MIT Open Access Articles

*Fast Distributed Multi-agent Plan Execution with Dynamic Task Assignment and Scheduling*

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

**Citation:** Shah, Julie A., Patrick R. Conrad, and Brian C. Williams. "Fast Distributed Multi-agent Plan Execution with Dynamic Task Assignment and Scheduling." 19th International Conference on Automated Planning and Scheduling, September 19-23, 2009, Thessaloniki, Greece.

**Publisher:** American Institute of Aeronautics and Astronautics

**Persistent URL:** http://hdl.handle.net/1721.1/59433

**Version:** Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

**Terms of use:** Attribution-Noncommercial-Share Alike 3.0 Unported

Massachusetts Institute of Technology

# Fast Distributed Multi-agent Plan Execution with Dynamic Task Assignment and Scheduling

**Julie A. Shah, Patrick R. Conrad, and Brian C. Williams**
MIT CSAIL MERS
32 Vassar St. Room 32-D224, Cambridge, MA 02139
julie_a_shah@csail.mit.edu, prconrad@mit.edu, williams@mit.edu

**Abstract**

An essential quality of a good partner is her responsiveness to other team members. Recent work in dynamic plan execution exhibits elements of this quality through the ability to adapt to the temporal uncertainties of others agents and the environment. However, a good teammate also has the ability to adapt on-the-fly through task assignment. We generalize the framework of dynamic execution to perform plan execution with dynamic task assignment as well as scheduling.

This paper introduces Chaski, a multi-agent executive for scheduling temporal plans with online task assignment. Chaski enables an agent to dynamically update its plan in response to disturbances in task assignment and the schedule of other agents. The agent then uses the updated plan to choose, schedule and execute actions that are guaranteed to be temporally consistent and logically valid within the multi-agent plan. Chaski is made efficient through an incremental algorithm that compactly encodes all scheduling policies for all possible task assignments. We apply Chaski to perform multi-manipulator coordination using two Barrett Arms within the authors' hardware testbed. We empirically demonstrate up to one order of magnitude improvements in execution latency and solution compactness compared to prior art.

## Introduction

An essential quality of a good partner is her ability to robustly anticipate and adapt to other team members and the environment. Recent work in dynamic execution exhibits elements of this quality through an executive that schedules activities online, dynamically in response to disturbances, while guaranteeing the constraints of the plan will be satisfied. Many recent multi-agent systems exploit this type of adaptive execution, allowing agents to absorb some temporal disturbances online (Alami et al. 1998, Brenner 2003, Lemai et al. 2004, Smith et al. 2006). However, disturbances triggering task re-assignment still require re-planning or plan repair. We introduce a multi-agent executive named Chaski, which generalizes the state-of -the-art in dynamic plan execution by supporting just-in-time task assignment as well as scheduling.

Many recent multi-agent systems perform dynamic plan execution by exploiting a flexible-time representation of the plan to absorb temporal disturbances online (ex. Lemai et al 2004, Smith et al. 2006). These systems employ a planning process that performs *task assignment* to allocate activities among the agents, and *synchronization* to introduce ordering constraints among activities so that concurrent execution remains logically valid (Stuart 1985, Kabanza 1995, Brenner 2003). The process of task assignment and synchronization generates temporally flexible plans described as Simple Temporal Networks (STNs) (Dechter et al. 1991). Agents dynamically execute STNs by scheduling plan activities online, just before the activity is executed (Muscettola et al. 1998, Tsamardinos et al. 1998). This strategy allows the agent to adapt to some disturbances that occur prior to the activity without introducing unnecessary conservatism. However, disturbances triggering task re-assignment or re-synchronization still require a deliberative capability to generate a new plan or perform plan repair.

The key contribution of this paper is an executive named Chaski that enables execution of temporally flexible plans with online task assignment and synchronization. Chaski enables an agent to dynamically update its plan in response to disturbances in the task assignment and schedule of other agents. Using the updated plan, the agent then chooses, schedules, and executes actions that are guaranteed to be temporally consistent and logically valid within the multi-agent plan. This capability provides agents maximal flexibility to choose task assignments, and schedule and execute activities online without the need for re-planning or plan repair. Chaski is especially useful for agents coordinating in highly uncertain environments, where near-continual plan repair results in execution delays – we see this, for example, with agents that interact with or adapt to humans.

The key innovation of Chaski is a fast execution algorithm that operates on a compact encoding of the scheduling policies for all possible task assignments. The compact encoding is computed by applying a set of incremental update rules to exploit the causal structure of the plan, as with previous algorithms for incremental compilation of Simple and Disjunctive Temporal Constraint Networks (Shah et al. 2007, 2008). We generalize this work to multi-agent plan execution by identifying and compactly recording the logical consequences that a particular task allocation and synchronization imply for future scheduling policies. We empirically demonstrate that this compact encoding reduces space to encode the solution set and execution latency by up to one order of magnitude compared to prior art.

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

This paper presents the incremental algorithm for compiling this compact encoding and the algorithm for distributed execution of plans based on the compact encoding. We empirically demonstrate Chaski through multi-manipulator coordination of two Barrett Whole Arm Manipulators. We show that performing execution on the compact representation yields low execution latency and scales well with the size of the multi-agent plan.

## Practical Scenario: Multi-robot Coordination

We have successfully applied Chaski to perform multi-manipulator coordination using two Barrett Arms. In this section, we present the multi-manipulator coordination scenario as a motivating example for the rest of the paper. Fig. 1 shows the two manipulator robots and their workspace. The robots must coordinate to remove one ball from each of the four locations in their communal workspace. Each robot also has one striped ball located in its own private workspace and must give the striped ball to the other robot using a hand-to-hand exchange. The scenario includes temporal constraints specifying the task must be completed within sixty seconds.

