the only eventuality required by the ordering constraint is that the plan will always terminate. Manna and Wolper's algorithm puts a great deal of effort into satisfying eventualities. We will find a graph such that every finite path through the graph is the prefix of an interpretation and every interpretation is a path through the graph.

Thus, it is possible for an infinite path through this graph not to satisfy the ordering constraint. However, such a path must correspond to an execution sequence for which it can never be asserted during the execution that the plan cannot terminate.

We now describe the algorithm for constructing the graph.

First, take the formula produced by the previous stage of the program and put it in a standard form. The ordering constraint is left unchanged. The safety constraint is made into a disjunctive normal form – a disjunction of conjunctions of formulas which are either atomic or have a temporal operator as the major connective. Also, no operator occurs in the scope of a negation operator.

The conversion is by repeated application of the following rules to subformulas within the constraint.

- Replace $f_1 \supset f_2$ with $\neg f_1 \vee f_2$.
- Replace $\neg\neg f_1$ with f_1.
- Replace $\neg(f_1 \wedge f_2)$ with $(\neg f_1 \vee \neg f_2)$.
- Replace $\neg(f_1 \vee f_2)$ with $(\neg f_1 \wedge \neg f_2)$.
- Replace $\neg\Box f_1$ with $\Diamond\neg f_1$.
- Replace $\neg\Diamond f_1$ with $\Box\neg f_1$.
- Replace $\neg(f_1 \cup f_2)$ with $\neg f_2 \uplus \neg f_1$.
- Replace $\neg(f_1 \uplus f_2)$ with $\neg f_2 \cup \neg f_1$.
- Replace $f_1 \wedge (f_2 \vee f_3)$ with $(f_1 \wedge f_2) \vee (f_1 \wedge f_3)$ if the original is not in the scope of a temporal operator.
- For formulas of the form $f_1 \circ (f_2 \circ f_3)$ or $(f_1 \circ f_2) \circ f_3$, where $\circ$ is $\wedge$ or $\vee$, replace with $f_1 \circ f_2 \circ f_3$.

The nodes of the graph contain a PTL formula corresponding to the safety constraint, and a program formula corresponding to the ordering constraint. Start the graph with a single node containing the given ordering constraint and the standardized safety constraint.

For each node, proceed as follows:

1. Transform the safety constraint by applying one of the following rules to any subformula that has as its main connective a temporal operator other than $\bigcirc$, and that is not within the scope of any other temporal operator, until no such subformula remains.

- Replace $\Box f_1$ with $f_1 \wedge \bigcirc\Box f_1$.
- Replace $f_1 \cup f_2$ with $f_2 \vee (f_1 \wedge \bigcirc(f_1 \cup f_2))$.
- Replace $\Diamond f_1$ with $f_1 \vee \bigcirc\Diamond f_1$.
- Replace $f_1 \uplus f_2$ with $f_1 \wedge (f_2 \vee \bigcirc(f_1 \uplus f_2))$.

In fact, the second two rules are never used. This step divides a safety constraint into requirements on the current state, and on the rest of the execution sequence.

2. Get the arc set for the program formula, as described above. This corresponds to finding all the possible messages to occur next in the sequence.

3. For each (α, m, ρ) in the arc set, get a new safety constraint by the following modifications to the safety constraint of the node under consideration.

   (a) For any α not in the scope of a temporal operator, replace it with **true**.

   (b) For any other proposition not in the scope of a temporal operator, replace it with **false**.

   (c) For any proposition at any point in the safety constraint which is not in the new ordering constraint (m, ρ), replace that proposition with **false**.

   (d) Apply the following rules as long as they are applicable. (f ia an arbitrary subformula.)

      - Replace **true** $\wedge f$, $f \wedge$ **true**, **false** $\vee f$ and $f \vee$ **false** with f.
      - Replace **false** $\wedge f$, $f \wedge$ **false**, $\Diamond$**false**, $\Box$**false** and $\bigcirc$**false** with **false**.
      - Replace **true** $\vee f$, $f \vee$ **true**, $\Diamond$**true**, $\Box$**true** and $\bigcirc$**true** with **true**.
      - Replace **false** $\cup f$ with f and **false** $\uplus f$ with **false**.
      - Replace **true** $\cup f$ with **true**.
      - Replace **true** $\uplus f$ with $\Diamond f$.
      - Replace $f \cup$ **true** with **true** and $f \uplus$ **true** with f.
      - Replace $f \cup$ **false** with $\Box f$.
      - Replace $f \uplus$ **false** with **false**.

