Dynamic Controllability of Temporally-flexible Reactive Programs

Robert Effinger*[+], Brian Williams[+], Gerard Kelly[‡], and Michael Sheehy[‡]

*Charles Stark Draper Laboratory, Inc.
Autonomous Mission Control Group
555 Technology Square
Cambridge, MA 02139-3563
reffinger@draper.com

[+]Massachusetts Institute of Technology
Model-based Embedded and Robotic Systems
CSAIL, 32 Vassar St. Room 32-226
Cambridge, MA 02139
effinger@mit.edu, williams@mit.edu

[‡]University of Limerick
CTVR, Stokes Institute
Engineering Research Bld. ERO-006,
Limerick, Ireland
gkelly@mit.edu, msheehy@mit.edu

Abstract

In this paper we extend dynamic controllability of temporally-flexible plans to temporally-flexible *reactive programs*. We consider three reactive programming language constructs whose behavior depends on runtime observations; conditional execution, iteration, and exception handling. Temporally-flexible reactive programs are distinguished from temporally-flexible plans in that program execution is conditioned on the runtime state of the world. In addition, exceptions are thrown and caught at runtime in response to violated timing constraints, and handled exceptions are considered successful program executions. *Dynamic controllability* corresponds to a guarantee that a program will execute to completion, despite runtime constraint violations and uncertainty in runtime state. An algorithm is developed which frames the dynamic controllability problem as an AND/OR search tree over possible program executions. A key advantage of this approach is the ability to enumerate only a subset of possible program executions that guarantees dynamic controllability, framed as an AND/OR solution subtree.

Introduction

In the field of AI planning, temporal reasoning plays a central role. Many planning applications involve complex temporal relationships that must be satisfied for a plan to succeed. In order to be robust to uncertainty, many of these applications require flexibility in the timing of actions, both at planning and execution time. To achieve this robustness, the planning community has developed a family of constraint formalisms to model and reason efficiently over large networks of flexible temporal constraints. Such formalisms include; the temporal constraint network (TCN) (Dechter, Meiri, and Pearl 1991), the disjunctive temporal problem (DTP) (Stergiou and Koubarakis 1998), the temporal plan network (TPN) (Kim, Williams, and Abrahmson 2001), the simple temporal problem under uncertainty (STPU) (Vidal and Fargier 1999), and the conditional temporal problem (CTP) (Tsamardinos, Vidal and Pollack 2003). The CTP formalism is distinguished in that it supports both flexible temporal constraints and runtime observations. A host of temporal reasoning algorithms and complexity results have been developed along with these temporal formalisms, such as (Stergiou and Koubarakis 2001) and (Morris, Muscettola, and Vidal 2001). Central to these algorithms is the notion of *controllability*. Although the terminology differs between formalisms, the central question asked by each is: "Does an execution strategy exist in which the specified timing constraints are guaranteed to be satisfied?" The answer to this question may vary based on the information available to the plan's executive at runtime. This distinction has guided the community towards defining three separate notions of controllability: *weak, strong,* and *dynamic controllability*. Strong controllability requires no information at runtime in order to succeed, weak controllability requires all information ahead of time, and dynamic controllability requires information only as it becomes available.

In this paper, we extend *dynamic controllability* to temporally-flexible reactive programs with three reactive programming language constructs; conditional execution, iteration, and exception handling. The proposed language is an extension of the Reactive Model-based Programming Language (RMPL) (Williams, et. al. 2003) and is interpreted graphically as a Temporal Plan Network. RMPL also supports; simple temporal constraints, parallel and sequential composition, and non-deterministic choice. We choose to view the dynamic controllability problem as a two-player game between the executive and the environment. The executive picks times for controllable timepoints and makes choices for controllable choice points. Alternatively, the environment picks times for uncontrollable timepoints and makes choices for uncontrollable choice points. The executive is allowed to observe the outcome of uncontrollable timepoints and choice points as they occur, and then use that information to dynamically schedule future controllable timepoints and

choice points in order to guarantee successful program execution. An algorithm is developed which frames the dynamic controllability problem as an AND/OR search tree (Dechter and Mateescu 2007) over possible program executions.

Next we give a simple example of a temporally-flexible reactive program, and we review some necessary background material. Then, we define temporally-flexible reactive programs, and develop an algorithm for determining dynamic controllability. We conclude with a summary and discussion of the experimental results.

A Simple Example

Consider a whole-arm manipulator (WAM) pick-and-place task, depicted in Figure 1. The WAM moves to waypoint W1 (to its left), then down to the ground to grasp an object. After grasping the object, the WAM moves back to W1, travels through W2, and onto W3. Then, the WAM lowers to the ground to release the object. The portion of the task in-which the arm moves from W1 through W2 to W3 is encoded in RMPL in Figure 1. This example demonstrates the use of reactive language constructs to improve plan flexibility and robustness in achieving a goal; "to be at W3". Each "go" activity has a flexible timing constraint and a flexible operating region (depicted at the top right of Figure 1) which it should not leave. If an obstacle is observed between W1 and W2, the program specifies an alternate route through W2north. If the WAM is displaced from its operating region, a contingency is invoked; the WAM returns to the home position. The *until* construct is used to attempt the task two times before giving up and remaining at the home position. For example, consider the plot at the bottom right of Figure 1. The WAM was physically disturbed from its operating region several times between W1 and W3. The trajectory taken by the WAM each time to return to the home position is plotted. Dynamic controllability in this example corresponds to a guarantee that the WAM will end up either at the goal, W3, or at a safe contingency location, the home position, within 40 seconds (i.e. with no uncaught exceptions).

