A Formal Model for Planning with Time and Resources in Concurrent Domains

Michael Brenner
Institute for Computer Science
Albert Ludwigs University
Georges-Koehler-Allee, Geb. 052
D-79110 Freiburg, Germany
brenner@informatik.uni-freiburg.de

Abstract

While classical planning is still mostly concerned with sequential plans, numerous realistic domains are inherently concurrent. Especially planning for multi-agent systems necessitates a clear formal concept of plans in such domains. This paper presents a simple semantic model for concurrent plans using time and resources. It models actions with arbitrary effects over time, simultaneous accesses to shared resources, and describes asynchronous as well as synchronous concurrency in action execution. According to this model, a description language for concurrent domains is defined. The language BTPL (Basic Temporal Planning Language) allows also for rich specifications of initial and goal situations including triggered or exogenous events over time, and maintenance goals. Furthermore, BTPL is downward compatible to PDDL and may therefore be used for classical as well as multi-agent planning.

1 Introduction

In recent years, growing attention in AI research has been paid to dynamic and multi-agent systems. This interest reflects the fact that many realistic application domains of AI are inherently distributed and concurrent: numerous agents act simultaneously in environments that are themselves changing dynamically. Research in AI Planning, on the other hand, mostly still focuses on the classical paradigm of sequential state transitions. The field of multi-agent planning that could bridge this gap seems remarkably underdeveloped and unconnected. We believe that the main reason for this is the lack of a clear formal model of concurrent plans in concurrent domains that can be used to plan *for* as well as *by* multi-agent systems.

The aim of this paper is to provide a simple model that incorporates only the features basically needed to describe concurrent plans. By this we hope to provide basics for more specialized research and to help unify existing approaches in multi-agent planning that now seem incompatible because of overspecialized description languages and semantics.

The syntactic realization of the semantic model, the language BTPL (Basic Temporal Planning Language), was designed to be compatible to the classical planning standard PDDL (Planning Domain Description Language, [McDermott & others, 1998; Bacchus, 2000]. In fact, BTPL is downward compatible to the STRIPS subset of PDDL and includes also parts of PDDL's ADL extension.

Two of the basic issues that must be dealt with when introducing concurrency into the planning paradigm are the role of time and the treatment of resources. Realistic models of concurrency require actions with duration and delayed effects; resources may or may not be accessed by several agents simultaneously. The semantic model of BTPL is thus designed to describe actions with effects at arbitrary times and maintenance conditions. Additionally, we propose three important extensions to the model that can be utilized separately or in combination (because of limited space, in this paper we present each one only as an individual extension to the basic model): The first extended model allows conditions over time, and richer initial and goal situations. To model concurrent access to shared resources we then introduce the concept of *asynchronous concurrency* that describes possible but not necessarily simultaneous events. When really simultaneous events cause specific effects not derivable from the basic events, domain designers can model this using the last extension to the basic semantics, so-called *causal domain laws*.

The reader should be aware of the purpose of this paper which is unusual in so far that it merely consists of definitions rather than algorithms or empirical results. We mainly intend to describe and solve some problems that seem to appear in many existing extensions of planning formalisms to concurrency. We do not promote a single algorithmic approach to planning in BTPL domains, but are convinced that many planning algorithms should be adaptable to our formal model. However, the main motivation is to provide a simple model of concurrency to be used in developing distributed planning algorithms for distributed concurrent plan execution, i.e. the field of multi-agent planning.

The rest of the paper is organized as follows: we first describe relevant previous work. Section 3 introduces the basic formal model, especially the semantics of temporal plans. In Section 4 we extend the basic model to allow richer specifications of planning problems. Section 5 defines the concept of asynchronous concurrency and its use for the treatment of resources. Causal domain laws are presented in Section 6. A brief summary concludes the paper.

2 Previous Work

Among the large body of research in the fields of temporal reasoning, temporal planning, and planning with resources (see exemplarily [Pelavin, 1991; Ghallab & Laruelle, 1994; Penberthy & Weld, 1994]) this paper follows a specific recent line of work extending the limited form of planning for concurrency used in GRAPHPLAN [Blum & Furst, 1997]. GRAPHPLAN uses STRIPS actions but allows them to be executed in "parallel" if they are not *mutually exclusive* (mutex), i.e. they do not delete their mutual preconditions or have conflicting effects. While this prevents parallel actions to interfere with each other no "realistic" model of concurrency is provided: Fig. 1 shows an example in which finding a realistic concurrent plan depends not only on the possibility of applying actions in parallel but also on their duration.

Figure 1: A simple logistics example. The goal is to get the goods to `Loc2`. Time-optimal planning depends on which actions can be done in parallel (e.g. can goods be loaded into a truck simultaneously?) and how long these actions take. If, e.g., each truck can load only one good at a time, the best plan still depends on the duration of loading actions in comparison to driving between locations: if driving is sufficiently fast, `T2` could "help" `T1` by driving to `Loc1` and loading some of the goods.

