Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2018-006 February 1, 2012

Risk-minimizing program execution in robotic domains

Robert Effinger

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Risk-minimizing Program Execution in Robotic Domains

by

Robert T. Effinger IV

B.Sc., Texas A&M University (2003)
S.M., Massachusetts Institute of Technology (2006)

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

DOCTOR OF SCIENCE
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2012

Author...
Department of Aeronautics and Astronautics
February 2, 2012

Certified by.. ....
Brian C. Williams
Thesis Supervisor, Professor of Aeronautics and Astronautics

Certified by................................
Andreas Hofmann
Vice President of Autonomous Systems, Vecna Technologies

Certified by... ....
Emilio Frazzoli
Associate Professor of Aeronautics and Astronautics

Certified by......................
Lauren Kessler
Principal Member of the Technical Staff, Draper Laboratory

Accepted by...
Prof. Eytan H. Modiano
Professor of Aeronautics and Astronautics
Chair, Graduate Program Committee

Risk-minimizing Program Execution in Robotic Domains

by

Robert T. Effinger IV

Submitted to the Department of Aeronautics and Astronautics
on February 2, 2012, in partial fulfillment of the
requirements for the Degree of Doctor of Science in
Aeronautics and Astronautics

Abstract

In this thesis, we argue that autonomous robots operating in hostile and uncertain environments can improve robustness by computing and reasoning explicitly about risk. Autonomous robots with a keen sensitivity to risk can be trusted with critical missions, such as exploring deep space and assisting on the battlefield. We introduce a novel, risk-minimizing approach to program execution that utilizes program flexibility and estimation of risk in order to make runtime decisions that minimize the probability of program failure.

Our risk-minimizing executive, called Murphy, utilizes two forms of program flexibility, 1) flexible scheduling of activity timing, and 2) redundant choice between subprocedures, in order to minimize two forms of program risk, 1) exceptions arising from activity failures, and 2) exceptions arising from timing constraint violations in a program. Murphy takes two inputs, a program written in a nondeterministic variant of the Reactive Model-based Programming Language (RMPL) and a set of stochastic activity failure models, one for each activity in a program, and computes two outputs, a risk-minimizing decision policy and value function. The decision policy informs Murphy which decisions to make at runtime in order to minimize risk, while the value function quantifies risk. In order to execute with low latency, Murphy computes the decision policy and value function offline, as a compilation step prior to program execution.

In this thesis, we develop three approaches to RMPL program execution. First, we develop an approach that is guaranteed to minimize risk. For this approach, we reason probabilistically about risk by framing program execution as a Markov Decision Process (MDP). Next, we develop an approach that avoids risk altogether. For this approach, we frame program execution as a novel form of constraint-based temporal reasoning. Finally, we develop an execution approach that trades optimality in risk avoidance for tractability. For this approach, we leverage prior work in hierarchical decomposition of MDPs in order to mitigate complexity.

We benchmark the tractability of each approach on a set of representative RMPL programs, and we demonstrate the applicability of the approach on a humanoid robot simulator.

Thesis Supervisor: Brian C. Williams
Title: Professor of Aeronautics and Astronautics

Acknowledgements

First and foremost, I would like to thank my wife, Veronica, for being my best friend and most devout supporter throughout my graduate program. From my first winter in Boston to this one, your presence is the fire that warms my heart. I love you! I truly appreciate your excitement and interest in my successes, as well as your uncanny ability to give sound advice and guidance on any topic. Veronica, we are finally done! I can't wait to see what adventure we embark upon next.

To my family, I sincerely thank you for your unwavering support. You have always been there for me. Thank you for teaching me the importance of education, both in individual pursuit and in outreach to others. Thank you for teaching me the value of hard work, and for infusing within me a love of science and nature. I can't wait to come home for a visit!

I would like to thank my advisor, Brian, for his significant investments of time and energy over these past 8 years to mold me into a more confident and mature researcher and writer. I am grateful for your deep reservoir of patience, the depths of which I surely tested, in completing this daunting task. I appreciate your ability to think and operate 20 years ahead of current practice, and to shape a research agenda that promises to greatly impact the fields of autonomy and robotics. I am excited to see your vision unfold in the coming years. Thank you, Brian, for always believing in me, and for continuously pushing me to do my best.

To my thesis committee and readers, thank you for your combined efforts in guiding my research, and in transforming my dissertation into a coherent, technical document. To Andreas, thank you for sharing with me your expertise at the interface of dynamical systems and autonomy: a truly fascinating topic. To Lauren, thank you for helping me focus on the bigger picture, teaching me to adhere to a schedule, and for offering a listening ear. To Emilio, thank you for your technical critique of my dissertation, and for your prior work on Maneuver Automata. To my thesis readers, Mitch and Paul, thank you for taking the brunt of my rougher drafts, and for providing feedback on short notice. Also, thank you for your prior work in the specification, compilation, and execution of model-based programs.

To my fellow members of the MERS group, thank you for being a constant source of inspiration and fun. I fear that wherever I go next, my colleagues will pale in comparison. Most importantly I would like to thank Seung Chung, Jon Kennel, Ishiang Shu, and Paul Elliott, each of which took considerable time out of their own schedules to help me get up to speed in the MERS group: I couldn't have done it without you! Also, I would like to thank Larry Bush, Lars Blackmore, and Patrick Conrad for important technical discussions which have helped to shape my thinking and research directions.

I would also like to extend thanks to my colleagues at Draper Laboratory. To Bob Brown and Dino DiBiaso, thank you for helping me learn the Timeliner scripting language, and for

giving me a glimpse into how aerospace and embedded systems are designed and built in practice. I would like to thank Lance Page and John West for their work on the Timeliner-Questus integration project: a fun and challenging project during my fellowship at Draper.

Finally, this thesis was sponsored by Draper Laboratory, through NASA Johnson Space Center under Draper contract number NNJ11HA31C.

Table of Contents

Table of Figures

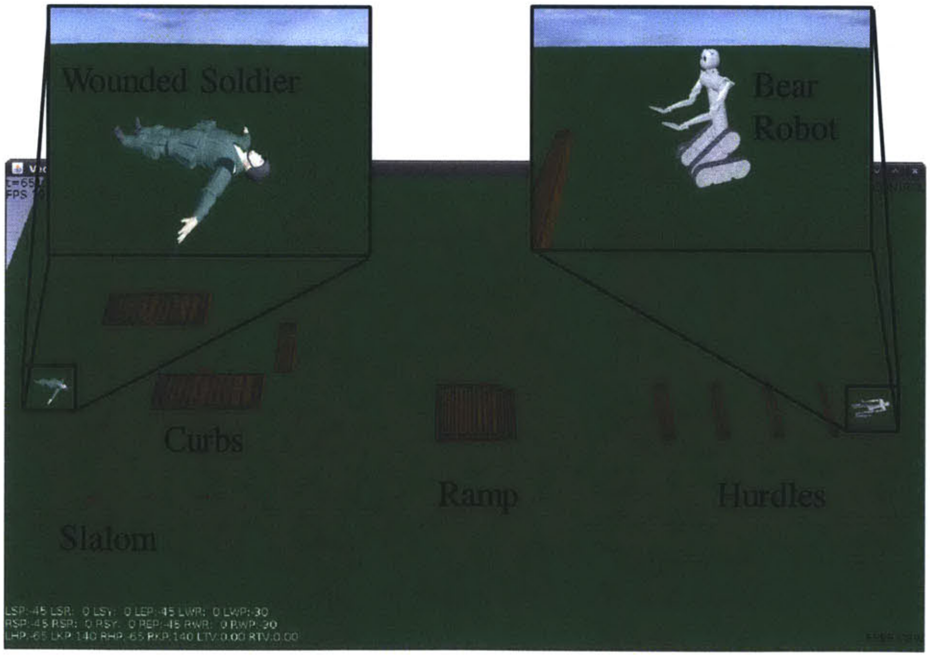
Chapter 1 – Introduction

The field of autonomous robotics promises to increase human safety and capability by extending our physical presence into remote and dangerous places. For example, NASA envisions autonomous robots exploring deep space [Ambrose et al., 2000], while the Army envisions humanoid robots rescuing wounded soldiers from the battlefield [Atwood and Klein, 2007]. These robots are expected to operate in hostile and uncertain environments, and to continuously strive to achieve their mission goals, despite inevitable task failures and equipment malfunctions. In addition, they are expected to be self-aware and capable of monitoring their own progress towards the achievement of mission goals. When an important goal is likely to be missed, such as failing to reach a wounded soldier within a deadline, the robot must realize it and ask for help.

In this thesis, we argue that in order to delegate great responsibility to autonomous robots, we must endow them with a keen sense of risk awareness. Risk awareness implies the capacity to measure and analyze the risk inherent in executing a control program, and then to execute a control program in such a way as to minimize risk. To this end, in this thesis we explore risk-minimizing program execution in robotic domains. We define *risk* as the probability that a control program will fail to execute to completion, and we frame our approach as a risk-minimizing variant of the Reactive Model-based Programming Language (RMPL) [Williams et al., 2003], which supports four key features, namely 1) flexible scheduling of activities, 2) choice between redundant subprocedures, 3) constraints on program timing, and 4) probabilistic failure models for activities. The main focus of this thesis is to develop execution algorithms that compute program risk and then utilize program flexibility to minimize risk at runtime. In the next section, we introduce a simple motivating example to ground the idea of risk-minimizing program execution.

1.1 A Motivating Example

Consider a humanoid Battlefield Extraction-Assist Robot (BEAR) [Atwood and Klein, 2007], tasked with rescuing a wounded soldier in a hostile environment. The robot must traverse an environment with multiple obstacles, shown in Figure 1.1, in order to reach the wounded soldier. Due to the proximity of enemy forces, it is desirable for the robot to attempt the rescue autonomously in order to prevent additional casualties. If it becomes apparent that the robot will be unable to reach the wounded soldier in 4 minutes, however, additional soldiers should be dispatched to assist in the rescue. This example highlights the need for an autonomous robot that is capable of monitoring its own progress towards the achievement of a mission goal, "to reach the wounded soldier within 4 minutes". To be trusted with this important mission, the robot must continually estimate its risk of failure, so that it can alert the commanding officer as soon as success becomes unlikely.



```
LSP:-45 LSR:  0 LSY:  0 LEP:-45 LWR:  0 LWP:-30
RSP:-45 RSP:  0 RSY:  0 REP:-45 RWR:  0 RWP:-30
LHP:-65 LKP:140 RHP:-65 RKP:140 LTV:0.00 RTV:0.00
```

Figure 1.1: Autonomous rescue of a wounded soldier.

In this section, we introduce risk-minimizing program execution in several steps. First, we demonstrate how the motivating example can be encoded into RMPL. Second, we show how the programmer specifies flexible scheduling and probabilistic failure for each activity in an RMPL program. Finally, we define program *risk* and explain how our risk-minimizing program executive, Murphy, utilizes the program flexibility and probabilistic failure models to dynamically select a program execution that minimizes risk.

The motivating example is encoded in RMPL in Figure 1.2. In order to rescue the wounded soldier, the robot must traverse a series of obstacles. Each obstacle highlights an important feature of the RMPL language. The first obstacle highlights exception handling; if the robot falls while traversing the hurdles, an exception is thrown and the robot must recover. The next obstacle demonstrates sensing; the robot traverses the ramp only if it is clear at runtime. The final two obstacles demonstrate redundant choice among subprocedures; the robot may choose to traverse either the slalom or curbs in order to reach the wounded soldier. Finally, we encode the mission goal, "to reach the wounded soldier within 4 minutes" as a constraint on program timing. Note the metric timing constraint, [0,240]{}, encapsulating the program in

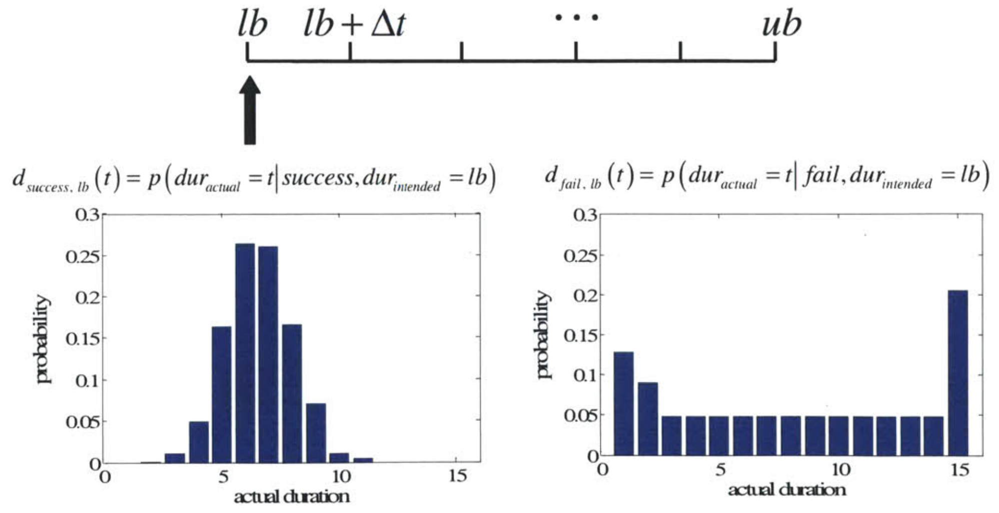
```
Wounded Soldier Program:

[0,240]{
  sequence{
    try{ hurdles() }
      catch{ hurdles_recovery() }
    if(clear){ramp()}
    choose{ slalom() , curbs() }
  }
}
```

Figure 1.2: Wounded soldier example RMPL program.

Figure 1.2. This constraint enforces an upper bound on program execution of 240 seconds. In general, a metric timing constraint, denoted [lb,ub], enforces minimum and maximum durations, *lb* and *ub* respectively, on an encapsulated subprogram. When a timing constraint is violated, the program executive throws an exception.

Next, we demonstrate how the programmer specifies flexible scheduling and probabilistic failure for each activity via a *stochastic activity model*. In Figure 1.3, we show a stochastic activity model for the hurdles activity. First, a lower and upper bound are defined, lb and ub respectively, as well as a time increment, Δt, which together indicate allowable target durations for activity scheduling, $\{lb, lb+\Delta t, \cdots, ub\}$. For each target duration, a probability distribution over success and failure as a function of time is provided. The success and failure distributions for the hurdles activity with the target duration of 45 seconds are shown in Figure 1.3. Our risk-minimizing executive reasons probabilistically from the stochastic activity models in order to make scheduling decisions at runtime that minimize risk. The stochastic activity model is defined formally in Chapter 4.



$d_{success,\,lb}(t) = p\left(dur_{actual} = t \middle| success, dur_{intended} = lb\right)$

$d_{fail,\,lb}(t) = p\left(dur_{actual} = t \middle| fail, dur_{intended} = lb\right)$

Figure 1.3: Stochastic activity model for the hurdles activity.

Finally, we define program *risk* and we explain how our risk-minimizing program executive, Murphy, exploits program flexibility and the probabilistic failure models to minimize risk. In this thesis, we define *risk* as the probability that an uncaught exception will occur at runtime. Note that this interpretation allows for exceptions to occur, as long as they are caught and handled by an exception handler. In the variant of RMPL defined in this thesis, an exception originates from one of two places, either from a failed activity or from a violated timing constraint. Alternatively, a program running to completion with no uncaught exceptions is interpreted as program success.

We choose this definition for risk with an important goal in mind. We wish to develop an executive that computes risk, even through exceptions, and that utilizes program flexibility to minimize risk. This concept is essential to improving robustness in unstructured robotic domains where exceptions are inevitable, and it represents an important departure from prior work in temporally-flexible planning, in-which any exception results in the loss of all timing guarantees. In summary, our definition of risk is chosen in order to isolate and study the following problem, "How can we best utilize program flexibility in order to minimize the risk of program failure that is incurred by runtime exceptions?"

The scope of this thesis is limited to two forms of program flexibility, 1) flexible scheduling of activity timing, and 2) redundant choice between subprocedures. This thesis is also restricted to minimizing two forms of risk, 1) exceptions arising from activity failures, and 2) exceptions arising from timing constraint violations. There are many other types of program flexibility and risk that are beyond the scope of this thesis. For example, a programming language might offer flexibility in allocating activities to robots, or an executive could improve robustness by minimizing the risk of depleting limited resources, such as fuel or supplies.

