Concurrent Action Execution with Shared Fluents

Michael Buro and Alexander Kovarsky
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{mburo,kovarsky}@cs.ualberta.ca

Abstract

Concurrent action execution is important for plan-length minimization. However, action specifications are often limited to avoid conflicts arising from precondition/effect interactions. PDDL — the planning domain definition language — for example, implements the "no moving targets" rule, which means that no two actions can simultaneously make use of a value if one of the two is updating the value. This rule poses problems for resource allocation planning in which resource values are accessed in preconditions and effects. A simple example is construction actions that consume certain amounts of a resource. For speeding up plan execution, we would like to be able to dispatch several construction actions simultaneously. Because action preconditions depend on resource values and action effects change them, the "no moving targets" rule does not allow concurrent execution. However, if sufficient resources are available, executing actions simultaneously poses no problems. This paper addresses the problem of deciding whether a set of actions produced by a planning system can be executed concurrently in the presence of fluent variables that occur in both action preconditions and effects. We first motivate the concurrent action execution problem by introducing a fair action scheduling algorithm for real-time strategy (RTS) games. Then we prove that the general decision problem, when restricting effects and preconditions to polynomial time computations, is co-NP complete. Thereafter, we focus on problem restrictions based on commutative operators which allow us to specify sufficient conditions for concurrent executability that can be checked quickly if the number of shared fluents is small. Finally, we apply these findings to action execution with shared resources in RTS games.

Introduction

Planning is an important cognitive process in which steps towards reaching a goal are laid out before we start acting. For planning problems that involve multiple acting entities it is preferable to schedule actions concurrently so that the total plan execution time is minimized.

In classical planning actions cannot execute concurrently. Temporal planning addresses this problem and is thus applicable to a wider range of problems. Concurrent execution of actions, however, poses new challenges. Take for instance the job scheduling problem in which we try to assign jobs to processors with the objective of minimizing the parallel execution time. Deciding whether there exists a schedule that runs for at most time T is known to be NP-complete, whereas all sequential schedules are valid and have the same easy to compute execution time span. In the general planning setting, concurrent action executability depends on the interdependence among action preconditions and effects. If all actions are independent of one another, such actions can be executed concurrently. In many situations, however, several actions access the same variables. We call such variables (or fluents) shared. Consider, for example, production problems in which a final product is built from raw materials called resources. In such domains there can be dependencies between actions: some actions may accumulate certain resources, while other actions consume resources to produce something. In a situation where there is a single resource we need to check whether several resource accumulating and resource consuming actions are executable simultaneously. The executability of such a set of actions depends on the accumulated resource amount so far, the amount being produced, the amount being consumed, and other variables that may effect preconditions and effects. In general, such a computation may not be trivial.

In the popular genre of real-time strategy (RTS) video games concurrent execution of actions that use shared resources is common. Typical RTS games are fast-paced simulations, in which players try to eliminate opponents in real-time by securing resources which are used to build military equipment. At the beginning of an RTS game, players send out workers to gather resources, such as gold and lumber, which are needed to build structures (e.g., barracks and tank factories) that in turn train/produce other units such as infantry or tanks. After scouting and creating a sufficient attack force, units are sent into battle. There are many challenges in RTS games that are of interest to AI researchers, including multi-object pathfinding, learning squad strategies, plan recognition, and opponent modeling (Buro 2003).

Our research focus is on the build-order optimization problem for RTS games, in which resource gathering and the creation of unit-producing buildings and units themselves in the initial stage of an RTS game is optimized. In typical problem instances, several actions can execute in parallel at any given time as long as there are sufficient shared re-

sources available for each action. For example, two workers can start building two structures that require shared resources at the same time, as long as there is enough building material available. In such a situation the ability to execute actions concurrently will result in more efficient (shorter) plans when compared with the sequential execution. Thus, efficiently determining whether a set of actions that uses shared resources is executable concurrently — which is the focus of this paper — is important.

In what follows, we first give an overview of related previous work in the area of planning with concurrency and fluents. Then we describe our model of concurrent action execution, which is motivated by action execution in RTS games. Thereafter, we show that the general decision problem of determining concurrent execution of actions is co-NP complete, followed by presenting tractable problem simplifications and applying them to the build-order optimization problem in RTS games. We also discuss the problem of generating concurrent action sets, which is closely related to this research and finish the paper with conclusions and future research directions.

