Abstract

Dynamic execution is a flexible plan execution technique in which a plan executive schedules and executes tasks dynamically at runtime in response to disturbances in order to satisfy plan constraints. In this paper, we extend dynamic execution to temporally and spatially flexible plans which,
1) execute tasks conditionally based on runtime state, and
2) support error recovery for anticipated runtime constraint violations. To accomplish these goals, we broaden our focus from dynamic execution of flexible plans to dynamic execution of flexible reactive programs.

First, we introduce the Reactive Model-based Programming Language (RMPL) which, in addition to modeling temporal and spatial flexibility, includes three reactive programming language constructs: conditional execution, iteration, and exception handling. Then, we develop a probabilistic particle-sampling based dynamic execution algorithm which reasons efficiently over future program states to schedule tasks dynamically at runtime in order to satisfy program constraints. In addition, the algorithm monitors its own progress and notifies the executive if at any time the likelihood of successful program execution drops below a specified probability bound, δ.

Introduction

For complex robotic systems, such as a humanoid robot, temporally and spatially flexible plans increase robustness by specifying flexible ranges over state and time within which task execution may occur [Hofmann 1996, Hofmann and Williams 2006]. A flexible plan executive is capable of executing tasks dynamically at runtime in response to disturbances in order to satisfy plan constraints. This type of plan execution is called dynamic execution [Effinger 2009, Tsamardinos 2003], and is distinct from traditional plan-execute-replan approaches. At runtime, a flexible plan executive monitors the current state and reasons over future states to dynamically schedule tasks (from a flexible set of alternatives) in order to satisfy plan constraints. Despite a plan’s flexibility, when a disturbance becomes too large, the state may fall outside of the flexible plan’s constraint boundaries, causing the plan to fail. If executing a plan within constraints becomes impossible at runtime, the algorithm halts execution. In prior work, dynamic execution has been accomplished by translating an RMPL specification of the desired system behavior into a temporally-flexible plan graph, called a Temporal Plan Network [Kim, et. al. 2001, Effinger 2006], and then using network shortest path algorithms [Dechter et. al. 1991] to determine constraint satisfaction at runtime. This type of flexible plan execution still has two significant drawbacks: 1) Although flexible, a plan is still unconditional, meaning it can’t change which tasks are executed at runtime, and 2) Any constraint violation results in plan execution failure, which halts all subsequent plan execution and invalidates any subsequent guarantees of plan success. In the case of a humanoid robot, and other underactuated systems, plan execution failure may result in a catastrophic system failure.

To resolve these issues, in this paper we broaden our focus from dynamic execution of flexible plans to dynamic execution of flexible reactive programs. We accomplish this by adding three reactive constructs to the RMPL language which are prevalent in embedded systems programming; conditional execution, iteration, and exception handling. Temporally and spatially flexible reactive programs have three defining characteristics:
1) They model temporal and spatial flexibility,
2) They are conditioned on the runtime state of the world,
3) They support error recovery for anticipated runtime constraint violations.

Reactive programs support error recovery by throwing and catching exceptions at runtime in response to constraint violations. Handled exceptions are considered successful program executions, while uncaught exceptions are considered program execution failures. Dynamic
execution of a reactive program corresponds to monitoring program state and reasoning over likely future states in order to schedule future tasks such that the program will execute to completion with no uncaught exceptions. Note that constraint violations (thrown exceptions) are allowed as long as they are caught and handled by an exception handler. In this context, exception handling is analogous to a limited form of replanning. To perform real-time dynamic execution of RMPL programs, we develop a new probabilistic particle-sampling based dynamic execution algorithm which reasons efficiently over program state in order to guarantee successful program execution within a specified probability bound, $\delta$.

**Simple Example**

Consider a humanoid robot moving three cargo boxes from location A to location B within a specified time-limit of 120 seconds. To create a plan specification for this task, we would ideally like to be able to use all of the features described in the introduction: temporal and spatial flexibility, conditions on runtime state, and error recovery for anticipated failures (like dropping a box). Using RMPL, we write such a specification below. To enforce the timing deadline, we encompass the program within a timing constraint, 0 to 120 seconds. The specification program incorporates conditions on the state of the world (e.g. box_delivered), and anticipates likely task failures (e.g. dropped_box). In this paper, we develop a dynamic execution algorithm which can reason from the following specification to ensure task success.

```
[0,120]{
    until(box_delivered, 3) {
        travel_to_A() 
        grab_a_box() 
        travel_to_B_with_box() 
        put_down_box() 
    }
}
```