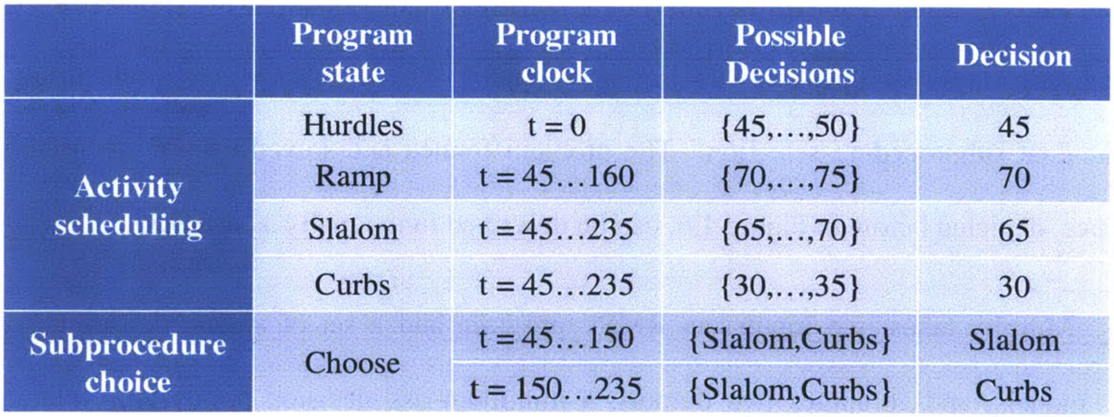
Next, we introduce our risk-minimizing program executive, called Murphy. Here we focus on describing the executive functionally, by describing its inputs and outputs. We postpone a discussion of the internal workings of the executive until the Problem Approach section (Section 1.4).

Murphy takes as input an RMPL program and a set of stochastic activity models, one for each activity in the program. Murphy then performs a compilation step in which it reasons over the RMPL program and the stochastic activity models in order to construct two outputs, a *minimum-risk decision policy* and a *value function.* The minimum-risk decision policy informs Murphy at runtime which scheduling decisions to make and which subprocedures to select in order to minimize risk. The value function computes the amount of risk in each program state as a program executes.

To give a concrete example, consider the minimum-risk decision policy for our motivating example, presented in Figure 1.4. A decision policy makes two types of decisions, 1) scheduling an activity's duration and 2) choosing between redundant subprocedures. Murphy schedules an activity's duration just prior to its execution by reading from the decision policy at runtime. In general, this duration may vary depending on the current program state and clock. However, in this motivating example, the scheduling decisions are invariant to program state and program clock. The tight deadline on program execution enforces a minimum-risk strategy of always choosing the shortest possible duration for each activity.

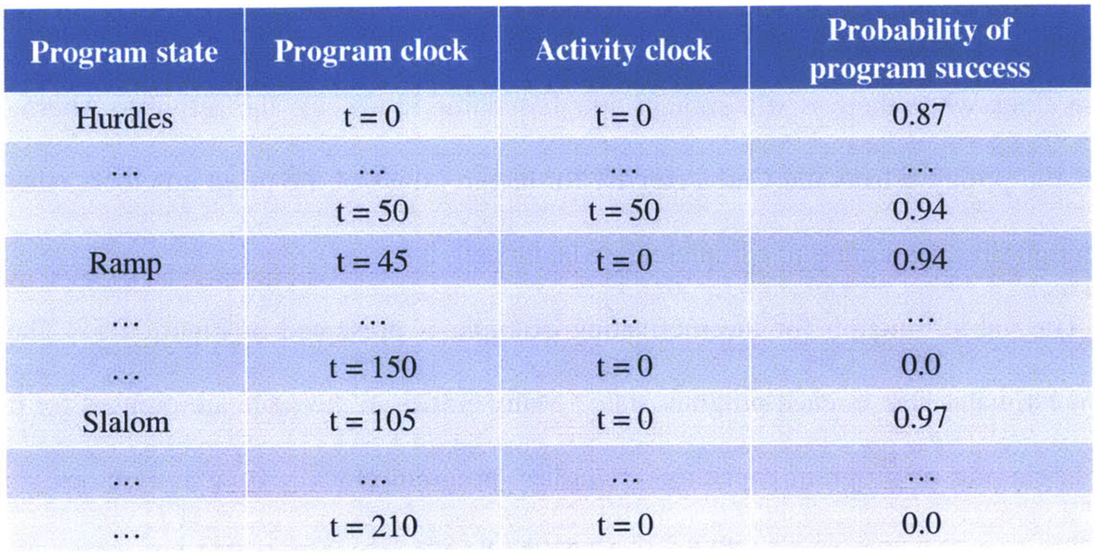
Murphy also chooses between redundant subprocedures at runtime by consulting the decision policy. Each time a *choose* construct is encountered, the decision policy informs Murphy which subprocedure to execute as a function of the current program state and clock in order to minimize risk. In this example, the Slalom activity incurs less risk but takes more time to execute. Therefore, the decision policy selects the Slalom activity at small values of the program clock when there is still enough time remaining to execute the activity. Alternatively, the decision policy selects the Curbs activity for larger values of the program clock, when there isn't enough time remaining to complete the Slalom activity.

The value function for the motivating example is presented in Figure 1.5. The value function estimates risk in each program state. Many entries in the table are omitted for brevity. At the beginning of program execution, {Hurdles, program clock = 0, activity clock = 0} the probability of program success is 87 percent. As the hurdle activity finishes execution, {Hurdles, program clock = 50, activity clock = 50}, the probability of success jumps to 94 percent. This is because the executive has learned at runtime that no exceptions were thrown during the Hurdles activity, thus eliminating some of the risk that existed before the program started execution. Finally, if the Hurdles activity completes execution late, after 150 seconds, and the ramp is taken at runtime, denoted by {Ramp, program clock = 150, activity clock = 0}, the probability of program success (meeting the deadline) is zero. We see that the executive should halt execution instead of executing the program further with no chance of success.



	Program state	Program clock	Possible Decisions	Decision
Activity scheduling	Hurdles	t = 0	{45,...,50}	45
	Ramp	t = 45...160	{70,...,75}	70
	Slalom	t = 45...235	{65,...,70}	65
	Curbs	t = 45...235	{30,...,35}	30
Subprocedure choice	Choose	t = 45...150	{Slalom,Curbs}	Slalom
		t = 150...235	{Slalom,Curbs}	Curbs

Figure 1.4: Risk-minimizing decision policy for the motivating example



Program state	Program clock	Activity clock	Probability of program success
Hurdles	t = 0	t = 0	0.87
…	…	…	…
	t = 50	t = 50	0.94
Ramp	t = 45	t = 0	0.94
…	…	…	…
…	t = 150	t = 0	0.0
Slalom	t = 105	t = 0	0.97
…	…	…	…
…	t = 210	t = 0	0.0

Figure 1.5: Value function for the motivating example

1.2 Problem Statement

In this thesis, we argue that autonomous robots can improve robustness when operating in harsh and uncertain environments by computing risk at runtime, and by utilizing program flexibility in order to minimize risk. To this end, we design a risk-minimizing executive, called Murphy, which minimizes two sources of program risk, namely 1) exceptions arising from activity failures, and 2) exceptions arising from timing constraint violations. To minimize risk, Murphy exploits two types of program flexibility, namely, 1) flexible activity scheduling, and 2) redundant subprocedure selection. The problem framed and solved by our risk-minimizing executive, depicted below in Figure 1.6, can be described functionally as follows:

> "Murphy takes two inputs, an RMPL program and a set of stochastic activity models, and computes two outputs, a minimum-risk decision policy and value function, which are used during program execution to minimize risk".

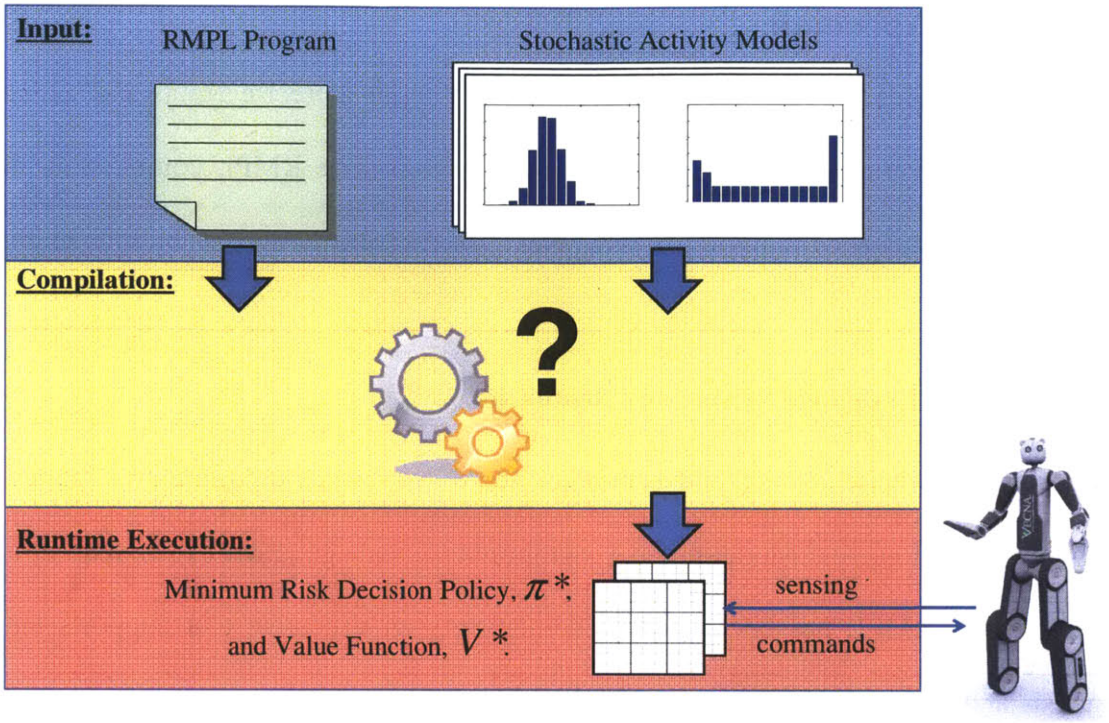

Input:
RMPL Program
Stochastic Activity Models
Compilation:
?
Runtime Execution:
Minimum Risk Decision Policy, π^*,
and Value Function, V^*.
sensing
commands

Figure 1.6: Visual depiction of the problem statement.

An example of each input and output was presented in the previous section, and a formal definition of each is provided in Chapter 3, Language Syntax and Semantics.

Next we discuss important properties of the outputs subject to the inputs, as well as important properties of the function that maps inputs to outputs. The outputs, namely, a decision policy and a value function, must be queried quickly at runtime, ideally within the reactive control loop of an autonomous robot (~10 Hz), so they must scale well in the size of the inputs. In addition, the computation time from inputs to outputs is important in applications where an autonomous robot must author and compile its own control programs as it operates. Finally, we identify an important tradeoff, which is optimality in risk avoidance versus scalability of the outputs and compilation time. For example, in some applications, it may be beneficial to sacrifice optimality in risk avoidance in return for a decision policy and compilation time that fit within specific size and time requirements.

1.3 Problem Approach

Our approach makes two key contributions. First, we define a precise syntax and semantics for a risk-minimizing variant of RMPL. Second, we develop three RMPL compilation algorithms, each of which is designed to minimize risk. Next, we describe these contributions in more detail.

Contribution 1: Language Syntax and Semantics

We develop a precise syntax and semantics for a risk-aware programming language as an extension to the Reactive Model-based Programming Language (RMPL) [Ingham et al., 2001]. In prior work, RMPL has been used to demonstrate an executive that supports autonomous diagnosis and repair [Williams et al. 2003; Williams et al., 2004], robust control of dynamical systems [Hofmann and Williams, 2006], and model-learning [Blackmore et al., 2007]. The RMPL language builds upon previous work on robotic execution languages, such as RAPS [Firby, 1987], ESL [Gat, 1996], and TDL [Simmons, 1998], and is distinguished by its emphasis on incorporating model-based constraint reasoning within the reactive control loop of embedded and robotic systems.

Next, we elaborate on three key features of our risk-minimizing variant of RMPL, a) it incorporates program flexibility in the form of runtime activity scheduling and subprocedure selection, b) it subsumes traditional embedded programming constructs, and c) it has a simple execution semantics in terms of hierarchical timed automata.

Flexibility in program execution:

We incorporate program flexibility in two forms, 1) activities have flexible windows on possible duration, from which a target duration is chosen at the start of activity execution, and 2) the *choose{}* language construct, from which exactly one subprogram is chosen for execution when encountered at runtime. The introduction of flexibility into a program is inspired by prior work in nondeterministic programming languages, such as A-Lisp [Andre and Russel, 2002], DT-Golog [Boutilier et al., 2000], and RAPs [Firby, 1987].

Delaying decisions until runtime presents an executive with a valuable opportunity to improve robustness. The reason being that information is learned continually as a program executes, which reduces uncertainty that existed during the beginning of program execution. Is the program running ahead or behind schedule based on current environmental conditions? Have exceptions occurred during program execution delaying progress and increasing the risk of violating a program's timing constraints? These questions can be answered at runtime, and enable a flexible executive to compensate for disturbances.

To identify a program execution that minimizes the risk of failure, our executive uses a stochastic model of each activity to evaluate the success probabilities of candidate executions. Each *stochastic activity model* characterizes the timing behavior of an activity probabilistically, under failure and success, over a range of target activity durations. An example is provided in Section 1.2. Our stochastic activity model builds upon a rich heritage of prior work in temporal constraint modeling, namely the Temporal Constraint Network (TCN) [Dechter et al., 1991], the Conditional Temporal Problem (CTP) [Tsamardinos et al., 2003], the Simple Temporal Problem with Uncertainty (STNU) [Vidal and Fargier, 1999], and the Temporal Plan Network (TPN) [Kim et al., 2001].

<u>Subsumes traditional embedded programming constructs:</u>

To be useful in robotic domains, it is important that our risk-aware programming language includes traditional embedded programming constructs, such as: sequential execution (sequence), parallel execution (multi-threading), exception handling (try-catch), and conditional execution (if-else). A key contribution of this thesis is the inclusion of exception handling and sensing into the RMPL language. Sensing is crucial when operating autonomously in an unstructured environment, while exception handling enables graceful recovery from failures by localizing error identification and recovery, rather than halting the entire executive, i.e. "safing".

A simple execution semantics in terms of hierarchical timed automata:

Execution semantics describe the behavior that a computer should follow when executing a program in the language [Slonneger 1995]. We describe RMPL execution semantics in terms of the execution of hierarchical timed automata (HTA). Timed automata [Alur and Dill, 1994] offer an intuitive and precise execution model, while allowing for the explicit representation of time. The semantics developed in this thesis draw upon prior work on automata-based semantics such as the synchronous languages [Halbwachs 1993, Harel 1987], probabilistic model-checking [Parker, 2002], and verification of timed systems [Pettersson 1999, Ingham 2003].

Contribution 2: Three RMPL Compilation Algorithms that Minimize Risk

In our pursuit of a risk-minimizing executive for RMPL programs, we have developed three RMPL compilation algorithms, ranging from exact to approximate, and we have explored two types of activity model, set-bounded and stochastic. Next, we summarize each of the algorithms, a) Minimum-risk Program Execution, b) Set-bounded Dynamic Execution, and c) Risk-minimizing Execution with Limited Coupling. We discuss the relative strengths and weaknesses of each algorithm, and we discuss the types of programs for which each algorithm is best suited.

Minimum-risk Program Execution (Chapter 4):

The goal of this algorithm is to provide a risk minimizing executive that generates an exact, optimal solution, while executing with low latency. The algorithm functions by formulating an HTA and a set of stochastic activity models into a Markov Decision Process (MDP). The MDP is then solved using Value Iteration, which results in an optimal decision policy and value function. The resulting value function measures risk at each state in a program, while the optimal decision policy informs the executive at runtime which decisions to make in order to minimize risk. Next, we describe each component of the MDP in more detail. Then, we discuss the strengths and weaknesses of the exact algorithm, and its general applicability in robotic domains.

The MDP transition function is constructed from the HTA by simulating all possible observations, control actions, and resulting program states, from each state in the HTA. The probability associated with each transition in the MDP is calculated from the stochastic activity models that are provided as input. Transitions ending in successful program termination are assigned a reward of one, while all other transitions, including those leading to the program uncaught exception state, are assigned a reward of zero. This reward function results in a value function that measures the probability of success, or inversely, 1 - the probability of program failure. The actions in the MDP represent the decisions that an executive makes when scheduling activity durations and choosing between redundant subprocedures in the HTA. Value Iteration solves the MDP by finding the decision policy that optimizes the value function, or in other words, finds the decisions that minimize execution risk at every point in a program where a decision is made.

The primary strength of the minimum-risk algorithm is that it finds the optimal solution:

the minimum risk decision policy. Also, the decision policy and value function are computed explicitly, in tabular form, ensuring low execution latency when they are queried at runtime. The weakness of this approach is its computational complexity. An analysis performed at the beginning of Chapter 6 shows that the complexity of the MDP is worst-case exponential, meaning that the number of states and transitions in the MDP is exponentially greater than the number of activities in the RMPL program. The poor scaling from inputs to outputs means that this algorithm is only acceptable for small programs, for example, up to 30 lines in length. For larger programs, the compilation time for this algorithm can take on the order of hours, and the resulting decision policy becomes too large to store and query in real-time given the computing and storage limitations on conventional robotic systems.

Set-bounded Dynamic Execution (Chapter 5):

The goal of this algorithm is to produce a fast execution algorithm that doesn't require an exact probabilistic model to be specified for each activity. The algorithm achieves greater efficiency by operating on an abstracted model of uncertainty and by shifting from a probabilistic to a constraint formulation. First, we discuss prior work leading to the development of this algorithm. Then we discuss the algorithm in more detail. Finally, we discuss the algorithm's strengths and weaknesses, and its general applicability in robotic domains.