Related Work

Execution of actions with numeric fluents has been studied extensively (Lee & Lifschitz 2001) (Erdem & Gabaldon 2005). The main focus of this research was the formalization of the semantics of concurrently executed actions with fluents. (Brenner 2001) discussed various forms of concurrency and event interactions and devised a description language for concurrent domains. (Boutilier & Brafman 2001) developed a partial-order non-temporal planner (POMP) that can generate plans for actions with concurrent interacting effects. To achieve this the STRIPS action representation language was modified. (Rohanimanesh & Mahadevan 2001) showed that a planning model under uncertainty, which allows concurrent action execution, produces shorter plans than a model that does not. In (Bacchus & Ady 2001) the focus was extending the types of problems that could be addressed by planners to problems with concurrent execution and metric resources. Using forward chaining with heuristics and domain specific information the TLplan planner generated concurrent plans.

As far as we know, however, the computational effort required for determining concurrent executability of actions with numerical fluents has not been addressed in previous research. Furthermore, the most widely used planning language PDDL (McDermott & AIPS'98 Committee 1998) sidetracked the issue of concurrent execution by not allowing simultaneous use of fluents. PDDL was later extended to deal with temporal planning problems (version 2.1, (Fox & Long 2003)) by allowing actions to execute simultaneously as long as there is no potential conflict. In PDDL 2.1, when one action obtains a certain resource to update its value, PDDL adopts a no "moving targets" rule on that resource, subsequently preventing other actions from accessing that resource until the original action finishes its execution.

Concurrent Action Execution

In order to better understand the issues of concurrently executing actions, let us examine the structure of actions. In most planning languages (both classical and temporal) an action is divided into two parts: preconditions and effects. A precondition is a Boolean function that must evaluate to true for an action to execute, while an effect is a certain consequence of executing that action. For example, a precondition could be of the form iron ≥ 100, where iron is a resource fluent, and the associated effect could be "build control center and deduct 100 from iron amount". In non-temporal planning, actions are executed sequentially one after another, thus only one action is executed at a time. Also, actions are instantaneous and the effects are executed immediately.

Temporal planning allows for concurrent execution of several actions and for actions to take more than one step to execute. Allowing temporal execution means that preconditions are checked only at certain times during action execution and that effects are also executed only at certain times. The focus of this paper, however, is the concurrent execution of actions. If we want several actions to execute at the same time, it is not sufficient to check whether each action's individual preconditions are satisfied. The preconditions of two or more actions in the set might conflict with each other, allowing the actions to be executed individually but not in concert with one another. For example, given a resource whose current amount is 100 and two actions that each require 100 of that resource, each action is executable individually, but not together. Similarly, effects can also determine whether two actions can be executed simultaneously. For example, if one action requires a certain resource amount which a second action creates, then we may want to allow the two actions to execute concurrently, irrespective of the current resource amount.

Additionally, in the planning literature a distinction is drawn with respect to *serializable* and *non-serializable* action sets. A set of actions is called serializable if the result of executing the actions sequentially is the same as the result of executing the actions concurrently. Non-serializable actions produce different results. Two actions are non-serializable if the precondition of each action requires the effect of the other action to be executed.

Given these complications and multiple views of what constitutes concurrent action execution, we need to give a precise definition of our execution model before we proceed. We motivate our choice by describing a fair action execution framework for RTS games.

A Fair Execution Framework for RTS Games

In typical RTS games multiple players are connected to either a central server — which sends out game state information regularly and receives action sequences from players — or directly to their peers, in which case all player computers run identical game simulations and exchange actions. Because the peer-to-peer mode is vulnerable to client hacks that potentially reveal state information the player is not supposed to know, server-client RTS game architectures are preferable.