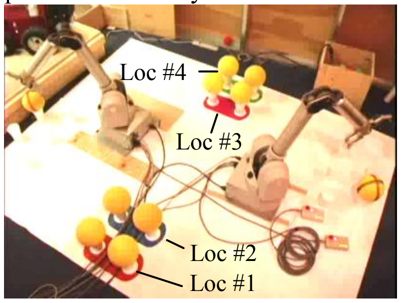

**Figure 1:** Multi-manipulator coordination scenario

This scenario is interesting because it contains both loosely and tightly coupled interaction, and a temporal constraint on the completion of the task. Also, some activities are not a-priori allocated to a particular robot. For example, "Remove one ball from Loc #1" can be performed by either robot. Finally, the robots have heterogeneous temporal capabilities. For example, the left robot has a shorter reach distance to Loc. #1 than the right robot. As a result, removing a ball from Loc. #1 takes the left robot 8-10 seconds and takes the right robot 11-13 seconds. We discuss how to perform fast, distributed execution of multi-agent plans such as this one.

## Background

Multi-agent systems typically employ a planning process that performs *task assignment* to allocate activities among the agents, and *synchronization* to introduce ordering constraints among activities so that concurrent execution remains logically valid. For example consider the following *task allocation* in the practical scenario described previously: the left robot performs both the activities: (1) Remove one ball from Loc. #1, and (2) Remove one ball from Loc. #2. Any *synchronization* of this task allocation would introduce ordering constraints to exclude concurrent execution of these two activities. The process of task assignment and synchronization generates temporally flexible plans described as Simple Temporal Networks (STNs). Agents exploit this flexible-time representation of the plan to adapt to some temporal disturbances online.

### Simple Temporal Networks

A Simple Temporal Problem (STN) is composed of a set of variables $X_1,\ldots X_n$, representing executable events. Events have real-valued domains and are related through binary temporal constraints. Binary constraints are of the form:

$$(X_k - X_i) \in [a_{ik}, b_{ik}].$$

A *solution* to an STN is a schedule that assigns a time to each event such that all constraints are satisfied. An STN is said to be *consistent* if at least one solution exists. Checking an STN for consistency can be cast as an all-pairs shortest path problem. The STN is consistent iff there are no negative cycles in the all-pairs distance graph. This check can be performed in $O(n^3)$ time (Dechter et al. 1991).

The all-pairs shortest path graph of a consistent STN is also a *dispatchable* form of the STN, enabling real-time scheduling. A network is dispatchable if for each variable $X_A$ it is possible to arbitrarily pick a time *t* within its timebounds and find feasible execution times in the future for other variables through one-step propagation of timing information. The constraints in the dispatchable form may then be pruned to remove all redundant information (Muscettola et al. 1998). The resulting network is a *minimal dispatchable* network, which is the most compact representation of the STN constraints that still contains all solutions present in the original network.

In the remainder of this paper we present Chaski, a multi-agent executive that extends recent work in fast execution of Disjunctive Temporal Constraint Networks to perform online task assignment and synchronization.

### Disjunctive Temporal Constraint Networks

A Disjunctive Temporal Constraint Network, otherwise known as a Temporal Constraint Satisfaction Problem (TCSP), extends an STN by allowing multiple intervals in constraints, given by the power set of all intervals:

$$(X_k - X_i) \in P(\{[a_{ik}, b_{ik}] \mid a_{ik} \le b_{ik}\}).$$

Determining consistency for a TCSP is NP-hard (Dechter et al. 1991). In previous work, a TCSP is viewed as a collection of component STNs, where each component STN is defined by selecting one STN constraint (i.e. one interval) from each TCSP constraint. Checking the consistency of the TCSP involves searching for a consistent component STN (Dechter et al. 1991). This approach is the basis of most modern approaches for solving temporal problems with disjunctive constraints

(Stergiou et al. 2000, Oddi and Cesta 2000, Tsamardinos and Pollack 2003).

Recent work in dispatchable execution of Disjunctive Temporal Constraint Networks (Shah et al. 2008) increases efficiency of execution by reasoning on a compact encoding of all consistent component STNs. The compact encoding is generated by an incremental algorithm in the spirit of other incremental algorithms for truth maintenance (Doyle 1979, Williams et al. 1998), informed search (Koenig et al. 2001), and temporal reasoning (Shu et al. 2005). The incremental compilation algorithm exploits the dependency structure of the network to identify and record the logical consequences that a particular simple interval constraint (or set of constraints) implies on the other constraints in the network. The compilation process first relaxes the TCSP to an STN and then compiles the STN to dispatchable form. Next, the algorithm applies Dynamic Back-Propagation (DBP) rules introduced in (Shah et al. 2007) to recursively propagate the logical consequences of a constraint change throughout the network. The incremental compilation algorithm results in a compiled plan that compactly represents the solution set in terms of the differences among viable component STNs.

## Problem Statement

Chaski takes as its input a multi-agent plan composed of P=(A,V,C,L), where A is a set of agents, V is a set of activities, A→V is an function describing the set of feasible activities and temporal capabilities of each agent, C is a set of temporal constraints over activities, and L is a set of logical constraints (for example, resource or agent occupancy constraints). The output of Chaski is a dynamic execution policy that guarantees temporally consistent and logically valid task assignments.

In this section, we reformulate a multi-agent plan as a Disjunctive Temporal Constraint Network, and provide insight into the challenges that arise in extending recent work in dynamic plan execution to perform online task assignment and synchronization.