This exemplifies that the GRAPHPLAN definition of time-step parallelism is little helpful for realistic models of such problems: each action taking exactly one time step a more complex interplay between overlapping and meeting action intervals cannot be modeled.

Temporal Graphplan [Smith & Weld, 1999] solves some of these problems by allowing actions to have durations. However, all action effects are guaranteed to hold only at the end of the action execution and are undefined before. This leads to a very strict notion of mutual exclusivity between actions and thus unnecessary prohibition of concurrency. E.g. a `fly(airplane,X,Y,dur)` action might "block" location (runway) `X` for other planes during the whole duration `dur` because the effect of deleting `airplane` being at `X` will become true only at the end of `dur`, while in reality `X` would be free to use directly after `airplane` left it.

The GRAPHPLAN-based planner IPP has been extended in [Köhler, 1998] to deal with explicitly modeled resources and time. The *mutex* definition is extended to prohibit conflicting accesses to resources.

Actions with effects at arbitrary times are integral parts of two planning formalisms based on specific temporal logics: TLPLAN [Bacchus & Ady, 2000] and TALPLANNER [Kvarnström, Doherty, & Haslum, 2000] have both been extended to deal with concurrency and resources recently. However, the semantics of concurrent executions of *dependent* actions, i.e. actions that have interfering or synergistic effects, has not been clearly specified yet or uses oversimplified assumptions on concurrent applicability (see Section 5 for a brief discussion). Furthermore, all approaches based on temporal logics differ from our work in relying on description languages equivalent to first order logic whereas for the sake of conceptual simplicity and wide applicability it is our goal to stay close to STRIPS-like classical planning languages.

3 Temporal plans in BTPL

In this section we describe the semantics of plans containing temporal actions. BTPL actions can be started at arbitrary times and may have effects with arbitrary delays. Therefore the semantics must describe the interleaving of effects of different actions during plan execution. This is accomplished by breaking down action effects to sets of so-called *unit events* that happen at a specific time point.

We model time by sequences of states at specific times.

Definition 1 *$\mathcal{S}$ is the set of possible* states *where each $s \in \mathcal{S}$ is a finite and consistent set of atoms. $\mathcal{T}$ is the set of possible* time stamps *modeled by the rational numbers $\mathbb{Q}$. A sequence of tuples $h = \langle(s_1, t_1), \ldots, (s_n, t_n)\rangle$ is called a* world history *where $s_i \in \mathcal{S}$ and $t \in \mathcal{T}$. One such tuple $p = (s, t)$ is called a* time point.

This does not mean that we adopt a discrete model of time, but a model of instantaneous changes of states: time points are to be seen as moments of transition between states. Intuitively, in a world history $\{(s, t), (s', t')\}$ the state s remains unchanged during the interval $[t, t')$ if there is no other time point (s'', t'') where $t < t'' < t'$.

In classical planning action preconditions are checked once and if they hold in the current state the effects take place. In temporal planning, effects may happen a long time after action invocation and precondition testing, so the world may have changed in ways that disturb the intended action execution. But it is also not always reasonable to demand that preconditions must hold over the whole duration of an action. BTPL's approach is therefore to provide events with individual conditions that must hold at the moment before the event is to be triggered.

Definition 2 *A* unit event *is a tuple $ue = (c, e, d)$ where c is a set of literals called the* condition *of ue, e is a set of literals called the* effects*, and $d \in \mathcal{T}$ is the* delay *of ue. The set of unit events is named $\mathcal{E}$.*

The delay d of a unit event is relative to an action's application. In the basic version of the formal model presented in this section d is the same for both the condition and the effect, i.e. if the condition holds directly before t, effect e becomes true *at* t. A unit event is thus basically a conditional effect, as used, e.g., in the ADL extension of PDDL [Bacchus, 2000], with a delay. However, in the next section we will describe an enhanced semantics and an extended dialect of the language (BTPL$^+$) that allows to specify arbitrary event conditions over world histories, e.g. conditions that must hold over intervals in the past. Restraining unit events to conditional effects allows to check event conditions by evaluating only the current state. Even with temporal actions it is thus possible to define a transition function for plans that maps states to states like in classical planning. As a consequence, the language BTPL could be designed to downward compatible to STRIPS: STRIPS actions are also BTPL actions, and STRIPS plans can be mapped to BTPL plans by a simple function that will be given at the end of the section.

Definition 3 *A* ground action *is a tuple $a = (pre, eff)$ where pre is a set of literals called the* preconditions, *and $eff = \{ue_1, \dots, ue_n\}$ where $ue_i \in \mathcal{E}$ is a set of unit events called the* effects *of the action.*

Intuitively, a can be executed in time point $p = (s, t)$ if the preconditions *pre* hold directly before that time point. Similarly, if condition c of unit event (c, e, d) holds before time point $(s', t + d)$ the effect e will become true at that time point. Unconditional effects are of the form $(\emptyset, e, d)$, whereas immediate, i.e. undelayed effects are of the form $(c, e, 0)$.