In one RTS game simulation cycle the server receives actions from players, executes them in addition to internally scheduled actions, and sends out the resulting game state information back to the players (Figure 1). Typically, RTS game actions are scripted and action invocation triggered by players can be viewed as remote procedure calls that check preconditions, and if they succeed, *sequentially* execute effects that change the game state and potentially schedule subsequent actions by appending them to an action queue. Actions in RTS games span a wide spectrum of preconditions and effects. For example, units can move, collide, and attack each other, resources can be gathered and consumed to create game objects, and area effects can reveal portions of the playing field or alter the landscape. In addition, actions may not be commutative (e.g., whose attack action is executed first or who is moving first may have an advantage) and action preconditions may depend on other actions' effects. The question now becomes how actions from multiple sources can be executed in a concurrent yet fair fashion. We propose to shuffle all actions that are due to be executed in the current game simulation cycle (Figure 2) and execute all actions sequentially. In the long run, this strategy ensures fairness for all players involved in the game. Executing actions sequentially does not mean that we have given up on the concept of concurrency. Instead, we define concurrency as a serialized execution of actions within a very short time period. Looking from the perspective of a planner we can identify two time scales. The first is the *environment time scale* in which the temporal planner operates. It is divided into longer time periods called game cycles. The second is the much shorter *concurrent execution time scale*, in which action sets that were determined to be concurrently executable by their planners are executed in a *randomly-serialized ordering* by the server. Because the actual action execution ordering is unknown to the players, they have to make sure that all execution sequences of the transmitted action set are valid, if the action throughput is to be maximized. This leads us to the following formal definitions which form the basis for subsequent deliberations:

- An **action** is a pair (p, e), where p — the precondition — is a procedure that computes a Boolean value over fluent variables and e is an effect, i.e., a procedure that changes fluent variables. An action is **executable** if its precondition evaluates to true. If an action is executed, its effect procedure is run — changing fluent variables. Precondition and effect computations are considered atomic, i.e., at any given time only one action is being processed.
- An action sequence $\vec{a} = (a_1, \ldots, a_n)$, where $a_i = (p_i, e_i)$, is called **executable** starting with fluent vector $\vec{f}$ if and only if — when actions are executed sequentially in order $i = 1, \ldots, n$ — precondition p_i evaluates to true after $i - 1$ steps, i.e., at the time a_i is to be executed, its precondition is true.
- An execution instance $(\vec{f}, (a_1, \ldots, a_n))$ can be **executed concurrently** if and only if for all execution orderings $\pi \in S_n$ (the symmetric group of degree n) $(a_{\pi(1)}, \ldots a_{\pi(n)})$ is executable starting with fluent vector f and each execution order results in the same fluent vector.

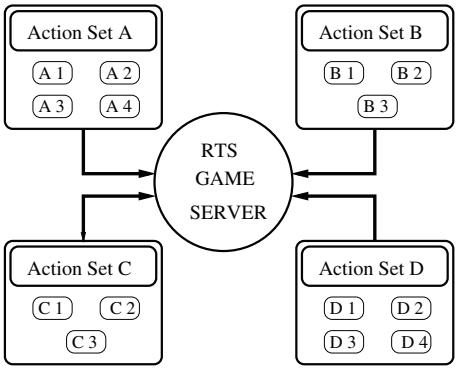

Figure 1: The RTS game server has received four action sets from players A, B, C, and D in the current game cycle. All actions need to be executed.

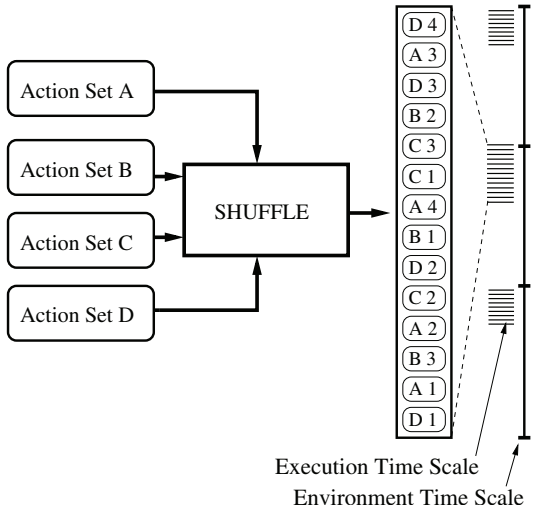

Figure 2: To ensure fairness, actions are shuffled and then executed sequentially.

The proposed action execution model is not restricted to RTS games. It can be applied to discrete simulation systems where the environmental time proceeds at a slower pace than the time for executing interacting actions in sequence.