Consider two robots that must coordinate to perform the following four activities in the practical scenario: Remove one ball each from Loc. #1 (RB1), Loc. #2 (RB2), Loc. #3 (RB3), and Loc. #4 (RB4). The robots have heterogeneous temporal capabilities. For example, removing a ball from Loc. #1 or #2 takes the left robot takes 8-10 seconds and takes the right robot 11-13 seconds. We also impose the temporal constraint that all four activities must be completed within twenty seconds. Fig. 2 presents this plan described as a Disjunctive Temporal Constraint Network.

Each activity is composed of a begin event and end event. For example, "a" and "b" represent the begin and end events, respectively, for activity RB1. The amount of time each agent takes to perform the activity is represented as a disjunctive binary constraint. For example, the disjunctive constraint L[8,10] V R[11,13] between events "a" and "b" specifies that the left robot "L" takes 8-10s to

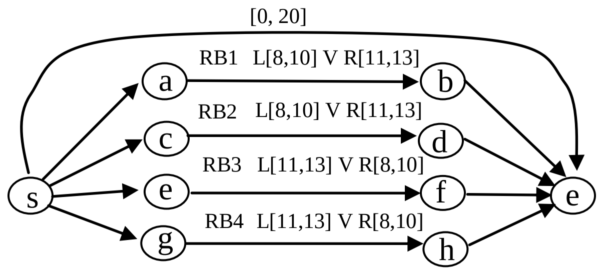

**Figure 2:** Multi-robot plan described as a Disjunctive Temporal Constraint Network

perform activity RB1, while the right robot "R" takes 11-13s. The execution order of the four activities is initially unspecified. The network includes ordering constraints of the form [0, inf] to specify that the activities must be executed after the epoch start event "s" and must be completed before the plan's end event "e". The temporal constraint [0,20] between events "s" and "e" constrains the time available to accomplish all four activities. Note that agents do not "own" the execution of particular activity events because the plan does not specify task assignments.

Dispatching this network using the method described in (Shah et al. 2008) will ensure temporally consistent task assignments at execution. However, this method does not perform synchronization of the task assignments and therefore, the execution may not be logically valid. For example, this plan contains implied *agent occupancy constraints*, meaning that each agent may only perform one activity at a time. We must introduce ordering constraints among activities to ensure that concurrent execution does not violate these occupancy constraints. We would like to compactly compile all feasible synchronizations for dynamic execution, just as we compile the feasible task assignments. However, the synchronization problem requires reasoning on a more general type of disjunctive constraint: disjuncts in intervals over three or more events. As a result, the synchronization problem cannot be framed as a Disjunctive Temporal Constraint Network and solved using the method presented in (Shah et al. 2008). In the next section, we address the challenge of compactly compiling the scheduling policies for all possible task assignments *and* their synchronizations.

## Incremental Compilation Algorithm for Multi-agent Temporal Plans

In this section we present an Incremental Compilation Algorithm (ICA-MAP) for compiling a multi-agent plan (MAP) to a compact dispatchable form. This compact representation is compiled by incrementally computing constraint modifications for task assignments and synchronizations, and then aggregating common information among synchronizations. The key idea behind ICA-MAP is to apply the Dynamic Back-Propagation (DBP) rules, described in (Shah et al. 2007), to systematically investigate and record the logical consequences that a particular task allocation and synchronization imply for future scheduling policies. As we empirically show in the next sections, this compact representation drastically reduces the number of constraints necessary to encode the feasible scheduling policies and supports fast dynamic execution.

## Pseudo-Code for the ICA-MAP

ICA-MAP takes as input a multi-agent plan (P=A,V,C,L), where A, V, A→V, and C are described as a Disjunctive Temporal Constraint Network G (ex. Fig. 2). The pseudo-code for ICA-MAP is presented in Figures 3-5.

The algorithm is composed of four main steps. The first two steps mirror the incremental compilation algorithm for TCSPs (Shah et al. 2008). **Step 1** relaxes the Disjunctive Temporal Constraint Network (G) to a Simple Temporal Network (S) (Line 1). This is accomplished by relaxing each disjunctive binary constraint to a simple interval. For each disjunctive constraint, a new simple temporal constraint is constructed using the lowerbound and upperbound of the union of intervals in the disjunctive constraint. **Step 2** then compiles the resulting STN to dispatchable form (Line 2). If the STN representing the relaxed plan is inconsistent, then there is no solution to the multi-agent plan and ICA-MAP returns false (Line 3). If the STN is consistent, then Line 4 initializes a data structure L(T,C) to record the scheduling policies for feasible task allocations (T) and their synchronizations (C).

In **Step 3**, the algorithm iterates through the set of full task assignments (Line 5). For each full task assignment $T_i$, the constraints associated with $T_i$ are placed on a queue $Q_t$ (Line 6). For example, consider the following full task assignment for the multi-agent plan in Fig. 2: Left Robot performs RB1 and RB2, and Right Robot performs RB3 and RB4. The interval constraints associated with each of these assignments are placed on the queue: $Q_t$ = {ab|L[8,10], cd|L[8,10], ef|R[8,10], gh|R[8,10]}.

Each constraint in $Q_t$ implies the tightening of a constraint in the relaxed, compiled network S. The function BACKPROPAGATE-TASK-ASSIGN propagates the effect of these constraint tightenings throughout S (Line 7). This process derives the necessary constraint modifications to ensure temporally consistent execution of the task assignment $T_i$. The modified constraints associated with task assignment $T_i$ are recorded in $L(T_i)$. During this process, typically only a subset of the constraints in the relaxed network S must be modified and recorded, contributing to the compactness of the representation. If back-propagation results in an inconsistency, then the task assignment $T_i$ is temporally inconsistent and the algorithm continues with the next full task assignment (Line 8).

