# Theory for Coordinating Concurrent Hierarchical Planning Agents Using Summary Information

**Bradley J. Clement** and **Edmund H. Durfee**
University of Michigan
Ann Arbor, MI 48109
{bradc, durfee}@umich.edu

## Abstract

Interacting agents that interleave planning, plan coordination, and plan execution for hierarchical plans (e.g. HTNs or procedures for PRS) should reason about abstract plans and their concurrent execution before they are fully refined. Poor decisions made at abstract levels can lead to costly backtracking or even failure. We claim that better decisions require information at abstract levels that summarizes the preconditions and effects that must or may apply when a plan is refined. Here we formally characterize concurrent hierarchical plans and a method for deriving summary information for them, and we illustrate how summary conditions can be used to coordinate the concurrent interactions of plans at different levels of abstraction. The properties of summary conditions and rules determining what interactions can or might hold among asynchronously executing plans are proven to support the construction of sound and complete coordination mechanisms for concurrent hierarchical planning agents.

## Introduction

The study of concurrent action in relation to planning (Georgeff 1984) has improved our understanding of how agents can reason about their interactions in order to avoid conflicts during concurrent plan execution. Conflicts can be avoided by reducing or eliminating interactions by localizing plan effects to particular agents (Lansky 1990), and by merging the individual plans of agents by introducing synchronization actions (Georgeff 1983). In fact, planning and merging can be interleaved, such that agents can propose next-step extensions to their current plans and reconcile conflicts before considering extensions for subsequent steps. By formulating extensions in terms of constraints rather than specific actions, a "least commitment" policy can be retained (Ephrati & Rosenschein 1994).

For many applications, planning efficiency can be enhanced by exploiting the hierarchical structure of planning operations. Rather than building a plan from the beginning forward (or end backward), hierarchical planners identify promising classes of long-term activities (abstract plans), and incrementally refine these to eventually converge on specific actions. Planners such as NOAH (Sacerdoti 1977) and NONLIN (Tate 1977) have this character, and are often considered instances of a class of planners called Hierarchical Task Network (HTN) planners. By exploiting the hierarchical task structure to focus search, HTN planners often converge much more quickly to effective plans. They are also becoming increasingly well understood (Erol, Hendler, & Nau 1994).

Using HTN planning for concurrently-executing agents is less well understood, however. If several HTN planning agents are each generating their own plans, how and when should these be merged? Certainly, merging could wait until the plans were fully refined, and techniques like those of Georgeff (mentioned previously) would work. But interleaving planning and merging holds greater promise for identifying and resolving key conflicts as early in the process as possible to try to avoid backtracking or failure. Such interleaving, however, requires the ability to identify potential conflicts among abstract plans.

Corkill (Corkill 1979) studied interleaved planning and merging in a distributed version of the NOAH planner. He recognized that, while most of the conditions affected by an abstract plan operator might be unknown until further refinement, those that deal with the overall effects and preconditions that hold no matter how the operator is refined can be captured and used to identify and resolve some conflicts. He recognized that further choices of refinement or synchronization choices at more abstract levels could lead to unresolvable conflicts at deeper levels, and backtracking could be necessary. Our work is directed toward avoiding such backtracking by improving how an abstract plan operator represents all of the potential needs and effects of all of its potential refinements.

Our motivation for doing this is not simply to make interleaved planning and merging with HTNs more efficient, but also to support another crucial use of HTN concepts—specifically, flexible plan execution systems such as PRS (Georgeff & Lansky 1986), RAPS (Firby 1989), etc., that similarly exploit hierarchical plan spaces. Rather than refine abstract plan operators

---

This work was supported in part by NSF (IRI-9158473) and DARPA (F30602-98-2-0142).

into a detailed end-to-end plan, however, these systems interleave refinement with execution. By postponing refinement until absolutely necessary, such systems leave themselves flexibility to choose refinements that best match current circumstances. However, this means that refinement decisions at abstract levels are made and acted upon before all of the detailed refinements need be made. If such refinements at abstract levels introduce unresolvable conflicts at detailed levels, the system ultimately gets stuck part way through a plan that cannot be completed. While backtracking is possible for HTN planning (since no actions are taken until plans are completely formed), it might not be possible when some (irreversible) plan steps have already been taken. It is therefore critical that the specifications of abstract plan operators be rich enough to summarize all of the relevant refinements to anticipate and avoid such conflicts. In this paper, we formally characterize methods for deriving and exploiting such rich summaries to support interleaved local planning, coordination (plan merging), and execution.

## Simple Example

This example illustrates the use of summary information, explains some terminology, and further motivates the formalism of a theory for concurrent hierarchical plans (CHiPs) and summary information.

Suppose that two agents wish to go through a doorway to another location, (*row*, *column*), as shown in Figure 1. Agent A has a hierarchical plan, $p$, to move from (0,0) to (0,4), and B also has a plan, $q$, to move from (2,0) to (2,4), but they need to coordinate their plans to avoid collision. Agent A could have preprocessed plan $p$ to derive its summary information. The set of *summary preconditions* of $p$ includes all its preconditions and those of its subplans that must be met external to $p$ in order for $p$ to execute successfully: $\{At(A,0,0), \neg At(B,0,1), \neg At(B,1,0), \ldots, \neg At(B,0,4)\}$. The proposition $At(A,0,0)$ is a *must* condition because no matter how $p$ is executed, the condition must hold. $\neg At(B,1,0)$ is *may* because it may be required depending on the path A takes. Likewise, the *summary postconditions* of $p$ are its effects and those of its subplans that are seen externally: $\{At(A,2,0), \neg At(A,0,0), \neg At(A,1,0), \ldots\}$. The *summary inconditions* are any conditions that must hold within the interval of time that the plan is executing and can be *must* or *may* and *always* or *sometimes*. An *always* condition is required to hold throughout the duration of any execution of the plan. For example, a *must*, *always* incondition of $p$ could be $PowerOn(A)$– the power must always be on. $At(A,1,0)$ is a *may*, *sometimes* incondition of $p$ because A *may* choose that path and would only be there at *some time*. These conditions and descriptors, such as *must* and *always*, provide the necessary information to reason about what conditions must or may be achieved or clobbered when ordering a set of plan executions.

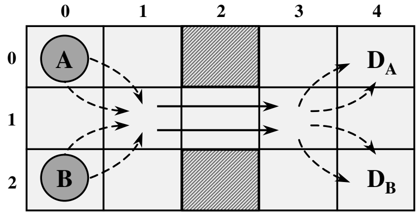


Figure 1: Agents A and B go through a doorway.

Now suppose A sends B $p_{sum}$, the summary information for $p$. Agent B can now reason about the interactions of their plans based on their combined summary information. For instance, based only on the summary information, B can determine that if $p$ is restricted to execute before $q$, then the plans *can* be executed (refined) in *any way*, or $CanAnyWay(\mathrm{b},p_{sum},q_{sum})$.[1] So, B could tell A to go ahead and start execution and to send back a message when $p$ is finished executing. However, B may instead wish to overlap their plan executions for better efficiency. Although $CanAnyWay(\mathrm{o},p_{sum},q_{sum})$ is not true, B could use the summary conditions to determine that there *might* be *some way* to overlap them, or $MightSomeWay(\mathrm{o},p_{sum},q_{sum})$. Then, B could ask A for the summary information of each of $p$'s subplans, reason about the interactions of lower level actions in the same way, and find a way to synchronize the subplans for a more fine-grained solution.

Consider another case where A and B plan to move to the spot directly between them, (1,0), and can choose from different routes. $MightSomeWay(\mathrm{b}, p_{sum}, q_{sum})$ would be false since the postconditions of $p$ must always clobber the preconditions of $q$. If we wanted to describe a rule for determining whether two actions can or might overlap, it is not obvious how this should be done. The difficulty of composing such rules stems from an imprecise specification of concurrent plan execution and the large space of potential plans that have the same summary information. If the $MightSomeWay(\mathrm{o}, p_{sum}, q_{sum})$ rule is not specified in a complete way, the agent may not determine that the *overlaps* relation cannot hold until it has exhaustively checked all synchronizations of $p$ and $q$'s primitive subplans. As the number of subplans grows, this becomes an intractable procedure (Vilain & Kautz 1986). Even worse would be if, for the sake of trying to be complete, a rule is specified in an unsound way leading to a synchronization choice that causes failure. We give an example of this in (Clement & Durfee 1999b), where we also implement a hierarchical plan coordination algorithm that uses summary information in the manner described above. Our evaluations show

[1] We will often abbreviate Allen's thirteen temporal relations (Allen 1983). Here, "b" is for the *before* relation. "o" is for *overlaps*.

that coordinating at different levels of abstraction for different cost scenarios results in better performance. Thus, formalizing concurrent hierarchical plans, their execution, and the derivation of summary conditions is necessary to avoid costly, irreversible decisions made during planning, plan execution, and coordination.

### Overview

In the next section we describe the semantics of hierarchical plans and their concurrent execution to ground our theory. The simple theory of action consistently describes all temporal interactions among primitive or hierarchical plans. We basically add a set of *inconditions* to popular STRIPS-style plan representations to reason about concurrent plan execution. In addition, we formalize traditional planning concepts, such as *clobbers* and *achieves*, and reintroduce *external conditions* (Tsuneto, Hendler, & Nau 1998) for reasoning about CHiPs. We then describe the semantics of plan summary information and a correct method for deriving it efficiently. This, in turn, is used to describe the construction of sound and complete rules for determining how plans can definitely or might possibly be temporally related. The result is a theory for proving correct coordination and planning mechanisms.

## A Model of Hierarchical Plans and their Concurrent Execution

The original purpose of developing the following theory was to provide, as simply as possible, a consistent model of execution to generally reason about the concurrent interaction of hierarchical plans. However, we also wanted the model to share important aspects of plans used by PRSs, HTNs, Allen's temporal plans, and many STRIPS-style plan representations. As such, this theory of action tries to distill appropriate aspects of other theories, including (Allen & Koomen 1983), (Georgeff 1984), and (Fagin *et al.* 1995).

### CHiPs

A concurrent hierarchical plan $p$ is a tuple $\langle pre,\ in,\ post,\ type,\ subplans,\ order\rangle$. $pre(p)$, $in(p)$, and $post(p)$ are sets of literals ($v$ or $\neg v$ for some propositional variable $v$) representing the preconditions, inconditions, and postconditions defined for plan $p$.[2] The *type* of plan $p$, $type(p)$, has a value of either *primitive*, *and*, or *or*. An *and* plan is a non-primitive plan that is accomplished by carrying out all of its subplans. An *or* plan is a non-primitive plan that is accomplished by carrying out one of its subplans. So, *subplans* is a set of plans, and a *primitive* plan's *subplans* is the empty set. $order(p)$ is only defined for an *and* plan $p$ and is a set of temporal relations (Allen 1983) over pairs of subplans that together are consistent; for example, $before(p_i, p_j)$ and $before(p_j, p_i)$ could not both be in *order*. Plans left unordered with respect to each other are interpreted to potentially execute in concurrently. For the example in Figure 1, A's highest level plan $p$ is the tuple $\langle\{\},\ \{\},\ \{\},\ and,\ \{m_1,\ m_2,\ m_3\},\ \{before(m_1,\ m_2),\ before(m_2,\ m_3)\}\rangle$. Here, $m_1$, $m_2$, and $m_3$ correspond to $p$'s subplans for moving to (1,1), (1,3), and (0,4) respectively. There are no conditions defined because $p$ can rely on the conditions defined for the primitive plans for moving between grid locations. The primitive plan for moving agent A from (1,3) to (0,3) is the tuple $\langle\{At(A,1,3)\},\ \{At(A,1,3),\ \neg At(B,1,3),\ \neg At(B,0,3)\},\ \{At(A,0,3),\ \neg At(A,1,3),\ \neg At(B,0,3),\ \neg At(B,1,3)\},\ primitive,\ \{\},\ \{\}\rangle$.

We also require postconditions to specify whether the inconditions change or not. This helps simplify the notion of inconditions as conditions that hold only *during* plan execution whether because they are *caused* by the action or because they are *necessary conditions* for successful execution. If a plan's postconditions did not specify the truth values of the inconditions' variables at the end of execution, then it is not intuitive how those values should be determined in the presence of concurrently executing plans. By requiring postconditions to specify such values, we resolve all ambiguity and simplify state transitions (described in the section below on Histories and Runs).

The decomposition of a CHiP is in the same style as that of an HTN as described by Erol *et al.* (Erol, Hendler, & Nau 1994). An *and* plan is a task network, and an *or* plan is an extra construct representing a set of all tasks that accomplish the same goal or compound task. Tasks in a network are subplans of the plan corresponding to the network. High-level effects (Erol, Hendler, & Nau 1994) are simply the postconditions of a non-primitive CHiP. CHiPs can also represent a variety of interesting procedures executable by PRSs.

### Executions

We recursively describe an execution of a plan as an instance of a decomposition and ordering of its subplans' executions. This helps us reason about the outcomes of different ways to execute a group of plans, describe state transitions, and formalize other terms.

The possible executions of a plan $p$ is the set $\mathcal{E}(p)$. An *execution* of $p$, $e \in \mathcal{E}(p)$, is a triple $\langle d, t_s, t_f\rangle$. $t_s(e)$ and $t_f(e)$ are positive, non-zero real numbers representing the start and finish times of execution $e$, and $t_s < t_f$. $d(e)$ is a set of subplan executions representing the decomposition of plan $p$ under this execution $e$. Specifically, if $p$ is an *and* plan, then it contains one execution from each of the subplans; if it is an *or* plan, then it contains only one execution of one of the subplans; and it is empty if it is *primitive*. In addition, for all subplan executions, $e' \in d$, $t_s(e')$ and $t_f(e')$ must be consistent with the relations specified in $order(p)$. Also, the first subplan(s) to start must

[2] Functions such as $pre(p)$ are used for referential convenience throughout this paper. Here, $pre$ and $pre(p)$ are the same, and $pre(p)$ is read as "the preconditions of $p$."

start at the same time as $p$, $t_s(e') = t_s(e)$; and the last subplan(s) to finish must finish at the same time as the $p$, $t_f(e') = t_f(e)$. An execution for agent A's top-level plan $p$ (described previously in the section on CHiPs) would be some $e \in \mathcal{E}(p)$. $e$ might be $\langle\{e_1, e_2, e_3\}, 4.0, 10.0\rangle$ where $e_1 \in \mathcal{E}(m_1)$, $e_2 \in \mathcal{E}(m_2)$, $e_3 \in \mathcal{E}(m_3)$, and $e$ begins at time 4.0 and ends at time 10.0. $e_1$ also starts at 4.0, and $e_3$ ends at 10.0.

