Fast Dynamic Scheduling of Disjunctive Temporal Constraint Networks through Incremental Compilation

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

Abstract

Autonomous systems operating in real-world environments must plan, schedule, and execute missions while robustly adapting to uncertainty and disturbance. One way to mitigate the effect of uncertainty and disturbance is to dynamically schedule the plan online, through *dispatchable execution.* Dispatchable execution increases the efficiency of plan execution by introducing (1) a compiler that reduces a plan to a *dispatchable* form that enables real-time scheduling, and (2) a temporal plan *dispatcher* that schedules start times of activities (or controllable events) dynamically in response to disturbances. Previous work addresses efficient dispatchable execution of plans described as Simple Temporal Problems (STPs). While STPs have proven useful for many applications, Temporal Constraint Satisfaction Problems (TCSPs) provide a more rich language by introducing disjunctive constraints. However, previous approaches to dispatchable execution of disjunctive temporal plans are intractable for moderately-sized problems.

The key contribution of this paper is an efficient algorithm for compiling and dynamically scheduling TCSPs. We present an incremental algorithm that compiles a TCSP to a compact representation, encoding the solution set in terms of the differences among solutions. We empirically demonstrate that this novel encoding reduces the space to encode the solution set by up to three orders of magnitude compared to prior art, and supports fast dynamic scheduling.

Introduction

Real-world autonomous agents must be able to schedule, and execute mission plans while robustly anticipating and adapting to uncertainty and disturbance. One way to mitigate the effect of uncertainty and disturbance is to dynamically schedule plan activities online, just before the activity is executed. This allows the scheduler to adapt to disturbances that have occurred prior to the activity without introducing unnecessary conservatism; this type of dynamic scheduling is called *dispatchable execution.* Dispatchable execution increases the efficiency of plan execution by introducing a *compiler* and a *dispatcher.* The compiler reduces a plan to a form that enables real-time scheduling. A temporal plan dispatcher then schedules activities online, dynamically in response to disturbances, while guaranteeing that the constraints of the plan will be satisfied. Dispatchable execution is domain independent and has been successfully applied to scheduling within the avionics processor of commercial aircraft (Tsamardinos et al. 1998), space probes (Muscettola et al. 1998b), autonomous air vehicles (Stedl 2004**)**, and walking robots (Hofmann et al. 2006).

Previous work has addressed efficient real-time scheduling of plans whose temporal constraints are described as Simple Temporal Problems (STPs) or STPs with Uncertainty (STPUs) (Dechter et al. 1991, Muscettola et al. 1998, Tsamardinos et al. 1998). STPs and STPUs provide a language for temporal constraints that is made tractable through simple interval constraints. Although STPs and STPUs have proven useful for important applications, their applicability to many problems is limited by their lack of disjunctive constraints.

Temporal Constraint Satisfaction Problems (TCSPs) extend STPs by allowing binary constraints to be expressed as sets of disjunctive intervals. Previous work in dispatchable execution has been developed for disjunctive temporal constraint networks (Tsamardinos 2001). First, offline, the disjunctive temporal constraint network is searched for all consistent component STPs, where each component STP is defined by selecting one simple interval from each disjunctive constraint (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). Dispatching the disjunctive temporal constraint network then involves (1) maintaining data structures for each of the viable component STPs, (2) computing interrelationships among the component STPs online, and (3) propagating timing information online simultaneously to multiple STPs (Tsamardinos 2001). However, this approach becomes costly for moderately-sized problems. First, a significant amount of space is used to represent the solution set of disjunctive temporal plans. Second, repeated online computations, and propagation of timing information in multiple STPs contribute to execution latency.

We propose an efficient approach to compiling and dynamically scheduling disjunctive temporal constraint networks. We present an incremental algorithm that compiles a TCSP to a compact representation that encodes the solution set in terms of only the differences among viable component STPs. The algorithm is developed as a set of incremental update rules 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 update rules exploit the causal structure of the TCSP to propagate constraint information as previous algorithms for incrementally maintaining dispatchability of STPs and STPUs (Shah et al. 2007). We apply incremental update rules to identify and record the logical consequences that a particular simple interval constraint (or a set of simple intervals) implies on the other constraints in the TCSP. We empirically show that the resulting compact representation reduces the space necessary to encode the TCSP solution set by up to three orders of magnitude, compared to prior art.

The compact representation also enables fast dynamic scheduling by addressing two major sources of execution latency: (1) the need to compute interrelationships among component STPs online, and (2) the need to propagate timing information online simultaneously to multiple STPs. The compact representation of the TCSP solution set already encodes interrelationships among viable component STPs. This reduces the amount of work necessary to identify interrelationships among component STPs online. Performing dynamic scheduling on this compact representation also eliminates the need to propagate timing information simultaneously to multiple STPs. We empirically show that performing dispatchable execution on this compact representation yields low execution latency, and scales well with the size of the disjunctive temporal constraint network.