---

**Prior Work**

This paper builds upon two areas of prior work:

- Temporally and spatially flexible plan execution
- Dynamic execution of conditional temporally-flexible plans

**Temporally and Spatially Flexible Plan Execution.**

Robust control of complex robotic systems, such as a humanoid robot operating in a dynamic and unstructured environment, poses a challenge for traditional task and motion planners which operate on a plan-execute-replan cycle. These planners assume that a new plan is capable of being generated fast enough to replace an invalidated plan at runtime. However, in complex robotic systems with a large number of continuous and discrete state variables, highly-coupled dynamics, and real-time demands, the time to create a new plan is often several orders of magnitude longer than the time available to replan. In response, temporally and spatially flexible plan representations have been developed [Hofmann 1996, Hofmann and Williams 2006] which are valid over a range of initial, operating, and goal states. These plans are capable of executing despite uncertainty and disturbances at runtime as long as the system state remains within allowable bounds.

**Definition 1:** A temporally and spatially flexible task.

- State constraints on $R_{init}$, $R_{op}$, and $R_{goal}$, are convex, polygonal, piecewise-affine, and time invariant.
- $t_{goal} \in [lb, ub]$ is a simple temporal constraint[Dechter 91]
- Compile offline a set of control policies, called a flowtube, which collectively represent an explicit control law of the form, $u_t = f(x_t, t)$, such that for a model of the system dynamics, $M$, $\forall$ (for every) $x_{init} \in R_{init}$ and $\exists$ (for some) $x_{goal} \in R_{goal}$ there exists a control policy $u^* = \{u_1, u_2, \ldots, u_{t_{goal}}\}$ that takes $M$ from $x_{init}$ to $x_{goal}$ $\forall$ (for every) $t_{goal} \in [lb, ub]$ without violating $R_{op}$.

---

Figure 2: a) Successful execution, b) Unsuccessful execution.

For small disturbances, displacement will occur within the flowtube allowing an activity to still execute successfully (Figure 2a). If a disturbance is too large, the trajectory is pushed outside the flowtube, indicating task execution...
failure (Figure 2b). When a constraint is violated, the task executive throws an exception and halts execution. Temporally and spatially flexible plans are constructed by chaining together flexible tasks in sequence and parallel into a flexible task network (Figure 3). This approach has been used to demonstrate humanoid walking amidst runtime disturbances [Hofmann and Williams 2006]. Network shortest path algorithms [Dechter, et. al. 1993] are used to ensure plan consistency, and to schedule task completion times at runtime which satisfy plan constraints.

In this paper, two significant drawbacks to this approach are addressed: 1) These flexible plans are not conditioned on the state of the environment, and 2) When disturbances become too large, the system state falls outside of the flexible plan’s boundaries, causing plan execution to fail. When plan execution fails, all subsequent plan execution is halted, and any guarantees of plan success are invalidated.

Dynamic Execution of Conditional Temporally-flexible Plans. Dynamic execution of temporally-flexible plans that are conditioned on the state of the world has been attempted before in the literature [Effinger 2009, Tsamardinos 2003]. These prior approaches enumerate a constraint network that is comprised of all possible candidate plan executions, and then perform runtime constraint propagation over this network in order to guarantee successful plan execution. While this type of runtime constraint propagation works well for unconditional plans, it causes plan execution to grind to a standstill while propagating over an exponential number of candidate executions. These prior approaches are not fast enough to perform real-time dynamic execution, and would be more aptly described as existence proofs for a real-time dynamic execution strategy, should P = NP. Another drawback of these prior approaches is that they are set-bounded, meaning that they plan for the worst possible case in order to provide a guarantee of successful plan execution. This can lead to conservative plan execution when combinations of off-nominal situations which are very unlikely are still planned for. In this paper, we overcome these drawbacks by developing a more practical and tractable dynamic execution algorithm which guarantees successful execution within a specified probability bound, δ.

Problem Statement

In general, the problem we are trying to solve can be described as follows:

“Given a specification of desired behavior, S, develop an execution strategy, E, which guarantees within a probability bound, δ, that the system state, x, will obey S, x ∈ S.”

“In addition, monitor the runtime progress of the execution strategy, E, and provide notification if at any time the likelihood of successful execution, x ∈ S, drops below δ.”

δ ∈ [0,1] is the desired minimum probability of successful program execution. Next, we develop the specification, S and dynamic execution strategy, E.