The *subexecutions* of an execution $e$, sometimes referred to as $subex(e)$, is defined recursively as the set of subplan executions in $e$'s decomposition unioned with their subexecutions. For agent A, $subex(e) = \{e_1, e_2, e_3\} \cup subex(e_1) \cup subex(e_2) \cup subex(e_3)$. For convenience, we say that a condition of a plan with an execution in the set containing $e$ and $e$'s subexecutions is a *condition of* $e$. So, if A executes its top-level plan, since $\neg At(B,1,2)$ is an incondition of the primitive for A to move from (1,1) to (1,2), it is also an incondition of the primitive's execution, $e_2$, and $e$.

## Histories and Runs

We describe hypothetical possible worlds, called histories, so that we can determine what happens in all worlds, some, or none. We then can describe how the state transforms according to a particular history. A state of the world, s, is a truth assignment to a set of propositions, each representing an aspect of the environment. We treat a state as the set of true propositional variables.

A *history*, $h$, is a tuple $\langle E, s_I\rangle$. $E$ is a set of plan executions including those of all plans and subplans executed by all agents, and $s_I$ is the initial state of the world before any plan is begun. So, a history $h$ is a hypothetical world that begins with $s_I$ as the initial state and where only executions in $E(h)$ occur.

A *run*, $r$, is a function mapping time to states. It gives a complete description of how the state of the world evolves over time. We take time to range over the positive real numbers. $r(t)$ denotes the state of the world at time $t$ in run $r$. So, a condition is *met* at time $t$ if the condition is a non-negated propositional variable $v$, and $v \in r(t)$ or if the condition is a negated propositional variable $\neg v$, and $v \notin r(t)$.

For each history $h$ there is exactly one run, $r(h)$[3], that specifies the state transitions *caused* by the plan executions in $E(h)$. The interpretation of a history by its run is defined as follows. The world is in the initial state at time zero: $r(h)(0) = s_I(h)$. In the smallest interval after any point where one or more executions start and before any other start or end of an execution, the state is updated by adding all non-negated inconditions of the plans and then removing all negated inconditions (as specified in Axiom 1 below). Similarly, at the point where one or more executions finish, the state is updated by adding all non-negated postconditions of the plans and then removing all negated postconditions (as specified in Axiom 2). Lastly, if no execution of a plan begins or ends between two points in time, then the state must be the same at those points (Axiom 3).

**Axiom 1**

$$\begin{aligned}
&\forall h, e_1, t, e_1 \in E(h) \wedge \\
&\quad [t > t_s(e_1) \wedge (\forall e_2 \in E(h), (t_s(e_2) > t_s(e_1) \rightarrow \\
&\quad t \leq t_s(e_2)) \wedge (t_f(e_2) > t_s(e_1) \rightarrow t < t_f(e_2)))] \rightarrow \\
&\quad r(h)(t) = r(h)(t_s(e_1)) \cup \{v \mid \exists p, e_3 \in E(h), \\
&\qquad e_3 \in \mathcal{E}(p) \wedge t_s(e_3) = t_s(e_1) \wedge v \in in(p)\} - \\
&\qquad \{v \mid \exists p, e_3 \in E(h), e_3 \in \mathcal{E}(p) \wedge \\
&\qquad t_s(e_3) = t_s(e_1) \wedge \neg v \in in(p)\}.
\end{aligned}$$

**Axiom 2**

$$\begin{aligned}
&\forall h, e_1, t, e_1 \in E(h) \wedge \\
&\quad [t < t_f(e_1) \wedge (\forall e_2 \in E(h), (t_s(e_2) < t_f(e_1) \rightarrow \\
&\quad t > t_s(e_2)) \wedge (t_f(e_2) < t_f(e_1) \rightarrow t \geq t_f(e_2)))] \rightarrow \\
&\quad r(h)(t_f(e_1)) = r(h)(t) \cup \{v \mid \exists p, e_3 \in E(h), \\
&\qquad e_3 \in \mathcal{E}(p) \wedge t_f(e_3) = t_f(e_1) \wedge v \in post(p)\} - \\
&\qquad \{v \mid \exists p, e_3 \in E(h), e_3 \in \mathcal{E}(p) \wedge \\
&\qquad t_f(e_3) = t_f(e_1) \wedge \neg v \in post(p)\}.
\end{aligned}$$

**Axiom 3**

$$\begin{aligned}
&\forall h, t_1, t_2, t_1 < t_2 \wedge (\neg \exists e \in E(h), t_1 \leq t_s(e) < t_2 \\
&\quad \vee t_1 < t_f(e) \leq t_2) \rightarrow Env(t_2) = Env(t_1).
\end{aligned}$$

Now we can define what it means for a plan to execute successfully. An execution $e = \langle d, t_s, t_f\rangle$ *succeeds in* $h$ if and only if the plan's preconditions are met at $t_s$; the inconditions are met throughout the interval $(t_s, t_f)$; the postconditions are met at $t_f$; and all executions in $e$'s decomposition are in $E(h)$ and succeed. Otherwise, $e$ *fails*. So, in a history $h$ where agent A successfully executes a plan (as described previously in the section on Executions) to traverse the room, $E(h) = \{e\} \cup subex(e)$, and all conditions of all plans with executions in $E(h)$ are met at the appropriate times. Given the example primitive conditions in the section on CHiPs and the axioms just described for state transitions, if agent B happened to start moving into A's target location, (0,4), at the same time as A, then either A's primitive plan execution $e_A$ finishes before B's and the $\neg At(A,0,4)$ incondition of B's primitive execution $e_B$ is not met (clobbered) at $t_f(e_A)$; $e_B$ finishes before $e_A$ and similarly clobbers $e_A$'s incondition; or they both finish simultaneously clobbering each others' $At(A/B,0,4)$ postconditions. If $e_A$ fails, then $e_3$ and the top-level execution $e$ must also fail.

## Asserting, Clobbering, Achieving, and Undoing

In conventional planning, we often speak of *clobbering* and *achieving* preconditions of plans (Weld 1994). In CHiPs, these notions are slightly different since inconditions can clobber and be clobbered, as seen in the previous section. Formalizing these concepts and another, *undoing* postconditions, helps prove properties of summary conditions. However, it will be convenient to define first what it means to *assert* a condition.

[3] For convenience, we now treat $r$ as a function mapping histories to runs, so $r(h)(t)$ is a mapping of a history and a time to a state.

An execution $e$ of plan $p$ is said to *assert* a condition $\ell$ at time $t$ in a history $h$ if and only if $\ell$ is an incondition of $p$, $t$ is in the smallest interval beginning after $t_s(e)$ and ending before a following start or finish time of any execution in $E(h)$, and $\ell$ is satisfied by $r(h)(t)$; or $\ell$ is a postcondition of $p$, $t = t_f(e)$, and $\ell$ is satisfied by $r(t)$. So, asserting a condition only *causes* it to hold if the condition was not previously met. Otherwise, the condition was already satisfied and the action requiring it did not really *cause* it.

A *precondition* $\ell$ of plan $p_1$ is [*clobbered*, *achieved*][4] in $e_1$ (an execution of $p_1$) by $e_2$ (an execution of plan $p_2$) at time $t$ if and only if $e_2$ asserts [$\ell'$, $\ell$] at $t$; $\ell \Leftrightarrow \neg\ell'$; and $e_2$ is the last execution to assert $\ell$ or $\ell'$ before or at $t_s(e_1)$. An [*incondition*, *postcondition*] $\ell$ of plan $p_1$ is *clobbered* in $e_1$ by $e_2$ at time $t$ if and only if $e_2$ asserts $\ell'$ at $t$; $\ell \Leftrightarrow \neg\ell'$; and $[t_s(e_1) < t < t_f(e_1)$, $t = t_f(e_1)]$. Achieving inconditions and postconditions does not make sense for this formalism, so it is not defined. In the previous section when $e_A$ finished first and asserted $At(A,0,4)$, it clobbered the incondition $\neg At(A,0,4)$ of B's primitive plan in $e_B$ at $t_f(e_A)$.

A *postcondition* $\ell$ of plan $p_1$ is *undone* in $e_1$ (an execution of $p_1$) by $e_2$ (an execution of plan $p_2$) at time $t$ if and only if $e_2$ asserts $\ell'$ at $t$; $\ell \Leftrightarrow \neg\ell'$; and $e_2$ is the first execution to assert $\ell$ or $\ell'$ at or after $t_f(e_1)$.

### External Conditions

As recognized in (Tsuneto, Hendler, & Nau 1998), external conditions are important for reasoning about potential refinements of abstract plans. Although the basic idea is the same, we define them a little differently and call them *external preconditions* to differentiate them from other conditions we call *external postconditions*. Intuitively, an external precondition of a group of partially ordered plans is a precondition of one of the plans that is not achieved by another in the group and must be met external to the group. External postconditions, similarly, are those that are not undone by plans in the group and are net effects of the group.

Formally, an *external precondition* $\ell$ of an *interval* $(t_1, t_2)$ in history $h$ is a precondition of a plan $p$ with some execution $e \in E(h)$ for which $t_1 \le t_s(e) < t_2$, and $\ell$ is neither achieved nor clobbered by an execution at a time $t$ where $t_1 \le t \le t_s(e)$. An *external precondition* of an *execution* $e = \langle d, t_s, t_f\rangle$ is an external precondition of an interval $(t_1, t_2)$ in some history where $t_1 \le t_s$; $t_f \le t_2$; and there are no other plan executions other than the subexecutions of $e$. An *external precondition* of a *plan* $p$ is an external precondition of any of $p$'s executions. It is called a *must* precondition if it is an external precondition of all executions; otherwise it is called a *may* precondition. $At(A,0,0)$ is an external precondition of agent A's top-level plan $p$ (Figure 1) since no subplan in $p$'s hierarchy achieves $At(A,0,0)$. $At(A,1,1)$ is not an external precondition of $p$ because it is achieved internally by the execution of subplan $m_1$ (described in the section on CHiPs).

Similarly, an *external postcondition* $\ell$ of an *interval* $(t_1, t_2)$ in $h$ is a postcondition of a plan $p$ with some execution $e \in E(h)$ for which $t_1 \le t_f(e) \le t_2$; $\ell$ is asserted by $e$; and $\ell$ is not undone by any execution at a time $t$ where $t_f(e) < t \le t_2$. External postconditions of executions and plans can be defined in the same way as external preconditions. $At(A,0,4)$ is an external postcondition of agent A's top-level plan $p$ since no subplan in $p$'s hierarchy cancels the effect. $At(A,1,3)$, an external postcondition of $m_2$ is not an external postcondition of $p$ because it is cancelled internally by the execution of subplan $m_3$ when it later asserts $\neg At(A,1,3)$.

## Plan Summary Information

With the previous formalisms, we can now define summary information and describe a method for computing it for non-primitive plans. The *summary information* for a plan $p$ is $p_{sum}$. Its syntax is given as a tuple $\langle pre_{sum}, in_{sum}, post_{sum}\rangle$, whose members are sets of *summary conditions*. The *summary* [*pre*, *post*] *conditions* of $p$, $[pre_{sum}(p), post_{sum}(p)]$, contain the external [pre, post] conditions of $p$. The *summary inconditions* of $p$, $in_{sum}(p)$, contain all conditions that must hold within some execution of $p$ for it to be successful. A condition $c$ in one of these sets is a tuple $\langle \ell, existence, timing\rangle$. $\ell(c)$ is a literal. The *existence* of $c$ can be *must* or *may*. If $existence(c) = must$, then $c$ is called a *must* condition because $\ell$ holds for every successful plan execution ($\ell$ *must* hold). For convenience we usually write $must(c)$. $c$ is a *may* condition ($may(c)$ is *true*) if there is at least one plan execution where $\ell(c)$ must hold. The *timing* of $c$ can take the values *always*, *sometimes*, *first*, *last*. $timing(c)$ is *always* for $c \in in_{sum}$ if $\ell(c)$ is an in-condition that must hold throughout the execution of $p$ ($\ell$ holds *always*); otherwise, $timing(c) = sometimes$ meaning $\ell(c)$ holds at one point, at least, within an execution of $p$. The *timing* is *first* for $c \in pre_{sum}$ if $\ell(c)$ holds at the beginning of an execution of $p$; otherwise, $timing = sometimes$. Similarly, *timing* is *last* for $c \in post_{sum}$ if $\ell(c)$ holds at the end of an execution of $p$; otherwise, it is *sometimes*. Although *existence* and *timing* syntactically only take one value, semantically $must(c) \Rightarrow may(c)$, and $always(c) \Rightarrow sometimes(c)$. See the Introduction for an example of summary conditions derived for an abstract plan.

### Deriving Summary Conditions

The method for deriving the summary conditions of a plan $p$ is recursive. First, summary information must be derived for each of $p$'s subplans, and then the following procedure derives $p$'s summary conditions from those of its subplans and its own sets of conditions. This procedure only applies to plans whose expansion

[4] We use braces [ ] as a shorthand when defining similar terms and procedures. For example, saying "[$a$, $b$] implies [$c$, $d$]" means $a$ implies $c$, and $b$ implies $d$.

is finite and which may *or may not* have the downward solution property. This procedure is adapted from that in (Clement & Durfee 1999a) that only accounts for plans having the downward solution property. This is the property where every *or* plan in the hierarchy can be refined successfully through one or more of its subplans.

Summary conditions for primitives and non-primitives

- First, for each literal $\ell$ in $pre(p)$, $in(p)$, and $post(p)$, add a condition $c$ with literal $\ell$ to the respective set of summary conditions for plan $p$. $existence(c)$ is $must$, and $timing(c)$ is $first$, $always$, or $last$ if $\ell$ is a pre-, in-, or postcondition respectively.

Summary [pre, post] conditions for *and* plan

- Add a condition $c$ to the summary [pre, post] conditions of *and* plan $p$ for each summary [pre, post] condition $c'$ of $p$'s subplans that is not [*must-achieved*, *must-undone*] or *must-clobbered*[5] by another of $p$'s subplans, setting $\ell(c) = \ell(c')$.[6]
- Set $existence(c) = must$ if $\ell(c)$ is a [pre, post] condition of $p$ or is the literal of a $must$ summary [pre, post] condition in a subplan of $p$ that is not [*may-achieved*, *may-undone*] or *may-clobbered* by any other subplans. Otherwise, set $existence(c) = may$.
- Set $timing(c) = [first, last]$ if $\ell(c)$ is a [pre, post] condition of $p$ or the literal of a [$first$, $last$] summary [pre, post] condition of a [least, greatest] temporally ordered subplan (i.e. no others are constrained by $order(p)$ to [begin before, end after] it).[5] Otherwise, set $timing(c) = sometimes$.