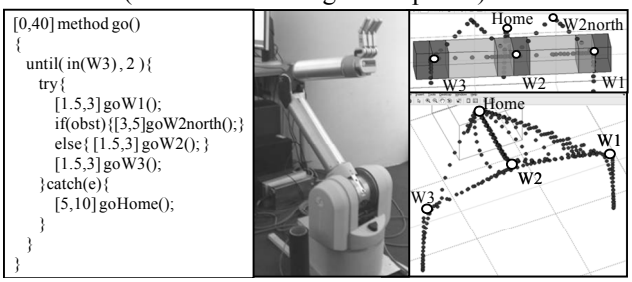


Figure 1: A Simple WAM Example

Background

We briefly review some necessary background material:

- Consistency for STPs, DTPs, and TPNs
- Weak, strong, dynamic controllability for STNU
- Weak, strong, dynamic consistency for CTP
- AND/OR search

Consistency for STPs, DTPs, and TPNs

The simple temporal problem (STP), the disjunctive temporal problem (DTP), and the temporal plan network (TPN) are all graphical models for analyzing temporally flexible networks of activities. In each model, a graph is constructed which consists of nodes and edges, $\langle v, e \rangle$. Nodes represent instantaneous timepoints (real-valued variables), while edges represent timing constraints between nodes. An example of each is shown in Figure 2. In an STP, each edge from node *i* to node *j* is labeled with an interval, [*lb*,*ub*], which represents a metric timing constraint, $lb <= X_j - X_i <= ub$. DTPs and TPNs generalize STPs by allowing disjunctions between timing constraints. The DTP and TPN differ from one another slightly; the DTP can be mapped into a meta-CSP while the TPN can be mapped into a meta-Conditional CSP as demonstrated in (Tsamardinos and Pollack 2003) and (Effinger 2006).

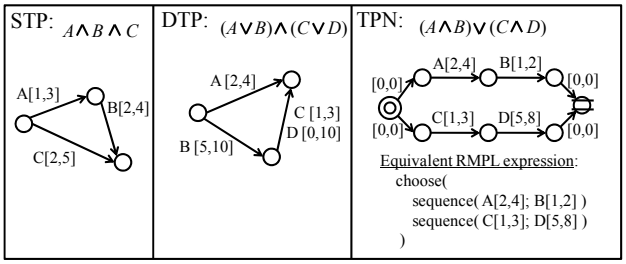


Figure 2: An Example STP, DTP, and TPN

Consistency for an STP is defined as finding one or more complete set of time assignments to each timepoint such that all constraints are satisfied. STP consistency can be determined in polynomial time by framing the problem as an all-pairs shortest path problem (Dechter, Meiri, and Pearl 1991). A consistent STP may then be dispatched efficiently by reformulating the all-pairs graph into a minimal dispatchable network (Muscettola, et. al. 1998). Consistency for DTPs and TPNs is determined by framing disjunctive choices among timing constraints as a meta-CSP. The meta-CSP is solved by traditional CSP search. A solution to a meta-CSP is a complete set of choices to the disjunctions among timing constraints in-which the selected component STP is consistent. Once found, a consistent component STP can be dispatched using the STP technique described above. An algorithm has also been developed to dispatch DTPs directly, retaining more flexibility at runtime (Tsamardinos, et. al. 2001).

Weak, Strong, Dynamic Controllability for STPU

In many applications, activity durations are not under the control of the executing agent and instead depend on external factors. To address this issue, the Simple Temporal Problem with Uncertainty (STPU) was developed. STPUs are similar to STPs except that edges are divided into *controllable* and *uncontrollable* edges (Vidal and Fargier 1999). The uncontrollable edges represent activities of uncertain duration, whose finish timepoints are controlled by nature, termed *uncontrollable*

timepoints. All other timepoints are *controllable timepoints*, which are freely controllable by the executive.

The definition of consistency for an STP needs to be extended for an STPU because the executive does not get to pick the times of execution for uncontrollable timepoints. Instead, an execution strategy is needed for assigning times to just the controllable timepoints such that all constraints are satisfied regardless of the values taken by the uncontrollable timepoints. Several such execution strategies exist, and they are classified in Figure 3, based on the information available to the agent at runtime.

STNU Controllability	Description
Weak Controllability	An execution strategy that is guaranteed to succeed for only one set of values taken by the uncontrollable timepoints at runtime. This strategy assumes that uncontrollable durations are known or can be predicted a-priori.
Strong Controllability	A static execution strategy that is guaranteed to succeed for all possible sets of values taken by the uncontrollable timepoints at runtime. This strategy doesn't need any knowledge at runtime to succeed.
Dynamic Controllability	A dynamic execution strategy that is guaranteed to succeed for all possible sets of values taken by the uncontrollable timepoints at runtime. This strategy relies on knowledge acquired at runtime in order to dynamically schedule future timepoints in order to satisfy all timing constraints. Hence, this strategy needs access to all runtime information as it becomes available.