Worst Case Time Complexity

To study the complexity of deciding whether an execution instance is concurrently executable, we define the *Concurrent Action Execution* decision problem **CAE** as follows:

$$\text{CAE} := \{r = (\vec{f}, \vec{a}) \mid r \text{ can be executed concurrently}\}.$$

Moreover, CAE_P is the subset of CAE for which the fluent vector as well as precondition and effect computations are discrete and their runtime is polynomial. We now proceed to show that testing membership of CAE_P is a hard problem.

Theorem 1: CAE_P is co-NP-complete.

Proof: We show that the complement of CAE_P — CAE_P^c — is NP-complete by reducing the Subset Sum problem (SUBS) to CAE_P^c and proving that CAE_P^c is in NP.

The following algorithm shows that CAE_P^c can be recognized by a polynomial time bounded non-deterministic Turing machine: first we check whether the input encodes an action execution problem, say $(\vec{f}, (a_1, \ldots, a_n))$. If not, we accept. Otherwise, we proceed by guessing two execution orderings $\pi_1, \pi_2 \in S_n$ and checking whether a precondition is violated when executing π_1 or whether executing π_1 and π_2 create different effects. If so, we accept. Otherwise, the input is rejected. This algorithm runs in polynomial time and recognizes CAE_P^c, which therefore belongs to NP.

We now show $\text{SUBS} \leq \text{CAE}_P^c$, which completes the proof, because

$$\text{SUBS} := \{(K, \{s_1, \ldots, s_n\}) \mid K, s_i \in \mathbb{N}, \exists I \subseteq \{1, \ldots, n\} : \sum_{i \in I} s_i = K\},$$

is NP-complete (Cormen *et al.* 2001). We need to construct a polynomial time transformation function g with the following property:

$$w \in \text{SUBS} \;\Leftrightarrow\; g(w) \in \text{CAE}_P^c$$

Transformation g maps $(K, \{s_1, \ldots, s_n\})$ to $((s = 0), (a_1, \ldots, a_{n+1}))$, with

- a single numerical fluent s initialized with 0,
- $a_i = (\text{“ } s \neq K \text{ ”}, \text{“ } s := s + s_i \text{ ”})$ for $1 \leq i \leq n$, and
- $a_{n+1} = (\text{“ } s \neq K \text{ ”}, \text{“ ”})$

If $w \in \text{SUBS}$, then $\exists I \subseteq \{1, \ldots, n\} : \sum_{i \in I} s_i = K$. This means that action sequences that start with indexes from I fail to execute, because s reaches value K and the precondition of the subsequent action evaluates to false. Therefore, $g(w) \in \text{CAE}_P^c$.

Conversely, if $g(w) \in \text{CAE}_P^c$ then there exists an execution ordering where at least one precondition fails, because by construction we know that $g(w)$ is syntactically correct and its effects are commutative. We collect action indexes up to the failure point and call this set I. Then, by definition of the action effects in $g(w)$, $\sum_{i \in I} s_i = K$ follows, which concludes the proof. □

Problem Simplifications

Although the co-NP-completeness of CAE_P suggests that it may not be decidable in polynomial time, checking the executability of an instance $r = (\vec{f}, (a_1, \ldots, a_n))$ can be done in time $\Theta(n! \cdot poly(|\text{code}(r)|))$ by enumerating all execution orderings and checking preconditions and total effects. If n is small, this brute-force approach may still lead to acceptable runtimes. In this section we turn our attention to the case where n is big compared to the number of shared fluent variables, in which case – under certain restrictions – we will be able to check executability more quickly, while still maintaining sufficient expressiveness for many practical applications.

For an execution instance to be executable concurrently, each ordering has to result in the same fluent vector. One way to ensure this condition is to base effects on commutative operators $\circ$, i.e., effects are sequences consisting of assignments of the following form: $f_i := f_i \circ c$, where c is a constant. We call such effects commutative. The second condition for an execution instance to be executed concurrently is for all preconditions to hold in all execution orderings. Thus, as seen earlier, we could check every execution ordering to satisfy the second requirement. However, if we restrict preconditions, fewer checks may be required.

Consider, for example, operator $+$ applied to integers. Looking at a sequence of $+c_i$ operations applied to a fluent variable, we can quickly establish lower and upper bounds for that variable that are valid for any execution ordering. A sufficiently simple form of preconditions may then allow us to quickly test executability by simply checking preconditions at the extreme points.