Before describing the semantics of plans we will give an example to show the realization of the action model in the domain description language BTPL. In this paper, we will not describe its syntax in detail. However, the mapping to the formal model should become clear by the example in Fig. 2 It shows a `fly` operator as known from logistics domains. When invoked `fly` will immediately delete the airplane being at its starting point. After a delay of `?dist` (facts of the form `(distance ?x ?y ?d)` must of course supply numeric values for the third parameter) the effect of landing at the destination will occur, but only if the condition of the plane moving still holds after time `?dist`. This allows to change or destroy *some* effects without necessarily needing to cancel the whole action, e.g. a plane might be shot (and thus not land) without changing the effect of the runway being cleared. A less cynical example showing the use of this feature in multi-agent domains would be a skeet shooting example: one agents releases the skeet, the standard effect of which would be its being in the air for some time and then dropping on the ground. Another agent's action of shooting might result in hitting the skeet and destroying the dropping effect. Fig. 2 also shows that effects need neither be conditional nor must they have a time delay.

```
(:action FLY
  :parameters (?airplane ?from ?to ?dist)
  :precondition
     (and (at ?airplane ?from)
          (distance ?from ?to ?dist))
  :effect
      (and
         (not (at ?airplane ?from))
         (moving ?airplane)
         [+?dist](when (moving ?airplane)
                    ((not (moving ?airplane))
                     (at ?airplane ?to)))))
```

Figure 2: Temporal operator `fly` in the Logistics domain.

Since there is no total order on the actions in concurrent plans, such plans cannot be described by action sequences like in classical planning; instead we attribute each action in a plan an absolute beginning time stamp.

Definition 4 *A* plan *is a set of tuples $P = \{(a_1, t_1), \dots, (a_n, t_n)\}$ where $a_i \in \mathcal{A}, t_i \in \mathcal{T}$. The set of plans is named $\mathcal{P}$.*

In order to define the transition function for plans we use the following auxiliary functions to access components of plans:

Definition 5 $A(P, t) \stackrel{def}{=} \{(a, t') \in P \mid t' = t\}$ *selects all actions in plan P that start at time t.*

applicable$(A, s) \stackrel{def}{=} \{a \in A \mid pre(a) \subseteq s\}$ *selects all actions in A that are applicable in state s.*

absolute$(A, t) \stackrel{def}{=} \bigcup_{a \in A}\{(c, e, t + d) \mid (c, e, d) \in eff_a\}$ *returns a set of unit events with absolute times that corresponds to the execution of action set A at time t.*

triggered$(E, s) \stackrel{def}{=} \bigcup_{(c,e,d) \in E \,\wedge\, c \subseteq s} e$ *gives the set of effects from a set of unit events E the conditions of which hold in state s.*

dels$(L) \stackrel{def}{=} \{a \mid \neg a \in L\}$ *and adds*$(L) \stackrel{def}{=} \{a \mid a \in L\}$ *select negative and positive literals from a set L that will be interpreted as delete and add effects, respectively.*

Now we can define the transition function for plans that computes the state resulting from the execution of a plan P in a state s. Since actions can be arbitrarily interleaved, unit events become the base unit of transition. To determine the order of these events and to compute their effects, we use an extended transition function *trans* : $\mathcal{S} \times \mathcal{P} \times \mathcal{E} \times \mathcal{T} \to \mathcal{S}$. Intuitively, *trans*$(s, P, E, t)$ will return the state resulting from executing plan P at time point (s, t) where the events in the event queue E are already scheduled for future time points.

The function *trans* is defined inductively as follows:

Definition 6 *trans* : $\mathcal{S} \times \mathcal{P} \times \mathcal{E} \times \mathcal{T} \to \mathcal{S}$

$$trans(s, P, E, t) \stackrel{def}{=} \begin{cases} s & \text{if } P = E = \emptyset \\ trans(s', P', E', t') & \text{otherwise} \end{cases}$$

where the auxiliary values are attributed as follows:

$$\begin{array}{lcl} E_0^+ & = & E \cup absolute(applicable(A(P,t),s),t) \\ E_0 & = & \{(c, e, t') \in E_0^+ \mid t' = t\} \\ E' & = & E_0^+ \setminus E_0 \\ effs & = & triggered(E_0, s) \\ s' & = & s \setminus dels(effs) \cup adds(effs) \\ P' & = & P \setminus A(P, t) \\ t' & = & \min(t'' \mid (a, t'') \in P \vee (c, e, t'') \in E') \end{array}$$

The *trans* function is directly transferable to a plan execution algorithm (Fig. 3) that should further explain the *trans* function.