Given a consistent task assignment $T_i$, **Step 4** collects the set of feasible synchronizations for $T_i$ (Line 9), and then iterates through each synchronization *y* (Line 10). Each synchronization *y* imposes a set of ordering constraints on the plan activities. For example, consider the task assignment: Left Robot performs RB1 and RB2, and Right Robot performs RB3 and RB4. One possible synchronization of this task assignment is: {bc|[0,inf], fg|[0,inf]}. This set of ordering constraints is added to the queue $Q_y$ (Line 11). (Note that our implementation of Chaski performs synchronization based on agent occupancy constraints. However, ICA-MAP generalizes to other synchronizations as well.)

```
function ICA-MAP (P={G,L})
1.  S ← Relax-Network-to-STN(G)
2.  S ← Compile-STN-to-Dispatchable-Form(S)
3.  if S is inconsistent return FALSE
4.  L(T,C) ← Initialize-Task-Allocation-Synchronization-List
5.  for each full task assignment ($T_i$)
6.      $Q_t$ ← add- $T_i$ -constraints-to-queue
7.      L($T_i$ ) ← BACKPROPAGATE-TASK-ASSIGN($Q_t$,S, L($T_i$ ))
8.      if BACKPROPAGATE-TASK-ASSIGN returns false,
        clear L($T_i$ ) and goto Line 5
9.      $C_y$ ← Synchronize-Task-Assignment($T_i$,L)
10.     for each synchronization y in $C_y$
11.         $Q_y$ ← add- $C_y$ - ordering-constraints-to-queue
12.         L($T_i$ , $C_y$) ← BACKPROPAGATE-SYNCH($Q_y$,S, L($T_i$ , $C_y$))
13.         if BACKPROPAGATE-SYNCH returns FALSE, clear
            L($T_i$ , $C_y$) and goto Line 10
14.     end for
15. end for
16. if L(T,C) is empty return FALSE
17. else return S and L(T,C)
```

**Figure 3:** Pseudo-code for ICA-MAP

The function BACKPROPAGATE-SYNCH then propagates the effect of these ordering constraints throughout the network (Line 12). If back-propagation of a synchronization *y* results in an inconsistency, then that synchronization *y* and its derived constraints are removed from L(T,C), and the algorithm continues with the next synchronization (Line 13). If L(T,C) remains empty after iterating through all full task allocations and synchronizations, then there is no solution to the multi-agent plan and ICA-MAP returns false. Otherwise, *ICA-MAP returns S and L(T,C), which compactly encode the scheduling policies for feasible task assignments and synchronizations.*

The key to compactly encoding the scheduling policies for feasible task allocations and synchronizations lies in the details of the two functions BACKPROPAGATE-TASK-ASSIGN and BACKPROPAGATE-SYNCH. Next, we walk through each of these functions.

The function BACKPROPAGATE-TASK-ASSIGN takes as its input the queue of task assignment constraints $Q_t$, the relaxed network S, and the data structure $L(T_i)$ that records the constraint modifications for task assignment $T_i$. Lines 1 and 2 add each constraint $e_i$ in $Q_t$ to $L(T_i)$. Line 3 applies the DBP rules to propagate the effect of each constraint $e_i$. For example, consider applying BACKPROPAGATE-TASK-ASSIGN to the queue of task assignments in our example: $Q_t$ = {ab|L[8,10], cd|L[8,10], ef|R[8,10], gh|R[8,10]}. First, we create the network S' associated with task assignment $T_i$ by intersecting the constraints in $L(T_i)$ with the constraints in S. We then apply the DBP rules to propagate the effect of ab|L[8,10] and the other constraints on $Q_t$ throughout the network S'.

If back-propagation deduces a new constraint $z_i$, and $z_i$ is a positive loop then $z_i$ does not have to be recursively propagated and the algorithm continues at Line 3. If $z_i$ is a negative loop then propagation has exposed an inconsistency and the function returns false. If $z_i$ is neither a positive nor negative loop, then Line 7 checks to determine whether $z_i$ is tighter than the corresponding constraint in S'. If so, $z_i$ is recorded in $L(T_i)$ and added to the queue $Q_n$ for further propagation (Lines 8 and 9). The constraints of $Q_n$ are recursively propagated through the network in Line 13. If recursive propagation of the synchronization constraints does not result in an inconsistency, then the function returns true (Line 14). *The output of BACKPROPAGATE-TASK-ASSIGN is the data structure $L(T_i)$, which records the constraint modifications to S that ensure temporally consistent execution of the task assignment $T_i$.*

Next, we present the function BACKPROPAGATE-SYNCH, which encodes the set of feasible synchronizations for each full task assignment.

The function BACKPROPAGATE-SYNCH is called for each synchronization *y* of a task assignment $T_i$. The function takes as its input the queue of synchronization constraints $Q_y$, the relaxed network S, and the data structure $L(T_i, C)$ that records the constraint modifications for task assignment $T_i$ and $T_i$'s set of synchronizations C.

Lines 1 and 2 add each constraint $e_i$ in $Q_y$ to $L(T_i, C_y)$, which records the constraint modifications for $T_i$'s synchronization *y*. Line 3 applies the DBP rules to propagate the effect of each constraint $e_i$. For example, consider applying BACKPROPAGATE-SYNCH to the queue of ordering constraints: $Q_y = \{bc|[0,inf], fg|[0,inf]\}$. First we create the network S" for task assignment $T_i$ and synchronization *y* by intersecting the constraints in $L(T_i)$ and $L(T_i, C_y)$ with the constraints in S. We then apply the DBP rules to propagate the effect of bc|L[0,inf] and the other constraints in $Q_y$ throughout the network S".