In what follows, we will first generalize this idea and then present some specializations.

Corner and Box Tests

For the bounds checking idea to work we need the following:

- A totally ordered fluent value set,
- commutative effects of the form $f := f \circ c_i$, and
- a fast method of determining good lower and upper bounds l, u for f when considering all c_i values and arbitrary execution orders.

We then have the choice of either checking all preconditions for all values in interval $[l, u]$ — which, in case of being universally true, implies executability — or simply checking all preconditions at l and u, provided that being true there implies being true in (l, u) as well. This idea can be easily extended to multiple fluents:

Corner/Box Test. Given an execution instance $((f_1 \ldots f_m), (a_1 \ldots a_n))$ with $a_k = (p_k, e_k)$ and effects e_k of the form $f_{i_1} := f_{i_1} \circ c_1 \ldots f_{i_n} := f_{i_t} \circ c_t$, for each $k = 1 \ldots n$:

1. Compute vectors $(l_1^{(k)} \ldots l_m^{(k)})$ and $(u_1^{(k)} \ldots u_m^{(k)})$ of fluent variable lower and upper bounds for action a_k by considering all effects of actions a_i $(i \neq k)$.
2. Define box $B_k = \{(x_1 \ldots x_m) \mid \forall j : x_j \in [l_j^{(k)}, u_j^{(k)}]\}$ and corners $C_k = \{(x_1 \ldots x_m) \mid \forall j : x_j \in \{l_j^{(k)}, u_j^{(k)}\}\}$

Return true if and only if for all $k = 1 \ldots n$: $p_k(\vec{f})$ is true for all $\vec{f} \in C_k$ (Corner Test) or $\vec{f} \in B_k$ (Box Test)

The following theorem states that the Corner and Box Tests are sound and partially complete:

Theorem 2: Let $r = ((f_1 \ldots f_m), (a_1 \ldots a_n))$.

1. If the Box Test applied to r returns true, then r is executable concurrently.

2. If the Corner Test applied to r returns true and the **corner generalization** property

$$\forall k \in \{1 \dots n\} : (\forall \vec{f} \in C_k : \ p_k(\vec{f}\,)) \Rightarrow (\forall \vec{f} \in B_k : \ p_k(\vec{f}\,)) \quad (1)$$

holds, then r is executable concurrently.

3. If 1) all action effects in r only change a single fluent variable, 2) the Corner Test applied to r fails, and 3) the found lower and upper bounds can be reached by execution sequences, then r is not executable concurrently.

Proof: For each action a_k the Corner/Box Test determines lower and upper bounds for all fluents by taking all effects except e_k into account. If $p_k(\vec{f}\,)$ is true for all $\vec{f} \in B_k$, then irrespective of the action sequence preceding a_k, p_k will evaluate to true when a_k is to be executed, because the fluent vector at that time will be an element of B_k. This proves claim 1. Likewise, if $p_k(\vec{f}\,)$ is true for all $\vec{f} \in C_k$ and the corner generalization holds, then $p_k(\vec{f}\,)$ is true for all $\vec{f} \in B_k$, which proves claim 2 using the same argument. For claim 3, it suffices to produce an action sequence that violates a precondition. Because the upper and lower bounds are reachable and effects only change individual fluent variables, we can execute actions so that the extreme value of each fluent variable is reached. This generates a fluent vector for which the precondition fails. □

Both tests are therefore sound, but neither is generally complete, because not all fluent vectors in B_k or C_k may be reachable by any execution ordering. This can be easily verified by considering effects that change two fluents in a correlated fashion, say $f_1 := f_1 + 5; f_2 := f_2 + 10$ and $f_1 := f_1 - 3; f_2 := f_2 - 6$. The runtime of the Corner Test is $\Theta(2^m poly(|\text{code}(r)|))$, where m is the number of shared fluents and r is the execution instance. It can be much faster than the brute-force approach described earlier if $m < n$. The Box Test runtime depends on the size of the B_ks and is at least as high as for Corner Test.