Problem Approach

First, we define a flexible and reactive specification language, S, which enables the flexible encoding of a broad class of desired system behaviors. Then, we develop a dynamic execution strategy, E, which is capable of enforcing the desired system behavior, S, at runtime.

Specification Language (S)

In this paper, we pursue a specification language, S, with the following properties:

- Specifies flexible ranges over state and time within which task execution may occur,
- Is conditioned on the runtime state of the world, and
- Supports error recovery for anticipated runtime constraint violations.

We propose such a specification language by adding three new constructs to the Reactive Model-based Programming Language (RMPL), namely: conditional execution, iteration, and exception handling. The proposed language is presented below in Figure 4. For the specification of a primitive task, we borrow Definition 1 from prior work, and we note that each primitive task can be targeted to complete at any time between its lower and upper time bounds, (lb ≤ t_{goal} ≤ ub). The system behavior implied by an RMPL program is enforced by translating the program into a dual representation called Hierarchical Constraint Automata (HCA). An HCA enforces the desired system behavior by guiding program execution through a series of hierarchical state transitions guarded by transition labels. A simple example in Figure 5 shows the translation from RMPL syntax, to RMPL syntax tree, to HCA. Notice that primitive tasks occupy all of and only the leaves in a program syntax tree, and analogously, all of and only the most nested states in an HCA. Because RMPL syntax is tree-like, all HCA states, except for the root state, have a single parent.
Execution progress in an HCA is tracked using the notion of marked (i.e. active) states. The root state is the only initially marked state, and states become marked and unmarked as they are transitioned into and out of, respectively. In order to make scheduling decisions which satisfy program constraints, the execution strategy, \( E \), needs to reason over the specification structure at runtime. We define three functions which allow access to the specification structure: \( s_0 = S.getRoot() \), \( [s_1, \ldots, s_n] = S.getChildren(s_i) \), and \( s_p = S.getParent(s_i) \). In addition, a number of functions are needed to access information about individual states in the specification, \( s_i \). A few examples are: \( t_i = s_i.getStartTime() \), \( n = s_i.numChildren() \), \( string = s_i.getType() \), and \( string = s_i.getExecutionStatus() \).

**Execution Strategy (\( E \))**

Given an HCA specification, \( S \), as defined above, a valid execution strategy, \( E \), must satisfy three requirements:

**Requirement 1:** \( E \) must enforce the state transition model, \( S \), by only marking and unmarking states as they are transitioned into and out of.

**Requirement 2:** Because \( S \) is flexible in time and space, it is not fully specified. Therefore, \( E \) must make two types of scheduling decisions at runtime:

1. When a primitive task state in the HCA is activated, \( E \) must pick a target completion time for the task.
2. When a choose state is activated, \( E \) must pick exactly one of the available sub-programs for execution.

These decisions must be made at reactive timescales, as soon as a state is marked for execution. In addition, \( E \) must ensure that its scheduling decisions guarantee within a probability bound, \( \delta \), that system state, \( x \), will obey \( S, x \in S \).

**Requirement 3:** \( E \) must provide notification if at any time the likelihood of successful execution, \( x \in S \), falls below \( \delta \).

Next, we develop a probabilistic particle-sampling based dynamic execution algorithm which meets these three requirements.

**Probabilistic Particle-sampling Based Dynamic Execution Algorithm**

In this section we describe a new method for solving the dynamic execution problem framed above. In Section A, we summarize the general method and explain the key ideas, while in Section B we describe in more detail the technical details of the probabilistic particle-sampling based approach.
A. Outline of the General Method

The key observation behind the new method is that by limiting the amount of information available to the execution strategy, $E$, at runtime, we can make scheduling decisions at reactive timescales while still providing a guarantee on successful program execution. What the new method sacrifices is completeness. There will be some specifications, $S$, for which the new method fails to find a satisfactory dynamic execution strategy, but for which one does exist, and could be found by using the methods described in prior work. However, when the proposed method does find a satisfactory dynamic execution strategy it is sound, meaning the strategy is guaranteed to succeed with probability $\delta$ and it runs at reactive timescales. We hypothesize that a majority of the problems of interest in robotic applications are solvable with this method.

The general intuition behind the method is to limit the runtime scheduling information available to each state in the HCA to only the scheduling information computed and stored in its parent and child states. Using this approach, scheduling information must “bubble up” and “trickle down” the HCA family tree. This is in stark contrast to prior work, in which a large constraint network flattens the entire search space allowing any scheduling decision to depend on almost any other scheduling decision. One consequence of this new approach is that each HCA parent state must compute and retain the scheduling information required to dynamically schedule its children states. The high-level pseudocode for the dynamic execution algorithm, Algorithm 1, is presented twice below. Once using intuitive English prose, and again functionally. The algorithm details are presented in the following section.