Background

Simple Temporal Problems

A Simple Temporal Problem (STP) is composed of a set of variables $X_1,\ldots X_n$, representing timepoints with real-valued domains and binary constraints. Binary constraints are of the form:

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

A *solution* to an STP is an assignment to each timepoint such that all constraints are satisfied. An STP is said to be *consistent* if at least one solution exists. Checking an STP for consistency can be cast as an all-pairs shortest path problem. The STP 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 STP is also a *dispatchable* form of the STP, 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 tightened to remove all redundant information (Muscettola et al. 1998). The resulting network is a *minimal dispatchable* network, which is the tightest representation of the STP constraints that still contains all solutions present in the original network.

During plan execution, a significant disturbance may require modifying constraints in the STP online. The STP must then be quickly compiled back to a dispatchable form. Shah et al. (2007) introduce an incremental algorithm for maintaining dispatchability of STPs in response to plan changes. For example, when a constraint is tightened, the following Dynamic Back-Propagation (DBP) rules are used to propagate the logical consequences of this constraint change throughout the network:

Given a dispatchable STP with associated distance graph G: *(i) Consider any tightening (or addition) of an edge AB, such that d(AB) = y, where y>0 and A≠ B. For all edges BC such that d(BC)= u <= 0, it follows that d(AC) = y + u. (ii) Consider any tightening (or addition) of an edge BA such that d(BA)= x, where x <= 0 and A≠ B; for all edges CB such that d(CB)= v, where v >= 0, it follows that d(CA) = x+v.*

Recursively applying rules (i) and (ii), when an edge is tightened in a dispatchable distance graph, will either expose an inconsistency or result in a dispatchable graph. The key feature of DBP is its increased efficiency because it only requires a subset of the edges to be checked to ensure that the modified constraint is consistent, rather than all edges when the all-pairs graph is computed.

Temporal Constraint Satisfaction Problems

A Temporal Constraint Satisfaction Problem (TCSP) extends an STP 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 STPs, where each component STP is defined by selecting one STP constraint (i.e. one interval) from each TCSP constraint. Checking the consistency of the TCSP involves searching for a consistent component STP (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).

Previous work in dispatchable execution of disjunctive temporal plans (Tsamardinos 2001) first requires searching for all consistent component STPs. Once all component STPs have been identified, the next step is to summarize the information in the component STPs and to identify interrelationships among the solutions. This step is necessary to determine when events can be executed, and to ensure during execution that all viable solutions are not eliminated simultaneously. Previous work performs this step repeatedly online as events are executed and solutions are eliminated. Whenever an event is executed, timing

information is then propagated online to all remaining viable component STPs.

There are two shortcomings to this approach. First, a significant amount of space is necessary to represent the solution set of even moderately-sized TCSPs. Second, repeated online computation and propagation of timing information to multiple STPs contribute to execution latency. Next, we propose an efficient approach to compiling and dynamically scheduling disjunctive temporal constraint networks that addresses these shortcomings.

Incremental Algorithm for Compiling a TCSP to a Compact Dispatchable Form

In this section we present an Incremental Compilation Algorithm (ICA-TCSP) for compiling a TCSP to a compact dispatchable form. The key idea behind ICA-TCSP is to apply the Dynamic Back-Propagation (DBP) rules, described in (Shah et al. 2007), in a novel way to systematically investigate and record the logical consequences that a particular simple interval constraint implies on other constraints. The key innovation of this work is a compact representation that encodes a TCSP solution set in terms of only the differences among the component solution STPs. As we empirically show in the next section, this compact representation reduces the space to encode the solution by up to three orders of magnitude.

Since DBP rules may only be applied to STP plans with simple temporal constraints, mapped to a dispatchable form, we first compile a *relaxed version* of the TCSP to dispatchable form. The relaxed problem and DBP rules are then used to compile the original TCSP to the compact dispatchable form.

The algorithm is composed of four main steps. We use the example TCSP in Fig. 1(a) to illustrate the steps. **Step 1** relaxes the TCSP to an STP. This is accomplished by relaxing each disjunctive binary constraint in the TCSP (Fig. 1(a)-(b)). 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 STP to dispatchable form (Fig. 1(b)). **Step 3** reverts each of the relaxed constraints back to disjunctive form, and places each interval (disjunct) of the reverted constraints on a queue (Fig. 1(c)).

In **Step 4**, *relationship lists* are created for each simple interval constraint on the queue. The DBP rules are then applied to infer the effect of selecting one of these simple intervals on the other constraints in the problem. The logical consequences of each simple interval constraint are then recorded in the interval's *relationship list*. During this process, if two or more simple interval constraints are found to be inconsistent, the set of intervals is recorded as a *conflict*. The final compiled form, including the reverted TCSP, its *relationship lists* and *conflicts*, is presented in Fig. 1(d).