   This asserts that only α is true in the current state, and that propositions which are not in the ordering constrinat will never be true at any future state. The

resulting formula will be a disjunctive normal form of $\bigcirc$ operators, and represents the requirement on the next state in the execution sequence.

(e) If there is any formula outside the scope of temporal operators of the form $\Box\neg\alpha$ for some proposition α, then modify the ordering constraint (m, ρ) as follows:

   i. Let ρ' be the lowest-level subplan of ρ that contains α and that has $*$ or $|$ as its main connective.

   ii. If ρ' is of the form $\rho'^{\times}$, replace it with Λ; if it is of the form $(\rho'_1 \mid \rho'_2)$, then replace it with whichever of ρ'_1 and ρ'_2 does not contain α.

If no such subplan ρ' can be found, replace the safety constraint with **false**. Otherwise, restart step 3 since more propositions may be replaced with **false** in the safety constraint as a result of deletion from the ordering constraint.

This is a heuristic we have found useful since it is applicable fairly often in problems we have solved. The resulting formula is equivalent, and it may avoid generating extra nodes in the graph.

(f) Delete all $\bigcirc$ operators not in the scope of another $\bigcirc$ operator from the modified safety constraint. This formula is true in the next state.

4. If the safety constraint is not **false**, add an arc labeled α to a node where the safety constraint is as found in the previous step and the ordering constraint is (m, ρ).

It turns out that this procedure will always terminate, since only a finite number of nodes can be generated. Nodes where the ordering constraint is (m, Λ) for some m are called *final* nodes. The last step in producing the graph is to delete all nodes that have no path leading to a final node. If this deletes all nodes, then there is no appropriate safe deadlock-free plan, and the program returns an appropriate error.

An execution sequence is safe if, after deleting propositions not used in the initial formula, it corresponds to a path through this graph that is either infinite or terminates at a final node. The core of the proof is showing that the initial formula is true for all and only the safe execution sequences, and showing that the algorithm generates an appropriate graph in the manner of the proof of correctness for the PTL theorem prover by Wolper[17]. We state the result.

Proposition 47 *An execution sequence of the given plan is safe iff after deleting propositions not in the generated constraints, it corresponds to a path through the generated graph*

that either is infinite or ends at a final node.

Any such path through the graph can be made into a possible safe execution sequence of the given plan by the addition of propositions not in the generated constraints.

4.4 Synchronized Plan Generation

The algorithm for generating the output plan is very similar to that of Manna and Wolper[11].

First, synchronization primitives are inserted into the original plan. For every proposition of the form *(end α)*, replace α in the plan with *(α; (send (end α)))*. For every proposition of the form *(begin α)*, replace α in the plan with *((send (begin α)) ; α)*.

Second, a synchronization skeleton is generated.

1. Each node in the graph produced in the previous stage is given some unique identifier. For every arc in the graph from node n_1 to node n_2 labeled α, take the subplan:

   $$(\text{guard } v\ n_1\ \alpha);\ (\text{set } v\ n_2)$$

   where v is a special unique variable not used in the original plan.

2. Combine all these subplans with the selection operator.

3. Put the whole skeleton as produced up to this point in a loop.

4. Use the ";" operator to put (set v i) before the loop, where i is the initial node.

5. For every arc leading from an arbitrary node n to a final node f, take a subplan (guard v n α), where α is the label of the arc. Combine all these subplans using the selection operator, and place them after the loop with the ";" operator.

The resulting synchronization skeleton is exactly the graph represented as a plan.

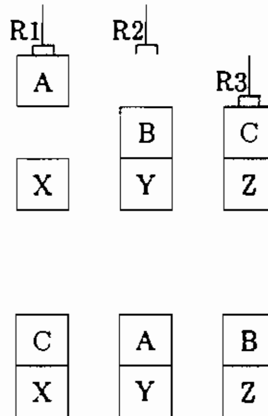
The synchronization skeleton is combined with the synchronized plan using the parallel operator to give the final output of the program.