If back-propagation deduces a new constraint $z_i$, which is tighter than the corresponding constraint in S", then Lines 8-16 perform computations to refactor $L(T_i, C)$ such that constraints common to all feasible synchronizations of $T_i$ are recorded in $L(T_i)$. In Line 17, $z_i$ is added to the queue $Q_n$ for further propagation. The constraints of $Q_n$ are recursively propagated through the network in Line 21. If recursive propagation of the synchronization constraints does not result in an inconsistency, then the function returns true (Line 22). *BACKPROPAGATE-SYNCH returns $L(T_i)$ and $L(T_i,C)$, which record the constraint modifications to S that ensure synchronized execution of the task assignment $T_i$.* The refactoring process in Lines 8-16 ensures that constraints common to all of $T_i$'s synchronizations are recorded once, contributing to the compactness of the encoding.

We provide a proof sketch that ICA-MAP is complete in that it compiles a multi-agent plan to a dispatchable form

```
function BACKPROPAGATE-TASK-ASSIGN ($Q_t$, S, $L(T_i)$)
1.  for each constraint $e_i$ in $Q_t$
2.      add $e_i$ to $L(T_i)$
3.      for each DBP incremental update rule propagating $e_i$
4.          deduce-new-constraint-$z_i$ ($e_i$, S, $L(T_i)$)
5.          if is-pos-loop($z_i$) then goto Line 2
6.          if is-neg-loop($z_i$) then return FALSE
7.          if $z_i$-is-tightening($z_i$, S, $L(T_i)$)
8.              $L(T_i) \leftarrow$ add $z_i$ to $L(T_i)$
9.              $Q_n \leftarrow$ add $z_i$ to $Q_n$
10.         end if
11.     end for
12. end for
13. BACKPROPAGATE-TASK-ASSIGN ($Q_n$, S, $L(T_i)$)
14. return $L(T_i)$
```

**Figure 4:** Pseudo-code for BACKPROPAGATE-TASK-ASSIGN

```
function BACKPROPAGATE-SYNCH (y, $Q_y$, S, $L(T_i, C)$)
1.  for each constraint $e_i$ in $Q_y$
2.      add $e_i$ to $L(T_i, C_y)$
3.      for each DBP incremental update rule propagating $e_i$
4.          deduce-new-constraint-$z_i$ ($e_i$, S, $L(T_i, C_y)$)
5.          if is-pos-loop($z_i$) then goto Line 2
6.          if is-neg-loop($z_i$) then return FALSE
7.          if $z_i$-is-tightening($z_i$, S, $L(T_i, C_y)$)
8.              if $L(T_i)$ contains a constraint $f$ with $e_i$'s start and end events
9.                  $L(T_i, C) \leftarrow$ add $f$
10.                 $L(T_i, C_y) \leftarrow$ replace $f$ with $e_i$
11.                 $L(T_{i,}) \leftarrow$ remove $f$
12.             end if
13.             if $L(T_i, C)$ all contain $e_i$
14.                 $L(T_i) \leftarrow$ add $e_i$
15.                 $L(T_i, C) \leftarrow$ remove $e_i$
16.             end if
17.             $Q_n \leftarrow$ add $z_i$ to $Q_n$
18.         end if
19.     end for
20. end for
21. BACKPROPAGATE-SYNCH (y, $Q_n$, S, $L(T_i, C)$)
22. return $L(T_i)$ and $L(T_i, C)$
```

**Figure 5:** Pseudo-code for BACKPROPAGATE-SYNCH

that preserves the set of execution possibilities attained using the [Tsamardinos, 2000] method. We know that to compile a dispatchable DTP, it is sufficient to compile each of the component STPs to a dispatchable form. First (i) we show that ICA-MAP enumerates all the component STPs for compilation. Second, (ii) we sketch that compiling each component STP using the Back-Propagation Rules is complete in that the compiled form contains the same set of dispatchable executions as the APSP-dispatchable form.

(i) ICA-MAP explicitly enumerates every possible task assignment and synchronization (Line 5,10). Each full task assignment corresponds to choosing one disjunct of each disjunctive constraint in the TCSP representation of the multi-agent plan (ex. Fig 2), thus enumerating all possible component STPs in the TCSP. Synchronization involves choices over ordering constraints among activities in a given task allocation, thus enumerating all component STPs of the full DTP.

(ii) Each component STP is compiled to dispatchable form by applying constraint tightenings to a relaxed dispatchable form of the original DTP. The relaxed dispatchable form is guaranteed to contain all possible successful executions of every component STP. The Dynamic Back-Propagation (DBP) Rules prune the relaxed problem so that it contains exactly the set of possible executions generated by dispatching the All-Pairs-Shortest-Path form of the component STP. Note that the DBP rules are not performing an incremental APSP computation. Instead, they perform a subset of the updates of an APSP computation, and rely on the propagation of timing information at execution to enforce full all-pairs-shortest-paths [see Shah 2007, 2008 for more details].

## Empirical Validation of ICA-MAP

In this section we empirically investigate the compactness of solutions compiled with ICA-MAP. In a later section, we empirically demonstrate that this compact encoding supports fast dynamic execution.