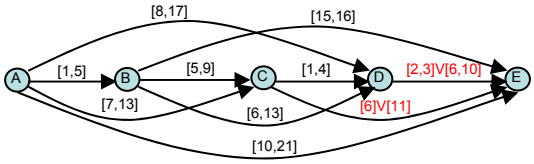

Figure 1(a): Example TCSP

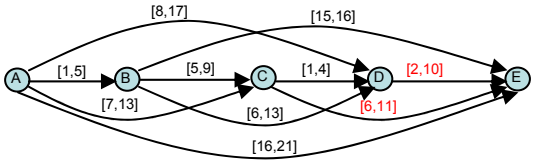

Figure 1(b): Example TCSP relaxed to a STP and compiled to dispatchable form.

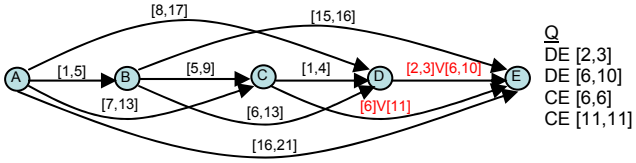

Figure 1(c): Reverted TCSP with Queue of reverted constraints.

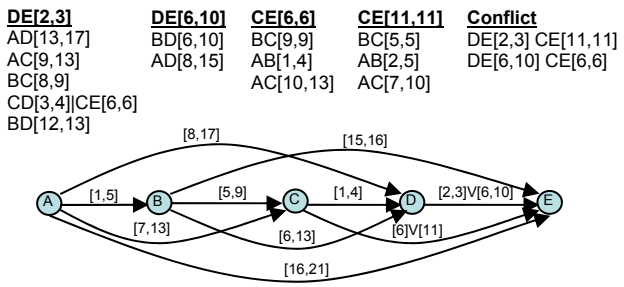

Figure 1(d): Compiled Compact Dispatchable Form of TCSP.

Next we explain, with examples, how the DBP rules are used to build the *relationship lists* and *conflicts*.

Applying Dynamic Back-Propagation Rules

Given a relaxed STP in dispatchable form, **Step 4** reintroduces disjunctive constraints and creates a *relationship list* for each simple interval constraint on the queue. Next, we use the DBP rules in a novel way to systematically investigate and record the logical consequences that a particular simple interval constraint implies on other constraints. We use to DBP rules to accomplish this as follows:

For each simple interval constraint on the queue, we first tighten the constraint in the reverted TCSP to match this simple interval. For example, in Fig. 1(c), consider the simple interval DE[2,3] shown at the top of the queue. We tighten constraint DE [2,3]V[6,10] in the reverted TCSP to [2,3], to match the simple interval DE[2,3] on the queue. This requires tightening constraint DE's highest upperbound from 10 to 3. We now propagate the effect of this tightened constraint throughout the rest of the reverted TCSP using the DBP rules. Recursively applying DBP rules (i) and (ii) modifies a subset of the constraints, as it derives the logical consequences of choosing DE[2,3] to be part of a solution to the TCSP. A record of these logical

consequences is maintained by recording each modified constraint in the DE[2,3] *relationship list.*

For example, consider propagating the tightened constraint DE[2,3] throughout the network. Constraint DE[2,3] is propagated through AE[16,21] using DBP rule (i), resulting in a new constraint AD[13,17]. Recursively applying the DBP rules means that the new constraint AD[13,17] is then propagated through CD[1,4] using DBP rule (ii), resulting in a new constraint AC[9,13]. Recursive propagation continues until no propagation remains, or an inconsistency is exposed.

Notice that a tightened constraint may need to be propagated through other disjunctive constraints. For example, DE[2,3] must also be propagated through CE[6]V[11]. We deal with this by propagating DE[2,3] through each disjunctive interval, as separate cases. First DE[2,3] is propagated through CE[6,6], and then DE[2,3] is separately propagated through CE[11,11]. In this case, any modified constraints must be recorded as conditioned on the appropriate disjunctive interval. For example, propagating DE[2,3] through CE[6,6] using DBP rule (i) results in a new constraint CD[3,4]. This new constraint is recorded in the *relationship list* as CD[3,4]|CE[6,6], meaning CD[3,4] is conditioned on choosing CE[6,6] to be part of the TCSP solution set.