The inspiration for this algorithm comes from a body of work in temporal planning called *dynamic controllability*. Dynamic controllability guarantees successful execution of a network of set-bounded activities [Morris et al., 2001, Tsamardinos et, al., 1998]. A set-bounded activity specifies a range on allowable activity duration, denoted [lb,ub], and which is interpreted by the executive as a metric timing constraint, $\left(lb \leq t_{end} - t_{start} \leq ub\right)$. The set-bounded activity model purposefully omits probability information, however, so we don't know where within the allowable range an activity is likely to complete. This makes proving dynamic controllability an all or nothing proposition; either a constraint network is provably dynamically controllable, or no guarantee is made at all. Also, dynamic controllability, as it was originally developed, is not capable of representing conditional branches, exception handling, or choice in subprocedures; three of the programming constructs we wish to support.

To add support for these constructs, in Chapter 5, we develop a new constraint formalism, called a Temporal Plan Network under Uncertainty (TPNU), with graphical nodes capable of representing subprocedure choice, sensing, and exception handling. Then, we frame dynamic controllability of a TPNU as a two-player game between the executive and the environment. An algorithm is developed which frames the dynamic controllability problem as AND/OR search [Dechter and Mateescu, 2007] over candidate program executions. To guarantee successful execution, the algorithm branches generatively as decisions in the program are made, either by the executive or by the environment. 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 AND/OR search tree considers all possible orderings of decisions that are able to be made between the executive and the environment. The original program is guaranteed to be executable if and only if an AND/OR solution subtree exists in the AND/OR search tree. The AND/OR solution subtree serves as a compact, dynamic execution strategy to ensure successful program execution.

The expressivity captured in the TPNU formalism comes with a high computational cost. Searching for AND/OR solution subtrees (a.k.a. the compilation time) takes time exponential in the number of activities in the RMPL program. The size of the resulting solution subtree (a.k.a.

the decision policy) is also exponential in the number of activities in the RMPL program. This algorithm is more efficient than the algorithm presented in Chapter 4, mainly since it is restricted to considering only candidate executions that are guaranteed to succeed, which rules out many of the candidate executions considered by the exact probabilistic algorithm. Practically speaking, however, this is a significant drawback in many robotic domains. Successful execution often can't be guaranteed, for example, ruling out all available candidate executions in our motivating example. In summary, the set-bounded execution algorithm is more efficient than the exact probabilistic algorithm (Chapter 4), but its complexity is still exponential in both compilation time and memory. In addition, while a set-bounded execution approach requires less information up front, it is ill-suited for applications in-which successful execution cannot be guaranteed.

Risk-minimizing Execution with Limited Coupling (Chapter 6):

The goal of this algorithm is to provide an executive that trades optimality in risk minimization for tractability in the resulting decision policy and value function. The approaches in Chapters 4 and 5 provide strong guarantees on risk-minimizing behavior, but they do nothing to mitigate the complexity of the resulting execution strategy. In Chapter 6, we identify the primary source of complexity in the prior two approaches as a state-space explosion associated with activities that are executing in parallel. We address this explosion by factoring the state space according to the hierarchy imposed by the HTA, and by limiting the coupling between parallel automata to coordinating the target execution times at each level of the hierarchy.

This approach eliminates the primary computational bottleneck of the exact approaches. However, some flexibility is lost. The limited coupling requires that parallel threads decide ahead of time on a shared target duration. The threads cannot decide to change their shared target duration in response to information learned at runtime in order to minimize risk. The resulting value function and decision policy will in general be suboptimal, implying a lower probability of successful program execution than is achievable with the minimum-risk approaches developed in Chapters 4 and 5.

By restricting our execution strategy in this manner, we may be neglecting valid execution strategies that incur lower risk. However, we justify this restriction only neglecting those execution strategies whose complexity exceeds the limits of our application of interest, autonomous robots with limited computational resources and demanding response times.

1.4 Thesis Roadmap

In Chapter 2, we summarize related work. In Chapter 3 we explain the language syntax and semantics. In Chapter 4 we present the minimum-risk execution algorithm. Chapter 5 presents set-bounded dynamic execution, while Chapter 6 presents the risk-minimizing execution algorithm with limited coupling. In Chapter 7 we perform experimental validation of the three algorithms developed in Chapters 4-6. In Chapter 8 we conclude and discuss promising directions for future work.

Chapter 2 – Related Work

This thesis draws upon three broad categories of related work, Nondeterministic Programming, Formal Modeling of Timed Systems, and Markov Decision Processes. In this chapter we briefly summarize the key developments in each field, and discuss their relation to this thesis in particular.

2.1 Nondeterministic Programming

In contrast to deterministic (i.e., procedural, imperative) programming languages such as C++ and Java, nondeterministic programming languages allow the specification of choice points in a program, leaving the program to decide at runtime which one of several alternative subprograms to execute. The programmer specifies a nested hierarchy of choices at compile time, and the program uses a reasoning algorithm to decide which choices to implement at runtime.

The overarching similarity between RMPL programs and programs written in other nondeterministic languages is that they can each be viewed as partially specified control programs. A nondeterministic program encodes the goals or otherwise the general behavior that needs to be accomplished, while leaving the lower-level implementation details up to the program at runtime. This approach has two benefits. First, robustness in execution is improved, because the program is able to respond to anomalous or unpredictable conditions at runtime by adjusting the choices that are made at the nondeterministic choice points. Second, tractability in planning is improved because the search for valid program completions is limited to the alternatives provided by the programmer. From this viewpoint, a nondeterministic language can be viewed as an "advice-giver" that provides valuable search control knowledge restricting search to only the alternatives deemed promising by the programmer. Next, we describe three nondeterministic languages, DT-Golog, A-lisp, and RAPs, and discuss how they relate to RMPL.

2.1.1 DT-Golog

The Golog language [Levesque et al., 1997] is a logic programming language developed for applications such as robotics, process control, intelligent agents, and discrete event simulation. Golog employs a first-order language called the situation calculus, comprised of actions, situations, fluents, and axioms, to model the way a system interacts with its environment. A simple description of Golog is as follows:

- All changes to the world state are the result of actions.
- A situation is a possible world history, and is simply a sequence of actions.
- Relational fluents are predicates with a situation as their last argument, *is_home*(Rob, s).
- Functional fluents are functions whose denotations vary from situation to situation, and are denoted by function symbols with the last argument taking a situation term, as in *location*(robot,s).
- Actions have preconditions that limit when they are possible, *Poss*(*pickup*(x),s).
- World dynamics are specified by effect axioms. These describe the effects of a given action on the fluents. For example, a robot dropping a fragile object causes it to be broken: $Poss(drop(r,x),s) \wedge fragile(x,s) \supset broken(x, do(drop(r,x),s))$.

In order to write control programs in Golog, rules are provided for converting basic programming constructs, such as if-else, iteration, and nondeterministic choice into logical expressions in the situation calculus. Then, the job of a Golog interpreter, which is really a general-purpose theorem prover, is to choose possible completions for each nondeterministic choice operator until a program has been fully expanded that is consistent with all axioms. For example, consider a simple mail delivery robot. A program can be specified in Golog which allows for nondeterministic choice between the intermediate locations that are visited along the mail route. The Golog interpreter binds intermediate locations as possible completions for the nondeterministic choice operators in the program, testing the precondition and postcondition axioms to ensure that a feasible "execution" has been expanded. In this example, the precondition and postcondition axioms would specify the start and goal locations, as well as locations that neighbor one another. Many possible "executions" may satisfy the original "specification", or nondeterministic program. It is the job of the interpreter to find and return one such successful execution, or otherwise return failure.

DT-Golog [Boutilier et al., 2000] extends Golog to handle decision-theoretic choice and stochastic outcomes via an interpreter that views optimal completion of a Golog program as a Markov decision process. DT-Golog allows the programmer to sense the state of the world, as in the conditional execution construct, *if(*cond*)else{ },* and is able to estimate the likelihood of program success.

DT-Golog differs from the work in this thesis in that it doesn't support concurrency, exception handling, or timing uncertainty. However, these capabilities are essential for autonomous robots operating in unstructured domains. In this thesis, as well in prior work [Ingham 2003], we argue that the key to incorporating such features into a nondeterministic programming language is the explicit representation and reasoning about time. Therefore, the key difference between DT-Golog and RMPL is the focus on an executive that reasons explicitly about time, and a language syntax and compiler that can support language constructs for concurrency, exception handling, and timing uncertainty. More recent work on DT-Golog [Ferrein et al., 2003] incorporates MDP options [McGovern et al., 1998] to improve tractability in interpreting DT-Golog programs. We develop a similar strategy in Chapter 6 to improve tractability in compiling RMPL programs. Our strategy differs in that it leverages a hierarchical MDP decomposition technique called *policy-based decomposition* [Wang and Mahadevan, 1999], instead of MDP options.

2.1.2 A-Lisp

A-Lisp [Andre and Russel, 2002] was developed to improve the tractability of the reinforcement learning problem for intelligent agents. The reinforcement learning problem consists of an agent, a set of states *S*, and a set of actions per state *A*. The agent receives a reward by taking actions, $Q: S \times A \rightarrow \mathbb{R}$. The problem is then solved via an algorithm called Q-learning [Watkins, 1989], which employs a stochastic asynchronous update step in order to compute an optimal decision policy. The drawback to this standard approach is that the number of states *S* grows exponential in the number of variables in the problem, even though many of the variables may be irrelevant. Quoting from [Andre and Russel, 2002], "When driving a taxi from A to B, decisions about which street to take should not depend on the current price of tea in china; when changing lanes, the traffic conditions matter but not the name of the street; and so on". A-Lisp is one in a family of partial programming languages proposed to improve tractability in reinforcement learning, including MAXQ [Dietterich, 2000], options [Precup and Sutton, 1998], and PHAM [Andre and Russel, 2001], although A-Lisp subsumes each of the prior languages.

A-Lisp introduces a nondeterministic choice operator and a hierarchical state abstraction operator, to enable hierarchical reinforcement learning in which the state abstraction differs at each level of the hierarchy. Similarly to RMPL, the nondeterministic choice operator in A-Lisp improves tractability by restricting the decisions made by an agent to those explicitly outlined by the programmer. A-Lisp differs form the work in this thesis, however, in that it doesn't support language constructs for expressing concurrency, exception handling, or timing uncertainty. As argued in the previous section, these constructs are essential for autonomous robots operating in unstructured domains, and developing a compiler and executive that supports these features while minimizing execution risk is the key focus of this thesis.

2.1.3 RAPs

RAPs [Firby, 1987] is a reactive planner that takes a purely reactive approach to robotic execution, meaning that the system does not attempt to perform any forward reasoning. Instead, the reactive planner simply maintains a queue of goals, as well as a list of Reactive Action Packages (RAPs) that may be invoked at execution time to achieve goals. As defined in [Firby, 1987], "A RAP is essentially an autonomous process that pursues a single goal until that goal has been achieved. If more than one goal is on the queue at runtime there will be an independent RAP trying to accomplish each one." A RAP consists of multiple strategies for achieving a goal. Once invoked, a RAP will repeatedly try strategies for achieving the goal until one succeeds, or all strategies have been exhausted. If a RAP fails, its goal goes back on the queue.

RAPs supports concurrency, exception handling, and nondeterminism, but does not perform any forward reasoning in order ensure goal satisfaction. This thesis develops a probabilistic look-ahead technique in order to minimize risk during program execution, in addition to supporting concurrency and exception handling.

2.2 Formal Modeling of Timed Systems

Analyzing and guaranteeing the correct timing behavior of safety and mission critical systems has been an active area of research for decades. Nuclear power plants, radiation-dosage chemotherapy machines, commercial airplanes, and even traffic lights are all examples of safety-critical software-controlled systems that operate in our daily lives. The first forays into formally proving timing correctness of such systems involved the addition of clock variables to automata [Alur and Dill, 1990], and then augmenting the state transition guards with metric timing constraints. Safety constraints – posed as invariants in a temporal logic – could then be proven or revoked by a model-checker that reasons over the timed automata [Alur and Dill 1994]. We begin this section by summarizing the recent advances in model-checking relevant to this thesis; primarily in probabilistic model-checking of timed systems.

Next, we move on to the synchronous languages. Synchronous languages build upon timed automata theory to provide formal tools for modeling, specification, and validation of real-time embedded applications [N. Halbwachs, 1993; Benveniste et al. 2003]. The synchronous languages employ a strict notion of concurrency, called the *synchrony hypothesis*, to guarantee fast system response times and error recovery, as well as precisely defined behavior. The synchronous languages do not attempt to perform any sort of look-ahead or temporal planning, however, which is an important aspect of RMPL.

Next, we move on to describe prior work in temporal planning. Temporal planning augments classical planning to account for the timing of planned activities as well as their logical pre- and post-conditions. This thesis builds upon prior work in set-bounded temporal planning, which accounts for the minimum and maximum timing durations that an activity can take to complete. We discuss algorithms that reason over plans consisting of networks of set-bounded timing constraints, in order to guarantee that the timing goals of a plan are satisfied. We finish this section by discussing recent advances in set-bounded temporal planning that account for timing uncertainty, nondeterministic choice, and runtime sensing. We conclude this section by discussing preceding work on Timed Model-based Programming in RMPL.

2.2.1 Probabilistic Model-checking of Timed Systems

Model-checking is concerned with the problem of ensuring whether a model of a mixed hardware/software system meets a certain specification, for example, safety requirements, the absence of deadlocks, or the avoidance of certain failure states. A model checker works by formulating both a model of the system and a specification of desired behavior into a mathematical language, such as propositional logic, and then attempts algorithmically to find a system trace that violates the specification. If all system traces obey the specification, then the specification is said to be satisfied, if a violating system trace is found, then it is returned as a counter-example to the specification.

The model-checking literature faces, and has dealt successfully, with some of the same challenges faced in this thesis. For example, the model-checkers UPPAAL [Pettersson, 1999] and PRISM [Parker, 2002] model metric time and probabilistic systems, respectively, using timed automata and Markov decision processes. They have developed symbolic approaches to dealing with the issue of tractability, and have grappled with interesting verification issues that are not addressed in this thesis, such as deadlocks, extended data-types, and hand-shaking synchronization.

The model-checking literature differs from timed model-based programming, however, in that it focuses on design-time verification rather than optimal runtime control. This thesis is concerned with the discovery of a robust runtime execution strategy to maximize probability of successful execution of a model-based control program. Model-checkers on the other hand, perform offline verification of system properties.

2.2.2 Synchronous Programming

The synchronous languages adhere to a strict notion of concurrency called the *synchrony hypothesis* to ensure that safety-critical systems are capable of responding in real-time, and that system behavior is defined in a mathematically precise way. The synchronous approach originated in France with the sister languages, Esterel [Berry and Gonthier, 1992], Lustre [Halbwachs et al., 1991], and Signal [Guernic, el. al., 1991], and then grew to include other languages such as Statecharts [Harel, 1987], a precursor to the Unified Modeling Language [Hunt, 2000] and Matlab's Stateflow [Hamon and Rushby, 2004].

As expounded in [Ingham, 2003],

> "The primary tenets of the synchronous languages are:
>
> 1) a set of standard constructs necessary for expressing reactive system behavior; conditional branching, iteration, parallel composition, sequential ordering, and preemption, and
>
> 2) to eliminate the gap between specifications about which properties of a system can be proven, and the programs that are supposed to implement those specifications.

> Timed model-based programming takes this idea one step further, by reasoning on executable specifications directly, in real time."

While the synchronous languages give us a strong model for concurrency and precision in the construction of reactive embedded systems, they do not attempt to perform any sort of look-ahead or temporal planning. The introduction of nondeterministic choice and timing uncertainty into RMPL leads us to a new body of literature for inspiration, namely, Temporal Planning.

2.2.3 Temporal Planning

In the field of autonomous robotics, temporal reasoning plays a central role. Many robotics applications, such as surgical robots and space satellites, 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 temporal 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 et al., 2001; Effinger, 2006], the simple temporal problem under uncertainty (STPU) [Vidal and Fargier, 1999], and the conditional temporal problem (CTP) [Tsamardinos et al., 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 1998; Morris et al., 2001; Tsamardinos and Pollack, 2003] 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.

A set-bounded timing constraint, [lb,ub], denotes the minimum (lb) and maximum (ub) duration of time that an activity can take to complete. Set-bounded temporal planners reason over networks of set-bounded timing constraints in order to guarantee a plan's goals are accomplished. In Chapter 5 of this thesis, we summarize set-bounded planning and execution in more detail.

In this thesis, we advance the state-of-the-art in the field of temporal planning by developing probabilistic temporal reasoning algorithms that simultaneously account for timing uncertainty, nondeterministic choice, runtime sensing, and exception handling. To do so, this thesis builds directly upon the work of [Tsamardinos et al., 2003], which augments the STP and DTP formalisms with observation nodes (called a CTP) to allow for the analysis of plans with conditional threads of execution and set-bounded temporal constraints. The work of [Tsamardinos et al., 2003] is summarized in more detail in Chapter 5.