Actually, the current version of the program puts the skeleton in CSP. Arcs are represented as guarded commands, and the loop is given a termination condition corresponding to the guards placed after the loop in the description above. The two plans are exactly equivalent.

Proposition 48 *The final plan output by the whole program is a maximal safe deadlock-free plan corresponding to the original input plan.*

5 An Example

Consider the problem of three robots, all trying to pick up a block and move it clockwise to a location that another robot will clear as it moves. The diagram shows the robots and blocks, and the initial and final positions.



The unsynchronized plan to achieve this is as follows:

```
( (START (R1 R2 R3) ((A X) (B Y) (C Z)))
  (PARALLEL  ((PICKUP R1 A X) (PUTDOWN R1 A Y))
             ((PICKUP R2 B Y) (PUTDOWN R2 B Z))
             ((PICKUP R3 C Z) (PUTDOWN R3 C X))))
```

The start action sets up the initial conditions, and then each robot in parallel executes a pickup and putdown. Clearly, collisions might result.

The synchronized plan produced by the program is

```
( (PARALLEL ((SEND (BEGIN 1))
             (START (R1 R2 R3) ((A X) (B Y) (C Z)))

             (PARALLEL ((PICKUP R1 A X)
                        (SEND (END 2 1 1))
                        (SEND (BEGIN 2 1 2))
                        (PUTDOWN R1 A Y))

                       ((PICKUP R2 B Y)
```

```
                    (SEND (END 2 2 1))
                    (SEND (BEGIN 2 2 2))
                    (PUTDOWN R2 B Z))

                   ((PICKUP R3 C Z)
                    (SEND (END 2 3 1))
                    (SEND (BEGIN 2 3 2))
                    (PUTDOWN R3 C X))))

         ((RECV (BEGIN 1))
          (SETQ N 2)
          (WHILE (NOT (EQ N 13))
              (SEL (IF (AND (EQ N 2)
                                (RECV (END 2 3 1)))
                      THEN (SETQ N 5))
                   (IF (AND (EQ N 2)
                                (RECV (END 2 2 1)))
                      THEN (SETQ N 4))
.......
                   (IF (AND (EQ N 5)
                                (RECV (END 2 1 1)))
                      THEN (SETQ N 6)))))))
```

The syntax may appear a little different. For example, the *guard* operation is replaced by an *IF*, which tests the memory and looks for a signal. Similarly, a condition is introduced on the loop. These are syntactic hangovers from generating pure CSP and can be translated directly into plans as they have been defined here. A large section of the synchronization skeleton has been removed in the example, since it contains 42 guarded commands - one for each arc in the model for the PTL formulas.

6 Conclusion and Future Work

In conclusion, there exists a program that synchronizes plans in such a way to allow any and all execution sequences which do not allow actions to fail and which do not deadlock. A second version is being developed that handles more general plans.

There is still plenty of scope for additional investigation, and such investigation is being conducted as part of the thesis work for the author's degree at Monash University in Melbourne, Australia. It is worthy of investigation to consider how to synchronize a plan by inserting all synchronization primitives in the main plan rather than provide a massive parallel synchronization skeleton, or how the synchronization skeleton could be made more modular. It could be possible to generate separate skeletons for each proposition used and changed by actions, and run these in parallel.

The definition of actions and environments given here enables very strong properties to be given to the synchronized plans: in particular, that *all and only* the correct executions of the initial plan are permitted. This is in contrast to previous means of synchronizing plans, such as in NOAH[13], which prohibit some execution sequences that might succeed.

By extending the definition of actions to include more general state transformations in events, a similar algorithm would generate a plan that is still less restrictive than that produced by previous plan-modifying techniques, but would disallow certain correct executions. The action descriptions would be much more complex and not capture all the essential properties in the same way as can be done in the simple case with ten sets of propositions. The current version of the theorem prover makes the same assumption as Manna and Wolper: that only one proposition of the PTL formula is true at any moment. For extended actions, new propositions might need to be included that do not have this property..