Summary [pre, post] conditions for *or* plan

- Add a condition $c$ to the summary [pre, post] conditions of *or* plan $p$ for each summary [pre, post] condition $c'$ in $p$'s subplans, setting $\ell(c) = \ell(c')$.
- Set $existence(c) = must$ if $\ell(c)$ is a [pre, post] condition of $p$ or a $must$ summary [pre, post] condition of all of $p$'s subplans. Otherwise, set $existence(c) = may$.
- Set $timing(c) = [first, last]$ if $\ell(c)$ is a [pre, post] condition of $p$ or the literal of a [$first$, $last$] summary [pre, post] condition in a subplan. Otherwise, set $timing(c) = sometimes$.

Summary inconditions for *and* plan

- Add a condition $c$ to the summary inconditions of *and* plan $p$ for each $c'$ in $C$ defined as the set of summary inconditions of $p$'s subplans unioned with the set of summary preconditions of the subplans that are not always $first$ in a least temporally ordered subplan and with the set of summary postconditions of the subplans that are not always $last$ in a greatest temporally ordered subplan[5], and set $\ell(c) = \ell(c')$.
- Set $existence(c) = must$ if $\ell(c)$ is an incondition of $p$ or a literal of a $must$ summary condition $c' \in C$, as defined above, and is always not a $first$ or $last$ condition.[5] Otherwise, set $existence(c) = may$.
- Set $timing(c) = always$ if $\ell(c)$ is an incondition of $p$ or a literal in an $always$ summary incondition in every subplan of $p$, and the intervals within the subplan executions must *cover* the interval within the execution of $p$.[5] Otherwise, set $timing(c) = sometimes$.

Summary inconditions for *or* plan

- Add a condition $c$ to the summary inconditions of *or* plan $p$ for for each summary incondition $c'$ in $p$'s subplans, setting $\ell(c) = \ell(c')$.
- Set $existence(c) = must$ if $\ell(c)$ is an incondition of $p$ or a $must$ summary incondition of all of $p$'s subplans. Otherwise, set $existence(c) = may$.
- Set $timing(c) = always$ if $\ell(c)$ is an incondition of $p$ or an $always$ summary incondition of all of $p$'s subplans. Otherwise, set $timing(c) = sometimes$.

Consider deriving the summary conditions of $m_2$ from its two primitive subplans (as introduced in the section on CHiPs). Suppose $p_1$ is the primitive subplan for moving agent A from (1,1) to (1,2), and $p_2$ moves A from (1,2) to (1,3). First, the summary conditions of the primitives must be derived. These are simply the conditions already defined for the primitives according to the first step of the procedure. $m_2$ has no conditions defined for itself, so all will come from $p_1$ and $p_2$. Since $m_2$ is an *and* plan, its only summary precondition is $At(A, 1, 1)$ from $p_1$ because $p_1$ *must achieve* $p_2$'s only precondition $At(A, 1, 2)$. $At(A, 1, 1)$ is a $must$ summary condition because it is a $must$ summary precondition in $p_1$, and no other subplan ($p_2$) *may achieve* $At(A, 1, 1)$. $At(A, 1, 1)$ is also $first$ because it is a $first$ summary precondition of $p_1$, and $p_1$ precedes $p_2$.

## Supporting Mechanisms

The procedure above ensures that external conditions are captured by summary conditions and $must$, $always$, $first$, and $last$ have their intended meanings. To prove this, we must first give definitions of several concepts that specify details of the procedure for deriving summary conditions and then prove that algorithms for deciding the truth of these concepts within their appropriate contexts are in accordance with their definitions. However, the algorithms for determining the following relations rely on the correctness of the procedure for deriving summary information. To avoid a circular argument, we shall assume that the summary information is correct for the set of plans over which these relations are defined, and when we give the proof of the properties of summary information, we will be able to use the following formalisms since we also assume the properties of summary information for the set of subplans in the inductive step. These concepts will also aid in proving the soundness and completeness of methods for determining whether certain temporal relations can or might hold among plans.

[5] See the proofs ending this section about how to determine *must-achieved*, *may-achieved*, *must-undone*, *may-undone*, *must-clobbered*, and *may-clobbered* as well as those about determining least and greatest temporally ordered subplans, always $first/last$ conditions, and interval covering.

[6] To resolve ambiguity with set membership, we say that any two summary conditions, $c$ and $c'$ are equal if $\ell(c) = \ell(c')$ and they belong to the same set of summary conditions for some plan.

**Summarizing, requiring, and attempting to assert summary conditions** A summary condition $c$ *summarizes* a *condition* $\ell$ in *condition set conds* of *plan* $p$ iff $c$ was added by the procedure for deriving summary information to a summary condition set of $p'$; $\ell = \ell(c)$; and either $c$ was added for $\ell$ in a condition set *conds* of $p = p'$, or $c$ was added for a summary condition of a subplan of $p'$ that summarizes $\ell$ in *conds* of $p$.

An execution $e$ of $p$ *requires* a *summary condition* $c$ *to be met at* $t$ iff $c$ is a summary condition in $p$'s summary information; there is a condition $\ell$ in a condition set *conds* of $p'$ that is summarized by $c$; if $first(c)$, $t = t_s(e)$; if $last(c)$, $t = t_f(e)$; if $always(c)$, $t$ is within $(t_s(e), t_f(e))$; and if $sometimes(c)$, there is an execution of a subplan of $p$ in $d(e)$ that requires a summary condition $c'$ to be met at $t$, and $c'$ summarizes $\ell$ in *conds* of $p'$.

Similarly, an execution $e$ of $p$ *attempts to assert* a *summary condition* $c$ at $t$ iff $c$ is a summary condition in $p$'s summary information; there is a condition $\ell$ in a condition set *conds* of $p'$ that is summarized by $c$; $\neg first(c)$; if $always(c)$, $t$ is in the smallest interval after $t_s(e)$ and before the start or end of any other execution that follows $t_s(e)$; if $last(c)$, $t = t_f(e)$; and if $sometimes(c)$, there is an execution of a subplan of $p$ in $d(e)$ that attempts to assert a summary condition $c'$ at $t$; and $c'$ summarizes $\ell$ in *conds* of $p'$.

**Definitions and algorithms for must/may asserting summary conditions and other things** The following definitions and algorithms are within the context of a set of plans $P$ with a corresponding set of summary information $P_{sum}$, a set of ordering constraints *order*, and a set of histories $H$ including all histories where $E(h)$ only includes an execution $e$ of each plan in $P$ and $e$'s subexecutions, and $E(h)$ satisfies all constraints in *order*. They are all concerned with the ordering of plan execution intervals and the timing of conditions. By themselves, they do not have anything to do with whether conditions may or must be met for a plan execution.

The algorithms for determining whether the defined relations hold between summary conditions for plans in $P$ use a point algebra constraint table as described in (Vilain & Kautz 1986). The point algebra table is constructed for the intervals corresponding to the executions of the plans in $P$—a row and column for both $p^-$ and $p^+$ are added for each plan $p \in P$. Then, the constraints in *order* are added to the table, and the transitive closure is computed to get all constraints entailed from those in *order*. This only needs to be done once for any $P$ and *order*. For *must-assert* relations, each constraint specified in an entry of the case tables must be a subset of the constraints in the point algebra table in order for the relation to hold for that case. For *may-assert* relations, each constraint specified in an entry of the case tables must be missing from the constraints in the point algebra table in order for the relation to hold for that case. These inferences using the point algebra table are shown to be sound and complete in (Vilain & Kautz 1986).

We determine that a plan $q$ in $p$'s subplans is temporally ordered [*least*, *greatest*] if and only if [$q^-$, $q^+$] is constrained [before, after] or equal to all other points in the point algebra table for $p$'s subplans. This is done by looking at each entry in the row for [$q^-$, $q^+$] and checking to see that the constraint is [$<$, $>$], $=$, or [$\leq$, $\geq$]. If this is not the case, then $q$ is not always [$first$, $last$]. A summary [pre, post] condition of plan $q$ in $p$'s subplans is *always* [$first$, $last$] if and only if $q$ is a [least, greatest] temporally ordered subplan. A summary [pre, post] condition of plan $q$ in $p$'s subplans is *always not* [$first$, $last$] if and only if in the row for [$q^-$, $q^+$] there is an entry with the [$>$, $<$] constraint.

We also determine that an *interval* $i_0$ is *covered* by a set of intervals $I = \{i_1, i_2, \ldots, i_k\}$ according to a conjunction of constraints *order* by constructing a point algebra table for all of these intervals, adding the constraints in *order* to the table, and computing the transitive closure. After this, we can check to see if $i_0$ is covered looking at all pairs of intervals to see if they overlap. If there are any intervals that have an endpoint that is only contained by $i_0$ and does not meet the opposite side of another interval, or there are no intervals overlapping $i_0$, then $i_0$ is not covered. Otherwise, it is.

For the following definitions and algorithms of must-assert, we assume $c$ and $c'$ are summary conditions of plans in $P$.

$p' \in P$ *must-assert* $c'$ [*by*, *before*] $c$ iff for *all* histories $h \in H$ and all $t$ where $e$ is the top-level execution in $E(h)$ of some $p \in P$ that requires $c$ to be met at $t$, and $e'$ is the top-level execution of $p'$ in $E(h)$, there is a $t'$ where $e'$ attempts to assert $c'$ at $t'$, and [$t' \leq t$, $t' < t$].

We determine that $p' \in P$ *must-assert* $c'$ [*by*, *before*] $c$ if and only if for the row of Table 1 describing $c'$ and $c$, all of the corresponding ordering constraints are imposed by (can be deduced from) *order* as described above using the point algebra table.

**Lemma** This algorithm is sound for the definition of must-assert; i.e. if we determine that $p' \in P$ *must-assert* $c'$ [*by*, *before*] $c$, then such is the case.

**Proof** by contradiction. If it were not the case that $p' \in P$ must-assert $c'$ [by, before] $c$, then there must be a history where for all $t'$ where $e'$ attempts to assert $c'$ at $t'$, there is a $t$ where $e$ requires $c$ to be met at $t$ and [$t' > t$, $t' \geq t$] since we assume that there exists a plan $p \in P$ for which there is a summary condition $c$; thus, there must be some $t$ where $e$ requires $c$ to be met at $t$. We need to show that this is false for any $c$ and $c'$. We claim that the table describing the constraints checked by the algorithm does this as it describes all cases of assertions by summary inconditions and postconditions and all cases of required summary conditions. For rows 1-4 in the table, the algorithm finds that the constraint [$p'^+ \leq p^-$, $p'^+ < p^-$]

| | | | $p'$ must-assert $c'$ by $c$ | $p'$ must-assert $c'$ before $c$ |
|---|---|---|---|---|
| | $c' \in post(p')$ | $c \in pre(p)$ | *order* must impose these constraints | *order* must impose these constraints |
| | $last$ | $first$ | | |
| 1 | T | T | $p'^{+} \leq p^{-}$ | $p'^{+} < p^{-}$ |
| 2 | T | F | $p'^{+} \leq p^{-}$ | $p'^{+} < p^{-}$ |
| 3 | F | ? | $p'^{+} \leq p^{-}$ | $p'^{+} < p^{-}$ |
| 4 | ? | ? | $p'^{+} \leq p^{-}$ | $p'^{+} < p^{-}$ |
| | $c' \in in(p')$ | | | |
| | $always$ | | | |
| 5 | T | T | $p'^{-} < p^{-}$ | $p'^{-} < p^{-}$ |
| 6 | T | F | $p'^{-} \leq p^{-}$ | $p'^{-} \leq p^{-}$ |
| 7 | F | ? | $false$ | $false$ |
| | $c' \in post(p')$ | $c \in in(p)$ | | |
| | $last$ | $always$ | | |
| 8 | T | ? | $p'^{+} \leq p^{-}$ | $p'^{+} \leq p^{-}$ |
| 9 | F | ? | $p'^{+} \leq p^{-}$ | $p'^{+} \leq p^{-}$ |
| | $c' \in in(p')$ | | | |
| | $always$ | | | |
| 10 | T | ? | $p'^{-} \leq p^{-}$ | $p'^{-} < p^{-}$ |
| 11 | F | ? | $false$ | $false$ |
| | $c' \in post(p')$ | $c \in post(p)$ | | |
| | $last$ | $last$ | | |
| 12 | T | T | $p'^{+} \leq p^{+}$ | $p'^{+} < p^{+}$ |
| 13 | T | F | $p'^{+} \leq p^{-}$ | $p'^{+} \leq p^{-}$ |
| 14 | F | T | $p'^{+} \leq p^{+}$ | $p'^{+} \leq p^{+}$ |
| 15 | F | F | $p'^{+} \leq p^{-}$ | $p'^{+} \leq p^{-}$ |
| | $c' \in in(p')$ | | | |
| | $always$ | | | |
| 16 | T | T | $p'^{-} < p^{+}$ | $p'^{-} < p^{+}$ |
| 17 | T | F | $p'^{-} \leq p^{-}$ | $p'^{-} \leq p^{-}$ |
| 18 | F | T | $false$ | $false$ |
| 19 | F | F | $false$ | $false$ |

Table 1: Table for *must-assert by/before* algorithm

can be derived from *order*. Since all histories must have to meet the constraints in *order*, $[t_f(e') \leq t_s(e),\ t_f(e') < t_s(e)]$ must be true. Because the summary postconditions of $p'$ are derived from its own postconditions or the summary postconditions of its subplans; and the summary preconditions of $p'$ are derived from its own preconditions or the summary preconditions of its subplans, these summary conditions are ultimately derived from postconditions of plans executed within $e'$ and preconditions of plans executed within $e$ according to the semantics of subplan executions. This means that $t_s(e') < t' \leq t_f(e')$ and $t_s(e) \leq t < t_f(e)$, and $[t' > t,\ t' \geq t]$ is, thus, false. Therefore the rules for rows 1-4 are sound in determining must-assert for their respective cases. [7] For row 5 by similar argument, $t_s(e') < t_s(e)$. As explained at the beginning of this section on Supporting Mechanisms, we assume that the summary conditions of $p$, $p'$, and all other subplans in $P$ have their intended properties, so since $c' \in in_{sum}(p')$, $always(c)$, and $e'$ attempts to assert $c'$ at $t'$, there is such $t'$ where $t_s(e') < t'$, and $t'$ is less than all $t_s > t_s(e')$ and all $t_f > t_s(e')$ for any execution in $E(h)$ including $e$ by the semantics of subplan executions and *always*. Similarly, $t_s(e) \leq t < t_f(e)$ since $\ell(c)$ is an external precondition of $p$, and $e$ requires $c$ to be met at $t$. Thus, $[t' > t,\ t' \geq t]$ is false. In this manner, all rows of the table used by the algorithm have been verified to show that $[t' > t,\ t' \geq t]$ is false for all cases and are, thus, sound. □

**Lemma** This algorithm is complete for the definition of must-assert; i.e. if $p' \in P$ must-assert $c'$ [by, before] $c$, then the algorithm determines this to be true.