2.2.4 Timed Model-based Programming

This thesis is the second PhD thesis dedicated to the inclusion of time into Model-based Programs [Williams et al., 2003; Williams et al., 2004]. The first PhD thesis, which was completed in 2003, [Ingham, 2003] focused on the inclusion of time into the physical plant model and control program so they obey the natural timing characteristics of the physical systems they model. This prior work focused on providing "reactive" robustness at execution time via in-the-loop diagnosis and recovery (i.e., reasoning on the plant model, but not the control program), whereas this thesis focuses on providing planning to minimize risk at execution time (reasoning on the control program).

This thesis extends the control program specifications in [Ingham, 2003] to also incorporate nondeterministic choice, more extensive exception-handling, timing uncertainty, and probabilistic reasoning over a program's timing constraints, into the model-based control program. The incorporation of these features improves the robustness of model-based programming by enabling a model-based executive to recover gracefully from failure, as well as to reason over alternative options at runtime, in order to make control decisions that maximize the probability of successful program execution.

This concludes our review of formal modeling of timed systems; in the next section we discuss related work on hierarchical Markov decision processes.

2.3 Markov Decision Processes

In this section, we first introduce Markov Decision Processes (MDP) and value iteration [Bellman 1960], an established technique for solving MDPs. Then, we summarize prior work in decomposition approaches for MDPs [Wang and Mahadevan, 1999].

2.3.1 Markov Decision Processes and Value Iteration

A brief summary of Markov Decision Processes (MDPs) is presented here. First, we formally define an MDP. Then, we define two components that comprise the solution to an MDP, a policy and value function. Then, we summarize value iteration, and we define its two outputs, an optimal policy and optimal value function. Finally, we describe MDP execution.

Definition 2.1 Markov Decision Process (MDP)

A Markov decision process (MDP) is a $\textit{4-tuple}(S, A, P, R)$ where,

- S is a finite set of states,
- $A(s)$ is a finite set of actions, available in each state, $s \in S$
- $P(s,a,s') = \Pr\left(s_{t+1} = s' \middle| s_t = s, a_t = a\right)$ is the probability that action a in state s at time t will lead to state s' at time $t+1$,
- $R_a(s,s')$ is the immediate reward received after transition to state s' from state s with transition probability $P_a(s,s')$.

Definition 2.2 MDP policy, $\pi(s)$

A policy, $\pi(s)$, specifies the action $a \in A_s$, to choose in each state, $s \in S$. A policy is computed prior to execution, and predetermines the action to take in each reachable state, $s \in S$.

Definition 2.3 Value Function, $V(s)$

A value function, $V(s)$, gives the expected discounted reward of following policy $\pi(s)$, from each state, $s \in S$, in the MDP. γ is a discount factor $0 < \gamma \leq 1$, which is typically close to 1.

$$V(s) = \sum_{s'} P_{\pi(s)}(s, s')\left(R_{\pi(s)}(s, s') + \gamma V(s')\right)$$

Definition 2.4 Value Iteration

In value iteration [Bellman 1957], the calculation of $\pi(s)$ is substituted into the calculation of $V(s)$ to get the combined step:

$$V(s) = \max_a \left\{ \sum_{s'} P_a(s, s')\left(R_a(s, s') + \gamma V(s')\right) \right\}$$

This rule is iterated until convergence, and is often called the recursive Bellman equations. The result of value iteration is an optimal value function and optimal policy, as defined below.

Definition 2.5 Optimal Value Function, $V^*(s)$

The result of value iteration is an optimal value function, $V^*(s)$, which indicates the action in each state that gives the highest expected return:

$$V^*(s) = R(s) + \max_a \gamma \sum_{s'} P(s' | s, a) V(s')$$

Definition 2.6 Optimal Policy, $\pi^*(s)$

An optimal policy, $\pi^*(s)$, is the policy which maximizes the cumulative expected reward for each state, $s \in S$, $\pi(s) = \arg\max_{a} \left\{ \sum_{s'} P_a(s,s')\big(R_a(s,s') + \gamma V(s')\big) \right\}$, and which results in the optimal value function, $V^*(s)$.

Next we describe MDP execution. MDP execution follows a Sense, Decide, Act loop. The policy, $\pi(s)$, informs the execution algorithm, at each state, s, which action to take. Next, the selected action is executed and the state transitions stochastically according to the transition function, T. Finally, the result of the stochastic transition is sensed, and the resulting state, s' replaces s in the policy lookup in the next iteration of the algorithm. Execution halts when a terminal state is reached.

Definition 2.7 MDP Execution Algorithm

Given an $MDP = (S, A, P, R)$, and a policy $\pi(s)$, an MDP executes as follows:

```
1. s = s_0    // initial MDP state
2. While not( terminal_state( s ) )
3.         Look at policy for action to take, $a_s = \pi(s)$              // Decide
4.         Execute action $a_s$                                            // Act
5.         Transition to state $s'$ (in accordance with transition function, $P(s,s')$)
6.         Observe which step, s' was transitioned into.                  // Sense
7.         s = s'
```

This concludes our review of MDPs, next we discuss hierarchical decomposition methods.

2.3.2 Decomposition Approaches for MDPs

In this subsection, we discuss decomposition approaches for MDPs. In [Wang and Mahadevan, 1999] MDP decomposition methods are classified into three basic types, state decomposition, action decomposition, and policy decomposition. First, we summarize state decomposition and action decomposition. Then, we describe policy decomposition in more detail, since we build upon that technique in Chapter 6.

State decomposition

In state decomposition [Dean and Lin, 1995; Lin and Makedon, 2000; Parr, 1998] an MDP's states are divided into subsets, each forming a new sub-MDP. A few states are shared between the sub-MDPs, thus resulting in a loose state coupling. For example, consider the floorplan example in Figure 2.1, below. The floorplan is discretized into grid cells, with each grid cell being an MDP state. The MDP states can then be subdivided into rooms, with grid cells between two rooms being interpreted as shared states that belong to both rooms. In performing state decomposition, a sub-MDP is solved independently for each room using value iteration. Then, for each shared state, the maximum value as a result of performing value iteration on all sub-MDPs, is shared with the other sub-MDPs. Then, the sub-MDPs are solved again using value iteration, and the maximum values of shared states are again shared between sub-MDPs. This process repeats until the value functions for each sub-MDP converge. In state decomposition, the actions, transition probabilities, and rewards remain the same as the original MDP, while the states are divided up into subsets.

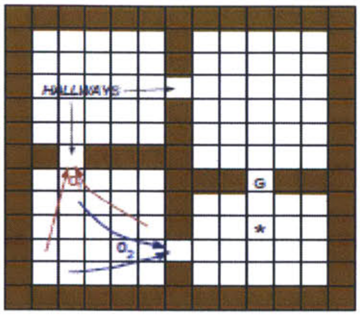


Figure 2.1: State decomposition on a floorplan example.

Action decomposition

In action decomposition [Meuleau et al., 1998; Singh and Cohn, 1998], an MDP is divided into sub-MDPs, with the sub-MDPs corresponding to weakly-coupled concurrent processes. The sub-MDPs are connected only through their action sets. Taking an action in one sub-MDP affects the actions available in other sub-MDPs. In [Meuleau et al., 1998], action decomposition is used in order to improve tractability in solving sequential stochastic resource allocation problems, such as military air campaign planning. Taking an action corresponds to consuming a resource. The reward function and transition probabilities for each sub-MDP are independent.

In [Singh and Cohn, 1998], the value functions produced in solving each sub-MDP are used to obtain upper and lower bounds on the global value function. The bounds are used to improve convergence for the global MDP. Alternatively, in [Meuleau et al., 1998], the solution for each sub-MDP is computed offline, and an on-line greedy heuristic search technique is developed to choose actions between sub-MDPs. This approach sacrifices optimality for computational feasibility, and avoids the need to reason a-priori about all possible contingencies.

Policy decomposition

In both state and action decomposition, the transition probabilities and rewards in sub-MDPs are the same as the original MDP, and they are independent from one another. Alternatively, policy decomposition [Wang and Mahadevan, 1999] allows for the state in one sub-MDP to affect the transition probabilities and rewards in another sub-MDP. Policy decomposition works by first identifying for each sub-MDP, the states in other sub-MDPs that affect its transition probabilities and rewards. These are called the sub-MDP's *parameter states*. Then, each sub-MDP is solved independently, once for each possible combination of parameter states, resulting in a parameterized decision policy for each sub-MDP. The policy employed at runtime to execute a sub-MDP varies based on which of its parameter states are occupied in the other sub-MDPs. For sub-MDPs with few parameter states, policy decomposition results in a computational improvement while still accounting for the weak coupling between sub-MDPs.

Consider a manufacturing line example [Wang and Mahadevan, 1999] depicted in Figure 2.2. A manufactuing line consists of a sequence of machines, m_i. The goal of a manufacturing line is to meet demand while minimizing the standing inventory. The state of each machine is characterized by five states, $m_i = \{working, n_i, s_i, d_i, t_i\}$. The first state, *working*, is Boolean to indicate whether the machine is either functional or broken. The remaining four states are integer valued. n_i is a machine's standing inventory, s_i is supply rate, d_i is demand rate, and t_i is a machine's uptime. Actions for each machine are $a_i = \{producing, idling, maintenance\}$. Reward is attained for producing and for minimizing the standing inventory.

To perform policy decomposition, we first identify the parameter states between sub-MDPs. As depicted in Figure 2.3, the demand rate of the previous machine, m_{i-1}, and the supply

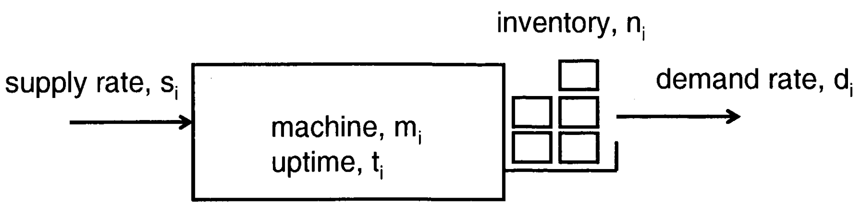


Figure 2.2: Manufactuing line example.

rate of the following machine, m_{i+1}, are a machine's parameter states in this example. Next, we solve each sub-MDP independently, once for each possible combination of parameter states, resulting in a parameterized decision policy for each sub-MDP, as depicted in Figure 2.3 below.

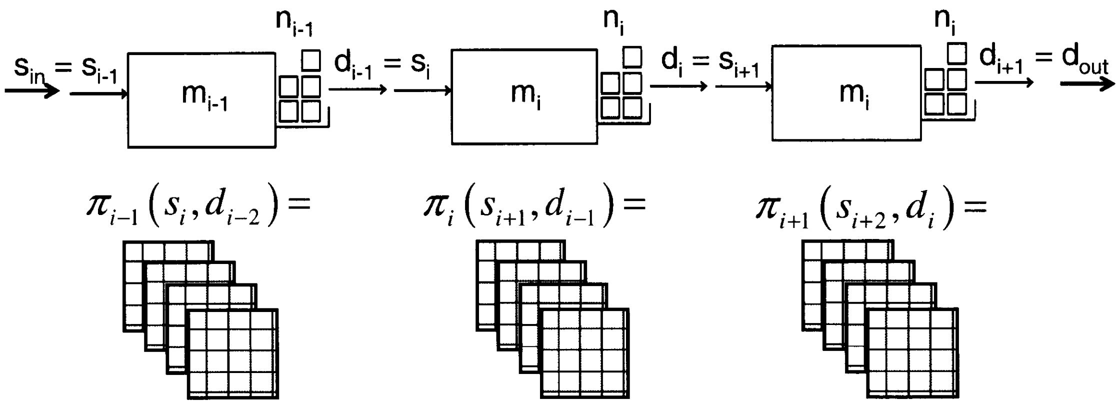


Figure 2.3: Identify parameter states and compute a parametized policy for each sub-MDP.

Policy decomposition improves tractability by not solving for the entire joint state space between sub-MDPs, but as a result, is susceptible to local maxima. Therefore, the increased tractability comes at the expense of suboptimal performance, and must be deemed either acceptable or unacceptable for a given application.

This concludes Chapter 2 – Related Work, next in Chapter 3 we introduce RMPL syntax and execution semantics.

Chapter 3: RMPL Syntax and Execution Semantics

In this chapter, we develop the syntax and execution semantics for a variant of the Reactive Model-based Programming Language (RMPL) [Williams, et. al., 2003, Ingham 2003] that emphasizes nondeterministic activity scheduling, nondeterministic subprocedure invocation, constraints on program timing, and recovery from failed activities at reactive timescales via exception handling. This rich combination of language constructs enables us to encode robust autonomous behaviors.

First, in Section 3.1, we develop a formal syntax for the RMPL language. We explain each language construct, and then outline the criteria for successful and failed program execution. Then, in Section 3.2, we define Hierarchical Timed Automata (HTA). HTA provide a compact encoding of RMPL programs, while enabling a simple execution semantics. In Section 3.3, we define the mapping from RMPL to HTA. Then, in Section 3.4, we define legal executions of an HTA, and in Section 3.5 we develop an execution semantics for HTA. Finally, in Section 3.6, we demonstrate HTA execution on a Bear robot example.

3.1 Syntax

In this section, we develop a formal syntax for a variant of the RMPL language comprised of eight constructs; primitive activities, conditional branching, iteration, parallel composition, sequential ordering, exception handling, nondeterministic choice, and metric timing constraints. The Backus-Naur form syntax for this variant of RMPL is presented in Figure 3.1. A primitive activity is denoted by $primitive(\,)$, where $primitive$ is the name of the primitive activity. $[lb,ub]\{expr\}$ enforces timing constraints by bounding the minimum and maximum duration of an encapsulated subprogram, *{expr}*. *Sequence* and *parallel* enable composition of subprograms, while *if-else*, *choose*, and *try-catch* enable conditional execution, nondeterministic choice between subprograms, and exception handling, respectively. A *noop()* is a primitive activity with zero duration. An exception is thrown at runtime when any timing constraint is violated, and when a primitive activity fails.

RMPL Backus-Naur form syntax:

```
          expr    ::= primitive( dur ) | primitive( ) |
                      [lb,ub]( name ){expr } |
                      sequence{ $expr_1$, $expr_2$,...} |
                      parallel{ $expr_1$,$expr_2$,...} |
                      if (obsv) { $expr_1$ } else { $expr_2$ } |
                      choose{ $expr_1$, $expr_2$,...} |
                      try{ expr }catch( exception( primitive | name ) ){ $expr_1$ }
                        catch(...){...}catch-all{ $expr_n$ }
    primitive( ) ::= choose{ primitive(lb), primitive( lb+$\Delta t$ ), ... , primitive(ub) }
                        for each primitive, lb, ub, and $\Delta t$, are global parameters
primitive(dur) ::= execute activity named "primitive" with target duration, dur
 [lb,ub](name) ::= a timing constraint on program execution, identified by "name"
           obsv ::= a Boolean observation variable queried at runtime
exception( primitive | name ) ::= a predicate indicating a thrown exception
```

Figure 3.1: RMPL Bakus-Naur Form Syntax

The *until* construct is implemented as a derived construct, through repeated invocation of the *if-else* construct, where *n* is a finite integer, as follows:

```
until( obsv , n ){ expr }::= if ( obsv ) { noop() }
                            else{ sequence { expr , if ( obsv ){ noop() }
                            else{ sequence { expr , if ( obsv ){ noop() }
                                "    ...repeated n-1 times...   "
                                                   else { expr }
                                }         }
                                }         }
```

An RMPL program terminates under two conditions, 1) the end of the program is reached, or 2) an uncaught exception occurs. In this thesis, we define successful program execution as any program execution that terminates with no uncaught exceptions. Note that exceptions, i.e. timing constraint violations and activity failures, are allowed in successful program executions, as long as they are caught and handled by an exception handler. For example, consider the Bear obstacle course example RMPL program in Figure 3.2. The primitive activity *hurdles* is encapsulated within an exception handling construct that invokes *hurdles_recovery* if the logroll activity fails. Program executions that include a logroll activity failure and subsequent recovery are considered successful program executions. However, if any of the other primitive activities, or the timing constraint, [0,180], throws an exception, the program will terminate via an uncaught exception.

a)

```
Obstacle Course Program:
[0,180]{
  sequence{
    try{ hurdles() }
      catch{ hurdles_recovery() }
    if(clear){ ramp() }
    choose{ slalom() , curbs() }
  }
}
```

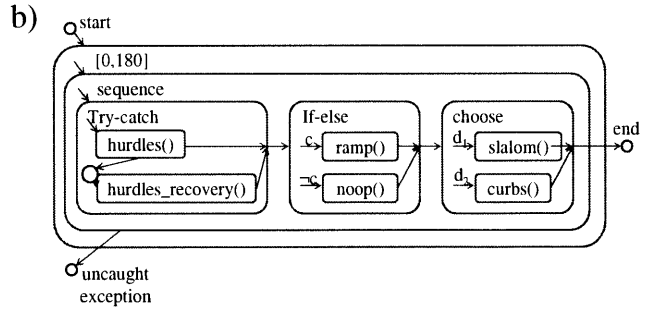


Figure 3.2: a) Bear obstacle course example RMPL program, and b) HTA representation.