B. Technical Approach

To retain tractability and to make scheduling decisions at reactive timescales, the quantity of information passed between HCA states at runtime must be very limited. In order to avoid a constraint representation explosion and to model arbitrary distributions over task completion times, we choose a particle-sampling based probabilistic representation [Doucet, et. al. 2001,Blackmore 2006]. To introduce this representation, we start with a primitive task.

**Primitive Task.** A primitive task (Definition 1) is capable of being targeted between, $t_{\text{target}} \in [lb, ub]$. We discretize time into integer valued bins, and for each (discrete) target time, a probability distribution over actual task end times is recorded. This distribution can be learned experimentally, or it can be approximated using common-sense. Consider a primitive task, go(5,10). For example, at $t_{\text{target}} = 7$, let us approximate the distribution over actual end times by 1000 samples from a Gaussian distribution with mean 7 and variance 3, as shown in Figure #a. A similar distribution is stored for each $t_{\text{target}} \in [lb, ub]$, resulting in an array of probability distributions, one for each target time, as shown in Figure #b. This array is the only

![Figure 6: a) A primitive task pointed at t=7, b) An array of p-dists for task go(5,10).](image-url)
scheduling information stored at runtime for a primitive task, and it is made available to the parent HCA state. A task failure probability is incorporated by normalizing the sum of all samples to \((1 - \text{probability of task failure})\).

Three functions that are needed to interact with a primitive task: \texttt{execute()}, \texttt{markAsExecuted()}, and \texttt{propagateSchedulingInfo()}, are developed below.

```cpp
function p_array = propagateSchedulingInfo(s)
1. return s.p_array /* simply return the task’s p_array,
// this is a leaf in the recursive function call
```

```cpp
function status = execute(s,t)
1. s.status = s.start(t); // Try to start the actual task
2. return s.status //Will be executing or an exception
```

```cpp
function markAsExecuted(s,t)
1. Unmark the HCA state
2. Update the p_array with the new execution information,
3. for i = s.lb to s.ub
4. for j = 1 to s.N // latest non-zero sample
5. if (j == t) p_array(i,j) = 1;
6. else p_array(i,j) = 0;
7. return
```

Sequence. Now that we have developed a probabilistic representation for primitive tasks, we address how to propagate scheduling information among other HCA states, such as the Sequence state. The theme taken in this paper will be to point each HCA state just as if it were a primitive task. Consider, for example, the RMPL fragment: \texttt{Sequence{ go1(1,5), go2(1,5) }}. We can define a lower and upper bound, \(lb^{-}\) and \(ub^{+}\), for the fragment where \(lb^{-} = lb_{1} + lb_{2} = 2\) and \(ub^{+} = ub_{1} + ub_{2} = 10\). Therefore, the sequence state can be targeted to complete at any time, \(t_{target} \in \{lb^{-}, ub^{+}\} = [2, 10]\). Next, we describe a local execution strategy (the \texttt{execute()} algorithm below) for the sequence HCA state which attempts to achieve the specified \(t_{target}\) only using the scheduling information available in its children HCA states at runtime. The \texttt{execute()} function takes as input the desired target completion time, \(t_{target}\), for the sequence, and distributes the time proportionally among each task. For example, for a sequence with two tasks, the formulas are:

\[
\begin{align*}
\text{tt}_1 &= \text{tt_seq} / (1 + \text{dur}_2 / \text{dur}_1) \\
\text{tt}_2 &= \text{tt_seq} / (1 + \text{dur}_1 / \text{dur}_2)
\end{align*}
\]

At runtime, if a task completes either ahead or behind target, the next task greedily tries to compensate up to but not exceeding its flexibility limit.