**Proof** by contradiction. If the algorithm did not determine that $p' \in P$ must-assert $c'$ [by, before] $c$, then it must have found the constraints in the must-assert table to differ from those in the point algebra table. So, we need to show that that could not be the case if $p' \in P$ actually must-assert $c'$ [by, before] $c$ for any $c'$ and $c$. For rows 1-4 in the table, the algorithm finds that the constraint $[p'^{+} \leq p^{-},\ p'^{+} < p^{-}]$ cannot be proven from *order*. Since all histories have to meet the constraints in *order*, $[t_f(e') > t_s(e),\ t_f(e') \geq t_s(e)]$ must be true for some histories. However, $e'$ attempts to assert $c'$ at $t'$ such that $t' = t_f(e')$, and $e$ requires $c$ to be met at $t$ where $t = t_s(e)$, so $[t' > t,\ t' \geq t]$ in such histories. This contradicts the assumption that $p'$ must-assert $c'$ [by, before] $c$, so the rules in rows 1-4 in the table are complete in determining must-assert for their respective cases. For row 5 by similar argument, $t_s(e') \geq t_s(e)$ for some histories. However, $e'$ attempts to assert $c'$ at $t'$ such that $t_s(e') < t'$, and $e$ requires $c$ to be met at $t = t_s(e)$. Thus, $t < t'$; the definition of must-assert is again contradicted; and the algorithm is complete for this case. In rows 7, 11, 18, and 19, $c'$ is

[7] Notice that whether $c$ and $c'$ are *must* or *may* does not make a difference since must-assert describes a temporal relation among executions only for cases when the conditions must be met in the executions of their respective plans.

*sometimes* incondition, which could be summarizing only some precondition of a plan in $p'$'s decomposition. In this case no plan can attempt to assert $c'$, so must-assert is false, contradicting the assumption that it was true. [8] In this manner, all rows of the table used by the algorithm have been verified to contradict the assumption for all cases and are, thus, complete. □

$p' \in P$ *may-assert* $c'$ [*by, before*] $c$ iff for *some* history $h \in H$, there exists a $p \in P$, $e$, $e'$, $t$, and $t'$ such that $e$ is the top-level execution of $p$ in $E(h)$; $e'$ is top-level execution of $p'$ in $E(h)$; $e'$ attempts to assert $c'$ at $t'$; $e$ attempts to assert $c$ at $t$, and $[t' \leq t, t' < t]$.

We determine that $p' \in P$ *may-assert* $c'$ [*by, before*] $c$ if and only if for the row of Table 2 describing $c'$ and $c$, none of the corresponding ordering constraints are imposed by (can be deduced from) *order* as described above using the point algebra table.

**Lemma** This algorithm is sound for the definition of may-assert; i.e. if we determine that $p' \in P$ may-assert $c'$ [by, before] $c$, then such is the case.

**Proof** by contradiction. If it were not the case that $p' \in P$ may-assert $c'$ [by, before] $c$, then there is no history where $e$ is the top-level execution of $p$ in $E(h)$; $e'$ is top-level execution of $p'$ in $E(h)$; $e'$ attempts to assert $c'$ at $t'$; $e$ requires $c$ to be met at $t$; and $[t' \leq t, t' < t]$. Since our algorithm assumes that there exists a plan $p \in P$ for which there is a summary condition $c$, we know that in all histories there is a top-level execution of $p$, $e$, in $E(h)$ and a top-level execution of $p'$, $e'$, in $E(h)$, and we know that in some history for some set of plans with summary information $P_{sum}$, $e'$ attempts to assert $c'$ at $t'$, and $e$ attempts to assert $c$ at $t$. Thus, $[t' > t, t' \geq t]$ for any such history. So, we must find a contradiction by showing that $[t' \leq t, t' < t]$ for any $c$ and $c'$ in some history using the corresponding cases in the table used by the algorithm. For row 1 in the table, the algorithm would find that the constraint $[p'^{+} > p^{-}, p'^{+} \geq p^{-}]$ cannot be derived from *order*. Since all histories must have to meet the constraints in *order*, $[t_f(e') \leq t_s(e), t_f(e') < t_s(e)]$ must be true for some history. By the same argument in the soundness proof for must-assert, $t' = t_f(e')$ and $t = t_s(e)$. However, this means that in $[t' \leq t, t' < t]$ in $h$, so we have a contradiction for this case. For row 7 by similar argument, $[t_s(e') \leq t_s(e), t_s(e') < t_s(e)]$ for some history $h$. Because $c' \in in_{sum}(p')$, $c \in in_{sum}(p)$, and $always(c)$, $t_s(e') < t' < t_f(e')$, $t_s(e) < t$, and $t$ is less than $t_s > t_s(e)$ and $t_f > t_s(e)$ for any execution in $E$. Thus, there must be a history where $t' < t$ for any such $c$ and $c'$. In this manner, all rows of the table used by the algorithm have been verified to show that $[t' \leq t, t' < t]$ for all cases and are, thus, sound. □

**Lemma** This algorithm is complete for the definition of may-assert; i.e. if $p' \in P$ may-assert $c'$ [by, before] $c$, then the algorithm determines this to be true.

**Proof** by contradiction. If the algorithm did not determine that $p' \in P$ may-assert $c'$ [by, before] $c$, then it must have found the constraints in the may-assert table to differ from those in the point algebra table. So, we need to show that that could not be the case if $p' \in P$ actually may-assert $c'$ [by, before] $c$. For row 1 in the table, the algorithm finds that the constraint $[p'^{+} > p^{-}, p'^{+} \geq p^{-}]$ is actually proved by *order*. Since all histories must have to meet the constraints in *order*, $[t_f(e') > t_s(e), t_f(e') \geq t_s(e)]$ must be true for all histories. But, since $last(c')$ and $first(c)$, $t' = t_f(e')$ and $t = t_s(e)$, so $[t' > t, t' \geq t]$ for all histories, contradicting the definition of may-assert [by, before]. Thus, the rule in row 1 in the table is complete in determining may-assert [by, before] for this case. For row 7 by similar argument, $[t_s(e') > t_s(e), t_s(e') \geq t_s(e)]$ for all histories. But, $t_s(e') < t' < t_f(e')$, $t_s(e) < t$, and $t$ is less than $t_s > t_s(e)$ and $t_f > t_s(e)$ for any execution in $E$. Thus, again $[t' > t, t' \geq t]$ for all histories, and we have a contradiction. In this manner, all rows of the table used by the algorithm have been verified to contradict the assumption for all cases and are, thus, complete. □

$p' \in P$ *must-assert* $c'$ *in* $c$ iff $always(c)$, and for *all* histories $h \in H$ and all $t$ where there's a plan $p \in P$ such that $c \in in_{sum}(p)$; $e$ is the top-level execution in $E(h)$ of $p$ that requires $c$ to be met at $t$; and $e'$ is the top-level execution of $p'$ in $E(h)$, there is a $t'$ where $e'$ attempts to assert $c'$ at $t'$, and $t_s(e) < t' < t_f(e)$.

We determine that $p' \in P$ *must-assert* $c'$ *in* $c$ if and only if for the row of Table 3 describing $c'$ and $c$, all of the corresponding ordering constraints are imposed by (can be deduced from) *order* as described above using the point algebra table.

**Lemma** This algorithm is sound for the definition of must-assert; i.e. if we determine that $p' \in P$ must-assert $c'$ in $c$, then such is the case.

**Proof** by contradiction. If it were not the case that $p' \in P$ must-assert $c'$ in $c$, then there must be a history where for all $t'$ where $e'$ attempts to assert $c'$ at $t'$, there is a $t$ where $e$ requires $c$ to be met at $t$ and either $t' \leq t_s(e)$ or $t' \geq t_f(e)$. We need to show that this is false for any $c$ and $c'$. For row 1 in the table, the algorithm finds that the constraints $p'^{+} > p^{-}$ and $p'^{+} < p^{+}$ can be derived from *order*. Thus, $t_f(e') > t_s(e)$, and $t_f(e') < t_f(e)$ for all histories. But, $t' = t_f(e')$, so $t_s(e) < t' < t_f(e)$, contradicting the assumption that must-assert is false. Therefore, the rule in the table for this case is sound. Similarly, for row 5 $t_s(e') \geq t_s(e)$, and $t_s(e') < t_f(e)$ for all histories. But, since $always(c')$, $p'$ attempts to assert $c'$ just after $t_s(e')$, so $t_s(e) <$

[8] It would be easy to add a flag to the summary information indicating whether $c'$ summarizes an in- or postcondition or not. Then we could alter the *false* entries in Table 1 to specify the ordering constraints under which must-assert is true for the cases when $p'$ does attempt to assert $c'$. These constraints would be $p'^{+} \leq p^{-}$ for rows 7, 11, and 19 and $p'^{+} \leq p^{+}$ for row 18.

| | | | $p'$ may-assert $c'$ by $c$ | $p'$ may-assert $c'$ before $c$ |
|---|---|---|---|---|
| | $c' \in post(p')$ | $c \in pre(p)$ | *order* cannot impose these constraints | *order* cannot impose these constraints |
| | *last* | *first* | | |
| 1 | T | T | $p'^{+} > p^{-}$ | $p'^{+} \geq p^{-}$ |
| 2 | T | F | $p'^{+} \geq p^{+}$ | $p'^{+} \geq p^{+}$ |
| 3 | F | T | $p'^{-} \geq p^{-}$ | $p'^{-} \geq p^{-}$ |
| 4 | F | F | $p'^{-} \geq p^{+}$ | $p'^{-} \geq p^{+}$ |
| | $c' \in in(p')$ | | | |
| | *always* | | | |
| 5 | ? | T | $p'^{-} \geq p^{-}$ | $p'^{-} \geq p^{-}$ |
| 6 | ? | F | $p'^{-} \geq p^{+}$ | $p'^{-} \geq p^{+}$ |
| | $c' \in post(p')$ | $c \in in(p)$ | | |
| | *last* | *always* | | |
| 7 | T | T | $p'^{+} > p^{-}$ | $p'^{+} > p^{-}$ |
| 8 | T | F | $p'^{+} \geq p^{+}$ | $p'^{+} \geq p^{+}$ |
| 9 | F | T | $p'^{-} \geq p^{-}$ | $p'^{-} \geq p^{-}$ |
| 10 | F | F | $p'^{-} \geq p^{+}$ | $p'^{-} \geq p^{+}$ |
| | $c' \in in(p')$ | | | |
| | *always* | | | |
| 11 | ? | T | $p'^{-} > p^{-}$ | $p'^{-} \geq p^{-}$ |
| 12 | ? | F | $p'^{-} \geq p^{+}$ | $p'^{-} \geq p^{+}$ |
| | $c' \in post(p')$ | $c \in post(p)$ | | |
| | *last* | *last* | | |
| 13 | T | T | $p'^{+} > p^{+}$ | $p'^{+} \geq p^{+}$ |
| 14 | T | F | $p'^{+} \geq p^{+}$ | $p'^{+} \geq p^{+}$ |
| 15 | F | T | $p'^{-} \geq p^{+}$ | $p'^{-} \geq p^{+}$ |
| 16 | F | F | $p'^{-} \geq p^{+}$ | $p'^{-} \geq p^{+}$ |
| | $c' \in in(p')$ | | | |
| | *always* | | | |
| 17 | ? | T | $p'^{-} \geq p^{+}$ | $p'^{-} \geq p^{+}$ |
| 18 | ? | F | $p'^{-} \geq p^{+}$ | $p'^{-} \geq p^{+}$ |

Table 2: Table for *may-assert by/before* algorithm

$t' < t_f(e)$, again contradicting the assumption. In this manner, all rows of the table used by the algorithm have been verified to show that $t' \leq t_s(e)$ and $t' \geq t_f(e)$ are false for all cases and are, thus, sound. □

**Lemma** This algorithm is complete for the definition of must-assert; i.e. if $p' \in P$ must-assert $c'$ in $c$, then the algorithm determines this to be true.

**Proof** by contradiction. If the algorithm did not determine that $p' \in P$ must-assert $c'$ in $c$, then it must have found the constraints in the must-assert table to differ from those in the point algebra table. So, we need to show that that could not be the case if $p' \in P$ actually must-assert $c'$ in $c$ for any $c'$ and $c$. For row 1 the algorithm finds that the constraints $p'^{+} > p^{-}$ and $p'^{+} < p^{+}$ cannot be proven from *order*. Thus, $t_f(e') \leq t_s(e)$, or $t_f(e') \geq t_f(e)$ for some histories. But, $t' = t_f(e')$, so it is not the case that $t_s(e) < t' < t_f(e)$ for such histories, contradicting the definition of must-assert. For row 5 $t_s(e') < t_s(e)$ or $t_s(e') \geq t_f(e)$ for some histories. But, $p'$ attempts to assert $c'$ just after $t_s(e')$, so there are some histories where $t' < t_s(e)$ and some where $t' > t_f(e)$, again contradicting the assumption. As explained in the *must-assert by* completeness proof, $c'$ for row 7 could be summarizing only a precondition, so must-assert is false for this case. In this manner, all rows of the table used by the algorithm have been verified to show that $t_s(e) < t' < t_f(e)$ is false for some histories for all cases and are, thus, complete. □

$p' \in P$ *may-assert* $c'$ *in* $c$ iff for *some* history $h \in H$, there exists a $p \in P$, $e$, $e'$, $t$, and $t'$ such that $c \in$

| | | | $p'$ must-assert $c'$ in $c$ |
|---|---|---|---|
| | $c' \in post(p')$ | $c \in in(p)$ | *order* must impose these constraints |
| | *last* | *always* | |
| 1 | T | T | $p'^{+} > p^{-}$ and $p'^{+} < p^{+}$ |
| 2 | T | F | *false* |
| 3 | F | T | $p'^{-} \geq p^{-}$ and $p'^{+} \leq p^{+}$ |
| 4 | F | F | *false* |
| | $c' \in in(p')$ | | |
| | *always* | | |
| 5 | T | T | $p'^{-} \geq p^{-}$ and $p'^{-} < p^{+}$ |
| 6 | T | F | *false* |
| 7 | F | T | *false* |
| 8 | F | F | *false* |
| | | | |
| | | | $p'$ may-assert $c'$ in $c$ |
| | $c' \in post(p')$ | $c \in in(p)$ | *order* cannot impose these constraints |
| | *last* | *always* | |
| 1 | T | T | $p'^{+} \leq p^{-}$ or $p'^{+} \geq p^{+}$ |
| 2 | T | F | $p'^{+} \leq p^{-}$ or $p'^{+} \geq p^{+}$ |
| 3 | F | T | $p'^{+} \leq p^{-}$ or $p'^{-} \geq p^{+}$ |
| 4 | F | F | $p'^{+} \leq p^{-}$ or $p'^{-} \geq p^{+}$ |
| | $c' \in in(p')$ | | |
| | *always* | | |
| 5 | T | T | $p'^{+} \leq p^{-}$ or $p'^{-} \geq p^{+}$ |
| 6 | T | F | $p'^{+} \leq p^{-}$ or $p'^{-} \geq p^{+}$ |
| 7 | F | T | $p'^{+} \leq p^{-}$ or $p'^{-} \geq p^{+}$ |
| 8 | F | F | $p'^{+} \leq p^{-}$ or $p'^{-} \geq p^{+}$ |

Table 3: Table for *must/may-assert in* algorithm

$in_{sum}(p)$; $e$ is the top-level execution $p$ in $E(h)$; $e'$ is top-level execution of $p'$ in $E(h)$; $e'$ attempts to assert $c'$ at $t'$; and $t_s(e) < t' < t_f(e)$.