We can now express what it means for a BTPL plan to solve a planning problem.

Definition 7 *A* planning problem *is a tuple $p = (I, G)$ where $I \in \mathcal{S}$ is called the initial state and $G \subseteq \mathcal{S}$.*

A plan P is a solution *to the problem p iff trans$(I, P, \emptyset, 0) \in G$.*

As STRIPS actions are also BTPL actions we can define

Definition 8 *The function m that maps* STRIPS *plans to* BTPL *plans is defined as*

$m(\langle a_1, \dots, a_k\rangle) \stackrel{def}{=} \{(a_1, 1), \dots, (a_k, k)\}$.

The actual unit of time is unimportant for the mapping of sequential plans or GRAPHPLAN's time-step parallel plans that both use only immediate effects. When a domain uses combined immediate and delayed effects, BTPL allows either to define the minimal time delay between two events where

For a plan P to be executed in time point (s, t)

0. If plan P is empty and future events $E = \emptyset$, return s

1. Select set A of actions in P that are to be applied at time t

2. For those $a \in A$ where $pre_a \subseteq s$ build the union over all their unit effects and join it with the scheduled effects E to form E_0^+

3. Select set $E_0 \subseteq E_0^+$ of all unit events at time t

4. Select set *eff* of effects e where unit effect $(c, e, t) \in E_0$ is triggered in s, i.e. $c \subseteq s$

5. Apply effects *eff* to s according to the STRIPS semantics

6. Remove executed/scheduled actions A from P, update E by removing executed unit events E_0 from E_0^+

7. Find the next time point t where either an action $a \in P$ shall be executed or an event $e \in E$ is scheduled

8. Go to step 0

Figure 3: Plan execution algorithm. Executes plan P in state s and returns final state s'.

the first enables the second or to describe that delay by a special infinitesimal time constant ϵ. The description of these features and their semantic properties lies beyond the scope of this paper.

The basic formal model for temporal planning presented in this section will be extended in three different ways in the following sections. For presentation and space reasons, all are presented as extensions to the basic model but can also be combined to a very expressive semantics for concurrent planning.

4 From states to world histories

The transition function *trans* returns a single state, the state of the world at the time when no further planned changes will occur. But a plan execution can also be seen to produce a complete world history: the changes to the world done by the plan over time. In this section we will adopt this point of view. While the semantics of a plan execution must be only slightly changed to map world histories to world histories instead of states to states, the real benefits of the new perspective lie in richer descriptions of planning problems and domains.

Initial situations become world histories, too, and goals are conditions on world histories (or sets of acceptable world histories, respectively). For the initial situation we allow one further extension: in addition to an initial history, i.e. the past, problem designers can also specify sets of events in the future. If these events are unconditional they describe exogenous changes over time that are not caused by planned actions, if they are conditional they model events that happen only when *triggered* by actions in the plan. Viewing goals as conditions on world histories allows to describe different goal states at different time points, limits of plan duration, maintenance goals over intervals etc. Operator descriptions are extended in similar ways: preconditions over intervals, maintenance conditions for unit events, conditions over time points in the more distant past can be specified. The extended language providing those features will be called BTPL$^+$.

We allow preconditions (unit event conditions) to range over world histories *in the past* of an action's (event's) execution time.

Definition 9 *An* interval condition *is a tuple* $ci = (c, t_1, t_2)$ *where* c *is a set of literals and* t_1 *and* t_2 *are time stamps with* $t_1 < t_2$. $\mathcal{I}$ *is the set of interval conditions. A* history condition *is a set of interval conditions* $ch = \{ci_1, \ldots, ci_n\}$. *When used in preconditions or unit effect conditions of operator descriptions, all time stamps used in the interval conditions are given in relation to the event's time stamp and must be of* negative *value to represent their being* before *the event.*

Definition 2 and Definition 3 are changed accordingly: preconditions and event conditions (as well as goals) become *history conditions*.

Definition 10 *To evaluate conditions over histories we define*
$iholds(ci, h)$ *iff* $\exists (s,t) \in h.(t \leq t_1) \wedge \forall t' \in [t, t_2).(s', t') \in h \rightarrow c \subseteq s'$ *where* $ci = (c, t_1, t_2)$
$hholds(ch, h)$ *iff* $\forall ci \in ch.iholds(ci, h)$ *where* $ch \subseteq \mathcal{I}$.

The definition of the transition function *trans*$^+$ is nearly identical to the function *trans* of standard BTPL, except that "past" time points are kept in the execution history so that goals and (pre-)conditions can be checked against it.