Propagation may also expose inconsistencies. An inconsistency is detected when propagation results in a negative cycle. This means that two or more simple intervals in the queue were found to conflict, in that together they result in an inconsistent solution. This set of conflicting intervals is composed of all the disjuncts that the negative self-loop is conditioned on at the time the inconsistency is detected. This set of intervals is then recorded is a *conflict*. *Conflicts* are used to increase the time efficiency of ICA-TCSP by avoiding exploring combinations of disjunctive intervals that were previously identified as dead ends. *Conflicts* are also necessary to correctly dispatch the plan, as will be described later. When propagation exposes an inconsistency, the *relationship list* must also be pruned to remove the logical consequences that result in the inconsistency.

Like previous algorithms for finding solutions to TCSPs, ICA-TCSP is exponential in the number of disjuncts. However, the result of ICA-TCSP offers a more compact representation on average, that encodes a TCSP solution set in terms of only the differences among the solution component STPs. This is in contrast to explicitly searching for and representing each solution component STP. The algorithm applies the DBP rules in a novel way to systematically investigate and record the logical consequences that a particular simple interval constraint implies on other constraints. During this process, typically only a subset of the constraints in the reverted TCSP must be modified and recorded, contributing to the compactness of the representation.

```
function ICA-TCSP (G)
1.  S ← Relax-Network-to-STP(G)
2.  S ← Compile-STP-to-Dispatchable-Form(S)
3.  if S is inconsistent return FALSE
4.  for each disjunctive temporal constraint x_i in G
5.      S ← Revert-Disjunctive-Constraint(S, x_i)
6.      Q ← Add-Disjunctive-Intervals-To-Queue(Q, x_i)
7.  end for
8.  R ← Initialize-Relationship-Lists(Q)
9.  M ←Initialize-Conflict-List
10. C ← Initialize-Condition-List
11. for each constraint e_i on Q and its relationship list R_i
12.     C ← Clear-List
13.     C ← Add-To-Condition-List(e_i)
14.     APPLY-DBP-RULES(S, e_i, R_i, M, C)
15. end for
16. if R is empty return FALSE
17. else return TRUE
```

Figure 2: Pseudo-code ICA-TCSP.

```
function APPLY-DBP-RULES (S,e_i R_i, M, C)
1.  for each DBP incremental update rule propagating e_i
2.      deduce a new constraint z_i
3.      if IS-POS-LOOP(z_i) then GOTO Line 16
4.      if IS-NEG-LOOP(z_i) then
5.          M ← Add-To-Conflict-Lists(C)
6.          R ← Remove-Thread-From-Relationship-List(R_i, C)
7.          GOTO Line 16
8.      end if
9.      if z_i is not equal to the corresponding constraint in S
10.         if e_i was propagated through a disjunctive interval d
11.             C ← Add-To-Condition-List(d)
12.         end if
13.         R ← Add-To-Relationship-List(z_i, C, R_i)
14.         if C not in M then APPLY-DBP-RULES(S,z_i,R_i,M,C)
15.     end if
16.     if e_i was propagated through a disjunctive interval d
17.         C ← Remove-Last-Constraint
18.     end if
19. end for
```

Figure 3: Pseudo-code APPLY-DBP-RULES.

Pseudo-Code for the ICA-TCSP

Figs. 2-3 show the pseudo-code for ICA-TCSP. Lines 1-2 relax the TCSP (G) to an STP (S) and compile S to dispatchable form. If S is inconsistent, then the TCSP is inconsistent and the algorithm returns false (Line 3). Else, Lines 4-7 revert each of the relaxed constraints in S back to the original disjunctive form found in G, and place each simple interval of the disjunctive constraints on a queue (Q).

Line 8 initializes the *relationship lists*, which are used to record the logical consequences of each simple interval constraint on Q. One *relationship list* is created for each simple interval constraint on Q. Line 9 initializes a *conflict list*, which is used to record sets of two or more simple intervals found to conflict. Line 10 initializes a *condition list*, which is used to maintain a history of the current propagation thread (i.e. which disjunctive intervals were selected during propagation).

Lines 11-15 apply the DBP rules to systematically investigate and record the logical consequences that a particular simple interval constraint on Q implies on other

constraints. For each interval constraint e_i on Q, the *condition list* is cleared, and then e_i is added to the *condition list*. The DBP incremental update rules are then applied to e_i according to APPLY-DBP-RULES in Fig. 2.

The function APPLY-DBP-RULES in Fig. 2 takes as input S, the reverted TCSP with queue (i.e., Fig. 1(c)), an interval constraint e_i, the interval constraint's *relationship list* R_i, the *conflict list* M, and the current propagation thread C. In Line 1, the DBP incremental update rules are used to propagate e_i. For each rule application, a new constraint z_i is deduced (Line 2). If z_i is found to be a positive loop (i.e., starts and ends at the same timepoint) then no additional propagation is required (Line 3). The algorithm then skips to Line 16. If e_i was propagated through a disjunctive interval, then the last constraint is removed from C. APPLY-DBP-RULES then proceeds to the next deduced constraint.