Next, we develop the \texttt{propagateSchedulingInfo()} function. Given an execution strategy, \texttt{execute()}, and the scheduling information of all child tasks (in the form of probability arrays as depicted above), we can compute an aggregate probability array for the sequence state which represents the probability distribution over likely end states for each of the sequence state’s target times, \(t_{target}\). The resulting probability array is analogous to the one constructed for a primitive task, and similarly, can be used by its parent HCA state to make scheduling decisions. A visual depiction of the probability array that results from running \texttt{propagateSchedulingInfo()} on the RMPL sequence fragment above is shown in Figure 7. The algorithm is similar to a discrete convolution algorithm over independent random variables except that it is more focused because of the corrective influence of \texttt{execute()}, as it greedily tries to attain the desired target times.

```cpp
function status = execute(s,t)
1. tt_1 = Calculate the desired target times for each task in seq by distributing proportionally among each task.
2. if no tasks have executed yet, execute the first child task \texttt{execute( s.child(1), tt_1 );}
3. Elseif all tasks executed, bubble execution up to parent \texttt{execute( s.parent, s.parent.tt );}
4. Else determine next task and what time to execute it
5. sn = s.getNextTaskToExecute( );
6. Calculate the difference between current time and desired target time for task sn, \(t_{diff} = t_{n} - \text{curr_time}\);
7. \(t_{diff} \in [s_{n}.lb, s_{n}.ub]\) // if the greedy time is ok
8. \(\text{greedy}_tt_{n} = t_{diff}\);
9. \(\text{else} bound t_{diff} \text{ frm above/below by } [s_{n}.lb, s_{n}.ub]\)
10. \(\text{if} t_{diff} \geq s_{n}.ub, \text{ greedy}_tt_{n} = s_{n}.ub\)
11. \(\text{else} t_{diff} < s_{n}.lb, \text{ greedy}_tt_{n} = s_{n}.lb\)
12. \texttt{execute( s.n, greedy_tt_n );}
```

```cpp
function p_array = propagateSchedulingInfo( )
1. First, recursively get scheduling info from children.
2. for each \(t_{target} \in \{s.lb, s.ub\}\)
3. \(\text{tt} = \text{Calculate the desired target times for each task.}\)
4. for \(a = 1:N_{1}\) // latest non-zero sample in task1
5. \(p_1 = \text{child_array}(a, tt_1)\) // a bin from task 1
6. for \(b = 1:N_{2}\) // latest non-zero sample in task2
7. \(\text{Calculate difference between } tt_{1} \text{ and } a\) (the simulated current time) \(t_{diff} = tt_{1}-\text{curr_time}\);
8. \(\text{Bound greedy}_tt_{1} \text{ as done in the execute( ) fcn.}\)
9. \(p_2 = \text{child_array2}(\text{greedy}_tt_{2}, b)\) //bin frm task b
10. \(p_{array}(t_{target}, a+b) ++ p_1*p_2;\)
11. return p_array
```

![Figure 7. Visual depiction of the distribution over actual end times for each target time of an RMPL sequence.](image-url)
If-Else. Analogously to the sequence statement, the if-else statement will be pointed between its lower and upper bounds, $lb^-$ and $ub^+$. Consider the RMPL fragment:

\[
\text{if}(c) \{ \text{task1}(1,5) \} \text{ else } \{ \text{task2}(6,10) \}
\]

We construct a local execution strategy, \text{execute}(), which attempts to greedily meet the desired target time, $t_{target} \in [lb^-, ub^+] = [1,10]$. For brevity, we forgo the pseudocode and give intuitive explanations of each function in the remaining sections.

Notice that if we choose a late $t_{target}$ and $c$ is true at runtime, or vice versa, the greedy strategy is limited to choosing an actual execution time of 5 or 6, respectively. By assuming a probability, $p$, that $c = \text{true}$ at runtime, we can compute an aggregate probability array for the if-else state which represents the probability distribution over likely end states for each of the if-else state’s target times, $t_{target}$. The resulting probability array is analogous to the one constructed for both the primitive task and sequence state, and similarly, can be used by its parent HCA state to make scheduling decisions. In Figure 8, a visual depiction is shown of the probability array that results from running \text{propagateSchedulingInfo}() on the RMPL if-else fragment.

Choose. In the choose state, a discrete choice must be made between sub-programs as well as choosing a target completion time. Consider the RMPL fragment, choose \{ go1(1,5), go2(6,10) \}. The choose operator can be a significant source of complexity due to the combinatorial nature of making compound choices. Keeping with our theme of pointing each state to a target completion time, we construct a local execution strategy, \text{execute}(), which greedily makes the discrete choice which maximizes the likelihood of success for a given $t_{target}$. This results in a discrete choice being coupled to each choice of $t_{target} \in [lb^-, ub^+] = [1,10]$. This is visually depicted in Figure 9, where red represents the choice to execute task $go1$ while green represents the choice to execute task $go2$.

\text{propagateSchedulingInfo}() simply applies a superposition.