Definition 11 *trans* : $\mathcal{H} \times \mathcal{P} \times \mathcal{E} \times \mathcal{T} \rightarrow \mathcal{H}$

$$trans^+(h, P, E, t) \stackrel{def}{=} \begin{cases} h & \text{if } P = E = \emptyset \\ trans+(h', P', E', t') & \text{otherwise,} \end{cases}$$

where P', E', and t' are computed as for standard BTPL except that preconditions and event conditions are checked with *hhold*, and that $h'_u = h_u \cup (s', t)$ where s' is computed as for standard BTPL.

The initial situation of a planning problem p in BTPL$^+$ is defined by an initial world history h and a queue of future unit events E where the future events must happen *after* the last time point in the history. Goals are sets of history conditions.

Definition 12 *A* planning problem *is a tuple* $p = (h, E, G)$ *where* $h \in \mathcal{H}$ *is called the* initial history, $E \in \mathcal{E}$ *is the* initial event queue *and* $G \subseteq 2^{\mathcal{I}}$, *and where the following condition holds:* $\max(t|(s,t) \in h) \leq min(t|(c,e,t) \in E)$.
A plan P *is a* solution *to the problem* $p = (h, E, G)$ *iff*

$$t >= \max(t'|(s,t') \in h) \wedge hholds(G, trans(h, P, E, t))$$

holds for $t = \min(t' \mid (a, t') \in P \ \vee \ (c, e, t') \in E')$.

5 Shared resources and asynchronous concurrency

The example given in Fig. 4 shows that the semantics for concurrent plans as defined in the previous sections may result in undesirable states! An example: for a state s where `(in-bowl 2)` holds we apply two operators `DROP-APPLES(3 2 5)` and `DROP-APPLES(4 2 6)` in parallel. The resulting state contains both `(in-bowl 5)` and `(in-bowl 6)` which is inconsistent with the obvious intention to have always *one* fact of form `(in-bowl ?x)` and to use it as a *resource variable*, namely a *counter* for the number of

```
(:action DROP-APPLES
  :parameters (?count ?old-val ?new-val)
  :precondition
    (and (in-bowl ?old-val)
         (sum ?old-val ?count ?new-val))
  :effect
    (and (not (in-bowl ?old-val))
         (in-bowl ?new-val)))
```

Figure 4: The fruit bowl domain. PDDL version.

apples in the bowl. The modeling of the domain is correct while only dealing with *sequential plans* but it fails in the concurrent case. What is even worse is that neither STRIPS nor PDDL allow to model the fruit bowl domain in any other way that would allow concurrent applications of actions like `DROP-APPLES` and give a meaningful result.

At the heart of the problem lies the perception that both operators are *interfering* with each other by deleting their mutual preconditions, i.e. they could not be executed after each other. This leads to GRAPHPLAN's solution to the problem: the actions are detected to be *mutually exclusive* and the planner will not attempt to execute them in parallel. However, in the fruit bowl domain (as well as in many others) we would like to *allow* concurrent executions of such actions. The solution we propose is therefore to define a less strict notion of mutual exclusivity (or broader notion of concurrent applicability, respectively) that allows more operators to be parallelized and give the concurrent executions a meaningful semantics[1].

A definition that resembles common specifications of concurrent transition systems is the following (adapted from [Wolper & Godefroid, 1993]):

Definition 13 *Two state transitions t_1 and t_2 over the set of states $\mathcal{S}$ can possibly be executed in* asynchronous concurrency *if the following two conditions are true in all states $s \in \mathcal{S}$:*

1. *if t_1 [t_2] is enabled in s and $t_1(s) = s'$ [$t_2(s) = s'$], then t_2 [t_1] is enabled in s iff t_2 [t_1] is enabled in s' (independent transitions can neither disable or enable each other); and*
2. *if t_1 and t_2 are enabled in s, then there is a unique state s' such that both $t_1(t_2(s)) = s'$ and $t_2(t_1(s)) = s'$ (commutativity of enabled independent transitions).*

The actions are then said to be (pair-)independent. *The result of the asynchronously concurrent execution in s is defined as s'.*

GRAPHPLAN's mutexes represent a direct (negative) application of this definition to STRIPS operators. However, intuitively, the problem in the fruit bowl example results from the "rigidity" of the STRIPS-style action description. The old and new number of apples must appear as previously fixed parameters and parts of the preconditions whereas intuitively these values do not matter for determining if the action is applicable but are just needed to compute the counter updates[2].

The key idea to solve this is to allow unbound parameters in actions that will only be instantiated during execution of the plan. These parameters can be different in differing serializations of a set of asynchronously concurrent actions A. As long as it can be guaranteed (like in GRAPHPLAN plans) that all serializations result in the same state the semantics for the concurrent execution of A is well-defined.