If z_i is found to be a negative loop (Line 4), then propagation has exposed an inconsistency, and a *conflict* composed of all the interval constraints that the negative loop is conditioned on is formed. In Line 5, this set of intervals C is recorded as a *conflict* in M. In Line 6, the *relationship list* R_i is also pruned to remove the logical consequences that result in the inconsistency. This requires pruning R_i of any modified constraint that is conditioned on the set of disjunctive intervals C. The algorithm then skips to Line 16. If e_i was propagated through a disjunctive interval, then the last constraint is removed from C. APPLY-DBP-RULES then proceeds to the next deduced constraint.

If z_i is found to be neither a positive nor negative loop (Line 9), then it is compared to the corresponding constraint in S. If z_i is not equal to the corresponding constraint in S, then the algorithm checks whether z_i resulted from propagation through a disjunctive interval d (Line 10). If so, then in Line 11, d is added to *condition list* C. Next, in Line 13, z_i and C (the disjunctive intervals z_i is conditioned on) are added to the *relationship list* R_i. If the current propagation thread C was not previously identified as a conflict, then z_i is propagated recursively using APPLY-DBP-RULES (Line 14). Propagation continues until each disjunctive interval constraint on Q has been systematically investigated, and the implied logical consequences have been recorded.

If, at the end of compilation, the set of *relationship lists* is empty, then the TCSP has no solution and ICA-TCSP returns false. If the set of *relationship lists* is not empty, then the TCSP has been compiled to a compact dispatchable form described by S, the reverted TCSP, R, the *relationship lists,* and M, the *conflicts.* Next, we empirically demonstrate that this novel encoding reduces the memory required to encode the solution set by up to three orders of magnitude compared to prior art.

Empirical Validation of Incremental Compilation Algorithm

We empirically validated ICA-TCSP by randomly generating TCSPs, and comparing the space necessary to represent our compact encoding of the TCSP solution set to the space necessary to represent all consistent component STPs, as used by (Tsamardinos 2001).

We computed all consistent component STPs using a depth-first, chronological backtracking search algorithm. The algorithm created STPs by selecting and assigning one STP constraint (i.e. one interval) from each disjunctive constraint in a depth-first manner. Each resulting STP was checked for consistency using the Floyd-Warshall algorithm. While there are more time efficient methods for searching for consistent component STPs (Tsamardinos et al. 2003), most modern approaches for solving problems with disjunctive constraints involve representing the solution as a set of consistent component STPs (Stergiou et al. 2000, Oddi and Cesta 2000, Tsamardinos and Pollack 2003). For a fair basis of comparison, we removed all redundant edges from each consistent component STP (Muscettola et al. 1998).

Both ICA-TCSP and the depth-first, chronological backtracking algorithm were implemented in JAVA. As a basis of comparison, we applied the two algorithms to randomly generated TCSPs. We generated random TCSPs using our implementation of the random Disjunctive Temporal Problem (DTP) generator (Stergiou et al. 2000), as is customary in the DTP literature (Oddi and Cesta 2000, Tsamardinos and Pollack 2003)**.** TCSPs were instantiated according to parameters $<k, N, m, L>$, where k is the number of disjuncts per constraint, N is the number of event nodes, and m is the number of constraints. The parameter L is the maximum upperbound, and $-L$ is the minimum lowerbound for each constraint. In this empirical validation, we used the settings: k= 2, L= 100, and N= [8, 10, 12, 14, 16]. The parameter m was set at 30% of the maximum number of possible constraints.

Fig. 4. 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 STPs. One hundred random TCSPs were generated for each N= [8, 10, 12, 14, 16]. The figure presents the median number of constraints reported, as well as the range over all 100 randomly generated TCSPs. Note that data is not reported for the component STP representation above N= 12 due to memory resource constraints.

Fig. 4 shows that the resulting compact representation reduces the space necessary to encode the TCSP solution set by up to three orders of magnitude, compared to prior art. Extrapolating trends in the data, TCSPs with up to 27 event nodes can now be addressed with the limited memory resources utilized in this empirical evaluation.

The key idea behind ICA-TCSP is to incrementally compile a TCSP to a compact representation that encodes the solution set in terms of only the differences among consistent component STPs. In applying the incremental update rules, typically only a subset of the constraints in the TCSP must be modified and recorded, contributing to the compactness of the representation.