Figure 3: STNU Controllability Classifications

Weak, Strong, Dynamic Consistency for CTP

The Conditional Temporal Problem (CTP) (Tsamardinos, Vidal, and Pollack 2003) augments the STP and DTP formalisms with observation nodes. Observation nodes have outgoing edges with labels attached that are associated with runtime observations. The CTP formalism allows for the analysis of plans with conditional threads of execution and flexible temporal constraints. Similarly to the STNU, in Figure 4, three notions of consistency are defined for the CTP; *weak, strong,* and *dynamic consistency*. The term consistency is used instead of controllability because all CTP timepoints are controllable.

CTP Consistency	Description
Weak Consistency	An execution strategy that is guaranteed to succeed for only one set of observations that may occur at runtime. This strategy assumes that all observations can be predicted or are known a-priori.
Strong Consistency	A static execution strategy that is guaranteed to succeed for all possible sets of observations that may occur at runtime. This strategy doesn't need any observational data at runtime to succeed.
Dynamic Consistency	A dynamic execution strategy that relies on observations acquired at runtime in order to dynamically schedule future choices and timepoints in order to satisfy all timing constraints. This strategy needs access to runtime observations as they become available.

Figure 4: CTP Consistency Classifications

The CTP dynamic consistency checking algorithm reformulates the original CTP into a DTP which explicitly encodes the requirements for dynamic consistency; namely, that scheduling decisions may only depend on observations that have occurred in the past. This is accomplished by enumerating all possible component STPs for a CTP, called *execution scenarios*, and placing them in a parallel network by conjoining each scenario's start and end nodes. Then, disjunctive constraints are added to the network to ensure that identical nodes across scenarios are only scheduled independently after a distinguishing observation has been made. The disjunctive constraints plus the underlying component STPs constitute a new DTP which is then checked for dynamic consistency using existing DTP solving techniques. If the reformulated DTP is consistent, then the original CTP is dynamically consistent. The reformulated DTP is then executed using the standard DTP dispatching techniques.

AND/OR Search

An AND/OR search tree is defined by a 4-tuple $\langle S, O, S_G, s_0 \rangle$ (Dechter and Mateescu 2007). S is a set of states partitioned into OR states and AND states. O is a set of two operators. The OR operator generates children states for an OR state, which represent alternative ways for solving a problem. The AND operator generates children states for an AND state, which represents a decomposition into subproblems, all of which need to be solved. The start state, s_0, is the only state with no parent state. States with no children are called *terminal states*, S_G and are marked as Solved (S) or Unsolved (U).

A *solution subtree*, *T*, of an AND/OR search tree is a subtree which:

(1) Contains the start state, s_0.
(2) If state *n* in *T* is an OR state, then exactly one child state of *n* must be in *T*.
(3) If state *n* in *T* is an AND state, then all child states of *n* must be in *T*.
(4) All terminal nodes are "Solved" (S).

A common example of AND/OR search is the two-player game. Consider a simple example shown in Figure 5. Both a hero and a villain pick a number from the set, {0,2}. If the additive total equals two, the hero wins, otherwise the villain wins. The possible outcomes are shown in Figure 5. In Figure 6a, the AND/OR tree is depicted for the case where the villain must choose first. In Figure 6b, the AND/OR tree is depicted for the case where the hero must choose first. Notice that a subtree which satisfies the definition of a *solution subtree* is available for case 6a but not for case 6b. To be guaranteed a win (i.e. to be "dynamically controllable") the hero must be allowed to choose last. Otherwise the villain can foil the hero.

Hero	Villain	Outcome
0	0	(U) ⊗
0	2	(S) ✓
2	0	(S) ✓
2	2	(U) ⊗

Figure 5: Hero/Villain Example

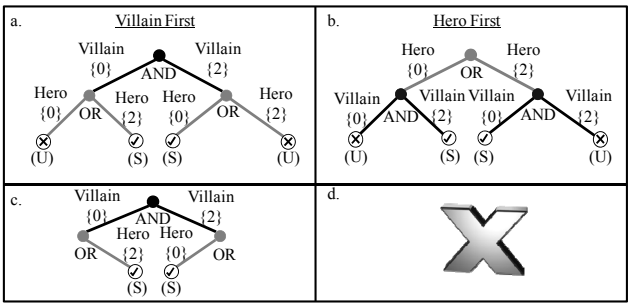


Figure 6: a. And/Or search tree when Villain goes first.
b. And/Or search tree when Hero goes first.
c. And/Or solution subtree for a.
d. No valid And/Or solution subtree exists for b.