We determine that $p' \in P$ *may-assert* $c'$ *in* $c$ if and only if for the row of the table describing $c'$ and $c$, none of the corresponding ordering constraints are imposed by (can be deduced from) *order* as described above using the point algebra table.

**Lemma** This algorithm is sound for the definition of may-assert; i.e. if we determine that $p' \in P$ may-assert $c'$ in $c$, then such is the case.

**Proof** by contradiction. If it were not the case that $p' \in P$ may-assert $c'$ in $c$, then there is no history where $e$ is the top-level execution of $p$ in $E(h)$; $e'$ is top-level execution of $p'$ in $E(h)$; $e'$ attempts to assert $c'$ at $t'$; and $t_s(e) < t' < t_f(e)$. Thus, in any history where $e'$ attempts to assert $c'$ at $t'$, $t' \leq t_s(e)$ or $t' \geq t_f(e)$. So, we must find a contradiction by showing that $t_s(e) < t' < t_f(e)$ for any $c$ and $c'$ in some history using the corresponding cases in the table used by the algorithm. For row 1 in the table, the algorithm would find that the constraints $p'^{+} \leq p^{-}$ and $p'^{+} \geq p^{+}]$ can neither be derived from *order*. Thus, $t_f(e') > t_s(e)$ and $t_f(e') < t_f(e)$ must be true for some histories. But, $t' = t_f(e')$ for such histories, so $t_s(e) < t' < t_f(e)$, contradicting the definition of may-assert. Similarly, for rows 5-8 $t_f(e') > t_s(e)$ and $t_s(e') < t_f(e)$ must be true for some histories. But, $t_s(e') < t' < t_f(e')$, so again $t_s(e) < t' < t_f(e)$, and we have a contradiction. In this manner, all rows of the table used by the algorithm have been verified to show that $t_s(e) < t' < t_f(e)$ for some histories for all cases and are, thus, sound. □

**Lemma** This algorithm is complete for the definition of may-assert; i.e. if $p' \in P$ may-assert $c'$ in $c$, then the algorithm determines this to be true.

**Proof** by contradiction. If the algorithm did not determine that $p' \in P$ may-assert $c'$ in $c$, then it must have found the constraints in the may-assert table to differ from those in the point algebra table. So, we need to show that that could not be the case if $p' \in P$ actually may-assert $c'$ in $c$. For row 1 in the table, the algorithm finds that either the constraint $p'^{+} \leq p^{-}$ or $p'^{+} \geq p^{+}$ is actually proved by *order*. Thus, $t_f(e') \leq t_s(e)$ or $t_f(e') \geq t_f(e)$ must be true for all histories. But, since $last(c')$, $t' = t_f(e')$ so in no history could $t_s(e) < t' < t_f(e)$ be true, contradicting the definition of may-assert. Thus, the rule in row 1 in the table is complete in determining may-assert for this case. For rows 5-8 by similar argument, $t_f(e') \leq t_s(e)$ or $t_s(e') \geq t_f(e)$ for all histories. But, $t_s(e') < t' < t_f(e')$, so again $t_s(e) < t' < t_f(e)$ is false for all histories, and we have a contradiction. In this manner, all rows of the table used by the algorithm have been verified to contradict the assumption for all cases and are, thus, complete. □

$p' \in P$ *must-assert* $c'$ *when* $c$ iff $c' \in post_{sum}(p')$; $last(c')$; and for *all* histories $h \in H$ and all $t$ where

| | | | $p'$ must-assert $c'$ when $c$ |
|---|---|---|---|
| | $c' \in post(p')$ | $c \in post(p)$ | *order* must impose these constraints |
| | *last* | *last* | |
| 1 | T | T | $p'^{+} = p^{+}$ |
| 2 | T | F | *false* |
| 3 | F | T | *false* |
| 4 | F | F | *false* |
| | | | |
| | | | $p'$ may-assert $c'$ when $c$ |
| | $c' \in post(p')$ | $c \in post(p)$ | *order* cannot impose these constraints |
| | *last* | *last* | |
| 1 | T | T | $p'^{+} \neq p^{+}$ |
| 2 | T | F | $p'^{+} \leq p^{-}$ or $p'^{+} \geq p^{+}$ |
| 3 | F | T | $p'^{+} \leq p^{+}$ or $p'^{-} \geq p^{+}$ |
| 4 | F | F | $p'^{+} \leq p^{-}$ or $p'^{-} \geq p^{+}$ |

Table 4: Table for *must/may-assert when* algorithm

$e$ is the top-level execution in $E(h)$ of some $p \in P$ that requires $c$ to be met at $t$, and $e'$ is the top-level execution of $p'$ in $E(h)$, there is a $t'$ where $e'$ attempts to assert $c'$ at $t'$, and $t = t'$.

We determine that $p' \in P$ *must-assert* $c'$ *when* $c$ if and only if for the row of Table 4 describing $c'$ and $c$, all of the corresponding ordering constraints are imposed by (can be deduced from) *order* as described above using the point algebra table.

**Lemma** This algorithm is sound for the definition of must-assert; i.e. if we determine that $p' \in P$ must-assert $c'$ when $c$, then such is the case.

**Proof** by contradiction. If it were not the case that $p' \in P$ must-assert $c'$ in $c$, then there must be a history where for all $t'$ where $e'$ attempts to assert $c'$ at $t'$, there is a $t$ where $e$ requires $c$ to be met at $t$, and $t \neq t'$. We need to show that this is false for any $c$ and $c'$. For row 1 in the table, the algorithm finds that the constraint $p'^{+} = p^{+}$ can be derived from *order*. Thus, $t_f(e') = t_f(e)$ for all histories. But, $t = t_f(e)$, and $t' = t_f(e')$, so $t = t'$, contradicting the assumption that must-assert is false. The other rows are sound because the "if" part of the lemma is false. Therefore, all rows of the table used by the algorithm have been verified for all cases to be sound. □

**Lemma** This algorithm is complete for the definition of must-assert; i.e. if $p' \in P$ must-assert $c'$ when $c$, then the algorithm determines this to be true.

**Proof** by contradiction. If the algorithm did not determine that $p' \in P$ must-assert $c'$ when $c$, then it must have found the constraints in the must-assert table to differ from those in the point algebra table. So, we need to show that that could not be the case if $p' \in P$ actually must-assert $c'$ in $c$ for any $c'$ and $c$. For row 1 the algorithm finds that the constraint $p'^{+} = p^{+}$ cannot be proven from *order*. Thus, $t_f(e') \neq t_f(e)$ for some histories. But, $t = t_f(e)$, and $t' = t_f(e')$, so it is not the case that $t = t'$ for such histories, contradicting the definition of must-assert. For row 2 there are no constraints on the executions. But, since $t$ and $t'$ could be at any point within $e$ and $e'$ respectively, so certainly there are histories where $t \neq t'$, again contradicting the assumption. In this manner, all rows of the table used by the algorithm have been verified to show that $t = t'$ is false for some histories for all cases and are, thus, complete. □

$p' \in P$ *may-assert* $c'$ *when* $c$ iff for *some* history $h \in H$, there exists a $p \in P$, $e$, $e'$, $t$, and $t'$ such that $c \in post_{sum}(p)$; $e$ is the top-level execution $p$ in $E(h)$; $e'$ is top-level execution of $p'$ in $E(h)$; $e'$ attempts to assert $c'$ at $t'$; and $t = t'$.

We determine that $p' \in P$ *may-assert* $c'$ *when* $c$ if and only if for the row of the table describing $c'$ and $c$, none of the corresponding ordering constraints are imposed by (can be deduced from) *order* as described above using the point algebra table.

**Lemma** This algorithm is sound for the definition of may-assert; i.e. if we determine that $p' \in P$ may-assert $c'$ when $c$, then such is the case.

**Proof** by contradiction. If it were not the case that $p' \in P$ may-assert $c'$ when $c$, then there is no history where $e$ is the top-level execution of $p$ in $E(h)$; $e'$ is top-level execution of $p'$ in $E(h)$; $e$ attempts to assert $c$ at $t$; $e'$ attempts to assert $c'$ at $t'$; and $t = t'$. Thus, in any history where $e$ attempts to assert $c$ at $t$, and $e'$ attempts to assert $c'$ at $t'$, $t \neq t'$. So, we must find a contradiction by showing that $t = t'$ for any $c$ and $c'$ in some history using the corresponding cases in the table used by the algorithm. For row 1 in the table, the algorithm would find that the constraint $p'^{+} \neq p^{+}$ cannot be derived from *order*. Thus, $t_f(e') = t_f(e)$ must be true for some histories. But, $t = t_f(e)$, and $t' = t_f(e')$, for such histories, so $t = t'$, contradicting the assumption that may-assert is false. Similarly, for row 2 $t_f(e') > t_s(e)$ and $t_f(e') < t_f(e)$ must be true for some histories. But, $t' = t_f(e')$, and $t_s(e) < t < t_f(e)$, so again $t = t'$ for some histories, and we have a contradiction. In this manner, all rows of the table used by the algorithm have been verified to show that $t = t'$ for some histories for all cases and are, thus, sound. □

**Lemma** This algorithm is complete for the definition of may-assert; i.e. if $p' \in P$ may-assert $c'$ when $c$, then the algorithm determines this to be true.

**Proof** by contradiction. If the algorithm did not determine that $p' \in P$ may-assert $c'$ when $c$, then it must have found the constraints in the may-assert table to differ from those in the point algebra table. So, we need to show that that could not be the case if $p' \in P$ actually may-assert $c'$ when $c$. For row 1 in the table, the algorithm finds that the constraint $p'^{+} \neq p^{+}$ is actually proved by *order*. Thus, $t_f(e') \neq t_f(e)$ must be true for all histories. But, since $t' = t_f(e')$, and $t' = t_f(e')$, $t \neq t'$ for all histories, contradicting the definition of may-assert. Thus, the rule in row 1 in the table is complete in determining may-assert for this case. For row 2 by similar argument, $t_f(e') \leq t_s(e)$ or $t_f(e') \geq t_f(e)$ for all histories. But, $t_s(e) < t < t_f(e)$, and $t' = t_f(e')$, so again $t = t'$ is false for all histories, and we have a contradiction. In this manner, all rows of the table used by the algorithm have been verified to contradict the assumption for all cases and are, thus, complete. □

**Definitions and algorithms for must/may clobber, achieve, and undo** The soundness and completeness proofs of these are actually trivial given the lemmas that established the soundness and completeness of must/may assert relations and the fact that the algorithms for determining these properties directly follow their definitions. Therefore, we shall only give proofs for the soundness of must-clobber and must-achieve. In order to prove the soundness and completeness of $CanAnyWay$ and $MightSomeWay$ algorithms in the next section, we give proofs describing when conditions are not met when they are must- and may-clobbered.

Plan $p' \in P$ *must-*[*achieve, clobber*] $c$ *in* $pre_{sum}(p)$ iff there is a $c'$; $\ell(c') \Leftrightarrow [\ell(c), \neg\ell(c)]$; and for *all* histories $h \in H$, where $e$ is the top-level execution $p$ in $E(h)$, and $e'$ is the top-level execution of $p' \in P$ in $E(h)$; $e'$ attempts to assert $c'$ before or at the time $e$ requires $c$ to be met and after any other attempt to assert some $c''$ (where $\ell(c'') \Leftrightarrow [\neg\ell(c), \ell(c)]$) before or at the time $e$ requires $c$ to be met; and there are no $p'' \in P$ and $c''$ where $p'' \neq p'$; $\ell(c'') \Leftrightarrow [\ell(c), \neg\ell(c)]$; and for *all* histories $h'' \in H$ where $e''$ is the top-level execution $p''$ in $E(h'')$, $e'$ attempts to assert $c'$ before $e''$ attempts to assert $c''$, and $e''$ attempts to assert $c''$ before or at the time $e$ requires $c$ to be met.

**Lemma** Any precondition summarized by $c \in pre_{sum}(p)$ will be met in *no* histories $h \in H$ if there is a plan $p' \in P$ that must-clobber $c$, and all of the $\neg\ell(c)$ conditions summarized by the $c'$ referred to in the definition of must-clobber are met in all $h \in H$.

**Proof** If $p'$ must-clobber $c$, and the conditions summarized by $c'$ are all met, then $p'$ asserts $\neg\ell(c)$ before $c$ is required to be met by the top-level execution $e$ of $p$. Thus, any precondition summarized by $c$ will be threatened by $p'$. We also know by the definition of must-clobber that there can be no other plan in $P$ that achieves the precondition since $\ell(c)$ cannot be asserted between the assertion of $\neg\ell(c)$ and the time that $c$ is required. Thus, any precondition summarized by $c$ must be clobbered. □

**Lemma** Assuming that the initial state of any $h \in H$ is such that it does not conflict with any preconditions external to the plans represented by $P_{sum}$, if any precondition summarized by $c \in pre_{sum}(p)$ will be met in *no* histories $h \in H$, then there is a plan $p' \in P$ that must-clobber $c$.