There is also the problem of types of nondeterminism. The current selection operator corresponds to the case where a plan may proceed in one of two directions, and tbe synchronizer is permitted to choose one over the other. This is *angelic* nondeterminism. However, it may be the case for some plans that the choice is critical but is made at execution time, in which case the synchronizer must allow both cases or none at all. This is *demonic* nondeterminism, and implies some additional structure to a plan that cannot be captured by agents as they are now defined. For added complexity, the decision may be based on the state of the world model, so that the synchronizer can determine the possible choices it must leave

open, depending on the possible world models it derives for the moment of choice.

Loops typically display demonic nondeterminism for choosing when to terminate. The synchronizer is not given total control over the number of times the loop will execute, but should always allow the loop to execute as often as necessary. Again, an explicit termination condition in terms of the world model adds further to the complexity.

The approach to this problem is to consider demonic nondeterminism a restriction on the way a plan may be synchronized, and possibly an extended definition of deadlock. We could allow a new type of ϵ arc in agents that is handled slightly differently: an agent may not backtrack over such an arc. This does not add to the expressive power of agents, but will allow demonic nondeterminism to be represented more intuitively.

Loops often have a termination condition that is a function of all the activity in the loop and yet may not be derived from the given information. Such a termination condition could be specified if a plan segment were treated as a single hierarchical action, and could be given properties similar to those for individual actions. For example, one could specify that a loop would always assert some condition. Consider a loop of an action that removes a single item from a box until none are left. To represent this in the formalism given here, the entire loop would be given an assert condition that the box become empty. To guarantee termination, the entire loop could be given a during condition that no one places anything in the box. In this case, the plan synchronizer certainly does not have full control of the choice of when to terminate the loop.

The theory for hierarchical actions and demonic nondeterminism is being explored, and these constructs may be included in the next version of the synchronizer.

There is also room for improvement to the theorem prover. It turns out that the graph produced for many problems is highly commutative. That is, distinct sequences from a given node that differ only in the ordering of propositions may often correspond to alternative paths to the same node. It would be interesting to relate this property to the syntax of PTL formulas, and exploit it to give more efficient theorem proving. The concept of a *program formula* may be related to the grammar operators of Wolper[17]. The development here is very domain specific. The possibilities should be further explored.

Acknowledgments

I would like to thank Michael Georgeff in particular for guidance and stimulating discussion, and for assistance in bringing this paper to its final form. Thanks are due also to SRI International for financial support, facilities, and an excellent environment for pursuit of this study.

A Notation Conventions

Symbol	Representing class
α, β	Operators and identifiers
γ	Correctness conditions
δ	Mappings on world states
ϵ	Null transitions for an agent
ς	Sequences of events
η	Nodes for an agent
θ	A distinguished operator
λ	A distinguished memory state
ν	A propositional formula or set of same
ξ	A sequence of string or agent execution states
π	A set of propositions
ρ	A plan, or arbitrary label on an agent
σ	A string, or general sequence
τ	A world status
ω	a world state

B Propositional Temporal Logic

B.1 Syntax

PTL *formulas* are built from:

1. A set P of atomic propositions.
2. Boolean connectives: $\wedge$ and $\neg$.
3. Temporal operators: $\bigcirc$ (next), $\Box$ (always), and $\cup$ (until).

The formation rules are:

1. An atomic proposition $p \in P$ is a formula.
2. If f_1, f_2 are formulas, then so are

$$f_1 \wedge f_2, \neg f_1, \bigcirc f_1, \Box f_1, f_1 \cup f_2$$

As usual, braces "()" will be used to make grouping clear, and the connectives $\supset$ and $\vee$ are defined as:

- $f_1 \vee f_2$ is $\neg(\neg f_1 \wedge \neg f_2)$.
- $f_1 \supset f_2$ is $\neg f_1 \vee f_2$.

Two additional temporal operators, $\Diamond$ and $\uplus$, are also defined:

- $\Diamond f_1$ is $\neg \Box \neg f_1$.
- $f_1 \uplus f_2$ is $\neg(\neg f_2 \cup \neg f_1)$.

B.2 Semantics

An *interpretation* for a PTL formula, with a set of propositions P, is a non-empty sequence of subsets of P. A finite sequence is extended to an infinite sequence by repetition of the last element. A PTL formula is true or false for a given interpretation according to the following rules:

Let an infinite sequence $\sigma \in (2^P)^\times$ be given.