Temporally-flexible Reactive Programs

In this section, we introduce temporally-flexible reactive programs consisting of seven constructs; conditional execution, iteration, exception handling, non-deterministic choice, parallel and sequential composition, and simple temporal constraints. The proposed language is an extension of the Reactive Model-based Programming Language (RMPL), which can be interpreted graphically as a Temporal Plan Network (TPN). The syntax for the proposed language is presented in Figure 7. A TPN explicitly represents the semantics implied by the RMPL syntax. The mapping from RMPL constructs to TPN elements is presented in Figure 10. A TPN is comprised of five node types and two edge types, as described in Figure 8. Arcs with no annotation are assumed to have bounds of [0,0]. The *until* construct consists of an uncontrollable choice between each number of possible iterations, bounded by *n*. The *try-catch* construct consists of an uncontrollable choice between the nominal thread of execution and each contingency. A contingency consists of an uncontrollable duration (to account for the uncertain timing of a thrown exception) followed by a handler for the exception type. Each catch clause, as well as the uncontrollable duration, in a *try-catch* construct is specified by the program's author, and represents the time window over which a program is robust to that particular exception. Threads of execution that include caught and handled exceptions are considered successful execution paths, and must be tested for temporal consistency with the rest of the program. Labels are attached to uncontrollable choice out-edges to represent the condition under which an edge is taken at runtime.

A TPN is comprised solely of TPN primitive activities, as defined in Definition 1. Each TPN primitive activity has an associated simple temporal constraint and a set of exceptions that it may throw. At a minimum, each TPN primitive activity has two exceptions, $\{e_{lb}, e_{ub}\}$, which are thrown if the activity completes before *lb* or doesn't complete by *ub*, respectively. More exceptions may be defined. For instance, in the WAM example, an exception is thrown when the WAM leaves its operating region.

An Example TPN

The TPN for the simple WAM example is depicted in Figure 9. A timing constraint [0,40] is placed in parallel with the rest of the TPN to enforce a 40 second time limit. Next, an uncontrollable choice is added for each possible iteration of the *until* construct, denoted by n=1 and n=2. Further iterations are bypassed once the halting condition, *in(W3)*, is met. The *if-else* construct branches on the presence of an obstacle, taking one of two possible routes. Finally, if an exception is thrown, the contingency is invoked by switching execution to the contingent thread.

RMPL Bakus-naur syntax:

wff ::= $a(p_1,p_2,\ldots)$ [lb,ub] | $a(p_1,p_2,\ldots)$ [lb?ub] | wff [lb,ub] |
try { wff } catch (cond) { wff_1 } catch-all { wff_2 } |
until(cond , n) { wff } |
sequence{wff_1,wff_2,…} |
parallel{ wff_1,wff_2,…} |
choose{ wff_1,wff_2,…} |
if (cond) { wff_1 } else { wff_2 }

$a(p_1,p_2,\ldots)$ ::= a primitive activity
[lb,ub] ::= a controllable metric timing constraint
[lb?ub] ::= an uncontrollable metric timing constraint
cond ::= a boolean statement which evaluates to true or false
n ::= a finite integer

Figure 7: RMPL Bakus-naur syntax

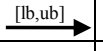
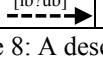
TPN element	Graphical Symbol	Description
Controllable timepoint		An instantaneous timepoint whose execution time is controlled by the executive.
Uncontrollable timepoint		An instantaneous timepoint whose execution time is NOT controlled by the executive and must be observed.
Controllable choice node		A non-deterministic choice point that the executive gets to pick at runtime.
Uncontrollable choice node		An uncontrollable choice point that the executive does NOT get to pick at runtime and must be observered.
Choice end node		An timepoint where alternative threads from a choice point reconverge.
Controllable time constraint		A solid arrow; A controllable metric timing constraint.
Uncontrollable time constraint		A dashed arrow; An uncontrollable metric timing constraint.

Figure 8: A description of TPN elements

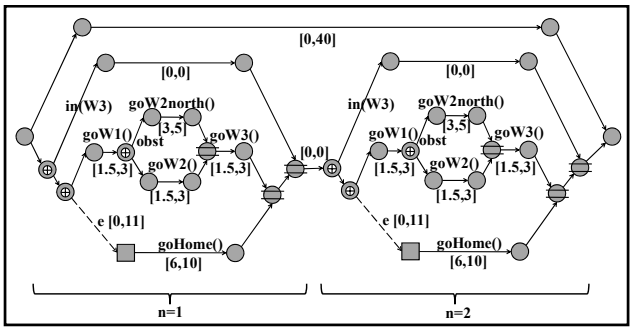
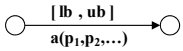
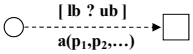
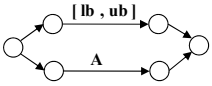
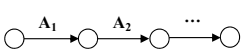
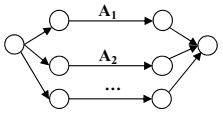
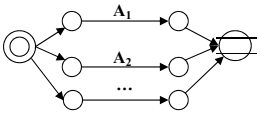
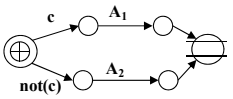
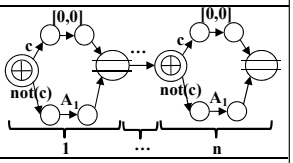
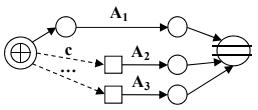


Figure 9: TPN for the Simple WAM Example

a(p_1,p_2,...) [lb , ub]	
a(p_1,p_2,...) [lb ? ub]	
A [lb , ub]	
sequence { A_1 , A_2 , … }	
parallel { A_1 , A_2 , … }	
choose { A_1 , A_2 , … }	
if (c) { A_1 } else { A_2 }	
until (c , n) { A_1 }	
try { A_1 } catch (c) { A_2 } catch(…) { A_3 }	

Definition 1-TPN Primitive Activity

A *TPN primitive activity* is a 5-tuple $\langle string, n_i, n_j, STC, e \rangle$.