3.2 Hierarchical Timed Automata (HTA)

In this section, we define Hierarchical Timed Automata (HTA). Let L denote a finite set of automata locations. RMPL program execution is defined in terms of a timed sequence of automata locations, which represent the "state" of the control program's execution at any given time. The automata change state according to a set of *input variables* Π partitioned into *control* *variables* Π^c and *observable variables* Π^o, as well as a set of *clock variables* C. A *control action* μ assigns a value to each variable in Π^c. Each variable in Π^c ranges over a finite domain, $\mu_i = \{\varnothing, val_1, val_2, ..., val_d\}$, including a null control action, $\mu_i = \varnothing$, which represents having taken no control action. Variables in Π^o are Boolean, and are observations of the robot (plant) under control. An *observation* o assigns a value to each variable in Π^o. A *clock interpretation* ν assigns a value in $\Re^+$ to each clock in C. Clock variables measure the elapsed time in program locations. Each location has a corresponding clock that is initialized upon transitioning into that location, $init(l) \subseteq C$. Transitions between locations are conditioned on constraints over clock variables, $\Phi(C)$, and transition guards, G.

Definition 3.2: Clock Constraints, $\Phi(X)$

The types of clock constraints allowed are simple comparisons of a clock value, *x*, with a constant, *c*, and Boolean combinations of such simple comparisons [Alur and Dill, 1994]. For a set X of clocks, the set $\Phi(X)$ of *clock constraints* δ is defined inductively by

$$\delta := x \leq c \mid c \leq x \mid \neg\delta \mid \delta_1 \wedge \delta_2 \mid \delta_1 \vee \delta_2,$$

where x is a clock in X and c is any constant in $\Re^+$.

Definition 3.3: Transition Guards, *G*

Each transition has a guard g that evaluates to "true" or "false" and consists of a Boolean combination of control assignments, observations, and clock constraints, defined inductively by

$$g := \left(\mu_i = \{\varnothing, val_1, val_2, ..., val_d\}\right) \Big| \left(o_j \in O_j\right) \Big| \left(\delta \in \Phi(C)\right) \Big| \neg g \Big| g_1 \wedge g_2 \Big| g_1 \vee g_2 .$$

When a transition guard evaluates to true, the transition occurs immediately. An empty guard is trivially true. The set of all feasible guards is denoted $g \in G$.

Transition Guard Examples: Figure 3.3a shows the HTA for a primitive activity, *hurdles*. The transition leaving the start location has no guard, and is thus enabled by default. Then, the scheduled duration of the hurdles activity is chosen based on the assignment made to control variable, u_1. The observation variable named "finished" on the outgoing transition into the "end" location indicates a successful execution, while the observation variable named "exception" on the outgoing transition into the "uncaught exception" location indicates a failed execution. Figure 3.3b shows a primitive activity encapsulated within a timing constraint [lb,ub]. The outgoing transition guards from the timing constraint location consist of Boolean operations over observation variables and clock constraints, enforcing the desired timing behavior. The exception transition is taken out of the timing constraint location when:

1. the clock *c* exceeds the upper timebound *ub*,
2. the hurdles activity finishes before the lower timebound *lb*, or
3. the hurdles activity throws an exception.

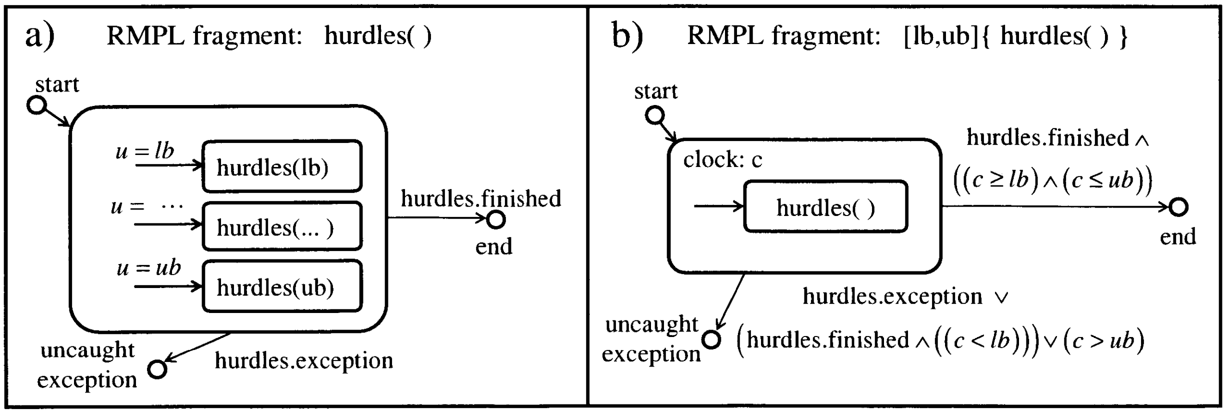


Figure 3.3: Transition guards for, a) a primitive activity, and b) a timing constraint.

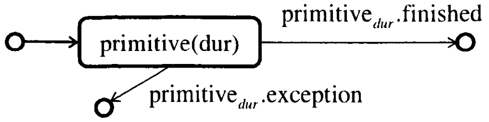
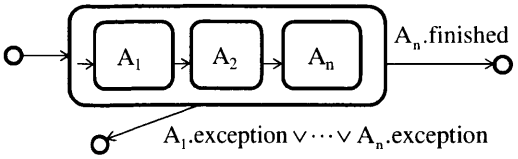
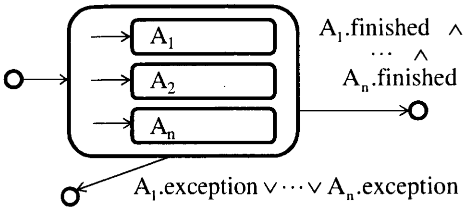
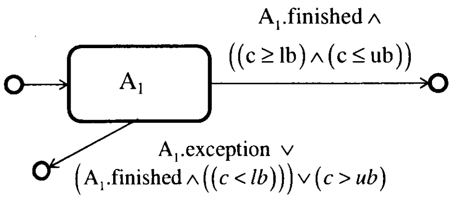
Definition 3.4: Hierarchical Timed Automata (HTA)

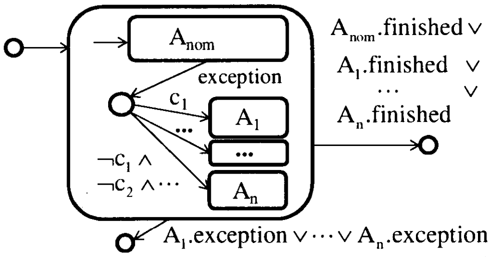
A Hierarchical Timed Automata (HTA) is a 9-tuple, $HTA = \langle L, l_0, L_T, \Pi, C, \eta, init, enabled, \tau \rangle$.

- L is a finite set of locations, partitioned into primitive and composite types, L_p and L_c :
  - $l \in L_p$, denotes primitive activities, $type(l \in L_p) =$ primitive, while
  - $l \in L_c$, denotes composite locations consisting of the following location types, $type(l \in L_c) \in \{$sequence, parallel, try-catch, lb-ub, if-else, choose$\}$.
- $l_0 \in L$ is a singular root location, $L_T \in L$, is a set of terminal locations.
- Π is a set of *input variables*, partitioned into *control variables* Π^c and *observable variables* Π^o. $o \in O$ denotes the set of all possible assignments over Π^o. $\mu \in U$ denotes the set of all possible assignments over Π^c.
- C is a set of *clock variables,* C, and v assigns a value in $\Re^+$ to each clock in C.
- $\eta : L \rightarrow 2^L$ maps each location to its child locations. η denotes the hierarchy among locations, giving rise to a tree structure with a single *root* location. We define two useful variants of the child function η:
  - $\eta *$ is the descendents function, the set of all nested locations of a location, l.
  - η^{-1} is the parent function, $\eta^{-1}(l) = \begin{Bmatrix} b, \text{ where } l \in \eta(b) \text{ if } l \neq root \\ \varnothing, \qquad\qquad \text{otherwise} \end{Bmatrix}$.
- $init(l) \subseteq C$, specifies the clock to initialize upon transitioning into each location, l.
- $enabled\left(L, v, o\right) \subseteq \Pi^o$, specifies the control variables that may be assigned a non-null value from their domains, given the active locations, clock valuations, and observations.
- $\tau : L \times G \times L$ is the set of transitions, $t \in \tau$, such that, $t = \langle l_1, g, l_2 \rangle$, with $l_1, l_2 \in L$ and $g \in G$, the set of all allowable transition guards (Definition 3.3).

3.3 Mapping from RMPL to HTA

In this section, we introduce a mapping from RMPL (Section 3.1) to HTA (Section 3.2), as depicted in Figure 3.4, below. Primitive activities are defined in Section 3.1. The sequence and parallel constructs enable sequential ordering and parallel composition of subprograms. The [lb,ub] construct is a timing constraint imposed on program execution. If a timing constraint is violated at runtime, an exception is thrown, as indicated by the constraints on clock variables present in the guard on the outgoing transition into the [lb,ub] location's exception state. The exception handling construct behaves in accordance with the behavior of exception handlers in other well-known languages such as C++ and Java. A thrown exception travels up the program stack until an exception handler is reached with a handler phrase that evaluates to "true". The if-else construct enables conditional execution, while the choose construct enables nondeterministic choice. The runtime condition that is evaluated in an if-else construct is assumed to be observable and translates directly into an observation variable in the resulting HTA. The observation variable is queried at runtime to determine which path to take. The nondeterministic choice construct is used to enable runtime activity scheduling and choice between subprogram invocation. The transition into each option for a nondeterministic choice location is labeled with a control variable assignment, which must evaluate to "true" at runtime for that branch to be taken. This concludes our explanation of RMPL to HTA translation.

primitive(dur)	 primitive(dur) primitive_{dur}.finished primitive_{dur}.exception
sequence { A_1 , A_2 , ... }	 A_1 A_2 A_n A_n.finished A_1.exception ∨ ⋯ ∨ A_n.exception
parallel { A_1 , A_2 , ... }	 A_1 A_2 A_n A_1.finished ∧ ⋯ ∧ A_n.finished A_1.exception ∨ ⋯ ∨ A_n.exception
[lb,ub] (name){ A_1 }	 A_1 A_1.finished ∧ $((c \geq lb) \wedge (c \leq ub))$ A_1.exception ∨ $(A_1.\text{finished} \wedge ((c < lb))) \vee (c > ub)$

try { A_{nom} } catch (c_1) {A_1}
catch(...) { ... }
catch-all { A_n }
	 A_{nom}
	exception
	c_1
	...
	A_1
	...
	A_n
	$\neg c_1 \wedge$
	$\neg c_2 \wedge \cdots$
	A_{nom}.finished ∨
	A_1.finished ∨
	⋯ ∨
	A_n.finished
	A_1.exception ∨ ⋯ ∨ A_n.exception

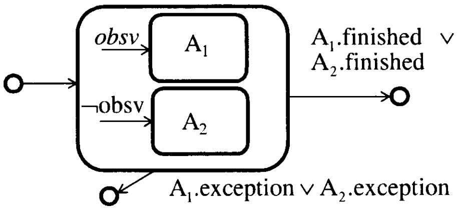
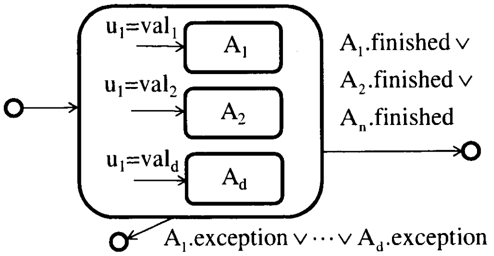
if (obsv) { A_1 } else { A_2 }	 *obsv* A_1 ¬obsv A_2 A_1.finished ∨ A_2.finished A_1.exception ∨ A_2.exception
choose { A_1 , A_2,... }	 u_1=val_1 A_1 u_1=val_2 A_2 u_1=val_d A_d A_1.finished ∨ A_2.finished ∨ A_n.finished A_1.exception ∨ ⋯ ∨ A_d.exception

Figure 3.4: Mapping from RMPL to HTA

3.4 Properties of a Legal Execution of an HTA

An HTA changes locations at absolute system times $t_0, t_1, \ldots, t_n$, called a timed trace, T, and maintains its locations between these discrete times. The duration of each time step $d_i = t_{i+1} - t_i$ is not constant and depends on the observation times of external events, $o \in O$. The times at which observations occur is denoted by a timed observation sequence $(o,t)^n$. A control action is taken by the HTA immediately in response to each observation, leading to a control action sequence $(\mu,t)^n$ with the same timed trace T as the observation sequence. The nondeterminism in an HTA originates from the choice over control actions after each observation. An execution e of an HTA is of the form:

$$e : \langle L_0, v_0 \rangle \xrightarrow[t_1]{o_1,\mu_1} \langle L_1, v_1 \rangle \xrightarrow[t_2]{o_2,\mu_2} \langle L_2, v_2 \rangle \xrightarrow[t_3]{o_3,\mu_3} \cdots$$

with $L_i \in L$ and $v_i \in [C \rightarrow \Re]$, for all $i \geq 0$. Each state in the sequence, $\langle L_i, v_i \rangle$, is an HTA labeling plus an assignment to clock variables. Each transition, $\xrightarrow[t_i]{o_i,\mu_i}$, specifies observed values at the time of transition, o_i, the time that has elapsed, t_i, and a control assignment, u_i. The notation used above borrows directly from timed automata theory [Alur and Dill, 1994] and differs only in its inclusion of a nondeterministic control assignment, u_i.

A legal execution of an HTA must satisfy the following requirements:

1. Initiation: L_0 is the initial state, and $v_0(x) = 0$ for all $x \in C$.
2. Transitions are Obligatory: for all $i \geq 0$, if there is a transition of the form $\langle l_{i-1}, g_i, l_i \rangle$ such that guard g_i evaluates to true given observations o_i, control assignments μ_i, and clock values v_i, then that transition must be taken.

3. Transition Ordering: Start from the root location, and search breadth-first for enabled transitions. Take the first transition encountered, and then repeat. Start from the root location again and search breadth-first for enabled transitions. This approach enforces preemption by allowing parent locations to transition prior to their children.
4. Clock Initialization: for each location l_i, transitioned into, $\langle l_{i-1}, g_i, l_i \rangle$, its associated clock variable $c_i = init(l_i)$ must be reset to zero, $v_i(c_i) \rightarrow 0$.
5. Clock Progression: all clocks not initialized to zero must progress by the amount of time elapsed between observations, $v_i = v_{i-1} + t_i - t_{i-1}$.
6. Hierarchy: Transitions $\langle l_{i-1}, g_i, l_i \rangle$ can be partitioned into two categories,
    - Parent-to-child: $l_i = \eta(l_{i-1})$. The originating location is the destination location's parent. The parent location is not removed from the list of active locations L_i, and the child location is added, $L_i = L_{i-1} + l_i$. (Parents enable their children)
    - Not parent-to-child: $l_i \neq \eta^{-1}(l_{i-1})$. The originating location is *not* the destination location's parent. First, remove the locations being transitioned out of from the list of active locations, L_i, which consist of all descendants of the destination's parent, $L_i = L_{i-1} - \eta^*(\eta^{-1}(l_i))$. Then, add the destination location, $L_i = L_i + l_i$.
7. Synchronicity: Given a set of observations o_i and control assignments μ_i all transitions resulting from Step 3 above are assumed to occur instantaneously and simultaneously at time t_i. This approach is taken in all synchronous languages [Berry and Gontheir, 1992].
8. Acquiescence: At each time step transitions occur until no more transitions are enabled. For example, at time step t_i a transition between sibling locations $\langle l_a, g_1, l_b \rangle$ adds l_b to

the list of active locations L_i while removing l_a. The addition of l_b to L_i subsequently enables transition g_2: $\langle l_b, g_2, l_c \rangle$ which adds l_c, the child of location l_b, to L_i. This process occurs in accordance with Step 3 until no more transitions are enabled. In accordance with Step 7 above, all resulting transitions are assumed to occur instantaneously at time t_i.

9. Mutual Exclusivity: A control variable may only be assigned one value from its domain, $\mu_i = \{\varnothing, val_1, val_2, ..., val_\mathrm{d}\}$, at each time step t_i. For example, only one of the following control assignments, $\mu_i = \varnothing, \mu_i = val_1, \cdots, \mu_i = val_\mathrm{d}$, is allowed to evaluate to true during each time step t_i.
10. Enabledness: In each time step t_i a control assignment, $\mu_i = val_d$, is allowed to be made if and only if that control variable is enabled, $\mathit{enabled}(L, v, o) \subseteq \Pi^o$, given the current HTA state. Otherwise, the control variable must be assigned its null value, $\mu_i = \varnothing$. This requirement ensures that control assignments are not assigned extraneously.