We apply ICA-MAP to a portfolio of parameterized, structured multi-agent plans in which parameters are generated randomly. We compute the number of constraints necessary to represent our compact encoding of the solution set, and compare this result to the number of constraints necessary to represent the solution set using naïve the approach proposed in prior art (Tsamardinos 2001). The naïve approach maintains a separate, minimally dispatchable STN for each feasible synchronization of a full task assignment.

Both ICA-MAP and the algorithm for computing component STNs via the naive approach are implemented in JAVA. As a basis for comparison, we apply the two algorithms to randomly generated multi-agent plans involving coordination of two agents. Plans are generated with n = 8, 12, and 16 activities. Each activity is composed of two events: a start event *S* and end event *E*. A binary disjunctive constraint of two intervals is randomly generated between each *S* and *E*, where each interval maps to one of the two agents. Intervals are randomly generated with upperbound time constraints between [1, max_duration =10], and lowerbound time constraints between [0, upperbound] so that the duration is nonzero and locally consistent. The method of generating upperbounds and lowerbounds for a disjunctive constraint ensures non-overlaping intervals. To derive constraints among activities, we randomly place each activity in a 2D plan space similar to a simple scheduling timeline, where overlapping activities represent concurrent activities. Simple interval constraints are generated with locally consistent values in order to constrain neighboring activities. This process ensures that the structure of randomly generated plans results in plan executions that generally flow from left to right in the plan space. The number of constraints in the plan increases with plan size according to O(3n).

Fig. 6 shows the number of constraints necessary to represent our compact encoding of the solution set, compared to the number of constraints necessary to represent all consistent component STNs. Thirty random

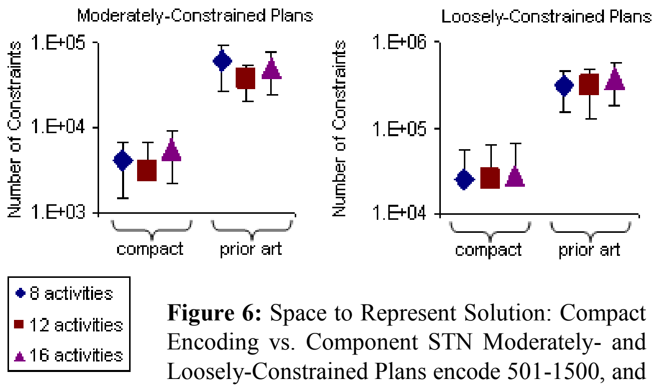

**Figure 6:** Space to Represent Solution: Compact Encoding vs. Component STN Moderately- and Loosely-Constrained Plans encode 501-1500, and 1501-5000 feasible component STNs, respectively.

multi-agent plans are each generated for plans with 8, 12, and 16 activities. We characterize each generated plan as tightly-, moderately-, or loosely-constrained based on the plan's number of feasible component STNs. The figure presents the mean and standard deviation in the number of constraints reported for each compilation method. Fig. 6 shows that the resulting compact representation reduces the space necessary to encode the multi-agent plan by one order of magnitude on average, compared to prior art.

## Algorithm for Fast Distributed Execution of Multi-agent Plans

We present the function FAST-MAP-DISPATCH, which performs fast distributed execution of multi-agent plans. FAST-MAP-DISPATCH is made efficient by performing online computations using the compact encoding generated by ICA-MAP. ICA-MAP drastically reduces the number of constraints necessary to encode the feasible scheduling policies, thereby reducing the amount of online computation required to propagate timing information FAST-MAP-DISPATCH also performs on-demand propagation of temporal information, contributing to its efficiency. Typically during dispatch, the temporal information of an executed event is propagated to all immediate neighbors. Instead, we only propagate temporal information to *enabled* events, or events that may possibly be executed at the next timestep, thereby reducing the amount of propagation for infeasible task assignments and synchronizations. We empirically show in the next sections that our dispatch method yields low execution latency and scales well with the size of the multi-agent plan.

## Pseudo-Code for the FAST-MAP-DISPATCH

The pseudo-code for FAST-MAP-DISPATCH is presented in Figures 7-8. The function takes as input the relaxed, compiled network S, and the data structure L(T,C) that records the necessary constraint modifications to ensure temporally consistent execution of the feasible task assignments (T) and their and synchronizations (C). Plan execution is "distributed" in that each agent makes its own execution decisions using its own copy of S and L(T,C). Agents coordinate through communicative acts that are used to (1) broadcast claims to execute events, (2) negotiate to resolve claim conflicts, and (3) broadcast the successful execution of events.

In performing distributed dispatch of the plan, each agent must keep a list E of the events currently *enabled* for other agents, and keep a list $E_{SELF}$ of the events currently *enabled* for itself. An event N is *enabled* for an agent A if there exists some feasible synchronization where: the event N is assigned to agent A and all events that are constrained to occur before event N have already been executed. Lines 1 and 2 initialize E and $E_{SELF}$. Initially, the plan's epoch start event is placed in either E, $E_{SELF}$ or both, depending on the event's enablement conditions. Line 3 initializes W and $W_{SELF}$, which maintain the feasible windows of execution for the events in E and $E_{SELF}$, respectively. In Line 4 the current time is initialized to zero.

The algorithm iterates through each enabled event N in E or $E_{SELF}$ until all plan events are executed (Lines 5,6). In Lines 7 and 8 the dispatcher compiles $W_{E,N}$ and $W_{SELF,N}$, the feasible execution windows of N for other agents and itself, respectively. If the current time is within another agent's feasible window of execution (Line 9) then the self-agent checks whether another agent has broadcast the successful execution of event N. If so, the self-agent records N's execution time as the current time, and labels N with the name of the agent that executed N (Line 10). If N has not yet been executed by another agent, the self-agent checks whether the current time is within its own feasible window of execution (Line 11). If so, then the self-agent broadcasts a claim to execute N (Line 12). In the case where another agent has also broadcast a claim to execute N, then the agents communicate to resolve the conflict. If after resolution, the self-agent owns the event N, then the self-agent records N's execution time as the current time, labels N with its own name, executes N, and broadcasts the successful execution of N (Line 13).