- ***string*** represents a command or timing constraint.
  - Commands take the form *a(p_1,p_2,...)*.
  - Timing constraints have the empty string, *null*.
- n_i is a time event representing the start of the activity.
- n_j is a time event representing the end of the activity.
- ***STC*** enforces a metric timing constraint on the activity:
  - [lb,ub] - A controllable metric timing constraint, or
  - [lb?ub] - An uncontrollable metric timing constraint.
- ***e*** is a set $\{e_{lb}, e_{ub}, e_1, e_2, \cdots, e_n\}$ of supported exceptions.

Definition 2 – Candidate Execution

A *candidate execution, s,* of a TPN is an assignment of choices to controllable and uncontrollable choice nodes in a TPN. V_c denotes the set of controllable choices, while V_u denotes the set of uncontrollable choices.

Definition 3 – Complete Execution

A *complete execution,* s_c , is a candidate execution which can be defined constructively in the following manner:

(1) A choice is made to the first choice node encountered along each parallel thread of execution emanating from the startnode.

(2) Each thread activated by a previous choice is traversed, and again a choice is made to the first choice node encountered along each parallel thread of execution. Repeat (2) until the end node is reached.

In a complete execution, all threads start at the start node and end at the end node. No encountered choices remain unassigned, and no unencountered choices are assigned extraneously.

Definition 4 - TPN Fringe

The *TPN fringe* is defined as the current state of execution progress in a TPN and is used to distinguish between already executed nodes and nodes that are next in line for execution. The TPN fringe consists of exactly the set of nodes that are capable of being executed next during TPN execution. This rules out any previously executed node; any node constrained to occur after an unexecuted node; and any node pairs on mutually exclusive program branches, (i.e. along different candidate executions).

Dynamic Controllability of Temporally-flexible Reactive Programs

In this section, we define dynamic controllability for temporally-flexible reactive programs. Then, we focus on developing an algorithm to determine dynamic controllability. Our approach is inspired by the approach taken in (Tsamardinos, Vidal, and Pollack 2003) to determine dynamic controllability of CTPs. However, our definition of *successful execution* differs from prior work. In temporally-flexible reactive programs, constraint violations (i.e. exceptions) are allowed as long as they are caught and handled by an exception handler. The programmer is given the flexibility to decide whether, how, and when exceptions should be caught. Because handled exceptions are considered successful program executions, they must be tested for consistency with the rest of the program. Each handler must be guaranteed to have time to complete in response to caught exceptions, or the TPN is identified as uncontrollable. This approach tends to overconstrain TPNs with many modeled exceptions. In response, we develop a more reasonable approach called *N-fault dynamic controllability* which ensures robustness to at most *n* exceptions at runtime.

Definition 5 – Dynamic Controllability

Dynamic controllability of temporally-flexible reactive programs can be viewed as a two-player game between the executive and the environment. The executive picks times for controllable timepoints and makes choices for controllable choice points. Alternatively, the environment

picks times for uncontrollable timepoints and makes choices for uncontrollable choice points. The executive is allowed to observe the outcome of uncontrollable timepoints and choice points as they occur, and then use that information to dynamically schedule future controllable timepoints and choices in order to guarantee successful execution. This strategy needs access to runtime observations as they become available.

An algorithm is developed which frames the dynamic controllability problem for temporally-flexible reactive programs as an AND/OR search tree over candidate program executions. To guarantee dynamic controllability, the algorithm starts from the beginning of a TPN and branches generatively over the *TPNfringe* as decisions are able to be made either by the executive or by the environment. As in the two-player game example, an OR state is constructed when a decision is made by the executive, while an AND state is constructed when a decision is made by the environment. The result is an AND/OR search tree that explicitly encodes the constraints implied by dynamic controllability. Just as in the two-player game example, the AND/OR search tree considers all possible orderings of decisions that are able to be made between the executive and the environment. Each state in the search tree is comprised of a set of simple temporal constraints. The original program is dynamically controllable if and only if an AND/OR solution subtree exists in-which all simple temporal constraints are satisfied. If a program is determined to be dynamically controllable, the AND/OR solution subtree serves as a compact, dynamic execution strategy to ensure successful program execution.

Definition 6 – *N*-fault Dynamic Controllability
To determine *N*-fault dynamic controllability, we artificially constrain the choices made by the environment when constructing an AND/OR search tree. Along each branch, we keep track of the number of exceptions the environment has "chosen" to throw so far, and limit that number to *N*. This results in an AND/OR solution subtree which is robust to at most *N* runtime exceptions, but is more compact than a full AND/OR solution subtree.

Definition 7–AND/OR Search Tree Encoding of a TPN
An AND/OR search tree encoding of a TPN that determines the property of dynamic controllability is a 4-tuple $\langle S, O, S_G, s_0 \rangle$. Where,

- ***S*** is a set of states partitioned into OR and AND states. Each state is comprised of a set of TPN primitive activities (Definition 1), $S=\{a_1,a_2,\cdots,a_j\}$.
- ***O*** is a set of operators used to construct the child states for each state in *S*. These are defined below.
- s_0 is the start state.
- S_G is a set of terminal states with no children. Terminal states are NOT simply marked as solved (S) or unsolved (U), however. To determine dynamic controllability of an AND/OR subtree, all timing constraints that comprise the subtree comprise a consistent STP. Consistency is determined by performing an all-pairs shortest path computation on each subtree, as it is constructed.