**Proof** Based on the assumption about the initial state, if all preconditions summarized by $c \in pre_{sum}(p)$ are not met, then they must be clobbered. The definition of must-clobber requires one or more plans to be responsible for asserting $\neg\ell(c)$ (summarized by $c'$) in all histories. It could not be the case that some set of plans assert $\neg\ell(c)$ in only some histories and only together clobber $c$ in all histories. If that were true, then the $c'$ in each plan would not be a *must* condition, and there would be some set of plans $P$ represented by $P_{sum}$ and some history in $H$ where no plan asserts $\neg\ell(c)$. Thus, there is a plan $p'$ represented in $P_{sum}$ that attempts to assert $c'$ in all $h \in H$. The definition of must-clobber also requires that no plan attempt to assert some $c''$ where $\ell(c'') \Leftrightarrow \ell(c)$ between when $c'$ is asserted and when $c$ is required in any history. It could not be the case that there are multiple plans ($p'_1$ and $p'_2$) asserting $\neg\ell(c)$ in all histories; some plan $p''$ asserts $c''$ after $p'_1$ and before $p'_2$ asserts $c'$ in some histories; and in other histories $p''$ asserts $c''$ after $p'_2$ and before $p'_1$ asserts $c'$. If this were true, then the plans would not have specific ordering constraints given in the *order* over $P$. This would mean that in some $h \in H$, $p''$ asserts $c''$ after both $p'_1$ and $p'_2$ assert $c'$, and $c$ is achieved and not clobbered. This contradicts our assumption that preconditions summarized by $c$ are never met. Thus, there can be no such $p''$, and there must be a $p'$ that must-clobber $c$. □

We can determine that $p' \in P$ *must*-[*achieve, clobber*] $c$ *in* $pre_{sum}(p)$ if and only if by checking inside two nested loops through the summary conditions in $P_{sum}$, we find a $c'$ such that $p' \in P$ must-assert $c'$ by $c$; $must(c')$; $\ell(c') \Leftrightarrow [\ell(c), \neg\ell(c)]$; and there is no $p''$ and $c''$ such that $p'$ may-assert $c'$ before $c''$; $p''$ may-assert $c''$ by $c$; and $\ell(c'') \Leftrightarrow [\neg\ell(c), \ell(c)]$; or $p'$ must-assert $c'$ before $c''$; $p''$ must-assert $c''$ by $c$; $\ell(c'') \Leftrightarrow [\ell(c), \neg\ell(c)]$; and $c''$ is *must*.

The complexity of walking through the summary conditions in two nested loops is $O(n^2c^2)$ for $c$ summary conditions for each of $n$ plans represented in $P_{sum}$. This can be improved in two ways. First, we can walk through the summary conditions just once and keep track of any candidate plans that must-assert a $c'$ or may-assert a $c''$ by/before $c$. We can discard must-assert candidates if there is another plan that may-assert a $c''$ between it and $c$, and if there are any remaining plans must-asserting a $c'$ at the end of the loop, then the algorithm returns $true$, else $false$. This reduces the complexity to $O(nc)$ since there is only one loop. Another possible way to improve the algorithm is to hash the summary conditions based on the literal they summarize and only process plans with matching or contradicting literals. Then we do not have to visit conditions summarizing different propositional variables. However, in the worst case, all summary conditions summarize the same propositional variable, and all $O(nc)$ conditions must be visited, so the worst case complexity is not further reduced. The complexity of the other must/may achieve/clobber/undo algorithms described later is the same by similar argument.

**Lemma** This algorithm is sound for the definition of must-[achieve, clobber]; i.e. if $p' \in P$ must-[achieve, clobber] $c \in pre_{sum}(p)$, then the algorithm determines this to be true.

**Proof** by contradiction. First, note that as explained in the beginning of this section on Supporting Mechanisms, we assume that the summary conditions in $P_{sum}$ have their intended properties. If the algorithm determines that it is not the case that $p'$ must-[achieve, clobber] $c \in pre_{sum}(p)$, then either $p'$ does not must-assert $c'$ by $c$; $c'$ is not *must*; $\ell(c') \Leftrightarrow [\neg\ell(c), \ell(c)]$; or there is a $p''$ and $c''$ that has the properties described in the algorithm. $p'$ must-assert $c'$ by $c$ because otherwise there would be no execution of $p'$ to attempt to assert $c'$ before or at the time $c$ is required to be met. $c'$ has to be *must* since $e'$ attempts to assert $c'$ in all $h \in H$. There must be a $c'$ such that $\ell(c') \Leftrightarrow [\ell(c), \neg\ell(c)]$ because otherwise there would not be an $e'$ that attempts to assert $c'$. Now we must show that there can be no $p''$ and $c''$. If there were a $p''$ and $c''$ such that $p'$ may-assert $c'$ before $c''$; $p''$ may-assert $c''$ by $c$; and $\ell(c'') \Leftrightarrow [\neg\ell(c), \ell(c)]$, then there would have to be an execution to attempt to assert $c''$ (where $\ell(c'') \Leftrightarrow [\neg\ell(c), \ell(c)]$) after $e'$ and before or at the time $e$ requires $c$ to be met. There can also be no $p''$ and $c''$ such that $p'$ must-assert $c'$ before $c''$; $p''$ must-assert $c''$ by $c$; and $\ell(c'') \Leftrightarrow [\ell(c), \neg\ell(c)]$ since that would entail having an $e''$ in $E(h'')$ for all $h'' \in H$ where $e'$ attempts to assert $c'$ before $e''$ attempts to assert some $c''$ (where $\ell(c'') \Leftrightarrow [\ell(c), \neg\ell(c)]$), and $e''$ attempts to assert $c''$ before or at the time $e$ requires $c$ to be met. The truth of all of the above statements rests on the proofs of the lemmas that priorly established the soundness and completeness of must/may assert relations. Thus, the assumption is false, and the algorithm for must-[achieve, clobber] is sound. □

Plan $p' \in P$ *may*-[*achieve, clobber*] $c$ *in* $pre_{sum}(p)$ iff there is a $c'$ where $\ell(c') \Leftrightarrow [\ell(c), \neg\ell(c)]$, and for *some* history $h \in H$, $e$ is the top-level execution $p$ in $E(h)$; $e'$ is the top-level execution of $p' \in P$ in $E(h)$; $e'$ attempts to assert $c'$ before or at the time $e$ requires $c$ to be met and after any other attempt to assert some $c''$ (where $\ell(c'') \Leftrightarrow \ell(c)$ or $\neg\ell(c)$ before or at the time $e$ requires $c$ to be met.

**Lemma** Assuming that the initial state of any history $h \in H$ is such that it does not conflict with any preconditions external to the plans represented by $P_{sum}$, any precondition summarized by $c \in pre_{sum}(p)$ will be met in *all* $h \in H$ if there is *no* plan $p' \in P$ that may-clobber $c$, or there is such a $p'$, and the $\neg\ell(c)$ conditions summarized by $c'$ referred to in the definition of may-clobber are clobbered in all $h \in H$.

**Proof** If there is no plan $p'$ that may-clobber $c$, or the $\neg\ell(c)$ condition summarized by $c'$ that the top-level execution of $p'$ attempts to assert is clobbered, then either there is no plan $p'$ that asserts $\neg\ell(c)$, or there is such a $p'$ and also a $p''$ that achieves $\ell(c)$ by asserting it after $p'$ asserts $\neg\ell(c)$ and before $c$ is required. Thus, there can be no plan that clobbers a precondition summarized by $c$, and all such preconditions will be met in all $h \in H$. □

**Lemma** If any precondition summarized by $c \in pre_{sum}(p)$ is met in *all* $h \in H$, then there is *no* plan $p' \in P$ that may-clobber $c$, or there is such a $p'$, and the $\neg\ell(c)$ conditions summarized by $c'$ referred to in the definition of may-clobber are clobbered in all $h \in H$.

**Proof** If all preconditions summarized by $c$ are met, then there could be no plan that clobbered any one of them. If it the case that a plan $p'$ attempted to assert $\neg\ell(c)$, but the assertion failed in every history, then the $\neg\ell(c)$ condition was clobbered. Thus, if there is any history $h$ where a precondition summarized by $c$ is not met, there must be a $p'$ whose top-level execution in $E(h)$ attempts to assert a $c'$ where $\ell(c') \Leftrightarrow \neg\ell(c)$, and there is a $\neg\ell(c)$ condition summarized by $c'$ that is not clobbered. Therefore, if all preconditions summarized by $c$ are met in all $h \in H$, there can be no $p'$ that may-clobber $c$, or the $\neg\ell(c)$ conditions summarized by $c'$ are clobbered in all $h \in H$. □

We determine that $p' \in P$ *may-*[*achieve*, *clobber*] *c in* $pre_{sum}(p)$ if and only if by checking inside two nested loops through the summary conditions in $P_{sum}$, we find a $c'$ such that $p' \in P$ may-assert $c'$ by $c$; $\ell(c') \Leftrightarrow [\ell(c), \neg\ell(c)]$; and there is no $p''$ and $c''$ such that $p'$ must-assert $c'$ before $c''$; $p''$ must-assert $c''$ by $c$; $\ell(c'') \Leftrightarrow \ell(c)$ or $\neg\ell(c)$; and $c''$ is $must$.

Plan $p' \in P$ *must-clobber c in* $in_{sum}(p)$ iff $always(c)$; there is a $c'$ where $\ell(c') \Leftrightarrow \neg\ell(c)$; and for *all* histories $h \in H$, where $e$ is the top-level execution $p \in P$ in $E(h)$, and $e'$ is the top-level execution of $p'$ in $E(h)$, $e'$ attempts to assert $c'$ within $e$.

**Lemma** An incondition summarized by $c \in in_{sum}(p)$ will *not* be met in all histories $h \in H$ if there is a plan $p' \in P$ that must-clobber $c$, and all of the $\neg\ell(c)$ conditions summarized by the $c'$ referred to in the definition of must-clobber are met in all $h \in H$.

**Proof** If $p'$ must-clobber $c$, and the conditions summarized by $c'$ are all met, then $p'$ asserts $\neg\ell(c)$ within $e$ when $c$ is required to be met since $always(c)$. Thus, in any $h \in H$ there must be some incondition summarized by $c$ that is clobbered since $c$ can only be $always$ when the open interval of $e$ is covered by plan executions with $\ell(c)$ inconditions that are summarized by $c$. □

**Lemma** If an incondition summarized by $c \in in_{sum}(p)$ is *not* met in all histories $h \in H$, then there is a plan $p' \in P$ that must-clobber $c$.

**Proof** If some incondition summarized by $c$ is not met, then it must be clobbered. It could not be clobbered internal to $p$ in all histories because, otherwise, there would be no summary condition added. $c$ cannot be $sometimes$ because there must be some history for some set of plans having the summary information $P_{sum}$ where $\neg\ell(c)$ is not asserted within the interval where $\ell(c)$ is required. By similar argument to that presented in the previous lemma for must-clobbering summary preconditions, there must be a plan $p'$ with execution $e'$ that asserts $\neg\ell(c)$ within $e$ in all $h \in H$. Thus, there must be a $c'$ summarizing $\neg\ell(c)$ that $e'$ attempts to assert within $e$ in all $h \in H$, so $p'$ must-clobber $c$. □

We determine that $p' \in P$ *must-clobber c in* $in_{sum}(p)$ if and only if by checking inside two nested loops through the summary conditions in $P_{sum}$, we find a $c'$ such that $p'$ must-assert $c'$ in $c'$; $c'$ is $must$; and $\ell(c') \Leftrightarrow \neg\ell(c)$.

Plan $p' \in P$ *may-clobber c in* $in_{sum}(p)$ iff there is a $c'$ where $\ell(c') \Leftrightarrow \neg\ell(c)$; and for *some* history $h \in H$, where $e$ is the top-level execution $p \in P$ in $E(h)$, and $e'$ is the top-level execution of $p'$ in $E(h)$, $e'$ attempts to assert $c'$ within $e$.

**Lemma** Any incondition summarized by $c \in in_{sum}(p)$ will be met in *all* $h \in H$ if there is *no* plan $p' \in P$ that may-clobber $c$.

**Proof** If there is no plan $p'$ that may-clobber $c$, then there is no plan $p'$ that asserts $\neg\ell(c)$ within $e$. Thus, there can be no plan that clobbers a incondition summarized by $c$, and all such inconditions will be met in all $h \in H$. □

**Lemma** If any incondition summarized by $c \in in_{sum}(p)$ is met in *all* $h \in H$, then either there is *no* plan $p' \in P$ that may-clobber $c$, or there is such a $p'$, and the $\neg\ell(c)$ conditions summarized by $c'$ referred to in the definition of may-clobber are clobbered in all $h \in H$.

**Proof** If all inconditions summarized by $c$ are met, then there could be no plan that clobbered any one of them. If it the case that a plan $p'$ attempted to assert $\neg\ell(c)$, then the assertion failed in every history because the $\neg\ell(c)$ condition was clobbered. Therefore, if all preconditions summarized by $c$ are met in all $h \in H$, there can be no $p'$ that may-clobber $c$, or the $\neg\ell(c)$ conditions summarized by $c'$ are clobbered in all $h \in H$. □

We determine that $p' \in P$ *may-clobber c in* $in_{sum}(p)$ if and only if by checking inside two nested loops through the summary conditions in $P_{sum}$, we find a $c'$ such that $p'$ may-assert $c'$ in $c$; and $\ell(c') \Leftrightarrow \neg\ell(c)$.

Plan $p' \in P$ *must-clobber c in* $post_{sum}(p)$ iff there is a $c'$ where $\ell(c') \Leftrightarrow \neg\ell(c)$, and for *all* histories $h \in H$, where $e$ is the top-level execution $p \in P$ in $E(h)$, and $e'$ is the top-level execution of $p'$ in $E(h)$, $e'$ attempts to assert $c'$ at the same time $e$ attempts to assert $c$.

**Lemma** A postcondition summarized by $c \in post_{sum}(p)$ will *not* be met in all histories $h \in H$ if there is a plan $p' \in P$ that must-clobber $c$, and all of the $\neg\ell(c)$ conditions summarized by the $c'$ referred to in the definition of must-clobber are met in all $h \in H$.

**Proof** If $p'$ must-clobber $c$, and the conditions summarized by $c'$ are all met, then $p'$ asserts $\neg\ell(c)$ when $e$ attempts to assert $c$. Thus, in any $h \in H$ there must be some postcondition summarized by $c$ that is clobbered and, thus, not met since $\neg\ell(c)$ is met at the same time the postcondition is required. □

**Lemma** If a postcondition summarized by $c \in post_{sum}(p)$ is *not* met in all histories $h \in H$, then there is a plan $p' \in P$ that must-clobber $c$.

**Proof** If some postcondition summarized by $c$ is not met, then it must be clobbered. It could not be clobbered internal to $p$ in all histories because, otherwise, there would be no summary condition added. By similar argument to that presented in the lemma for must-clobbering summary preconditions, there must be a plan $p'$ with execution $e'$ that asserts $\neg\ell(c)$ at the same time $e$ attempts to assert $c$ in all $h \in H$. Thus, there must be a $c'$ summarizing $\neg\ell(c)$ that $e'$ attempts to assert at the same time $e$ attempts to assert $c$ in all $h \in H$, so $p'$ must-clobber $c$. □

We determine that $p' \in P$ *must-clobber* $c$ *in* $post_{sum}(p)$ if and only if by checking inside two nested loops through the summary conditions in $P_{sum}$, we find a $c'$ such that $p'$ must-assert $c'$ when $c$; $c'$ is $must$; and $\ell(c') \Leftrightarrow \neg\ell(c)$.

Plan $p' \in P$ *may-clobber* $c$ *in* $post_{sum}(p)$ iff there is a $c'$ where $\ell(c') \Leftrightarrow \neg\ell(c)$; and for *some* history $h \in H$, where $e$ is the top-level execution $p \in P$ in $E(h)$, and $e'$ is the top-level execution of $p'$ in $E(h)$, $e'$ attempts to assert $c'$ at the same time $e$ attempts to assert $c$.