As an example consider cumulative effects which are common in practice. As we have seen in the introduction, resource gathering and spending can be modeled by effects of the form $f := f + c$ applied to integer fluent f. Lower and upper bounds for such effects can be established by only considering negative effects and positive effects, respectively: for action a_k, $l_i^{(k)}$ [$u_i^{(k)}$] is the sum of all negative [positive] effects on fluent f_i in effects e_j $(j \neq k)$ added to the initial value of f_i. These bounds can be reached, because executing the respective actions in sequence will set f_i to $l_i^{(k)}$ or $u_i^{(k)}$, respectively. Lower and upper fluent bounds for other commutative effects such as $f := f \cdot c$, $f := \min(f, c)$, and $f := f^c$ can be computed similarly.

Corner-Generalizing Preconditions

In this subsection we will explore types of preconditions that have the corner generalization property which is required by the more efficient Corner Test. If we start with preconditions e_j, we can form conjunctions such as

$$e_1 \wedge e_2 \dots \wedge e_t$$

which have the corner generalization property if and only if all e_js have it. Disjunctions of corner-generalizing preconditions also generalize corners. However, the specificity of such conditions suffers, because disjunctions of non-corner-generalizing preconditions can be corner-generalizing. Negations of corner-generalizing preconditions generally do not retain this property.

When considering numerical fluents $\vec{f}$, a common condition type is

$$g(\vec{f}\,) \geq 0.$$

For such conditions p the corner generalization property (1) becomes:

$$(\forall \vec{f} \in C : \ g(\vec{f}\,) \geq 0) \ \Rightarrow \ (\forall \vec{f} \in B : \ g(\vec{f}\,) \geq 0),$$

where C is the set of corners of box B which is determined by the lower and upper bounds of the particular fluent variables. This property holds if g assumes its minimum value in the box at one of its corners, i.e.,

$$\forall \vec{f} \in B : \ g(\vec{f}\,) \geq \min_{\vec{h} \in C} g(\vec{h}) \quad (2)$$

If it is possible to locate the minimal corner quickly without having to check all corners, the runtime of the Corner Test can potentially become polynomial. Linear functions

$$g(f_1 \dots f_m) = w_0 + \sum_{i=1}^{m} w_i \cdot f_i$$

have this property because a minimal corner is given by

$$f_i = \begin{cases} l_i, & \text{if } w_i \geq 0 \\ u_i, & \text{otherwise} \end{cases}$$

Thus, if preconditions have form $w_0 + \sum_{i=1}^{m} w_i \cdot f_i \geq 0$ (or ≤ 0) the quick Corner Test only requires one evaluation rather than 2^m.

Slightly more general, property (2) applies to concave functions. This is a well-known result which can be proved by induction over the number of variables: in the one-dimensional case it follows from the definition of concavity:

$$g((1-\lambda)\vec{f_1} + \lambda\vec{f_2}) \geq (1-\lambda)g(\vec{f_1}) + \lambda g(\vec{f_2})$$

for all $\lambda \in [0, 1]$. In the general case, for non-corner points, we pick a variable f_i and consider the two box planes defined by $f_i = l_i$ and $f_i = u_i$. We then know by the induction hypothesis that the function values at these two points are at least as big as the respective minimal corners. Applying the definition of concavity once more to the endpoints finishes the induction proof.

Again, if the concave precondition function is sufficiently simple, it may allow us to find minimal corners quicker than by complete enumeration, and thus speed up the Corner Test considerably.

Application to RTS Game Action Execution

In this subsection we look at action execution in RTS games in light of the just established ideas on checking concurrent executability. Typical RTS games begin with a build-order

optimization problem where workers are sent out to gather resources which are then consumed to build other units (including workers) and structures, until a certain condition — such as creating a certain amount of units — is met. Construction actions (p, e) usually have the form

$$p = \quad f_1 \geq c_1 \wedge \ldots \wedge f_t \geq c_t$$
$$e = \quad f_1 := f_1 - c_1; \ldots f_t := f_t - c_t; \text{ create-obj(type)},$$

where the f_i represent amounts of resources — such as iron, lumber, or oil — and the c_i indicate the cost of the object to be built. Additionally, object creation may be delayed to account for construction time. Resource-gathering actions (p, e) typically look like:

$$p = \quad \text{“worker } w \text{ close to control center”} \wedge \text{“}w \text{ has mineral”}$$
$$e = \quad f_m := f_m + 1; \text{ “worker } w \text{ has no mineral”}$$

Here, worker w is returning a mineral to the control center with the effect of increasing the mineral count f_m by one and emptying the worker's mineral storage bin.