Lb-Ub. The lb-ub state constrains the allowable executions of an RMPL program. This state is used to reign in, direct, or place boundaries on the flexibility of an RMPL program. Consider the RMPL fragment, \{ [5,10] \{ task1(1,15) \} \}. Task1 is capable of executing at any time in $[1,15]$. In this program fragment, however, task1 is encompassed by a more restrictive timing requirement of $[5,10]$. The \text{execute}() and \text{propagateSchedulingInfo}() functions are trivial for this state, simply pruning allowable target times to be within the restricted time window. Pruned probability mass represents program executions which violate program constraints.

Try-Catch. The try-catch state enables RMPL programs to recover from anticipated runtime constraint violations. The local execution strategy, \text{execute}() and \text{propagateSchedulingInfo}() are similar to the if-else state. Consider the RMPL fragment, try\{nom_task(1,5)\} catch\{recovery_task(6,10)\}. If the nominal task fails, the recovery task takes at least 6 seconds to complete. This causes the aggregated probability distribution over end times for the try-catch state to have a tail skewed in the direction of longer execution times. We must assume an exception probability and a distribution over exception occurrence times. By default we assume an exception probability of 0.1 and a uniform distribution over times.

Parallel. Each child state in a parallel state must assume the same target time, thus synchronizing the tasks. Each task must remain in execution until all tasks can transition.
Until. The until state can be considered syntactic sugar for the other RMPL constructs. For example, the following RMPL program is equivalent to: until(c,3){task1(1,5)}.

if(c){ [0,0] }  
else { sequence{ task1(1,5) , if(c){ [0,0] } } }  
else { sequence{ task1(1,5) ,if(c){[0,0]}  
else { task1(1,5) } } }  

Maximum Likelihood of Success. The maximum likelihood of success prior to executing an HCA is computed as follows, where \(s_0\) is the root state of an HCA:

\[
\max \left\{ p : p = \sum_{t=1}^{n} s_0.t_{\text{target}}(t).p_{\text{dist}}(t), t \in [s_0.\text{lb}, s_0.\text{ub}] \right\}
\]

The execution strategy developed in this paper is not allowed to change a \(t_{\text{target}}\) after it has been selected. Thus, the likelihood of successful program execution at runtime is:

\[
\left\{ p : p = \sum_{t=1}^{n} s_0.t_{\text{target}}(t).p_{\text{dist}}(t), t = s_0.t_{\text{target}} \right\}
\]

As tasks execute, propagateSchedulingInfo() updates \(p\).

Results

We constructed 240 programs for the experiments that follow. We focus our experimental results on two topics:

1. Clocking the execute() functions. To prove that our dynamic execution algorithm can perform at reactive timescales we need to show that the execute() function for each HCA state assigns target times/choices at reactive timescales. This should be expected, since each HCA state type employs a local execution strategy in-which all constraint reasoning, except for simple differencing, is performed offline. We recorded the maximum time taken by the execute( ) function for each state type over all 240 test programs. The results are tabulated below:

- Lb-ub: 3.07x10^-4, Primitive task : 1.3x10^-3
- If-else: 2.01x10^-4, Try-catch state: 4.81x10^-4
- Choose: 2.02x10^-4, Sequence state: 3.18x10^-4

We see that each state type takes fractions of a second to execute. The primitive task takes slightly longer because it has to make an external function call when starting a task.

2. Analyzing the propagateSchedulingInfo() function. The propagateSchedulingInfo() function is spawned in its own thread, so it doesn’t have to be strictly as fast as the execute() algorithm at runtime to ensure reactive execution. However, an accurate estimate on likelihood of success will lag by the running time of this function (up to 2 seconds as shown in Figure 11a). In Figure 11b we see that the propagate function footprint is linear in the number of HCA states and is on the order of kilobytes.

Discussion

Why does this algorithm dynamically schedule program timing, but not in the spatial dimensions? The explicit controllers underlying our primitive task definition (Definition 1) can be constructed more tractably under this assumption, and the resulting dynamic execution algorithm (Algorithm 1) must reason over less scheduling info at runtime. Extending the algorithm to point dynamically in the spatial dimensions would be interesting future work.

How often does this algorithm fail to find an execution strategy when one does exist? We haven’t tested the algorithm thoroughly enough to make predictions on the types or percentage of problems not solvable using the algorithm. A worthwhile future research direction is to investigate versions of the execute() algorithm which perform limited amounts of constraint reasoning online as time permits. This would enable the algorithm to perform a more complete search over a program’s schedule space.

References


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