- If $f \in P$, then f is true iff $f \in \sigma_1$.
- $\neg f_1$ is true iff f_1 is false.
- $f_1 \wedge f_2$ is true iff f_1 is true AND f_2 is true.
- $\bigcirc f_1$ is true iff f_1 is true for pop(σ).
- $\square f_1$ is true iff $\forall i \geq 0$. f_1 is true for pop$^i(\sigma)$.
- $f_1 \cup f_2$ is true iff $\forall i \geq 0$. f_1 is true for pop$^i(\sigma)$ OR
  $\exists i \geq 0$. f_2 is true for pop$^i(\sigma)$ AND
  $\forall 0 \leq j < i$. f_1 is true for pop$^j(\sigma)$

It can easily be shown that

- $\diamond f_1$ is true iff $\exists i \geq 0 : f_1$ is true for pop$^i(\sigma)$.
- $f_1 \uplus f_2$ is true iff $\exists i \geq 0 : f_2$ is true for pop$^i(\sigma)$ AND
  $\forall 0 \leq j \leq i, f_1$ is true for pop$^j(\sigma)$

Note that the operators $\cup$ and $\uplus$ are similar, except that $\uplus$ has an *eventuality* component; $f_1 \uplus f_2$ implies $\diamond(f_1 \wedge f_2)$. Also note that a state is included in its own successors, so that $\square f_1$ implies f_1.

A formula is *valid* if it is true for all interpretations, and it is *satisfiable* if it is true for some interpretation.

A *model* for a formula is the set of all interpretations for which it is true.

References

[1] Allen, J. F., *A General Model of Action and Time*, Technical Report 97, University of Rochester, Rochester, New York, 1981.

[2] Cheeseman, P., *A Representation of Time for Planning.* Technical Note 278, Artificial Intelligence Center, SRI International, Menlo Park, California, 1983.

[3] Emerson, E. A. and E. M. Clarke, *Using Branching Time Logic to Synthesize Synchronization Skeletons.* Science of Computer Programming 2, pages 241-266, 1982.

[4] Fikes, R.E. and N. J. Nilsson, STRIPS: A new approach to the application of theorem proving in problem solving. *Artificial Intelligence*,(2): 189-208, 1971.

[5] Georgeff, M.P., Communication and interaction in multi-agent planning. *Proceedings of the Third National Conference on Artificial Intelligence*, pages 125-129, Washington, D. C., 1983.

[6] Georgeff, M.P., A theory of action for multi agent planning. In *Proceedings of the Fourth National Conference on Artificial Intelligence*, Austin, Texas, 1984.

[7] Hendrix, G., Modeling simultaneous actions and continuous processes. *Artificial Intelligence*, 4, pages 121-125, 1973.

[8] Hoare, C.A.R., Communicating sequential processes. *Communications of the ACM*, (21):8, pages 666-677, 1978.

[9] Hopcroft, J.E. and J.D. Ullman, *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1979.

[10] Lansky, A. L., *Behavioral Specification and Planning for Multiagent Domains.* Technical Report TN 360, Artificial Intelligence Center, SRI International, Menlo Park, California, 1985.

[11] Manna, Z. and P. Wolper, *Synthesis of Communicating Processes from Temporal Logic Specifications.* Report STAN-CS-81-872, Stanford University Computer Science Department, Stanford, California, September 1981.

[12] McDermott, D., *A Temporal Logic for Reasoning about Plans and Processes*, Computer Science Research Report 196, Yale University, New Haven, Connecticut, 1981.

[13] Sacerdoti, E.D., *A structure for plans and behaviour.* Technical Note 109, Artificial Intelligence Center, SRI International, Menlo Park, California, 1975.

[14] Stuart, C.J., *Modeling the operation of agents in a continuously changing world.* Unpublished.

[15] Stuart, C.J., An implementation of a multi-agent plan synchronizer using a temporal logic theorem prover. In *Proceedings of the Ninth International Conference on Artificial Intelligence*, pages 1031-1034, Los Angeles, California, 1985.

[16] Wilkins, D., *Domain-Independent Planning: Representation and Plan Generation.* Technical Note 266R, Artificial Intelligence Center, SRI International, Menlo Park, California, 1983.

[17] Wolper, P., Temporal logic can be more expressive. In *Proceedings of the Twenty-Second Symposium on Foundations of Computer Science.* Nashville, Tennessee, October 1981.