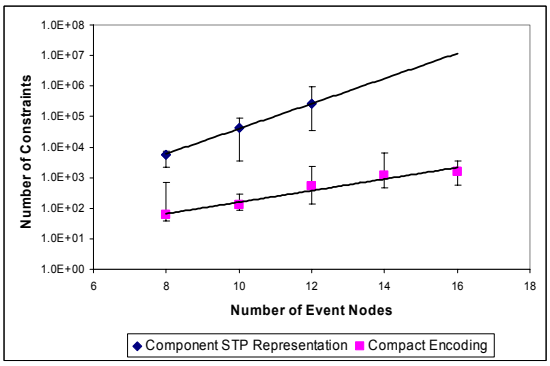

Figure 4: Space to Represent Solution: Compact Encoding vs. Component STP Representation

Algorithm for Fast Dynamic Scheduling

ICA-TCSP compiles a TCSP to a novel, compact encoding that supports fast dynamic scheduling. In this section, we describe how to schedule in real-time the compact compiled form.

In prior art (Tsamardinos 2001), dispatching the disjunctive temporal constraint network involves (1) maintaining data structures for each of the viable component STPs, (2) computing interrelationships among the component STPs online, and (3) propagating timing information simultaneously online to multiple STPs. Computing interrelationships among the component STPs is necessary to determine when events may be executed[1], and to ensure during execution that all viable solutions are not eliminated simultaneously (Tsamardinos 2001).

Recall that our compact representation enables fast dynamic scheduling by addressing two major sources of execution latency in prior art: (1) the repeated online computation necessary to compute interrelationships among the component STPs, and (2) the propagation of timing information online simultaneously to multiple STPs whenever an event is executed. Our compact representation of the TCSP solution set already encodes interrelationships among feasible component STPs. This reduces the amount of work necessary to identify interrelationships among component STPs online. Performing dynamic scheduling on this compact representation also eliminates the need to propagate timing information simultaneously to multiple STPs.

Fig. 5 shows the pseudo-code for INCREMENTAL-DISPATCH-TCSP. The algorithm takes as input the reverted TCSP S, the *relationship lists* R, and *conflicts* M. Notice that the algorithm is very similar to the dispatch algorithm for STPs (Muscettola et al. 1998). We have

[1] An STP event may be executed if it is *alive* and *enabled,* meaning that all the events that are constrained to occur before it have already been executed, and the temporal constraints of the event are satisfied (Muscettola et al. 1998).

```
function INCREMENTAL-DISPATCH-TCSP (S,R,M)
1.  E ← Add-Events-Without-Predecessors(S)
2.  current_time = 0
3.  X ← Initialize-Executed-List
4.  R ← Mark-All-Feasible-Disjuncts()
5.  while one or more events have not been executed
6.      wait until current_time has advanced such that some
        event N in E is live
7.      set N's execution time to current_time
8.      R,S ← UPDATE-FEASIBLE-DISJUNCTS(R,M,N,S)
9.      X ← Add-Event-To-Executed-List(N)
10.     if S is an STP then
11.         S ← Propagate-STP(S)
12.     else S ← PROPAGATE-COMPACT-ENCODING(S,R)
13.     end if
14.     E ← Add-Enabled-Event-Nodes(S)
15. end while
```

Figure 5: Pseudo-code INCREMENTAL-DISPATCH-TCSP.

highlighted the three lines in which they differ. In particular, INCREMENTAL-DISPATCH-TCSP requires two extra functions, provided in Fig. 6-7. We walk through the dispatch of Fig.1(d) to illustrate the algorithm.

First, in Line 1, all events without predecessors are added to the enabled list E. In our example, event A is added to E. Next, in Lines 2-3, the current time is initialized to zero, and list X is initialized to record executed events. In Line 4, we initially mark all disjuncts with non-empty *relationship lists* as *feasible,* meaning these disjuncts are currently feasible dispatch solutions. In our example, this means the disjuncts DE[2,3], DE[6,10], CE[6,6], and CE[11,11] are marked *feasible.* In Line 6, the algorithm waits until the current time has advanced such that some event N in the enabled list E is live, meaning the temporal constraints of event N are satisfied. Event N's execution is then set to the current time (Line 7). In the example, event A is executed at time t=0.

At this point, Line 8 contains the first major modification to the dispatching algorithm. Initially we mark each disjunct as *feasible.* However, each time an event is executed, each feasible disjunct must be checked to ensure it is still feasible. This is necessary since a subset of the disjuncts may not be consistent with the event's execution time. We update the feasible disjuncts by calling the function UPDATE-FEASIBLE-DISJUNCTS. In the example, the execution time of A at t=0 does not require any updates. In Line 9, we add the executed event A to the executed list X. In Line 10-11, if the reverted TCSP is found to be an STP, then the dispatcher proceeds according to the STP dispatch algorithm (Muscettola et al. 1998). However, if the reverted TCSP is not an STP, then we propagate event A's timing information using the compact encoding of the TCSP solution set. We do this in Line 12 using the function PROPAGATE-COMPACT-ENCODING. Once timing information has been propagated, all enabled events are added to the enabled list E. An event is enabled if all the events that are constrained to occur before it have already been executed. Finally,

Lines 5 through 15 repeat until all events have been executed.