The total effect of a series of such resource gathering and construction actions is the same for each action ordering provided all preconditions evaluate to true regardless of the actual execution sequence. The construction action preconditions are conjunctions of linear inequalities that qualify for the quick Corner Test which only checks the minimal corner. Furthermore, the minimal corner for a specific action can be determined by summing up all costs for single resources excluding the costs accrued in the current action effect and subtracting these values from the initial resource amounts.

Generation of Concurrent Action Sets

A research issue closely related to concurrent action executability is the *generation* of action sets that can be executed concurrently by a planning system. Given a set of actions that are individually executable at a given time, the goal is to generate potentially concurrent sets of actions. Instead of generating action sets and then checking their concurrent executability — which can be inefficient — we could try to directly generate concurrently executable sets. Indeed such an approach can be used in domains with simple preconditions and effects. For example, when simplifying the just described RTS game scenario further to a bare minimum of one type of precondition $f \geq c$ and effect $f := f - c$, we can sequentially add actions to the concurrently executable set, as long as the available resource amount is bigger than the cumulative resource amount needed for all actions added so far. However, if several resources are used in the same precondition or if preconditions have a more complex structure such straightforward computation may not be sufficient and consequently directly generating actions sets may be infeasible. Furthermore, even in the simplest case it is possible to have a large number of potential action sets to generate, which compromises efficient real-time planning. A more effective approach to generating action sets in real-time could be to produce a small number of “promising” action sets using heuristic methods and then use the tests such as those proposed in this paper to check concurrent executability.

Conclusion and Future Work

The focus of this paper has been on deciding whether a given action sequence can be executed concurrently in the presence of fluents that are shared by preconditions and effects. By defining concurrent executability in terms of serializability and generating the same overall effect regardless of execution order, the “no moving targets” rule implemented by PDDL can be mitigated with the result that more actions can be scheduled in one time frame for simultaneous execution. In addition, the following research contributions have been made:

- a technique for fair and concurrent action execution in RTS games,
- proving the co-NP-completeness of the concurrent action execution decision problem for polynomial-time preconditions and effects, and
- establishing concurrent executability tests that can work faster than complete enumeration for a restricted yet expressive class of preconditions and effects.

We have started to look at the build-order optimization problem in RTS games as a prototypical planning domain that features object creation. The potentially large number of independently acting entities and relatively short action durations make it necessary to maximize the number of actions executed in a single execution frame. While optimization problems like this are generally NP-hard, we hope that the heuristics presented in this paper can help us to quickly test action sequences for their concurrent executability potential.

Acknowledgments

We thank Timothy Furtak for feedback on an earlier draft. Financial support was provided by NSERC and iCORE.

References

Bacchus, F., and Ady, M. 2001. Planning with resources and concurrency: A forward chaining approach. In *Proceedings of IJCAI*, 417–424.

Boutilier, C., and Brafman, R. 2001. Partial-order planning with concurrent interacting actions. *Journal of Artificial Intelligence* 14:105–136.

Brenner, M. 2001. A formal model of planning for concurrency. *Technical Report. Institute for Computer Science, Albert Ludwigs University, Freiburg, Germany*.

Buro, M. 2003. Real-time strategy games: A new AI research challenge. In *Proceedings of IJCAI*, 485–486.

Cormen, T.; Leiserson, C.; Rivest, R.; and Stein, C. 2001. *Introduction to Algorithms (2nd edition)*. MIT Press.

Erdem, E., and Gabaldon, A. 2005. Cumulative effects of concurrent actions on numeric-valued fluents. In *Proceedings of the AAAI Conference*, 627–632.

Fox, M., and Long, D. 2003. PDDL 2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence* 20:61–124.

Lee, J., and Lifschitz, V. 2001. Additive fluents. In *Proceedings of the AAAI Spring Symposium: Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, 116–123.

McDermott, D., and AIPS'98 Committee. 1998. PDDL – the planning domain definition language. *Technical Report. Department of Computing Science, Yale University*.

Rohanimanesh, K., and Mahadevan, S. 2001. Decision-theoretic planning with concurrent temporally extended actions. In *Proceedings of UAI*, 100–107.