**Lemma** Any postcondition summarized by $c \in post_{sum}(p)$ will be met in *all* $h \in H$ if there is *no* plan $p' \in P$ that may-clobber $c$.

**Proof** If there is no plan $p'$ that may-clobber $c$, then there is no plan $p'$ that asserts $\neg\ell(c)$ when $e$ attempts to assert $c$. Thus, there can be no plan that clobbers a postcondition summarized by $c$, and all such postconditions will be met in all $h \in H$. □

**Lemma** If any postcondition summarized by $c \in post_{sum}(p)$ is met in *all* $h \in H$, then either there is *no* plan $p' \in P$ that may-clobber $c$, or there is such a $p'$, and the $\neg\ell(c)$ conditions summarized by $c'$ referred to in the definition of may-clobber are clobbered in all $h \in H$.

**Proof** If all postconditions summarized by $c$ are met, then either there could be no plan that clobbered any one of them, or there is a plan $p'$ that attempted to assert $\neg\ell(c)$, and the assertion failed in every history because the $\neg\ell(c)$ condition was clobbered. Therefore, if all postconditions summarized by $c$ are met in all $h \in H$, there can be no $p'$ that may-clobber $c$, or the $\neg\ell(c)$ conditions summarized by $c'$ are clobbered in all $h \in H$. □

We determine that $p' \in P$ *may-clobber* $c$ *in* $post_{sum}(p)$ if and only if by checking inside two nested loops through the summary conditions in $P_{sum}$, we find a $c'$ such that $p'$ may-assert $c'$ when $c$; and $\ell(c') \Leftrightarrow \neg\ell(c)$.

Plan $p' \in P$ *must-undo* $c$ *in* $post_{sum}(p)$ iff there is a $c'$ where $\ell(c') \Leftrightarrow \neg\ell(c)$, and for *all* histories $h \in H$, where $e$ is the top-level execution $p$ in $E(h)$, and $e'$ is the top-level execution of $p' \in P$ in $E(h)$; $e$ attempts to assert $c$ before or at the time $e'$ attempts to assert $c'$ and after any other attempt to assert some $c''$ (where $\ell(c'') \Leftrightarrow \ell(c)$) before $e'$ attempts to assert $c'$; and there are no $p'' \in P$ and $c''$, where $p'' \neq p'$, $\ell(c'') \Leftrightarrow \neg\ell(c)$, and for *all* histories $h'' \in H$ where $e''$ is the top-level execution $p''$ in $E(h'')$, $e$ attempts to assert $c$ before or at the time $e''$ attempts to assert $c''$, and $e''$ attempts to assert $c''$ before $e'$ attempts to assert $c'$.

We determine that $p' \in P$ *must-undo* $c$ *in* $post_{sum}(p)$ if and only if by checking inside two nested loops through the summary conditions in $P_{sum}$, we find a $c'$ such that $p$ must-assert $c$ by $c'$; $must(c')$; $\ell(c') \Leftrightarrow \neg\ell(c)$; and there is no $p''$ and $c''$ such that $p$ may-assert $c$ by $c''$; $p''$ may-assert $c''$ before $c'$; and $\ell(c'') \Leftrightarrow \ell(c)$; or $p$ must-assert $c$ by $c''$; $p''$ must-assert $c''$ before $c'$; $\ell(c'') \Leftrightarrow \neg\ell(c)$; and $c''$ is $must$.

Plan $p' \in P$ *may-undo* $c$ *in* $post_{sum}(p)$ iff there is a $c'$ where $\ell(c') \Leftrightarrow \neg\ell(c)$, and for *some* history $h \in H$, $e$ is the top-level execution $p$ in $E(h)$; $e'$ is the top-level execution of $p' \in P$ in $E(h)$; $e$ attempts to assert $c$ before or at the time $e'$ attempts to assert $c'$ and after any other attempt to assert some $c''$ (where $\ell(c'') \Leftrightarrow \ell(c)$ or $\neg\ell(c)$) before $e'$ attempts to assert $c'$.

We determine that $p' \in P$ *may-undo* $c$ *in* $post(p)$ if and only if by checking inside two nested loops through the summary conditions in $P_{sum}$, we find a $c'$ such that $p \in P$ may-assert $c$ by $c'$; $\ell(c') \Leftrightarrow \neg\ell(c)$; and there is no $p''$ and $c''$ such that $p$ must-assert $c$ by $c''$; $p''$ must-assert $c''$ before $c'$; $\ell(c'') \Leftrightarrow \ell(c)$ or $\neg\ell(c)$; and $c''$ is $must$.

## Proving the Properties of Summary Information

The proof that the procedure derives summary conditions with their intended properties is all-inclusive since the truth of each property depends on those of others. These results ease the proofs of soundness and completeness for inference rules determining how CHiPs can definitely or potentially interact so that good planning and coordination decisions can be made at various levels within and among plan hierarchies.

**Theorem** The following properties of summary information are met by using the procedure for deriving summary conditions for plans. The set of external [pre, post] conditions for a plan is equivalent to the set of all literals in the plan's summary [pre, post] conditions. The set of conditions that must hold within (in the interval $(t_s, t_f)$ of) some execution of a plan in order for it to be successful is equivalent to the set of all literals in the plan's summary inconditions. A summary [pre, post] condition is *must* iff it is a *must* external [pre, post] condition of the plan. Otherwise, it is *may*.[9]

[9] This means that a *may* condition is one that must hold for some but not all executions of the plans that have this summary condition. However, a particular plan may have a

A summary incondition $c$ is *must* iff $\ell(c)$ must hold within all executions of the plan for it to be successful; otherwise, it is *may*. A summary [pre, post] condition $c$ is [*first*, *last*] iff $\ell(c)$ must hold at $[t_s,\ t_f]$ of some execution of the plan for it to be successful; otherwise, *sometimes*($c$). A summary incondition $c$ is *always* iff $\ell(c)$ must hold throughout $(t_s, t_f)$ for all executions of the plan in order for them to be successful; otherwise, *sometimes*($c$).[10]

**Proof** by induction over the maximum subplan depth. The base case is a primitive plan $p$ (subplan depth zero). According to the first step of the procedure, the summary [pre, post] conditions include a condition $c = \langle \ell, must,\ [first,\ last]\rangle$ for every [pre, post] condition $\ell$ of $p$, which must be an external [pre, post] condition of $p$. According to the semantics for successful execution, these conditions have to hold at $[t_s,\ t_f]$. Likewise, the summary inconditions include a condition $c = \langle \ell, must, always\rangle$ for every incondition $\ell$ of $p$, and the inconditions must hold throughout $(t_s, t_f)$ for all executions of $p$. Thus, the base case is satisfied.

Assume that the theorem is true for all plans of maximum depth $\leq k$. Any plan $p$ of maximum depth $k+1$ must have subplans with maximum depths $\leq k$. According to the first step of the procedure, summary conditions for non-primitive $p$ are added in the same way as in the base case for primitive plans based on $p$'s pre, in, and postconditions. The semantics for a successful execution based on these conditions is the same as for primitive plans, so the properties for these summary conditions are met. Additional summary conditions are derived from $p$'s subplans based on whether *type*($p$) is *and* or *or*.

Case 1—*type*($p$) = *and*:

In the step of the procedure for adding summary [pre, post] conditions for an *and* plan, summary conditions of the $p$'s subplans are added if they are not [*must-achieved*, *must-undone*] or *must-clobbered* by some other subplan or have already been added in the first step of the procedure. Any newly added condition $c$ is an external [pre, post] condition because it could not be [achieved, undone] or clobbered by a subplan of $p$'s subplan since $c$ is an external [pre, post] condition according to the inductive hypothesis, and if $c$ were [achieved, undone] or clobbered by another subplan of $p$, the procedures for determining [*must-achieve*, *must-undo*] and *must-clobber* would have identified such a subplan, and $c$ would not have been added. The *existence* of $c$ is set to *must* by the procedure iff it is a *must* external [pre, post] condition of $p$ because the procedure ensures that it is an external [pre, post] condition of all executions of $p$ by making sure it is a *must* summary [pre, post] condition (thus, a *must* external [pre, post] condition according to the inductive hypothesis) in a subplan and ensuring that there is no other subplan of $p$ that [*may-achieve*, *may-undo*] or *may-clobber* $c$. Otherwise, the procedure sets *existence*($c$) to *may*. The *timing* of $c$ is set to [*first*, *last*] by the procedure iff $\ell(c)$ must hold at $[t_s(e),\ t_f(e)]$ of some execution $e$ of $p$ by identifying a subplan $p'$ that is potentially [least, greatest] temporally ordered according to *order*($p$) with $\ell(c)$ as the literal of a [*first*, *last*] summary [pre, post] condition $c'$. By the inductive hypothesis, $c'$ must hold at $[t_s(e'),\ t_f(e')]$ of some execution $e'$ of $p'$, and if $p'$ is potentially ordered [least, greatest], then there must be some execution $e$ of $p$ where $[t_s(e) = t_s(e'),\ t_f(e) = t_f(e')]$. Otherwise, the procedure sets *timing* to *sometimes*.

In the step of the procedure that adds summary inconditions for an *and* plan, a condition $c$ is added iff it is a condition that must hold within (in the interval $(t_s(e), t_f(e))$ of) some execution $e$ of $p$ in order for $e$ to be successful. This is because the procedure adds summary inconditions to $p$ for all conditions of all of $p$'s subplans except those that can only hold at $t_s(e)$ or $t_f(e)$. It does this in three substeps. First, the procedure adds a summary condition for each summary incondition $c'$ of one of $p$'s subplans $p'$. $\ell(c')$ must hold within some execution $e'$ of $p'$ by the inductive hypothesis, and the inconditions of $p'$ must hold within $e$ of $p$ according to the semantics of subplan executions. In the [second, third] substep, an incondition $c$ is added for each summary [pre, post] condition $c'$ for each subplan $p'$ that is not always a [*first*, *last*] summary [pre, post] condition of $p$. Therefore, there is some execution $e'$ of $p'$ where $\ell(c')$ holds within $e'$ and, thus, must also hold within some execution $e$ of $p$. So, the only conditions of the subplans that are not inconditions of $p$ are those that hold at $t_s(e)$ or $t_f(e)$ for all executions $e$ of $p$. The *existence* of $c$ is set to *must* iff $\ell(c)$ must hold at some point within all executions $e$ of $p$ since either $\ell(c)$ is specified as an incondition of $p$; $c$ is derived from a *must* summary incondition of some subplan $p'$ of $p$ (in which case it must hold in every execution of $p'$ by the inductive hypothesis and, thus, in every execution $e$ of $p$ because $d(e)$ contains an execution of every subplan of $p$); or $c$ is derived from a *must* summary [pre, post] condition $c'$ of a subplan $p'$ that is always not [*first*, *last*]–thus, $\ell(c')$ must hold within all executions of $p'$ and $p$. Otherwise, the *existence* of $c$ is set to *may*. The procedure sets *timing*($c$) to *always* iff $\ell(c)$ must hold throughout all executions of $p$ since it requires that $\ell(c)$ is in an *always* summary incondition of each subplan; $\ell(c)$ must hold throughout the executions of each subplan by the inductive hypothesis; the intervals within the executions of the subplans

decomposition such that the *may* condition holds for all of its executions. In this case, this fact is lost in the summary information but can be discovered by looking deeper into the plan hierarchy.

[10] This requires that *always* summary inconditions are also *must*. Summary inconditions could have been derived so that *always* conditions could also be *may*, and mechanisms for determining legal temporal relations among plans would be constructed differently. We have not analyzed the advantages and disadvantages of these options.

together must cover the interval within the execution of $p$ according to *order*($p$); and there are no intervals in any execution of $p$ where there are no subplans executing. Otherwise, *timing* is set to *sometimes*.

Case 2—*type*($p$) = *or*:

The step of the procedure for adding summary [pre, post] conditions for an *or* plan only adds conditions for the external [pre, post] conditions of $p$ because there is a one-to-one matching of the summary [pre, post] conditions of every subplan $p'$ to the set external [pre, post] conditions of $p'$ by the inductive hypothesis, and there are no other subplans that could [achieve, undo] or clobber $p'$'s summary [pre, post] conditions since the execution of an *or* plan $p$ is only decomposed into an execution of one of $p$'s subplans. The procedure sets the *existence* of the added summary [pre, post] condition $c$ to *must* iff $\ell(c)$ is a *must* external [pre, post] condition of $p$ because $\ell(c)$ must hold for all executions of every subplan for them to be successful by the inductive hypothesis, and an execution of $p$ is only decomposed into an execution of one of $p$'s subplans. Otherwise, the *existence* of $c$ is set to *may*. The *timing* of $c$ is set to [*first*, *last*] iff $\ell(c)$ must hold at $[t_s, t_f]$ of some execution of the plan for it to be successful because there must be an execution $e'$ of a subplan $p'$ and a [*first*, *last*] summary [pre, post] condition $c'$ where $\ell(c') \Leftrightarrow \ell(c)$ and $\ell(c)$ must hold at $[t_s(e'), t_f(e')]$ and, thus, at $[t_s(e), t_f(e)]$ of some execution $e$ of $p$ by the inductive hypothesis and the semantics of *or* subplan executions. Otherwise, *timing* is set to *sometimes*.