Lines 15-18 describe the process of updating the plan in response to an executed event N. First, the enabled lists E and $E_{SELF}$ are cleared (Line 15), since the execution of N may make the task assignments and synchronizations that support the currently enabled events infeasible. Next, the function PRUNE-AND-UPDATE-ENABLED is called to remove infeasible task assignments and synchronizations from L(T,C), update the enabled lists E and $E_{SELF}$, and propagate timing information for the enabled events.

PRUNE-AND-UPDATE-ENABLED takes as input N, the recently executed event, S, the relaxed network, and L(T,C), which records the constraint modifications for the feasible task assignments and their and synchronizations. The function iterates through each full task assignment $T_i$ (Line 1), checking whether the execution of N implies task assignment $T_i$ is infeasible. $T_i$ may be infeasible due to inconsistent agent assignment (Line 2), inconsistent execution time (Line 3), or unsatisfied enablement conditions (Line 4). If $T_i$ is found to be infeasible, then $T_i$ and all its synchronizations are marked infeasible. If $T_i$ is found to be feasible, then the function iterates through each

```
function FAST-MAP-DISPATCH (S, L(T , C))
1.  E ← Initialize-other-agents'-enabled-list
2.  $E_{SELF}$ ← Initialize-self-agent's-enabled-list
3.  {$W_E$ ,$W_{SELF}$}← Initialize-execution-window-lists
4.  current_time = 0
5.  while one or more events have not been executed
6.      for each event N in E or $E_{own}$
7.          $W_{E,N}$ ← Compile-Other-Agents'-Windows(N, $W_E$)
8.          $W_{SELF,N}$ ← Compile-Self-Agent's-Windows(N, $W_{SELF}$)
9.          if current_time is in $W_{E,N}$ and E contains N
10.             if other agent has executed N then set N's execution time
                to current_time and label N with executing agent's name
11.         else if current_time is in $W_{SELF,N}$ and $E_{SELF}$ contains N
12.             claim N for self-agent and resolve any claim conflict
13.             if self-agent owns N then set N's execution time to
                  current_time, label N with self-agent's name, execute N,
                and broadcast the successful execution of N
14.         end if
15.         if N is executed
16.             E, $E_{SELF}$ ← clear-lists
17.             E, $E_{SELF}$,$W_E$ ,$W_{SELF}$ ←
                PRUNE-AND-UPDATE-ENABLED(N,S,L(T,C))
18.         end if
19.     end for
20. end while
```

**Figure 7:** Pseudo-code for FAST-MAP-DISPATCH

```
function PRUNE-AND-UPDATE-ENABLED (N, S, L(T , C))
1.  for each feasible full task assignment $T_i$
2.      if N's agent assignment is inconsistent with $T_i$ then mark $T_i$ and
        all its synchronizations as infeasible and goto Line 1
3.      if N's execution time is inconsistent with $T_i$ then mark $T_i$ and all
        its synchronizations as infeasible and goto Line 1
4.      if N's enablement conditions are not satisfied then mark $T_i$ and all
        its synchronizations as infeasible and goto Line 1
5.      for each feasible synchronization $y_n$ of $T_i$
6.          if N's execution time is inconsistent with $y_n$ then mark $y_n$ as
            infeasible and goto Line 5
7.          if N's enablement conditions are not satisfied then mark $y_n$ as
            infeasible and goto Line 5
8.          E, $E_{SELF}$ ← gather-enabled-events-using-( $y_n$ , $T_i$ ,S)
9.          $W_E$ ,$W_{SELF}$ ←
            update-windows-of-enabled-events-using -( $y_n$ , $T_i$ ,S)
10.     end for
11. end for
```

**Figure 8:** Pseudo-code for PRUNE-AND-UPDATE-ENABLED

feasible synchronization $y_n$ of $T_i$ (Line 5), checking whether the execution of N implies $y_n$ is infeasible. The synchronization $y_n$ may be infeasible due to inconsistent execution time or unsatisfied enablement conditions (Lines 6,7). If a given synchronization $y_n$ of task assignment $T_i$ is found to be feasible, then Line 8 gathers the enabled events. Line 9 then propagates the timing information of N and updates the feasible execution windows for the enabled events.

FAST-MAP-DISPATCH has the following properties: (1) it is correct in that any complete task assignment and execution sequence generated by the dispatcher also satisfied the constraints of the multi-agent plan, (2) it is deadlock-free in that any partial execution generated by the dispacther can be extended to a complete execution that satisfies the constraints of the multi-agent plan, and (3) it is maximally flexible in that the dispatcher generates the same set of complete execution sequences that are generated by dispatching the consistent component STPs of the multi-agent plan. Proofs are omitted for space.

In the next section we empirically show that FAST-MAP-DISPATCH reduces execution latency by up to one order of magnitude compared to prior art.

## Empirical Validation of FAST-MAP-DISPATCH

In a previous section we have shown that our incremental compilation method drastically reduces the number of constraints necessary to encode the set of feasible scheduling policies. In this section we empirically show that this compact representation supports fast dynamic execution of multi-agent plans.