Syntactically, we will extend BTPL by the `:vars` construct that was part of the first definition of PDDL [McDermott & others, 1998] but the semantics of which were never clearly specified and that is now no longer part of PDDL. We reintroduce it into BTPL because it provides a way to describe accesses to resource variables without more complex features like real numeric values, functions, global variables etc. Fig. 5 shows the fruit bowl example in the (only minimally different) BTPL version that allows Definition 13 to be applied. We apply `DROP-APPLES(3)` and `(DROP-APPLES(4)` in parallel in s where `(in-bowl 2)` holds. Although the values of `?old-val` and `?new-val` change after the application of one of the operators the other one stays enabled. In the resulting state s' `(in-bowl 9)` holds.

```
(:action DROP-APPLES
  :parameters (?count)
  :vars (?old-val ?new-val)
  :precondition
    (and (in-bowl ?old-val)
         (sum ?old-val ?count ?new-val))
  :effect
    (and (not (in-bowl ?old-val))
         (in-bowl ?new-val)))
```

Figure 5: The fruit bowl domain. BTPL version.

The semantics of `:vars` can be intuitively characterized as follows: for an action a to be applicable in a state s there must be exactly one instantiation of the set of variables v such that the preconditions *pre* of a hold in s. Assuming a closed world we can model the possible instantiations of variables and the corresponding possible unit events as sets of constants and sets of (ground) unit events, respectively.

Definition 14 *A* variable precondition *is a set of sets of literals $pre_v = \{pre_1, \ldots, pre_n\}$. A* variable unit event *is a tuple $ue_v = (ce, t)$ where t is a time stamp and $ce = \{(c_1, e_1), \ldots, (c_n, e_n)\}$ is a set of condition/effect pairs. The set pre of an operator's preconditions is a set of variable preconditions (where singleton sets denote invariable preconditions). The set eff of an operator's effects is a set of variable unit events. We call the set of variable unit events $\mathcal{E}_v$.*

We will now describe the changes to the transition function. We assume that a plan contains only actions that are independent in the sense of Definition 13; the question how a planner can find sets of independent actions is addressed later in this section. The key idea for determining the result of the

[1]The fruit bowl example is of course nothing more than addition and subtraction of natural numbers. The methods proposed in [Rintannen & Jungholt, 1999] can be used to check concurrent applicability in that case. However, we will give other examples with non-numeric variables and non-trivial combined effects of actions where these methods are not applicable.

[2]It should be stressed that for this problem to occur it is unimportant whether a plan description language supports numeric values or not.

concurrent variable events is (instead of accumulating all effects) to simulate it by a sequential execution. As the events are assumed independent each possible sequence will lead to the same state, so arbitrarily one is selected and executed.

The auxiliary functions dealing with unit effects are slightly changed and extended:

Definition 15 *We define*

triggered$(E, s) \stackrel{def}{=} \{(cv, t) \in E \mid \exists!(c_i, e_i) \in cv.c_i \subseteq s\}$ *gives the set of variable events in* E *where exactly one instantiation is triggered in* s.

instance(ce,s) $\stackrel{def}{=} \{(c_i, e_i) \in ce \mid c_i \subseteq s\}$ *selects that instance.*

The only change in the actual transition function is the updating of state s to s' by the triggered events *effs* $=$ *triggered*(E_0, s). The new state s' is determined by iteratively applying the events. We assign $s' = seq(s, effs)$ where

Definition 16 $seq : \mathcal{S} \times 2^{\mathcal{E}_v} \to \mathcal{S}$

$$seq(s, te) \stackrel{def}{=} \begin{cases} s & \text{if } te = \emptyset \\ seq(s', te') & \text{otherwise, where} \end{cases}$$

ce is some element of te
$te' = te \setminus ce$
$ie = instance(ce, s)$
$s' = s \setminus dels(ie) \cup adds(ie)$

While this semantics gives a reasonable meaning to the concurrent execution of sets of independent actions (events) the key question is of course how to determine such sets during planning. First of all, we must define how Def. 13 can be extended to sets of more than two transitions. The obvious definition would be:

Definition 17 *A set of transitions* $T = \{t_1, \ldots, t_n\}$ *is called* (set-)independent *if the following two conditions are true in all states* $s \in \mathcal{S}$ *for all* $t_i \in T$. *Let* $T_e \subseteq T$ *be the set of transitions that are enabled in* s.

1. $t_i \in T_e$ *iff* t_i *is enabled in all states* s' *where* s' *is the state resulting from the execution of some serialization* $seq_{T'}$ *of some set* $T' \subseteq T \setminus t_i$ *(independent transitions can neither disable or enable each other); and*
2. *there is a unique state* s' *such that for any sequence* $seq_{T_e} = \langle t_{i_1}, \ldots, t_{i_n} \rangle$ *where* $t_{i_j} \in T_e$, $seq_{T_e}(s) = s'$ *holds.*

For $n = 2$ this boils down to Def. 13. The definition expresses the idea of asynchronous concurrency, i.e. the order of events "doesn't matter" as long as the result is the same. However, in order to verify the conditions for an n element set, $n!$ serializations have to be checked which is intractable, especially when it has to be done for many possible action combinations during a search process. Yet it could be necessary: when arbitrary unit events (or conditional effects) are allowed it is easy to construct examples where all but one serialization of a set of (variable) events lead to the same state whereas one special serialization gives a different result and thus makes the set dependent[3].