3.5 Execution Semantics

In this section we develop an HTA execution semantics. We start by defining an HTA decision policy π that maps locations L_{i-1}, clock valuations v_{i-1}, and observations o_i, to control assignments μ_i at times t_i, $\langle L_{i-1}, v_{i-1} \rangle \xrightarrow[t_i]{o_i, \mu_i} \langle L_i, v_i \rangle$. Then, we present an algorithm for HTA execution that takes as input an HTA decision policy π and a timed sequence of observations $(o,t)^n$, and provides as output a timed sequence of control assignments $(\mu,t)^n$, resulting in a legal HTA execution, $e: \langle L_0, v_0 \rangle \xrightarrow[t_1]{o_1, \mu_1} \langle L_1, v_1 \rangle \xrightarrow[t_2]{o_2, \mu_2} \langle L_2, v_2 \rangle \xrightarrow[t_3]{o_3, \mu_3} \cdots$.

Definition 3.5: HTA Decision Policy, π

An HTA decision policy, π, determines control assignments, $\mu_i = U$, at runtime in response to observations, o_i, at observation times, t_i. π maps locations L_{i-1}, clock valuations v_{i-1}, and observations o_i, at times, t_i, to control assignments μ_i, denoted as $\mu_i = \pi(L_{i-1}, v_{i-1}, o_i, t_i)$.

The HTA execution algorithm, presented below in Algorithm 3.1, can be characterized as a Wait, Sense, Decide, Act loop. The Wait and Sense steps (steps 3 and 4) correspond to monitoring for and recording which observation variables evaluate to true at runtime. The Decide step corresponds to looking up, in the HTA decision policy, π, which control assignments to make given the current observations o_i and HTA state, $\langle L, v\rangle$. The Acting step corresponds to taking all transitions whose guards have become enabled based on the prior Wait, Sense, and Decide steps. Next, a walkthrough of the algorithm is presented in Section 3.6.

Algorithm 3.1: HTA Execution Algorithm

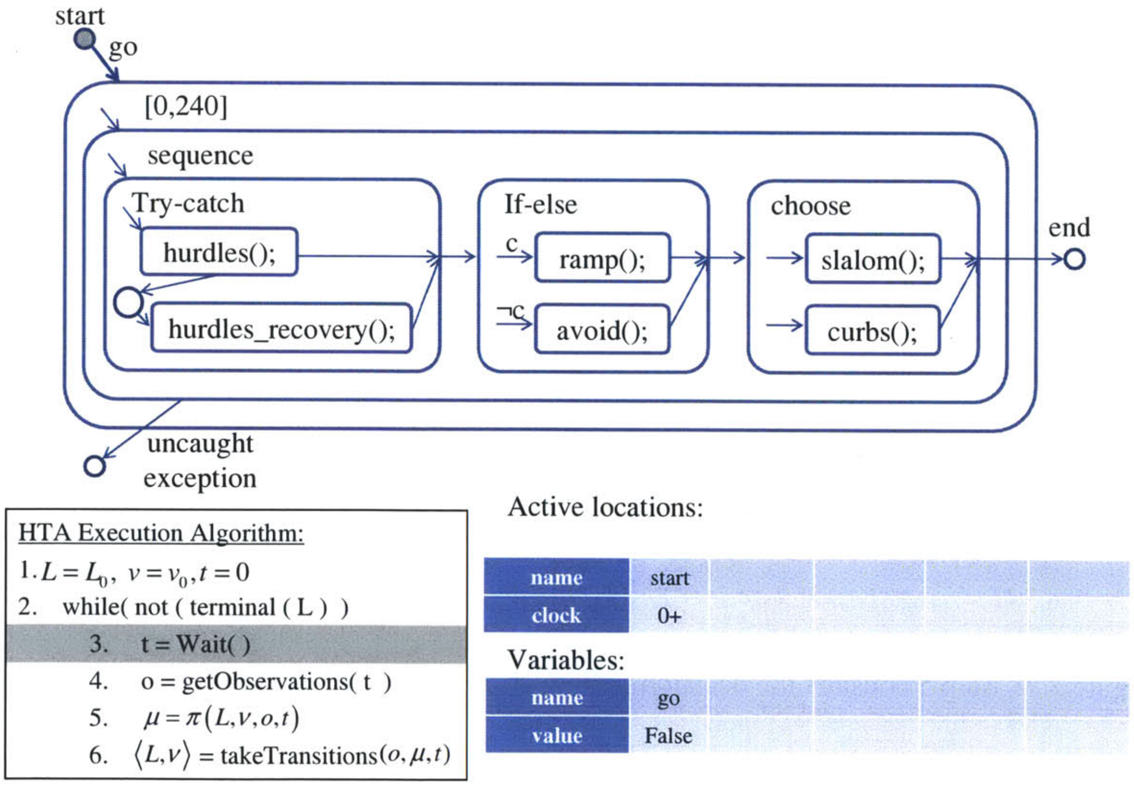
Inputs: Hierarchical Timed Automata (HTA)
HTA decision policy, π
Timed observation sequence, $(o,t)^n$

Output: Timed control sequence, $(\mu,t)^n$
A legal execution, $e: \langle L_0, v_0\rangle \xrightarrow[t_1]{o_1,\mu_1} \langle L_1, v_1\rangle \xrightarrow[t_2]{o_2,\mu_2} \langle L_2, v_2\rangle \xrightarrow[t_3]{o_3,\mu_3} \cdots$

```
1. $L = l_0$, $v = v_0, t = 0$                         // Initialization
2. while $not(terminal(L))$                            // Termination check
3.         $t = Wait()$                                 // Wait     - for next observation to arrive
4.         $o = getObservations(t)$                     // Sense    - get observations at time, t
5.         $\mu = \pi(L, v, o, t)$                      // Decide   - get control assignments from, $\pi$
6.         $\langle L, v\rangle = takeTransitions(o, \mu, t)$   // Act      - take enabled transitions
```

3.6 An HTA Execution Example: $\langle Wait, Sense, Decide, Act \rangle$

To get a feel for the HTA execution algorithm, we walk through an execution trace of the Bear obstacle course example. Execution begins with the only marked location being the start location, depicted in grey in Figure 3.5. The start location's outgoing transition has a guard with a Boolean observation variable named "go". When the variable "go" changes from "False" to "True", the HTA execution algorithm progresses from step 3, to 4, and then on to step 5. Next, depicted in Figure 3.6, the decision policy π determines what value the control variable u_l should take. In Chapter 4 we describe how this decision policy is computed. The value assigned to u_l determines the scheduled duration of the hurdles activity, which is 45 seconds, as depicted in Figure 3.7.



```
HTA Execution Algorithm:
1. $L = L_0$, $v = v_0, t = 0$
2.  while( not ( terminal ( L ) ) )
        3.  t = Wait( )
        4.  o = getObservations( t )
        5.  $\mu = \pi(L, v, o, t)$
        6.  $\langle L, v \rangle$ = takeTransitions$(o, \mu, t)$
```

Active locations:

name	start				
clock	0+				

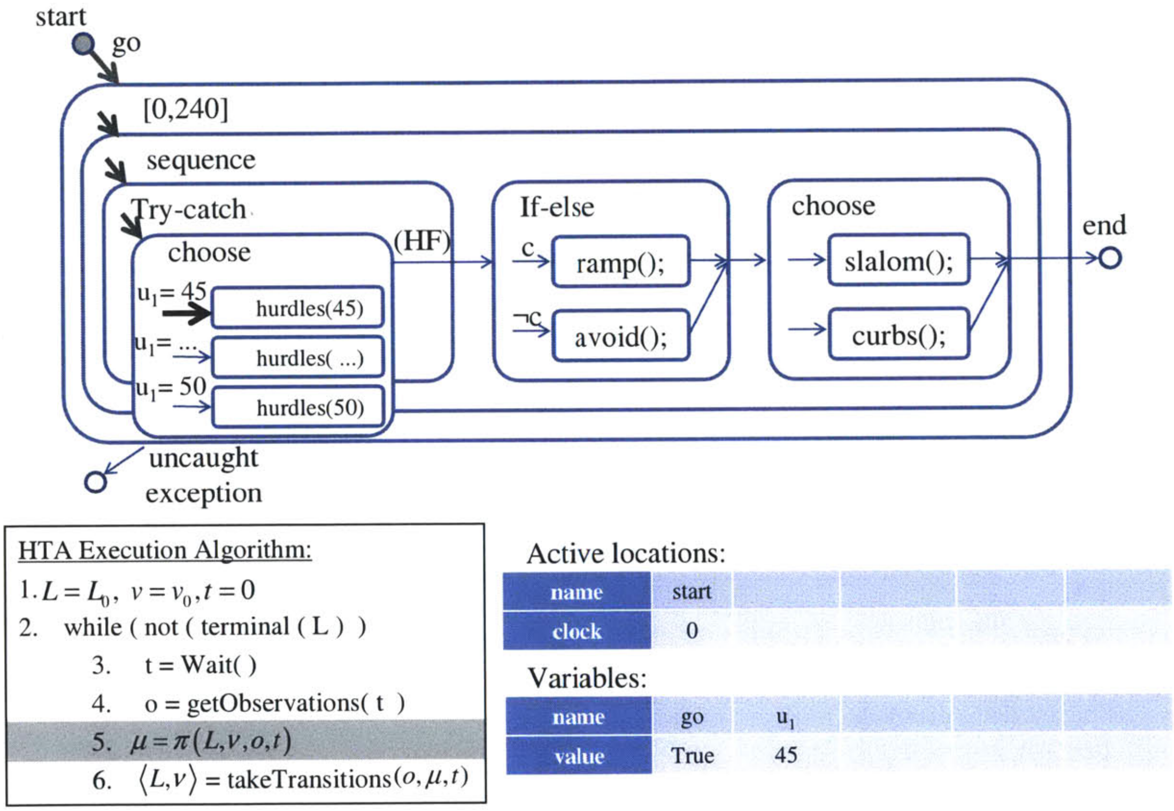
Variables:

name	go				
value	False				

Figure 3.5: The start location is active, waiting on the Boolean observation variable "go".

	Program state	Program clock	Possible Decisions	Decision
Activity scheduling	Logroll	t = 0	{45,…,50}	45
	Ramp	t = 45…160	{70,…,75}	70
	Slalom	t = 45…235	{65,…,70}	65
	Curbs	t = 45…235	{30,…,35}	30
Subprocedure choice	Choose	t = 45…150	{Slalom,Curbs}	Slalom
		t = 150…235	{Slalom,Curbs}	Curbs

Figure 3.6: Risk-minimizing decision policy for the motivating example.



HTA Execution Algorithm:

```
1. $L = L_0,\ v = v_0, t = 0$
2. while ( not ( terminal ( L ) )
        3.   t = Wait( )
        4.   o = getObservations( t )
        5.   $\mu = \pi(L, v, o, t)$
        6.   $\langle L, v \rangle$ = takeTransitions$(o, \mu, t)$
```

Active locations:

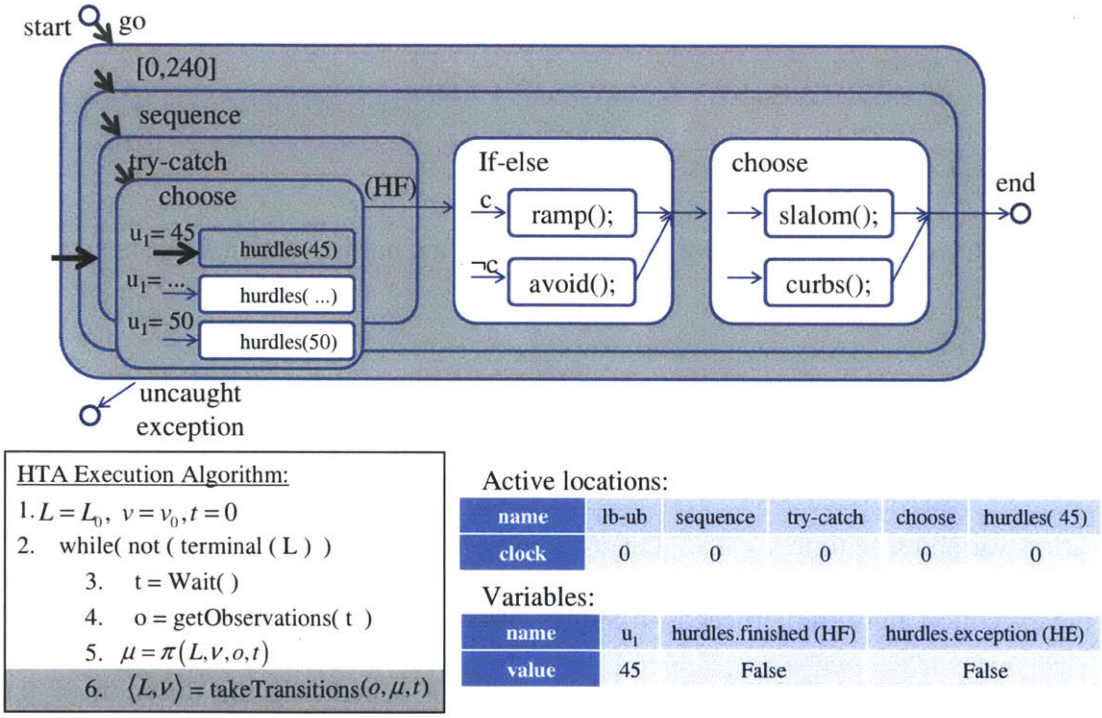
name	start
clock	0

Variables:

name	go	u_1
value	True	45

Figure 3.7: Step 5 consults the decision policy, π, in order to schedule the *hurdles* activity.

After the assignment to variable u_1 is made, the execution algorithm proceeds to step 6, depicted in Figure 3.8. In step 6, the *takeTransitions( )* function traverses the enabled transitions (highlighted in bold) from the start location, enters the hurdles activity, sets its clock value to zero, and sends the command, *hurdles(45)*, to the Bear robot. The transitions then cease since the observation variables on outgoing transitions from the hurdles activity, *hurdles.finished* and *hurdles.exception*, are both "False". The hurdles.finished variable is abbreviated (HF) in Figure 3.8. The hurdles.exception variable (HE) is not visible in the figure because of the expanded hurdles location. The clock of each newly marked location is initialized to zero. Then, the



```
HTA Execution Algorithm:
1. $L = L_0,\ v = v_0, t = 0$
2. while( not ( terminal ( L ) )
        3. t = Wait( )
        4. o = getObservations( t )
        5. $\mu = \pi(L, v, o, t)$
        6. $\langle L, v \rangle$ = takeTransitions($o, \mu, t$)
```

Active locations:

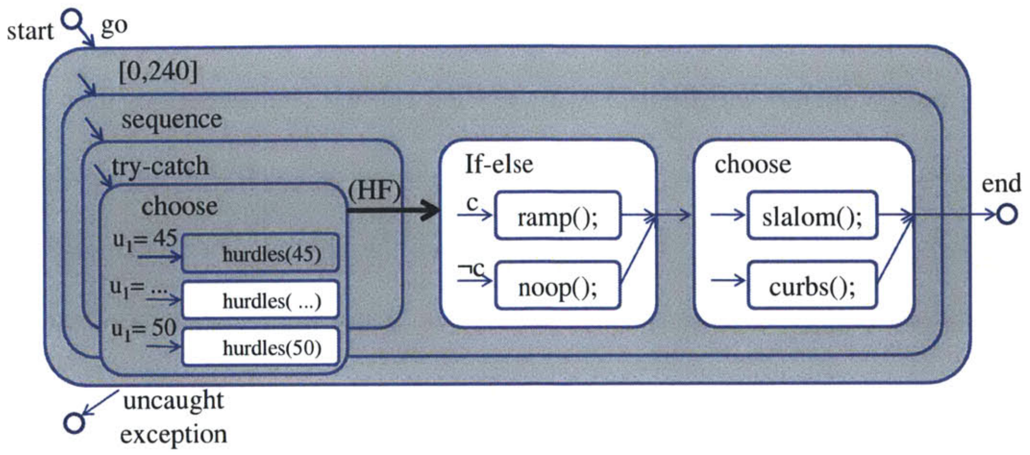
name	lb-ub	sequence	try-catch	choose	hurdles(45)
clock	0	0	0	0	0

Variables:

name	u_1	hurdles.finished (HF)	hurdles.exception (HE)
value	45	False	False

Figure 3.8: Enabled transitions are taken, and the hurdles activity is scheduled and executed.

execution algorithm returns to step 3 to wait for one of the observation variables to change from "False" to "True". We assume that the *hurdles(lb)* activity finishes in exactly 45 seconds, changing the value of the *hurdles.finished* variable to "True", and causing the execution



HTA Execution Algorithm:

```
1. $L = L_0$, $v = v_0$, $t = 0$
2. while( not ( terminal ( L ) )
      3. t = Wait( )
      4. o = getObservations( t )
      5. $\mu = \pi(L, v, o, t)$
      6. $\langle L, v \rangle$ = takeTransitions$(o, \mu, t)$
```