Next, we describe the computation performed by UPDATE-FEASIBLE-DISJUNCTS (Fig.6). The function takes as input the *relationship lists* R, *conflicts* M, the most recently executed event N, and the reverted TCSP S. We illustrate the function assuming that event A in Fig 1(d) has been executed at time t=0, and event B has just been executed at time t=1. In Lines 1-3, we search each *relationship list* R_j for constraints (with feasible conditions) between each previously executed event and the most recently executed event. In the example, A is the previously executed event and B is the most recently executed event. First, we search *relationship list* DE[2,3] for a constraint between A and B. There is no such constraint, meaning that the logical consequences DE[2,3] did not imply a change to constraint AB. Therefore [U] is given by the corresponding constraint in S, AB[1,5] (Line 6). In Line 8, we find that B's execution time satisfies AB[1,5], therefore DE[2,3] remains *feasible*. Next, we search *relationship list* DE[6,10]. Again, no constraint is found, [U] = AB[1,5], and DE[6,10] remains *feasible*. In searching the next list CE[6,6], we find {C} = {AB[1,4]}. Since AB[1,4] is not conditioned on any disjuncts, [U]=AB[1,4]. In Line 5, we calculate that the number of feasible disjunct combinations involving CE[6,6] is one. The size of {C} is also one, meaning the *relationship list* contains all feasible intervals for AB; we skip to Line 8. In Line 8, we find that B's execution time at t=1 does satisfy [U], and therefore CE[6,6] remains *feasible.* Finally, we investigate *relationship list* CE[11,11]. We find {C}=[U]=AB[2,5]. In this case, B's execution time at t=1 does not satisfy AB[2,5] and therefore the disjunct CE[11,11] is marked as *infeasible* (Line 9). Intuitively, CE[11,11] is infeasible since choosing it to be part of the TCSP solution implies that B cannot be executed at t=1.

Next, in Lines 13-15, we reduce the consequences of disjunctive constraints with one feasible disjunct, using the *conflicts* M to identify and mark conflicting disjuncts as *infeasible.* In the example, disjunctive constraint CE has one feasible disjunct: CE[6,6]. Using the *conflicts* we find that they imply that disjunct DE[6,10] is *infeasible.* Notice that in the general case of more than two disjuncts per constraint, Lines 13-15 can be generalized as a SAT problem. Finally, in Lines 16-18, we update the constraints in S with the selected disjuncts and their implied logical consequences, given by R.

Next, we describe the computations performed by PROPAGATE-COMPACT-ENCODING (Fig. 7). This function efficiently propagates timing information to compute feasible windows of execution for future events. The computation of feasible time windows involves computing interrelationships among the possible TCSP solutions. The resulting time windows ensure that, during execution, all viable solutions are not eliminated simultaneously. The function takes as input the reverted

```
function UPDATE-FEASIBLE-DISJUNCTS (R,M,N,S)
1.  for each previously executed e_i in S
2.      for each relationship list R_j
3.          {C} ← search R_j for constraints between e_i and N with
                  feasible conditions
4.          [U] ← union of all {C}
5.          if size({C}) < Num-Feasible-Disjunct-Combinations
6.              [U] ← union of [U] and (constraint[N][e_k] in S)
7.          end if
8.          if N's execution time does not satisfy [U]
9.              mark the disjunct associated with R_j as infeasible
10.         end if
11.     end for
12. end for
13. for each disjunctive constraint with one feasible disjunct
14.     R ← Mark-Conflicting-Disjuncts-Infeasible(M,R)
15. end for
16. for each disjunctive constraint with one feasible disjunct
17.     S ← Update-Network(S,R)
18. end for
```

Figure 6: Pseudo-code UPDATE-FEASIBLE-DISJUNCTS.

```
Function PROPAGATE-COMPACT-ENCODING(S,R,N)
1.  for each future event e_k
2.      for each disjunctive constraint h_i
3.          for each relationship list R_j associated with h_i
4.              {P} ← search R_j for constraints between N and
                      each future event e_k with feasible conditions
5.              [U] ← union of all {P}
6.              if size({P}) < Num-Feasible-Disjunct-Combinations
7.                  [U] ← union of [U] and (constraint[N][e_k] in S)
8.              end if
9.              set X_j = [U]
10.         end for
11.         set [Y_i] = Union{X}
12.     end for
13.     set [Z] = Intersection{Y}
14.     update e_k's feasible time window to be the intersection of
        e_k's current time window and (current_time + [Z])
15. end for
```

Figure 7: Pseudo-code PROPAGATE-COMPACT-ENCODING.

TCSP S, and the *relationship lists* R. We illustrate the function assuming that event A in Fig 1(d) has been executed at time t=0, and event B has just been executed at time t=4.