As arbitrary conditional effects or unit events are the key problem our solution is to restrict asynchronous concurrency to those sets of events E where the concatenation operator $(;)$ is *commutative* for all pairs in E in all states. This would allow to check and execute only a single arbitrary serialization because the property guarantees that any other permutation leads to the same state.

Theorem 1 *A set of transitions* $T = \{t_1, \ldots, t_n\}$ *where all pairs* t_i *and* t_j *(with* $i \neq j$*) are* pair-independent *is* set-independent.

We omit the proof by induction here. As a consequence of Theorem 1, we can greatly simplify the checks for concurrent applicability if we restrict asynchronous concurrency to sets of pair-independent events. Instead of checking if all $n!$ serializations of n events are enabled and lead to the same state we just need to check if all $n(n+1)/2$ pairs are pair-independent. If this is the case, it follows from Theorem 1 that any arbitrary serialization of a set of events might be used to check if the events might happen asynchronously parallel and if so what the unique resulting state will be. (This is trivially the case for GRAPHPLAN-like non-mutex events.)

Theorem 1 is interesting in so far that it does not restrict transitions, i.e. events or actions, to specific syntactical forms or demands certain properties of their semantics. This allows the principle to be applied in other formalisms than BTPL as well. It is also possible to enable domain designers to explicitly state pair-independence of certain operators in a domain. E.g. this could be done for arithmetic operators like addition and subtraction or explicitly be prohibited for others like addition and multiplication. In such cases the search space for verification of possible concurrency is significantly reduced.

If pair-independence is not explicitly stated it must be automatically shown by the planner. For sets of variable unit events we have developed an algorithm that runs in $O(n^2)$ where n is the maximum of the two variable events' number of unit events. We will not describe the algorithm here because of the paper's purpose to mainly present BTPL's possibilities to describe concurrent planning domains, but will give another example of a planning domain where asynchronous concurrency is of great value.

The Mixing Paint domain (Fig. 6) describes a setting where one or more agents mix color in a bucket in order to paint a wall. The problem for STRIPS-like description languages is of course that the color currently in the bucket must be input to the action of pouring a new one in the bucket in order to determine the resulting color. This makes several pouring actions mutually exclusive. Moreover, mixing colors is not easily mapped to arithmetic relations between numerical variables as the database of color mixtures in Fig. 6 shows;

[3]This turns out to be the problematic part in the definition of concurrent applicability in [Bacchus & Ady, 2000] where possible concurrency is checked only by testing *one* arbitrary serialization, namely the order in which the actions where entered into the plan. The problem can be seen in the simple example of checking if the actions `t1` (for "take one apple") and `ta` (for "take all apples") can be parallelized. If the two actions are entered into the plan as `t1;ta` this gives a meaningful result, whereas if `ta` is executed first `t1` is no longer enabled. {`t1`,`ta`} should thus be not allowed to be executed in parallel.

```
(:action POUR-COLOR
  :parameters (?color)
  :vars (?old-mix ?new-mix)
  :precondition
    (and (color-bucket ?old-mix)
         (mixes ?old-mix ?color ?new-mix))
  :effect
    (and (not (color-bucket ?old-mix))
         (color-bucket ?new-mix)))

(:database
  (mixes nothing red red)
  ...
  (mixes red blue violet)
  (mixes red yellow orange)
  ...
  (mixes green orange brown)
  ... )
```

Figure 6: The Mixing Paint domain.

Rintannen's approach thus seems not suited to describe the fact that realistically colors may be mixed in series as well as in parallel. Our verification algorithm is capable to use the POUR-COLOR operator plus the database to show that concurrent application of several different action instances is possible.

6 Synchronous concurrency and causal domain laws

The most interesting property of asynchronous concurrency is that the semantics of the parallel application of a set of actions can be derived from the individual actions. There are, however, effects of concurrent actions that are "more than the sum of the parts", i.e. a group of actions forms a compound action with new properties.

A classic example is the simple domain where two agents try to lift the ends of a table. Each time one of the agents lifts one end a vase sitting on the table falls down and breaks. Something completely different happens at the moment the two agent lift the table simultaneously: not only does the vase not fall down, but the table rises into the air, an effect produced by none of the two actions but by the compound action they have formed.

In a framework with delayed effects like BTPL it would of course be possible to model this by making the vase's falling down a delayed effect of the first action and thus giving the second action time to add facts that destroy the event condition for falling. The second event could also have the conditional effect of making the table rise if it is already pulled. However, from a conceptual point of view, this is problematic in two ways:

- immediateness of action effects is sacrificed only because of modeling reasons
- the second actions causes the rising of the table whereas conceptually it is neither action but their occurring together that produces the effect. One could even go further by saying that the laws of the domain have determined the rising of the table as a consequence of two simultaneous attempts to raise one end.