The step of the procedure for adding summary inconditions for an *or* plan only adds conditions that must hold within some execution $e$ of $p$ for it to be successful since all of the conditions that must hold within the executions of the subplans are captured by their summary inconditions by the inductive hypothesis, and no other conditions of the subplans can hold within $e$ because they are *first* or *last* and would only hold at $t_s(e)$ or $t_f(e)$ by the inductive hypothesis and semantics of *or* subplan executions. The procedure sets the *existence* of the added summary incondition $c$ to *must* iff $\ell(c)$ must hold within all executions of $p$ because it is a *must* summary incondition of all subplans and, therefore, must hold within the all of the subplans' executions by the inductive hypothesis and, thus, within all executions of $p$. Otherwise, the *existence* of $c$ is set to *may*. The *timing* of $c$ is set to *always* iff $\ell(c)$ must hold for any execution $e$ of $p$ throughout the interval $(t_s(e), t_f(e))$ since $\ell(c)$ must hold within all executions $e'$ of each subplan throughout $(t_s(e'), t_f(e'))$ by the inductive hypothesis and, thus, throughout $(t_s(e), t_f(e))$ by the semantics of *or* subplan executions. Otherwise, *timing* is set to *sometimes*. □

## Complexity

The procedure for deriving summary conditions works by basically propagating the conditions from the primitives up the hierarchy to the most abstract plans. Because the conditions of any non-primitive plan depend only on those of its immediate subplans, deriving summary conditions can be done quickly. The complexity of $O(n(log^2 n)c^2)$ for $n$ non-primitive plans with $c$ conditions in each plan's summary pre-, in-, and postconditions reported in (Clement & Durfee 1999a) does not tell us how the complexity grows as a result of summary conditions accumulating in greater and greater sets as they are propagated up the hierarchy. If $c'$ is the number of literals in each plan's pre-, in-, and postconditions, then the complexity is $O(n^2c'^2)$. Here, the worst case is when all plans are *and* plans, and the conditions of each plan are completely different than those of any other plan. Consider such a hierarchy with $n$ plans, $b$ subplans for each non-primitive plan, and depth $d$.[11] At the next-to-bottom depth level $d-1$, the procedure tests each of $c'$ conditions in each condition set of the $b$ primitive subplans of each of the $b^{d-1}$ plans to see if they are achieved/clobbered/undone by any other subplan attempting to assert $c'$ inconditions and $c'$ postconditions. Thus, a constant number of operations must be performed when comparing each condition in each subplan with every other condition in every other subplan resulting in $O(b^2c'^2)$ operations. So, for $b^{d-1}$ plans $O(b^{d-1}b^2c'^2)$ operations are performed to derive summary information at level $d-1$. At level $d-2$, there are $b^{d-2}$ plans, and the number of conditions that must be compared among their subplans additionally includes those propagated from the primitive level for a total of $c' + bc'$ conditions. Thus, $O(b^{d-2}b^2(c'+bc')^2)$ operations are performed at level $d-2$. This generalizes to $O(\sum_{i=1}^{d} b^{d-i}b^2(b^{i-1}c')^2)$ operations for the entire hierarchy. We can reduce this to $O(b^2c'^2\sum_{i=1}^{d} b^{d+i-2}) = O(b^{2d}c'^2)$, and since $n = O(b^d)$, the complexity can be simply stated as $O(n^2c'^2)$.

In this worst case, the number of summary conditions for an abstract plan grows exponentially as you go up the hierarchy. There are $3nc'$, or $O(b^dc')$, summary conditions at the root of the hierarchy—this is the total number of pre-, in-, and postconditions in the hierarchy. One level down, there are $O(b^{d-1}c')$ summary conditions, and there are $O(b^{d-2}c')$ at level 2 while there are only $c'$ at level $d$. One might argue that in such cases deriving summary information only increases computation. But, actually, exponential computation time is saved when decisions based on summary information can be made at abstract levels. With respect to the general search procedure described in (Clement & Durfee 1999b), the only case in which summarizing conditions is wasteful is when the search cannot be completed at abstract levels, and the summary information does not improve the search by avoiding backtracking or by directing it towards better quality solutions. This is because the only known

[11] We consider the root at depth level 0 and the leaves at level $d$.

algorithms for synchronizing plan steps to avoid conflicts are exponential with respect to the number of plans expanded in the hierarchy, which also grows exponentially with the depth. This exponential growth down the hierarchy outweighs the exponential growth of summary conditions in plans up the hierarchy. So the improvements made using summary information can make exponential savings while only incurring a small polynomial overhead in deriving and using summary information (as described in the next section).

Let's make this more clear. At the $i$th depth level in the hierarchy, each of the $O(b^i)$ plans has $O(b^{d-i}c'^2)$ summary conditions in the worst case. As described in the next section, the algorithm to check whether a particular ordering of these plans results in all plans executing successfully has a complexity of $O(n^2c^2)$ for $n$ plans and $c$ summary conditions for each of those plans. Checking such a synchronization for the plans at any level $i$ in a plan hierarchy is, thus, $O(b^{2i}b^{2d-2i}c'^2) = O(b^{2d}c'^2)$. So, since $i$ drops out, the complexity of doing this check is *independent of the depth level*. But, there is a huge space of $n! = O((b^i)!)$ sequential orderings[12] of the $n$ plans at level $i$ to be checked to find a valid synchronization. Thus, the search space grows wildly exponentially down the hierarchy despite the worst case when the number of conditions grows exponentially up the hierarchy. This argument assumes that finding a valid synchronization is intractable, so we show that it is actually NP-complete. We reduce HAMILTONIAN PATH to the THREAT RESOLUTION problem for STRIPS planning and claim that a similar reduction can be done for our problem that allows concurrent execution.

**Theorem** THREAT RESOLUTION is NP-complete. This is the problem of determining whether there is a set of ordering constraints that can be added to a partial order STRIPS plan such that no operator's preconditions are threatened by another operator's effects.

**Proof** If there is a set of ordering constraints that will resolve all threats, then there is at least one corresponding total order where there are no threats. Thus, the problem is in NP since orderings of operators can be chosen non-deterministically, and threats can be identified in polynomial time.

For a directed graph $G = (V, E)$ with nodes $v_1, v_2, \ldots, v_n \in V$ and edges $e_1, e_2, \ldots, e_m \in E$ (a set of ordered pairs of nodes), HAMILTONIAN PATH is the problem that asks if there is a path that visits each node exactly once. We build an instance of THREAT RESOLUTION (a partial order plan) by creating an operator for each node $v_i$. The only precondition of the operator is $A(i)$, representing the *accessibility* of the node. There is a postcondition $A(j)$ for each edge $e_k = (v_i, v_j)$, and a postcondition $\overline{A(l)}$ for all other nodes for which there is no edge from $v_i$. All operators are unordered and the initial state and goal state is empty.

If there is a Hamiltonian path for the graph, then the operators for the nodes can be ordered the same as the nodes in the path because the accessibility preconditions of each operator will be satisfied by the previous operator. If there is no Hamiltonian path for the graph, then there is no consistent ordering of the operators. We know this because there is a one-to-one mapping from an ordering of nodes to an ordering of operators. If the ordering of the nodes is such that there is no edge from one to a succeeding node, then the accessibility precondition of the corresponding operator will be clobbered. In addition, for any walk through the graph, there eventually will be an unvisited node for which there is no edge from the last node visited. In this case, the unvisited node will be clobbered because its accessibility precondition will not be met. Thus, THREAT RESOLUTION is NP-hard, and since it was shown to be in NP, it is NP-complete. □

There are only *and* plans in this worst case. In the case that there are *or* plans, by similar argument, being able to prune branches at higher levels based on summary information will greatly improve the search despite the overhead of deriving and using summary conditions. Obviously, the computational savings of using summary information will be even greater when there are conditions common to plans on the same level, and the number of summary conditions does not grow exponentially up the hierarchy. Still, surely there are cases where none of the details of the plan hierarchy can be ignored, and summary information would incur unnecessary overhead, but when the size of problem instances are scaled, dealing with these details will likely be infeasible anyway.

## Soundness and Completeness of Checking Temporal Relations

With the properties of summary information proven, we can safely reason about the interactions of plans without information about their subplans. Based on the *existence* and *timing* information carried by summary conditions, we can determine what relations can or might hold between CHiPs without searching their subplans. In many coordination tasks, if it could be determined that certain temporal relations can hold among plans no matter how they are decomposed ($CanAnyWay$) or that certain relations cannot hold for any decomposition ($\neg MightSomeWay$), then coordination decisions can be made at abstract levels without entering a potentially costly search for valid plan merges. Here we prove the soundness and completeness of rules determining $CanAnyWay$ and $MightSomeWay$ relations based on summary information. The use of these rules is illustrated in the Introduction and explored further in (Clement & Durfee 1999b). For convenience, we will abbreviate $Can$

[12]There are more for other orderings allowing for concurrent execution.

to $C$, *Any* to $A$, *Way* to $W$, and so on.

Informally, [$CAW$(*order*, $P_{sum}$), $MSW$(*order*, $P_{sum}$)] says that the temporal relations in *order* [*can*, *might*] hold for the set of plans $P$ whose corresponding summary information is in the set $P_{sum}$ for [*any way*, *some way*] that the plans may be executed. We could also describe $CanSomeWay$(*order*,$P_{sum}$) and $MightAnyWay$(*rel*,$P_{sum}$) in the same fashion, but because these relations depend on both decomposition choices and how the subexecutions are synchronized, their meanings are not clear. For example, $MightAnyWay(\{\text{d}\}, \{p_{sum}, q_{sum}\})$ could mean that the *during* relation might hold for some decompositions of the plans in such a way that the subplans can be synchronized in any way (within the constraints of *during* at the top level). However, it could also mean that for any way the plans are decomposed, there might be a synchronization of the plans such that they will succeed. We could call the former interpretation the $MightAnyWaySynch$ relation, and the latter the $MightAnyWayDecomp$ relation. The same can be done for $CanSomeWay$. $CanAnyWay$ and $MightSomeWay$ do not have such ambiguity.

Formally, [$CAW$(*order*, $P_{sum}$), $MSW$(*order*, $P_{sum}$)] iff for [all, some] sets of plans $P$ with summary information $P_{sum}$ and [all, some] histories $h$ where $E(h)$ includes an execution of each plan in $P$ as well as its subexecutions, and all executions meet the constraints in *order*, all executions in $E(h)$ succeed. We assume here that each of the plans in $P$ would successfully execute by itself (i.e. they cannot clobber their own conditions) and that the initial state of $h$ is such that it does not conflict with any preconditions external to the plans in $P$. In other words, we assume that the external conditions of the plans do not conflict with each other.[13] So the task of determining whether $CAW$ or $MSW$ is true is basically to detect whether any plan may or must clobber a condition of another.

We determine [$CAW$(*order*, $P_{sum}$), $MSW$(*order*, $P_{sum}$)] is *false* if and only if by checking inside two nested loops through the $P_{sum}$, we find a $p_{sum}$ and $p'_{sum}$ in $P_{sum}$ and a summary condition $c$ of $p_{sum}$ such that one or more of the procedures for determining $p'$ [may-clobber, must-clobber] $c$ (using $p'_{sum}$ as the summary information for $p'$) returns *true*. The complexity of this algorithm is $O(n^2c^2)$ since the $O(nc)$ procedures for determining must/may-clobber must be run for each of $nc$ conditions for $c$ summary conditions in each of $n$ plans represented by $P_{sum}$.

**Theorem** The algorithms for $CAW$(*order*, $P_{sum}$) and $MSW$(*order*, $P_{sum}$) are sound and complete.

**Proof** If the algorithm returns *true* for [$CAW$, $MSW$], then there is no $p'$ that [may-clobber, must-clobber] some $c$. By the proofs given in the section on Definitions and algorithms for must/may clobber, achieve, and undo, all preconditions summarized by the summary preconditions, inconditions summarized by summary inconditions, and postconditions summarized by summary postconditions must be met in [all, some] $h \in H$ for [all, some] set of plans $P$ whose summary information is $P_{sum}$. This means that all of the external preconditions, external postconditions, and inconditions of all executions and subexecutions are met. Since we assume that the plans in $P$ cannot clobber their own conditions, we also know that the internal (non-external) pre- and postconditions are also met. Thus, all of the conditions in all of the executions and subexecutions in $E(h)$ are met, so all executions succeed.

If all executions succeed for [all, some] $h \in H$ for [all, some] set of plans $P$ whose summary information is $P_{sum}$, then no conditions were clobbered. By the proofs given in the section on Definitions and algorithms for must/may clobber, achieve, and undo, there can be no $p'$ that [may-clobber, must-clobber] some summary condition $c$ unless the execution of $p'$ fails when it attempts to assert a $c'$ that summarizes conditions that would otherwise clobber $\ell(c)$. But, such a $p'$ cannot exist because then there would be some plan that clobbered a condition of $p'$, and we assume that no conditions were clobbered. □

[13] We could additionally check the summary preconditions of the plans against each other or against a particular initial state for conflicts. This could be simply done by treating the initial state as the external postconditions of a plan ordered before any plan in $P$.

## Conclusions and Future Work

Coordination and planning for hierarchical plans often involve the refinement of abstract plans into more detail. However, the cost of backtracking or making irreversible commitments makes it critical that the specification of abstract plan operators be rich enough to anticipate and avoid costly planning/execution decisions. We have addressed this directly by offering a formalism for describing concurrent hierarchical plan execution and methods for deriving summary conditions and determining legal plan interactions in a sound and complete fashion. This provides a foundation upon which provably correct coordination and planning mechanisms can be built. In prior work that motivates and validates this formalism, we describe a general algorithm for merging hierarchical plans using summary information, a specific implementation, and a preliminary evaluation of the approach (Clement & Durfee 1999b). Future work includes relaxing assumptions, such as the downward solution property, investigating other types of plan summary information, and constructing sound and complete algorithms for concurrent hierarchical planning and for interleaving planning, plan coordination, and plan execution.

## References

Allen, J. F., and Koomen, J. A. 1983. Planning using a temporal world model. In *Proc. IJCAI*, 741–747.

Allen, J. F. 1983. Maintaining knowledge about temporal intervals. *Communications of the ACM* 26(11):832–843.

Clement, B., and Durfee, E. 1999a. Theory for coordinating concurrent hierarchical planning agents. In *Proc. AAAI*.

Clement, B., and Durfee, E. 1999b. Top-down search for coordinating the hierarchical plans of multiple agents. In *Proc. Intl. Conf. Autonomous Agents*.

Corkill, D. 1979. Heirarchical planning in a distributed environment. In *Proc. IJCAI*, 168–175.

Ephrati, E., and Rosenschein, J. 1994. Divide and conquer in multi-agent planning. In *Proc. AAAI*, 375–380.

Erol, K.; Hendler, J.; and Nau, D. 1994. Semantics for hierarchical task-network planning. Technical Report CS-TR-3239, University of Maryland.

Fagin, R.; Halpern, J.; Moses, Y.; and Vardi, M. 1995. *Reasoning about knowledge*. MIT Press.

Firby, J. 1989. *Adaptive Execution in Complex Dynamic Domains*. Ph.D. Dissertation, Yale University.

Georgeff, M. P., and Lansky, A. 1986. Procedural knowledge. *Proc. IEEE* 74(10):1383–1398.

Georgeff, M. P. 1983. Communication and interaction in multiagent planning. In *Proc. AAAI*, 125–129.

Georgeff, M. P. 1984. A theory of action for multiagent planning. In *Proc. AAAI*, 121–125.

Lansky, A. 1990. Localized search for controlling automated reasoning. In *Proc. DARPA Workshop on Innov. Approaches to Planning, Scheduling and Control*, 115–125.

Sacerdoti, E. D. 1977. *A structure for plans and behavior*. Elsevier-North Holland.

Tate, A. 1977. Generating project networks. In *Proc. IJCAI*, 888–893.

Tsuneto, R.; Hendler, J.; and Nau, D. 1998. Analyzing external conditions to improve the efficiency of htn planning. In *Proc. AAAI*, 913–920.

Vilain, and Kautz, H. 1986. Constraint propagation algorithms for temporal reasoning. In *Proc. AAAI*, 377–382.

Weld, D. 1994. An introduction to least commitment planning. *AI Magazine* 15(4):27–61.