In Lines 1-4, for each future event e_k and disjunctive constraint h_i, and for each *relationship list* R_j associated with h_i, we search R_j for constraints (with feasible conditions) between the most recently executed event N and the event e_k. In the example, we first consider the future event C, the disjunctive constraint DE, and its disjuncts: DE[2,3] and DE[6,10]. We first search DE[2,3]'s *relationship list* for constraints between events B and C. We find {P} = [U] = $[X_{DE[2,3]}]$ =BC[8,9] (Line 4-7). Next we search DE[6,10]'s *relationship list* and find the size of {P} is zero, meaning that the logical consequences DE[6,10] did not imply a change to constraint BC. Therefore, $[X_{DE[6,10]}]$ = BC[5,9], where BC[5,9] is given by the corresponding constraint in S (Line 7-9). In Line 11 we take the union of { $X_{DE[2,3]}$, $X_{DE[6,10]}$ } = Y_{DE} = BC[5,9]. Intuitively, Y_{DE} = BC[5,9] allows for the selection and

execution of all currently feasible DE disjuncts. Next, considering disjunctive constraint CE, we find that Y_{CE} = BC[5,5]V[9,9]. Intuitively, Y_{CE} = BC[5,5]V[9,9] allows for the selection and execution of all currently feasible CE disjuncts. Now, in Line 13, we use an intersection operation to ensure that any execution within event C's feasible time window will include at least one feasible disjunct for each disjunctive constraint. In the example, [Z] = intersection{ Y_{DE} = BC[5,9], Y_{CE} = BC[5,5]V[9,9]} = BC[5,5]V[9,9]. Intuitively, this means that C must be executed exactly 5 time units after B, or exactly 9 time units after B, to avoid simultaneously eliminating all dispatch solutions. This information is then used to update event C's feasible time window (Line 14).

In the next section we empirically show that INCREMENTAL-DISPATCH-TCSP reduces execution latency by more than three orders of magnitude, compared to prior art.

Empirical Validation of INCREMENTAL-DISPATCH-TCSP

We empirically validated INCREMENTAL-DISPATCH-TCSP by dynamically scheduling randomly generated TCSPs. We compared the execution latency associated with dispatching our compact encoding to the execution latency of dispatching the component STP representation. As a conservative measure, we recorded the execution latency to propagate the timing of the first executed event. This is a conservative measure for execution latency because in the compact encoding, all *relationship lists* are still feasible, and in the component STP representation, all consistent component STPs are still viable, thus increasing the computation required to propagate timing information.

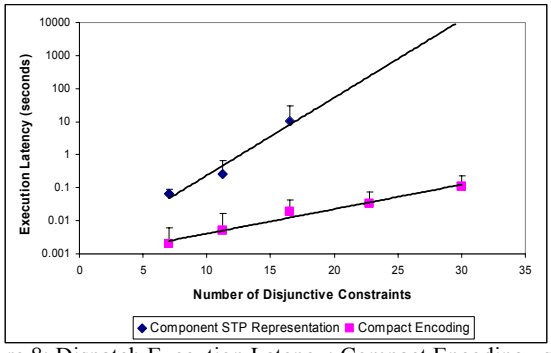

Figure 8: Dispatch Execution Latency: Compact Encoding vs. Component STP Representation

The results of the comparison are shown in Fig. 8. One hundred random TCSPs were generated for each *N*= [8, 10, 12, 14, 16]. 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 up to three orders of magnitude compared to dispatch of the component STP representation.

Conclusion

In this paper, we introduced an incremental compilation algorithm ICA-TCSP for compiling TCSPs to a compact dispatchable form that supports fast dynamic scheduling. The key innovation of this work is a compact representation that encodes a TCSP solution set in terms of only the differences among the solution component STPs. We empirically show that our compact encoding reduces the space necessary to encode the TCSP solution set by up to three orders of magnitude, compared to prior art. Also, we empirically show that our compact encoding supports fast dynamic scheduling, by reducing execution latency by more than three orders of magnitude, compared to prior art.

References

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

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

[Hofmann, A., Williams, B. 2006] Robust execution of temporally flexible plans for bipedal walking devices. *Proc.* ICAPS-06.

[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.*

[Muscettola, N., et el. 1998b] To boldly go where no AI system has gone before. *AI* 103(1):5-48.

[Oddi, A., and Cesta, A. 2000] Incremental Forward Checking for the Disjunctive Temporal Problem. In *Proc. 14th European Conf. on Artificial Intelligence*, 108–112.

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

[Stedl 2004] Managing Temporal Uncertainty Under Limited Communication: A Formal Model of Tight and Loose Team Communication, S.M. Thesis, MIT.

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

[**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 *Proceedings of the 6th European Conference on Planning*, 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.