Definition 8 – AND/OR Solution Subtree
A *solution subtree*, *T*, is a subtree of the AND/OR search tree which:

(1) Contains the start state, s_0.
(2) If state *n* in *T* is an OR state, then exactly one child state of *n* must be in *T*.
(3) If state *n* in *T* is an AND state, then all child states of *n* must be in *T*.
(4) All timing constraints that comprise the subtree form a consistent STP.

Defining the Operators, *O*

The operators to construct the AND/OR search tree from a TPN are encoded recursively in a function called *RecursivelyExpand()*. Pseudocode follows, along with a walkthrough of the algorithm on the WAM example.

```
function N-Fault Dynamically Controllable TPN ( startnode , N )
1.  TPNfringe = startnode;  i = 0;  //fault count
2.  s_0 = null;  parent(s_0) = null;   //andor start state
3.  ConvertUncontrollableTimepoints( );
3.  AND-OR-tree = RecursivelyExpand( startnode, s_0, TPNfringe, I, N )
4.  if SolutionSubtreeExists( AND-OR-tree ) return TRUE
5.  else return FALSE
```

```
function RecursivelyExpand ( startnode , s , TPNfringe , i , N)
1.  while not( empty( TPNfringe ) )
2.  if IntermediateTemporalConsistencyCheck( s , TPNfringe )
2.      if any node in TPNfringe is a controllable or uncontrollable choice
3.          ExpandFringe( s , TPNfringe , i , N)
4.      else  { s , TPNfringe}= BumpFringe( s , TPNfringe )
5.  else  Remove state s from AND-OR-tree.
```

```
function ExpandFringe ( s , TPNfringe , i , N)
1.   Create an (OR) constraint with parent s, and create a child state:
2.   for each node p in TPNfringe
3.       c = null; parent(c) = s; //Creates the new OR state c with parent s
4.       TPNfringe_copy = TPNfringe; // Creates a copy of the TPNfringe
5.       for each node a in TPNfringe and its copy, denoted a_copy
6.           e =  [0,0];  start(e) = a;  end(e)=a_copy;     //Creates [0,0] edge, e,
                                                  // between each node and its copy
7.           c += e;                              // puts edge e into the andor state c
8.       if node p is a controllable timepoint node
4.           ExpandNode( p_copy , c , TPNfringe_copy , i , N)
5.       else if node p is an uncontrollable choice node
6.           ExpandUncontrollableChoice( p_copy ,c ,TPNfringe_copy, i, N)
7.       else if node p is a controllable choice node
8.           ExpandControllableChoice( p_copy , c , TPNfringe_copy , i , N)
```

```
function ExpandNode ( p , s , TPNfringe , i , N )
1.  for each node a in TPNfringe except p
2.      e =  [0,inf];  start(e) = p;  end(e)=a;  //Constrains node p to occur
3.      s += e;                                  //before all other TPNfringe nodes
4.  Remove node p from TPNfringe
5.  Add all children d of node p to the TPNfringe
6.  Add all edges e traversed to children d into the andor state s, s +=  e;
7.  RecursivelyExpand( s , TPNfringe , i , N)
```

```
function ExpandUncontrollableChoice( p , s , TPNfringe, i , N )
1.  if node p comes from a try-catch statement, i = i + 1;
2.  Create an (AND) constraint with parent s, and create a child state c:
3.  for each uncontrollable choice out-edge, e_out, emanating from node p
4.      if i >= N and node p is from a try-catch statement   //N exceeded,
5.          if e_out is not the nominal thread, continue;  //skip contingencies
6.      c = null; parent(c) = s; //Creates a new AND state c with parent s
7.      TPNfringe_copy = TPNfringe; // Creates a copy of the TPNfringe
8.      for each node a in TPNfringe and its copy, denoted a_copy
9.          e = [0,0];  start(e) = a;  end(e)=a_copy;   //Creates [0,0] edge, e,
                                                  // between each node and its copy
10.         c += e;                               // puts edge e into new andor state c
11.     for each node c_copy in TPNfringe_copy
12.         if c_copy is p_copy, continue;//skip it
13.         e = [0,inf]; start(e) = p_copy; end(e) = a_copy; //Constrains node p_copy
14.         c +=  e;              //to occur before all other TPNfringe nodes
15.     Remove node p_copy from TPNfringe_copy
16.     TPNfringe_copy += end(e_out)        //Add chosen out-edge endnode
17.     c += e_out;          // Add chosen out-edge e_out into new andor state c
18.     RecursivelyExpand( c , TPNfringe_copy , i , N )
```

```
function ExpandControllableChoice ( p , s , TPNfringe , i , N )
1.  Create an (OR) constraint with parent s, and create a child state c:
2.  for each controllable choice out-edge, e_out, emanating from node p
3.      c = null; parent(c) = s; //Creates a new OR state c with parent s
4.      TPNfringe_copy = TPNfringe; // Creates a copy of the TPNfringe
5.      for each node a in TPNfringe and its copy, denoted a_copy
6.          e = [0,0];  start(e) = a;  end(e)=a_copy;   //Creates [0,0] edge, e,
                                                  // between each node and its copy
7.          c += e;                               // puts edge e into the andor state c
8.      for each node c_copy in TPNfringe_copy except p_copy
9.          if c_copy is p_copy, continue;//skip it
10.         e = [0,inf]; start(e) = p_copy; end(e) = a_copy; //Constrains node p_copy
11.         c += e;              //to occur before all other TPNfringe nodes
12.     Remove node p_copy from TPNfringe_copy
13.     TPNfringe_copy += end(e_out)        //Add chosen out-edge endnode
14.     c +=  e_out;          // Add chosen out-edge e_out into the andor state s
15.   RecursivelyExpand( c , TPNfringe_copy , i , N )
```

```
function BumpFringe ( s , TPNfringe)
1.  Find all nodes, n, in or extending from TPNfringe that are guaranteed
    to execute before or simultaneously with the next possible choice or
    uncontrollable choice node.
2.  for each node in n
3.      for each out-edge e emanating from n
4.          if endnode(e) is also in n
5.              s +=  e;
6.  Remove all nodes, n, from the TPNfringe.
7.  Add all children of nodes n, called c, to the TPNfringe.  Exclude any
        children, c, which are also in n.
8.  return { s , TPNfringe }
```

```
function ConvertUncontrollableTimepoints ( )
1.  Find all uncontrollable edges [lb?ub] in the TPN.
2.  for each edge e
3.      for each time increment t between lb and ub
4.          Create an uncontrollable choice with two possibilities
5.              choice1: the edge extends for one additional time increment
6.              choice2: the edge terminates
```

```
function IntermediateTemporalConsistencyCheck ( s , TPNfringe )
1.  Find all nodes, n, and edges, e, between the startnode and TPNfringe.
2.  if  TemporallyConsistent( startnode + n + TPNfringe , e )
3.      return true;
4.  else
5.      return false;
```