We empirically validate FAST-MAP-DISPATCH by dynamically executing randomly generated, structured multi-agent plans. We compare the execution latency associated with dispatching our compact encoding to the execution latency of dispatching the component STN representation. As a conservative measure, we record the execution latency to propagate the timing of the first executed event. This is a conservative measure for execution latency because all task allocations and synchronizations are still feasible, thus increasing the computation required to propagate timing information.

The results of the comparison are shown in Fig. 9**.** Thirty structured multi-agent plans are randomly generated for each n = 8, 12, and 16 activities. The figure presents the mean and standard deviation of execution latency for each dispatch method. The results indicate that dispatching the compact encoding significantly reduces execution latency, by one order of magnitude on average, compared to the dispatch of the component STN representation. Also, the results indicate that our method scales well with the size of the multi-agent plan. Doubling the size of loosely-constrained multi-agent plans from 8 to 16 activities increases the execution latency by no more than 0.03 seconds on average using the compact encoding.

By leveraging a compact encoding of multi-agent plans, FAST-MAP-DISPATCH enables agents to perform distributed dynamic execution while (1) reasoning on flexible scheduling policies for thousands of possible futures, and (2) achieving execution latency within the bounds of human reaction time (250 ms).

## Conclusion

In this paper, we introduced an executive named Chaski that enables execution of temporally flexible plans with online task assignment and synchronization. Chaski generalizes the state-of-the-art in dynamic plan execution by supporting just-in-time task assignment as well as scheduling. The key innovation of Chaski is a fast execution algorithm that operates on a compact encoding of the scheduling policies for all possible task assignments. We show that Chaski reduces execution latency and the number of constraints necessary to encode the randomly generated plans by one order of magnitude on average, compared to prior art.

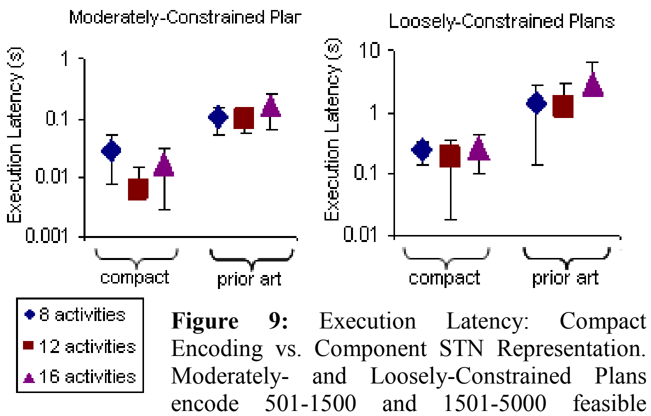

**Figure 9:** Execution Latency: Compact Encoding vs. Component STN Representation. Moderately- and Loosely-Constrained Plans encode 501-1500 and 1501-5000 feasible component STNs, respectively.

## References

**[Alami, R., Ingrand, F., and Qutub, S. 1998]** A Scheme for Coordinating Multi-robot Planning Activities and Plans Execution. In *Proc.ECAI*, Brighton, UK, 1998.

**[Brenner, M. 2003]** Multiagent planning with partially ordered temporal plans. In *Proc. AIPS DC.*

**[Dechter, R., et al. 1991]** Temporal constraint networks. *AI*, 49:61-95.

**[Doyle 1979]** A truth maintenance system. *AI,* 12:231-272.

**[Kabanza, F. 1995]** Synchronizing Multiagent Plans using Temporal Logic Specifications, *Proc. ICMAS.* Menlo Park, CA, 217-224.

**[Lemai, S., and Ingrand, F. 2004]** Interleaving temporeal planning and execution in robotics domains,. in *Proc. AAAI.*

**[Koenig, S., Likhachev, M. 2001]** Incremental A*. *Advances in Neural Information Processing Systems (14).*

**[Muscettola, N., et al. 1998]**. Reformulating temporal plans for efficient execution. *Proc.KRR-98.*

**[Oddi, A., and Cesta, A. 2000]** Incremental Forward Checking for the Disjunctive Temporal Problem. In *Proc. ECAI*, 108–112.

**[Shah, J., et al. 2007]** A Fast Incremental Algorithm for Maintaining Dispatchability of Partially Controllable Plans. *Proc. ICAPS-07.*

**[Shah, J., et al. 2008]** Fast Dynamic Scheduling of Disjunctive Temporal Constraint Networks through Incremental Compilation. *Proc. ICAPS-08.*

**[Smith, S.; Gallagher, A.; Zimmerman, T.; Barbulescu, L.; and Rubinstein, Z. 2006]**. Multi-Agent Management of Joint Schedules. In *AAAI Spring Symposium on Distributed Plan and Schedule Management*, 128.135. AAAI Press.

**[Stergiou, K., and Koubarakis, M. 2000]** Backtracking Algorithms for Disjunctions of Temporal Constraints. *AI* 120:81–117.

**[Stuart, C. 1985]** An implementation of a multi-agent plan synchronizer. *Proc. ICJAI*, Los Angeles, CA, pp. 1031–1033.

**[Tsamardinos, I., et al. 1998]** Fast transformation of temporal plans for efficient execution. *Proc. AAAI-98.*

**[Tsamardinos, I.; et al. 2001].** Flexible dispatch of disjunctive plans. In *Proc. ECP*, 417–422

**[Tsamardinos, I., and Pollack, M. E. 2003]** Efficient Solution Techniques for Disjunctive Temporal Reasoning Problems. *Artificial Intelligence* 151(1-2):43–90.

**[Williams, B.C., and Millar, B. 1998]** Decompositional, Model-based Learning and its Analogy to Model-based Diagnosis, *Proc. AAAI*, Milwaukee, Wisconsin,pp. 197-203.