Because of these reasons we introduce the possibility to describe so-called *causal domain laws*[4].

Definition 18 *A* causal domain law *is a unit event* $l = (c, e, 0)$. *The set of causal domain laws is called* $\mathcal{L}$.

Causal laws are not part of an operator but can be supplied by a domain designer to describe specific effects of synchronous events in a domain. Intuitively, every time that "something happens", i.e. the world history is updated, causal domain laws are triggered in the new state and may change it. The state as computed by the standard transition function is in a sense an *attempt*[5] which may or may not be consistent with the domain laws. In the table lifting domain this can be exploited by giving actions only effects like `(raised_left)` and let the real effects of this attempt be computed by the causal domain laws.

To incorporate causal domain laws into the basic transition function of section 3 we define it as before except

Definition 19 $s' = trans(s'', \emptyset, L, 0)$ *where*
s'' *is computed as* s' *in Definition 6*
L *is the set of causal laws*

An interesting extension to this concept proposed in [Thielscher, 1995] is to define a partial order on the set of causal laws that allows to apply only the "most specific" applicable laws in a state s. While the definition is easy it lies beyond the scope of this paper.

7 Summary and outlook

We have presented a basic formal model for planning in concurrent domains. Various extensions of the semantics allow to model increasingly complex situations, operators and their concurrent application. For all of these extensions reasonable semantics have been defined that above all will allow automatic planning for concurrent domains specified in the language BTPL. We did not describe any planning algorithms on purpose: we believe the described semantics to be general enough to be used in many planning applications using concurrency. Our own basic motivation comes from the field of multi-agent planning, i.e. distributed planning for distributed execution, but single-agent planning for multi-agent execution and other variants are also conceivable. Moreover, many algorithmic approaches should be usable with the formal model: total-order as well as partial-order planning, forward as well as backward search. Currently, expressive domain models like ours seem to be most easily tackled by forward search because explicit models (states) are searched rather than theories (sets of states), but this is only partially true for non-classical applications like multi-agent planning where some assumptions (like permanent omniscience of the world' state and changes) may no longer be kept. We have presented BTPL and its semantics in order to encourage domain engineers to use it in describing concurrent domains. Of course, to develop planning algorithms and evaluate their efficiency as well as the quality of the plans produced is the main task to be addressed in future work.

[4] This part of our work is mainly inspired by [Thielscher, 1995]

[5] See [Karlsson & Gustafsson, 1999] for a similar point of view.

References

[Bacchus & Ady, 2000] Bacchus, F., and Ady, M. 2000. Planning with resources and concurrency: a forward chaining approach. In *AAAI-2000 Workshop on the Integration of AI and OR Techniques for Combinatorial Optimization*.

[Bacchus, 2000] Bacchus, F. 2000. *Subset of PDDL for the AIPS-2000 Planning Competition*.

[Blum & Furst, 1997] Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1–2):279–298.

[Ghallab & Laruelle, 1994] Ghallab, M., and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. In *Proc. of AIPS '94*.

[Karlsson & Gustafsson, 1999] Karlsson, L., and Gustafsson, J. 1999. Reasoning about concurrent interaction. *Journal of Logic and Computation* 9(5):623–650.

[Köhler, 1998] Köhler, J. 1998. Planning under resource constraints. In *Proc. of ECAI '98*.

[Kvarnström, Doherty, & Haslum, 2000] Kvarnström, J.; Doherty, P.; and Haslum, P. 2000. Extending TALplanner with concurrency and resources. In *Proc. of ECAI '00*.

[McDermott & others, 1998] McDermott, D., et al. 1998. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Comitee.

[Pelavin, 1991] Pelavin, R. 1991. *Reasoning about Plans*. Morgan Kaufmann. chapter Planning with Simultaneous Actions and External Events.

[Penberthy & Weld, 1994] Penberthy, J., and Weld, D. 1994. Temporal planning with continuous change. In *Proc. of AAAI '94*.

[Rintannen & Jungholt, 1999] Rintannen, J., and Jungholt, H. 1999. Numeric state variables in constraint-based planning. In *Proc. of ECP '99*.

[Smith & Weld, 1999] Smith, D. E., and Weld, D. S. 1999. Temporal planning with mutual exclusion reasoning. In *Proc. of IJCAI '99)*, 326–337.

[Thielscher, 1995] Thielscher, M. 1995. The logic of dynamic systems. In *Proc. of IJCAI '95*.

[Wolper & Godefroid, 1993] Wolper, P., and Godefroid, P. 1993. Partial-order methods for temporal verifications. In *Proc. of CONCUR '93*, volume 715, 233–246. Springer.