A Simple Walkthrough

The algorithm starts by placing the TPN start node in the TPN_fringe, and then expands recursively via the function *RecursivelyExpand*(). Uncontrollable timepoints are handled by converting them into uncontrollable choices via *ConvertUncontrollableTimepoints*(). This function reflects the environment's ability to execute an uncontrollable timepoint at any time up to the last instant.

RecursivelyExpand() then proceeds in two possible ways. If the TPN_fringe contains either a controllable or uncontrollable choice node, *ExpandFringe*() is called. If not, then *BumpFringe*() is called, which "bumps" the fringe to the next choice or uncontrollable choice node to expand. *ExpandFringe*() creates a branch for each node in the TPN_fringe, and creates a new OR state which consists of a copy of the TPNfringe. This is depicted graphically in Figure 11 for the first call to *ExpandFringe*() for the WAM example. Next, *ExpandFringe*() proceeds differently depending on the type of node *p*. If at any point the subtree becomes temporally inconsistent, the function IntermediateTemporalConsistencyCheck() prunes the subtree from the ANDOR tree. In the WAM example, two nodes are expanded, a controllable timepoint node and an uncontrollable choice node, via functions *ExpandNode*() and *ExpandUncontrollableChoice*(), respectively. The ANDOR tree after these function calls is depicted in Figure 12. Intuitively, the AND/OR search tree ensures that for at least one ordering of the two nodes in TPN_fringe, both possible outcomes for the uncontrollable choice node will be consistent.

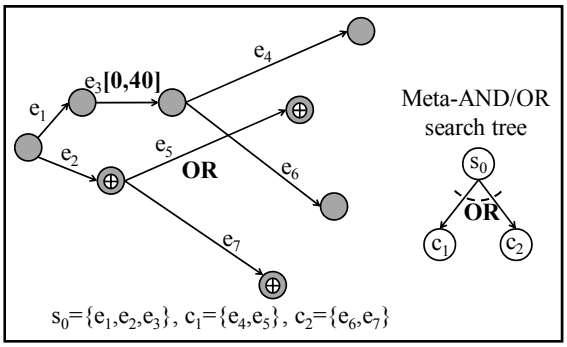


Figure 11: First call to *ExpandFringe*() in WAM example

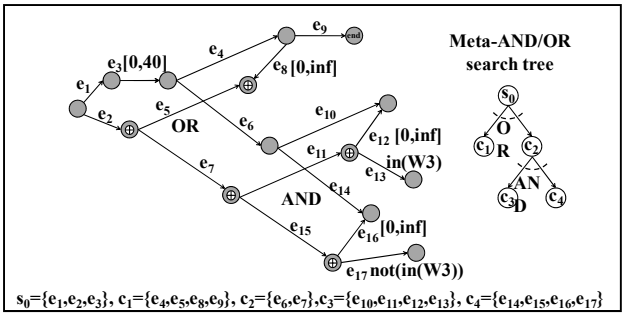


Figure 12: Function subcalls to *ExpandNode*() and *ExpandUncontrollableChoice*() in the WAM example.