Active locations:

name	lb-ub	sequence	try-catch	choose	hurdles(45)
clock	45	45	45	45	45

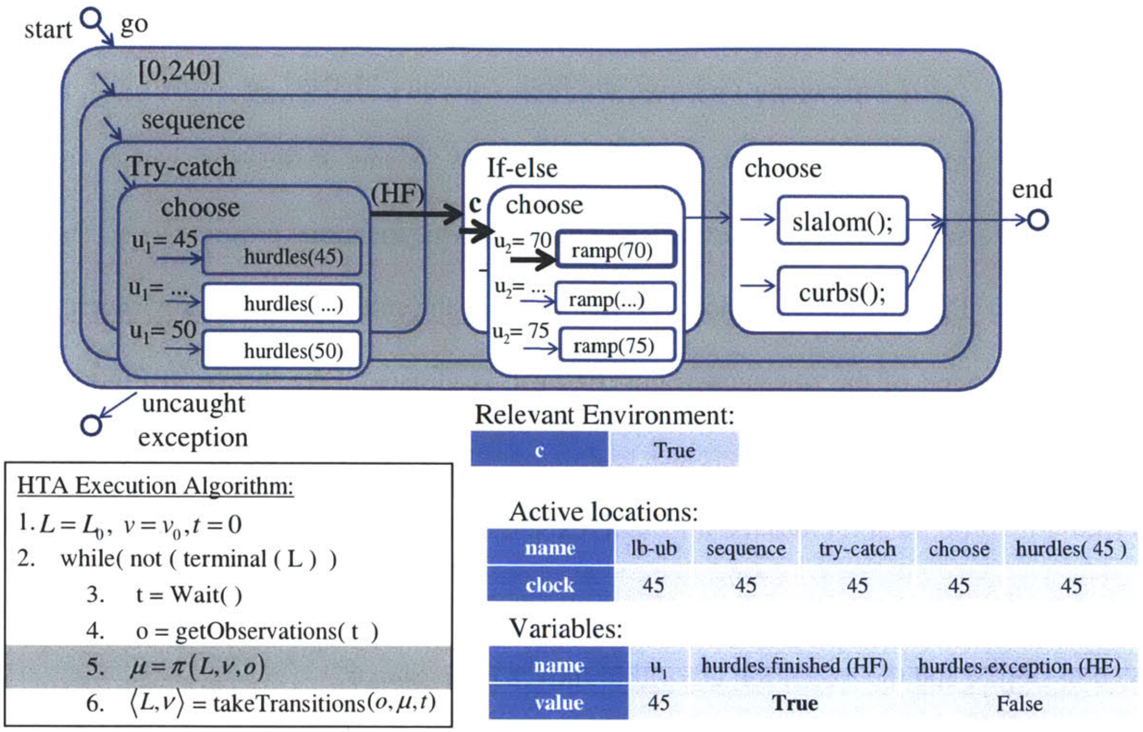
Variables:

name	u_1	hurdles.finished (HF)	hurdles.exception (HE)
value	45	**True**	False

Figure 3.9: Clock values increment in real time until the hurdles activity finishes executing.

algorithm to move on to step 4. The ramp activity is only executed if the Boolean observation variable "c" guarding the transition into the ramp activity evaluates to "True" as Step 4 polls the observation variables at time $t = 45$. Let us assume that the variable "c" evaluates to "True". Next, the execution algorithm proceeds to Step 5. As depicted in Figure 3.10, the decision policy must now make a choice in assigning variable u_2, thus scheduling the duration of the ramp activity. The decision policy returns the assignment {u_2=70}. After assigning the control variable in Step 5, the execution algorithm moves on to Step 6.

In step 6, *takeTransitions()* traverses the enabled transitions (highlighted in bold in Figure 3.10) from the hurdles activity, through the transition guard, c, through the control variable assignment {u_2=70}, and finally into the *ramp(70)* location sending the command to the Bear robot. The clock of each location transitioned into is initialized to zero. The transitions then



Relevant Environment:

c	True

HTA Execution Algorithm:

```
1. $L = L_0,\ v = v_0, t = 0$
2. while( not ( terminal ( L ) )
      3. t = Wait( )
      4. o = getObservations( t )
      5. $\mu = \pi(L, v, o)$
      6. $\langle L, v \rangle$ = takeTransitions$(o, \mu, t)$
```

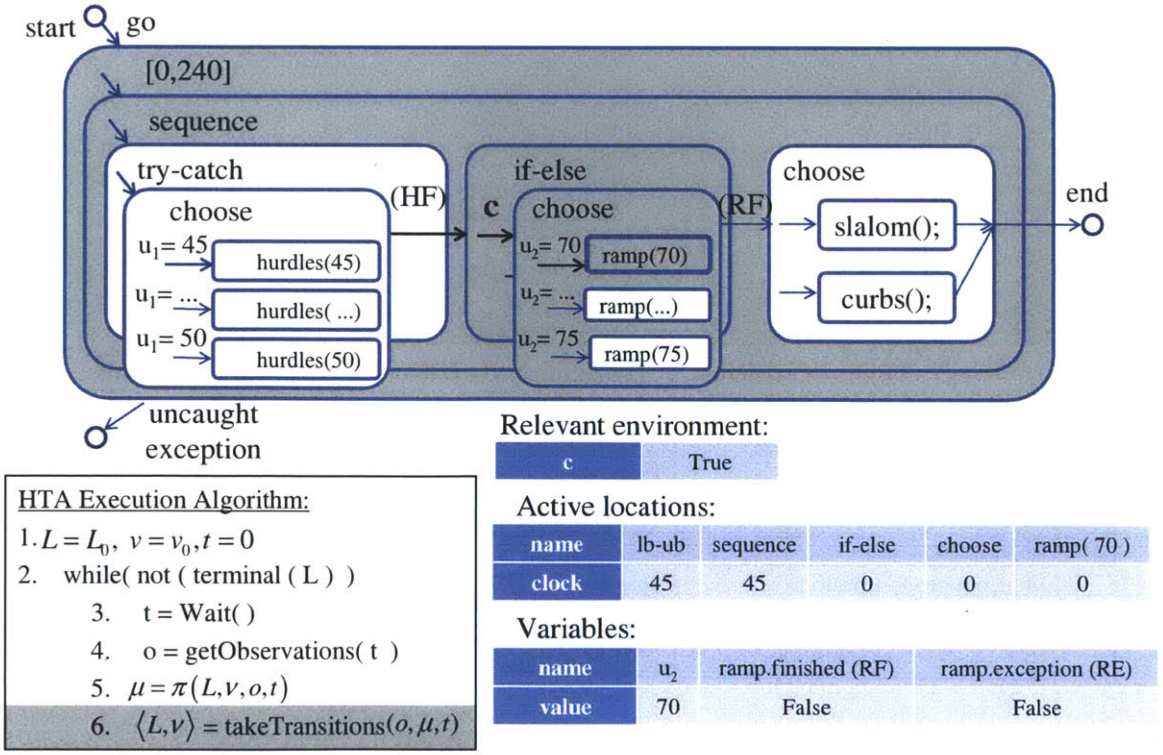
Active locations:

name	lb-ub	sequence	try-catch	choose	hurdles(45)
clock	45	45	45	45	45

Variables:

name	u_1	hurdles.finished (HF)	hurdles.exception (HE)
value	45	**True**	False

Figure 3.10: {c = True} at runtime, and the executive schedules the ramp activity, {u2=70}.



Relevant environment:

c	True

HTA Execution Algorithm:

```
1. $L = L_0,\ v = v_0, t = 0$
2. while( not ( terminal ( L ) )
      3. t = Wait( )
      4. o = getObservations( t )
      5. $\mu = \pi(L, v, o, t)$
      6. $\langle L, v \rangle$ = takeTransitions$(o, \mu, t)$
```

Active locations:

name	lb-ub	sequence	if-else	choose	ramp(70)
clock	45	45	0	0	0

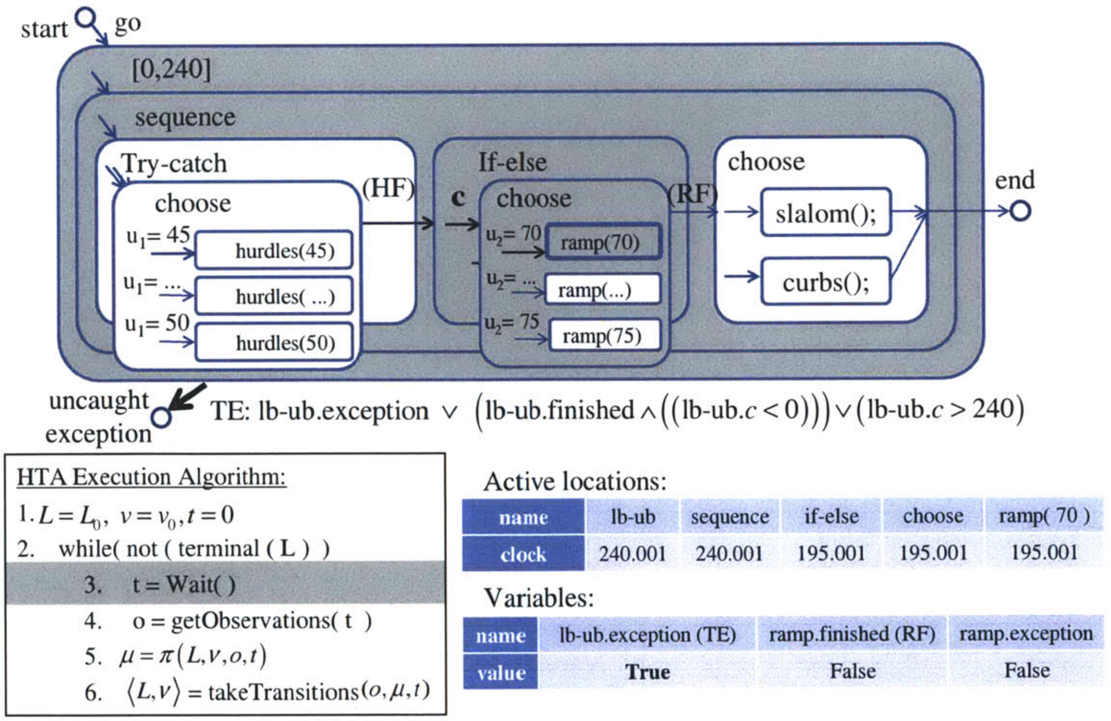
Variables:

name	u_2	ramp.finished (RF)	ramp.exception (RE)
value	70	False	False

Figure 3.11: The HTA transitions to executing the ramp activity.

come to a halt because the two observation variables on the outgoing transitions from the ramp activity, *ramp.finished* and *ramp.exception*, are both currently “False”, as shown in Figure 3.11. The execution algorithm then returns to Step 3 and waits for one of the observation variables to change from “False” to “True”, while the clocks of all active locations progress.

Next, to illustrate an exception, we assume that the ramp activity takes much too long, and the upper bound of the time constraint, [0,240], is violated. As shown in Figure 3.12, the clock time in the lb-ub location progresses to 240.001 enabling the guard on the transition named (TE) from the lb-ub location into the uncaught exception location. This transition indicates that a timing exception has occurred, and the algorithm progresses to Step 5. No control variables are enabled in step 5, and the execution algorithm moves on to Step 6.



TE: lb-ub.exception $\vee$ $\big(\text{lb-ub.finished} \wedge ((\text{lb-ub}.c < 0))\big) \vee (\text{lb-ub}.c > 240)$

HTA Execution Algorithm:

```
1. $L = L_0$, $v = v_0$, $t = 0$
2. while( not ( terminal ( L ) )
      3. t = Wait( )
      4. o = getObservations( t )
      5. $\mu = \pi(L, v, o, t)$
      6. $\langle L, v \rangle$ = takeTransitions($o, \mu, t$)
```

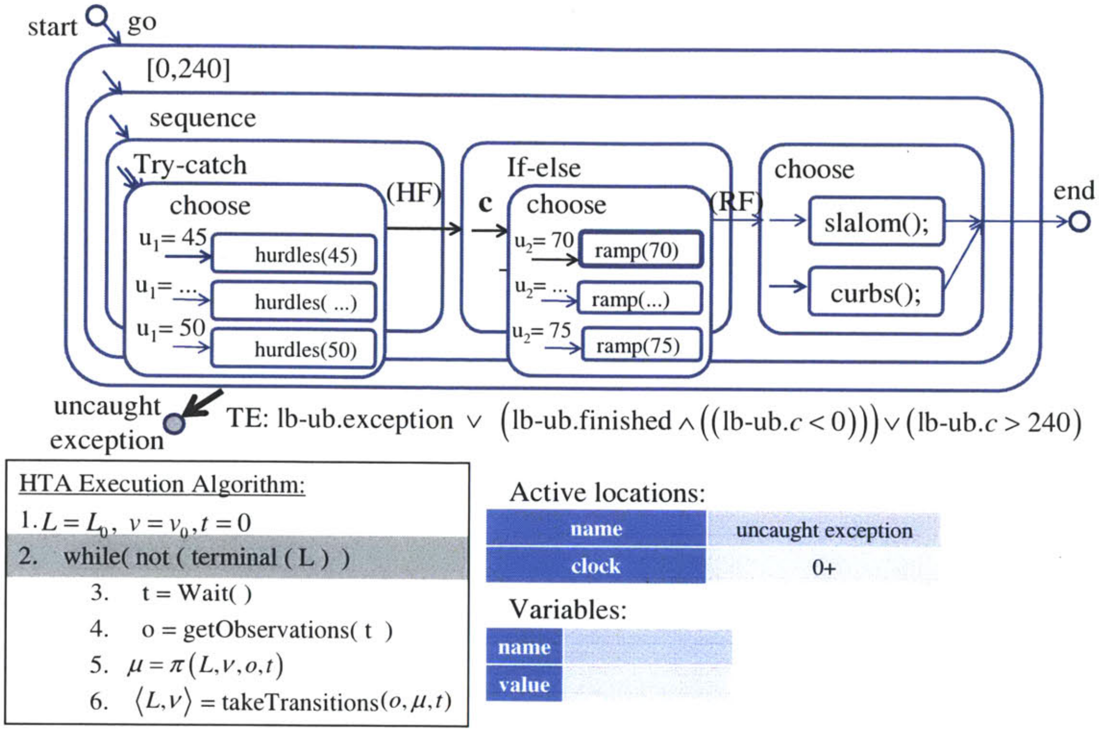
Active locations:

name	lb-ub	sequence	if-else	choose	ramp(70)
clock	240.001	240.001	195.001	195.001	195.001

Variables:

name	lb-ub.exception (TE)	ramp.finished (RF)	ramp.exception
value	**True**	False	False

Figure 3.12: The lb-ub location throws an exception when its clock exceeds 240 seconds.



HTA Execution Algorithm:

```
1. $L = L_0,\ v = v_0, t = 0$
2. while( not ( terminal ( L ) )
      3.  t = Wait( )
      4.  o = getObservations( t )
      5.  $\mu = \pi(L, v, o, t)$
      6.  $\langle L, v \rangle$ = takeTransitions($o, \mu, t$)
```

Active locations:

name	uncaught exception
clock	0+

Variables:

name	
value	

Figure 3.13: The uncaught exception location, a terminal location, is reached, ending execution.

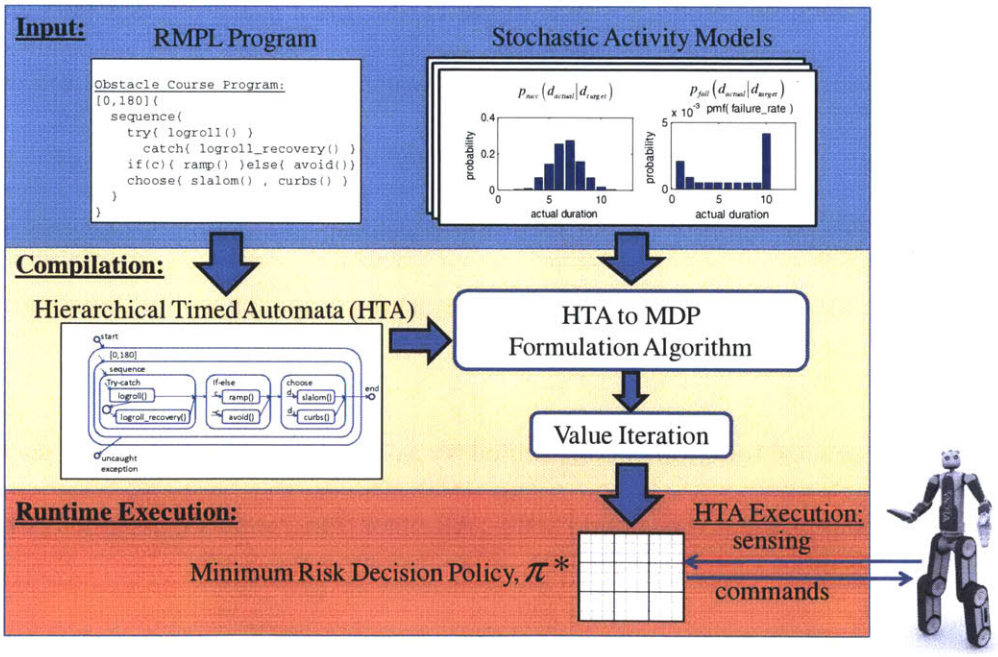
In Step 6, the transition (TE) is taken, as shown in Figure 3.13, the lb-ub location is unmarked, as are all of the descendants of the lb-ub location. This is an example of preemption as described in Section 3.3. When the uncaught-exception location becomes marked, as shown in Figure 3.12, program execution has terminated in failure. This constraint violation exemplifies the type of undesirable program outcome that our executive attempts to avoid. Our goal in this thesis is to construct a decision policy that maximizes the likelihood of successful program execution by minimizing the likelihood of timing constraint violations and activity failures leading to uncaught exceptions, such as this one.