Experimental Results and Discussion

AND/OR search trees can be solved using standard search techniques, such as depth-first search. One, some, or all solution subtrees may be found. The ability to enumerate just one solution subtree, which corresponds to a subset of possible program executions that guarantees dynamic controllability, is considered a key advantage of this approach. In Figure 13, we show that tractability can be improved by placing an upper limit on robustness to the number of faults at runtime. Four categories of randomly generated programs were tested for dynamic controllability. The program generator creates programs with three parallel threads of execution with randomly chosen RMPL constructs. To increase program complexity, the maximum depth of nested constructs was increased from 1 to 4. For example, in Category 3, a program may consist of a contingent choice construct, nested within a try-catch construct, nested within a controllable choice construct. We observe that computation time is driven almost solely by the number of faults we wish to be robust to, regardless of the complexity of the original plan. Approximately an order of magnitude improvement is seen for each unit reduction in the number *N* for *N*-fault controllability. Intuitively, this can be understood as reducing the number of variables in the underlying CSP problem for each fault that is ignored.

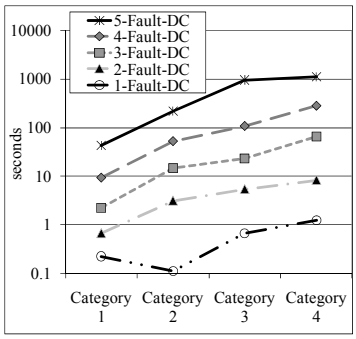


Figure 13: N-fault DC results

The approach taken in this paper was inspired by the work of (Tsamardinos, Vidal, and Pollack 2003) to determine dynamic consistency of CTPs. Our work is novel in a few key respects:

- The definition of a *successful execution* is extended to include execution paths with constraint violations (i.e. exceptions) as long as they are caught and handled by an exception handler.
- Temporally-flexible reactive programs bridge the gap between the most common embedded programming language constructs used in practice to build reactive systems (loops, conditions, and exceptions) and the formalisms employed by temporally-flexible planners.
- The proposed algorithm achieves greater compactness than prior art by exploiting two structural restrictions of temporally-flexible programs.
    1. Threads of execution in a TPN are physical processes which always proceed forward in time.
    2. Branches in a TPN always emanate from a single program point and coalesce to a single program point. The proposed algorithm could be extended to support arbitrary disjunctions, but this would require some additional bookkeeping.
- To model looping, conditions, and exception handling, a plan representation is provided which supports both activities of uncontrollable duration (to model the uncertain timing of a thrown exception) as well as uncontrollable outcomes (to model the uncertainty in the type of exception thrown, loop halting, etc…)

Conclusion

In this paper, we extend dynamic controllability of temporally-flexible plans to temporally-flexible reactive programs. In temporally-flexible reactive programs, program execution is conditioned on the runtime state of the world, and exceptions are thrown and caught at runtime in response to violated timing constraints. Handled exceptions are considered successful program executions, and *dynamic controllability* corresponds to guarantee that a program will execute to completion, despite runtime constraint violations and uncertainty in runtime state. An algorithm is developed which frames the dynamic controllability problem as an AND/OR search tree over possible program executions. A key advantage of this approach is the ability to enumerate only a subset of possible program executions that guarantees dynamic controllability, framed as an AND/OR solution subtree. To improve tractability, an approach is developed called *N-fault dynamic controllability* which ensures robustness to at most *n* exceptions at runtime.

References

Dechter, R.; Meiri, I.; Pearl, J., 1991. Temporal Constraint Networks. *Artificial Intelligence*, 49:61-95.

Kostas Stergiou and Manolis Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *AAAI-98.*

Kim, Williams, and Abrahmson, 2001. Executing Reactive, Model-based Programs through Graph-based Temporal Planning. *IJCAI*.

Vidal, T. and Fargier, H., 1999. Handling contingency in temporal constraint networks: from consistency to controllabilities. *Journal of Experimental & Theoretical AI,* 11, 23-45.

Tsamardinos, I., Vidal, T., Pollack, M., 2003, CTP: A New Constraint-Based Formalism for Conditional, Temporal Planning. Constraints 8 (4).

Stergiou, K. and Koubarakis, M., 2000. Backtracking algorithms for disjunctions of temporal constraints. *AI* 120, 81-117.

Morris, P., Muscettola, N., and Vidal, T., 2001. Dynamic Control Of Plans With Temporal Uncertainty. *IJCAI-01*. Seattle, WA.

Tsamardinos, Pollack, Ganchev. 2001. Flexible dispatch of disjunctive plans. Sixth European Conference on Planning.

Williams, et. al. 2003. Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers, *Proc. of the IEEE: Modeling and Design of Embedded Software*, vol. 91, pp. 212-237.

Dechter, R., and Mateescu, R., 2007. AND/OR search spaces for graphical models. *Artificial Intelligence* 171, Feb 2007, 73-106.

Tsamardinos, I., and Pollack, M., 2003. Efficient Solution Techniques for Disjunctive Temporal Reasoning Problems. *AI*, 151(1-2): 43-90.

Effinger, R. 2006. Optimal Temporal Planning at Reactive Time Scales via Dynamic Backtracking B&B. S.M. Thesis, MIT.

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

Morris, P., et. al. 2001. Dynamic Control of Plans with Temporal Uncertainty. IJCAI 2001.