This concludes the HTA execution algorithm walkthrough, as well as Chapter 3. Next, in Chapter 4, we develop a minimum risk execution approach for RMPL programs by formulating an HTA and a set of stochastic activity models into a Markov Decision Process (MDP).

Chapter 4: Minimum-risk Execution of RMPL Programs

4.1 Overview of the Approach

Recall that RMPL program execution corresponds to executing an HTA, and that executing an HTA involves two types of decisions, selecting the target duration for each primitive activity and selecting between subprocedures. Decisions are made by consulting a decision policy at runtime that maps program locations, clock valuations, and observations to a set of control variable assignments. In this chapter, we construct a decision policy that minimizes risk, therefore maximizing the probability of successful execution. Successful execution is defined as any legal HTA execution that does not end with an uncaught exception. As depicted in Figure 4.1, we construct a *minimum risk decision policy* by formulating the problem of *minimum risk HTA execution* as a Markov Decision Process (MDP). The MDP can then be solved by any suitable



```
Obstacle Course Program:
[0,180]{
  sequence(
    try{ logroll() }
      catch{ logroll_recovery() }
    if(c){ ramp() }else{ avoid()}
    choose{ slalom() , curbs() }
  )
}
```

Figure 4.1: Minimum-risk Execution of RMPL Programs

MDP solver. Here, we employ explicit state value iteration. Our contribution in this thesis is intended as the formulation of minimum risk execution as an MDP, rather than an efficient algorithm for solving the MDP.

The HTA to MDP formulation algorithm described in this chapter builds an MDP for HTA execution by simulating transitions in an HTA in a depth-first manner, using a queue and a visited list. For each state in the HTA, the algorithm simulates all possible observations, control actions, and resulting states within a specified time discretization, Δt. Then, the likelihood of each resulting state transition is computed, and a probabilistic transition is added to the MDP.

The HTA to MDP formulation algorithm proceeds in four steps:

1. First, we compute an MDP from the stochastic activity model for each primitive activity.

2. We simulate each possible transition in the HTA. Each transition is simulated using the HTA Execution Algorithm (Algorithm 3.1), which determines the resulting state from a given starting state, and an assignment to observation variables and control variables in the HTA. The probability of the resulting transition is computed, and the transition is added to the MDP. A queue and visited list ensure completeness and uniqueness.

3. The MDP's reward function is constructed by assigning a reward of one to each transition whose end state is the program end state. All other transitions are assigned zero reward.

4. The resulting MDP is solved using explicit state value iteration [Bellman 1960], resulting in a minimum risk (i.e. optimal) decision policy.

The minimum risk decision policy is then provided as input to the HTA Execution Algorithm (Algorithm 3.1), resulting in a minimum risk HTA execution (i.e. an execution that maximizes the probability of success). Meanwhile, the minimum risk value function computes the maximum likelihood of successful execution from all states in an HTA, given the minimum risk decision policy.

Roadmap for Chapter 4. First, in Section 4.2 we define *minimum risk HTA execution*, with the help of two new concepts, *minimum risk decision policy* and *minimum risk value function.* In Section 4.3, we introduce the stochastic activity model. Next, in Section 4.4, we develop the HTA to MDP Formulation Algorithm, which formulates an MDP from an HTA and a set of stochastic activity models. In Section 4.4, we give some examples of HTA to MDP formulation. Finally, in Section 4.5, we give a chapter summary.

4.2 Minimum Risk HTA Execution

In this section, we begin by defining successful and failed HTA execution. Then, we introduce two new concepts, *minimum risk decision policy* and *minimum risk value function.* Finally, we formally define *minimum risk HTA execution* as any legal HTA execution that adheres to the minimum risk decision policy.

Definition 4.1: Successful HTA Execution

A *successful HTA execution* is as any legal HTA execution (defined in Section 3.4) that ends in the HTA's end state.

Definition 4.2: Failed HTA Execution

A *failed HTA execution* is any legal HTA execution (defined in Section 3.4) that ends in the HTA's uncaught exception state.

Definition 4.3: Minimum Risk Decision Policy, π^*

The *minimum risk decision policy*, π^*, is the decision policy (Definition 3.5) that maximizes the HTA value function, $V(\pi)$: $\pi^* = \left\{\pi : \arg\max\left(V\left(\pi\right)\right)\right\}$

Definition 4.4: Minimum Risk Value Function, V^*

The minimum risk value function, $V^* = V\left(\pi^*\right)$, is the value function that results by following the minimum risk decision policy, π^*, from all states in an HTA.

Definition 4.5: Minimum Risk HTA Execution

Finally, a *minimum risk HTA execution* is defined as any legal HTA execution (defined in Section 3.4) that adheres to the minimum risk decision policy (Definition 4.3).

4.3 Stochastic Activity Model

In this section, we introduce the stochastic activity model used to determine the success probability of an execution. Recall that a set of stochastic activity models is provided as input to the executive, one for each primitive activity. A stochastic activity model characterizes the timing behavior of each primitive activity in the RMPL program.

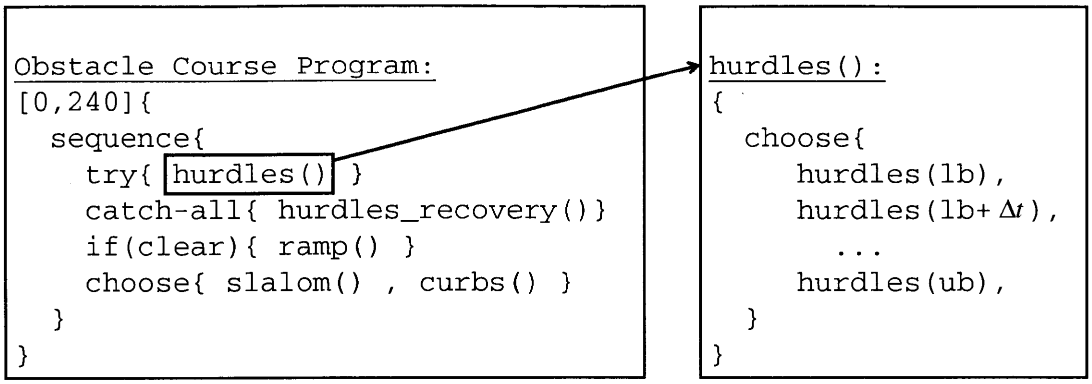
In our approach to program execution, an executive controls when a primitive activity finishes, by scheduling that activity's intended duration. However, there will inevitably be uncertainty in an activity's actual duration and its success or failure. A stochastic activity model characterizes this uncertainty along two separate dimensions: stochastic duration and stochastic success. Definition 4.8 gives a formal definition of the stochastic activity model.

Definition 4.8: Stochastic Activity Model

A *stochastic activity model* is a 6-tuple $\langle name, STC, \Delta t, P_{fail}, d_{success}, d_{fail} \rangle$,

- ***name*** is a unique symbol.
- ***STC*** enforces a time bound on the scheduled activity duration, $dur_{intended} \in [lb, ub]$.
- Δt, a time discretization, indicates a discrete set of target durations within the range lb to ub that the activity may be scheduled to complete, $dur_{intended} \in \{lb, lb+\Delta t, lb+2*\Delta t, \cdots, ub\}$.
- $P_{fail,\,i} \in [0,1]$, specifies a probability that the activity will fail to accomplish its intended function, which depends on the intended duration, *i*, where $i \in \{lb, lb+\Delta t, \cdots, ub\}$.
- $d_{success,\,i}(t) = p\left(dur_{actual} = t \middle| success, dur_{intended} = i\right)$ specifies a probability distribution over actual activity duration, *t*, when the activity succeeds, for each intended duration, *i*.
- $d_{fail,\,i}(t) = p\left(dur_{actual} = t \middle| fail, dur_{\text{int}ended} = i\right)$ specifies a probability distribution over actual activity duration when the activity fails, for each intended duration, *i*.

The first element of a stochastic activity model is simply its unique identifier, a *name*. The next two elements consist of a flexible bound on schedulable activity duration, $[lb, ub]$, and a time discretization, Δt. An activity is scheduled at runtime by choosing an intended duration within the range lb to ub at discrete intervals of Δt: $dur_{intended} \in \{lb, lb+\Delta t, lb+2*\Delta t, \cdots, ub\}$. As shown in Figure 4.2, if a primitive activity is called with no parameter for the intended duration, then the executive chooses the intended duration at runtime via the *choose* construct. Each primitive activity in an RMPL program adheres to this convention, enabling dynamic, runtime activity scheduling of RMPL programs.



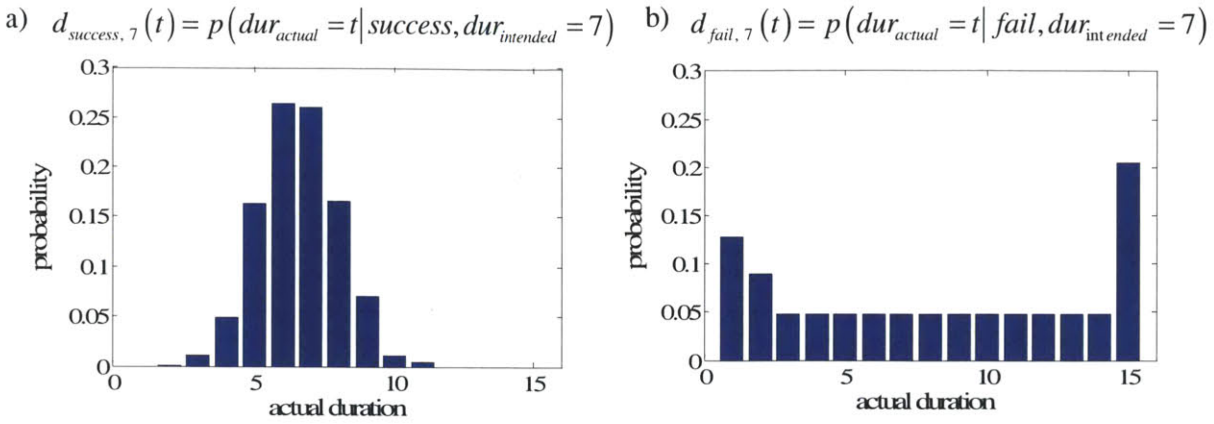
```
Obstacle Course Program:
[0,240]{
  sequence{
    try{ hurdles() }
    catch-all{ hurdles_recovery()}
    if(clear){ ramp() }
    choose{ slalom() , curbs() }
  }
}
```

```
hurdles():
{
  choose{
     hurdles(lb),
     hurdles(lb+Δt),
       ...
     hurdles(ub),
  }
}
```

Figure 4.2: A primitive activity called in RMPL with no parameter for intended duration.

The remaining three elements in the stochastic activity model, $P_{fail,i}$, $d_{success,i}$, and $d_{fail,i}$, characterize the uncertainty inherent in executing activities in robotic domains, and in general they may depend on the choice for intended duration, *i*. For example, an activity scheduled to complete as quickly as possible, $i = lb$, might have a higher overall likelihood of failure, $P_{fail,i}$, than an activity scheduled with the maximum allowable duration, $i = ub$. Also, $d_{success,i}$ will likely exhibit a large amount of probability mass concentrated near the activity's intended duration, *i*. $P_{fail,i}$, $d_{success,i}$, and $d_{fail,i}$ can be learned empirically, or estimated using common sense. An example of a stochastic activity model is shown in Figure 4.3. Figure 4.3a depicts the probability that an activity will end at time *t*, given that it succeeds, while Figure 4.3b depicts the probability that an activity will end at time *t*, given that it fails. Meanwhile, $P_{fail,i}$records the overall likelihood of activity failure, providing *a priori*, a relative weighting over the two distributions, $d_{success,i}$, and $d_{fail,i}$. For each choice of intended duration, *i*, the distributions $d_{success,i}$, and $d_{fail,i}$, when weighted by the overall likelihood of success and failure, $\left(1-P_{fail,i}\right)$ and $P_{fail,i}$, respectively, must sum to 1:

$$\forall i \in \{lb, lb+\Delta t, lb+2*\Delta t, \cdots, ub\}, \quad \sum_t \left(d_{success,\,i}(t)*\left(1-P_{fail,\,i}\right)+d_{fail,\,i}(t)*P_{fail,\,i}\right) \;= 1.$$



a) $d_{success,7}(t) = p\left(dur_{actual} = t \middle| success, dur_{intended} = 7\right)$ b) $d_{fail,7}(t) = p\left(dur_{actual} = t \middle| fail, dur_{intended} = 7\right)$

Figure 4.3: An example stochastic activity model with schedulable bound of [5,10], $\Delta t = 1$, $\mathrm{dur}_{intended} = 7$, and $P_{fail,7} = 0.3$. a) When the activity succeeds, the actual duration is modeled as a Gaussian distribution with a variance of 2, and mean centered on the choice for intended duration, $dur_{intended} = 7$. b) When the activity fails, it is assumed to fail close to uniformly, with a slightly higher probability of failure just after invocation, and at 15 seconds, indicating that the activity automatically times out after 15 seconds.

4.4 HTA to MDP Formulation Algorithm

In this section, we present the HTA to MDP formulation algorithm. We start by explaining its general intuition, and then we present the high level pseudocode. After that, we describe each step of the algorithm in more detail.

As depicted in Figure 4.4, the overall approach is to translate the elements of an HTA into corresponding elements of an MDP. Recall the Markov property, which states that a transition into a future state may depend only on the current state and action taken in that state. To satisfy this property, we require each state in the MDP to correspond to a unique state in the HTA, corresponding to a unique stack in the RMPL program. We achieve this by recording all realizable combinations of active locations, clock valuations, and observable variables in an

HTA. An HTA's control variables are translated directly into actions in the MDP, and the MDP transition probabilities are derived from the stochastic activity models.

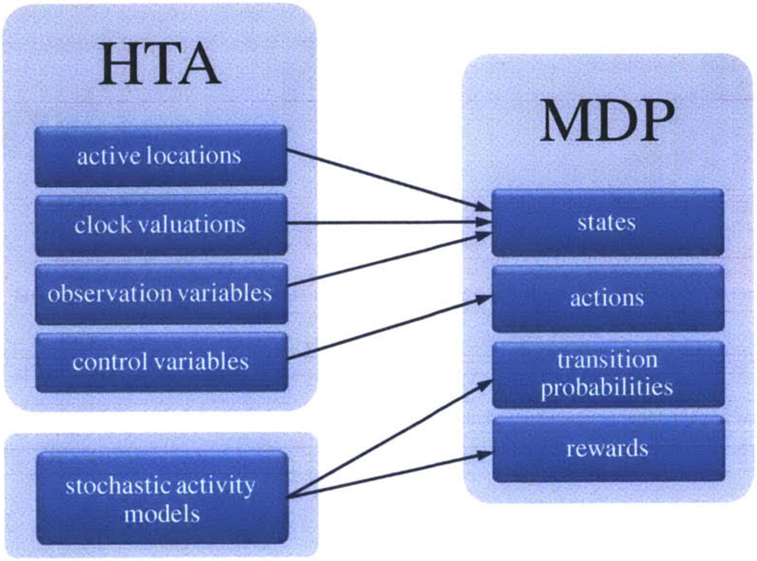


Figure 4.4: Translation of HTA elements into MDP elements. MDP states consist of realizable combinations of marked locations, clock valuations, and observable variables, in an HTA. MDP actions correspond to HTA observable variables, while the stochastic activity models determine transition probabilities and rewards.

The high-level pseudocode for the HTA to MDP Algorithm is presented below in Algorithm 4.1. The algorithm takes as input an HTA, a set of stochastic activity models (SAMs), and a time discretization, Δt. The output is an MDP, the solution of which constitutes the minimum risk (i.e. optimal) decision policy for executing an HTA.

Algorithm 4.1: HTA to MDP Algorithm

Inputs: *HTA*, a Hierarchical Timed Automata,
$sam_{1 \cdots N}$, a set of Stochastic Activity Models
Δt, a time discretization
Output: *MDP*, a Markov Decision Process

```
function HTA_to_MDP $(HTA, sam_{1...N}, \Delta t)$
for each $sam_i, i = 1 \cdots N$
    $MDP_{prims,1 \cdots N}$ = ComputePrimitiveMDPs $(sam_{1 \cdots N})$
$Q$.push$(s_0)$                                   // Push the HTA start state, $s_0$ onto $Q$
$MDP$.addState($s_0$), VisitedList $\leftarrow s_0$        // Add $s_0$ to $MDP$ and Visited List
while $\neg(\text{empty}(Q))$
    $s = Q.pop(\,)$                                // where, $s = \langle L, v \rangle$
    $O$ = get Relevant Observations $(s, HTA)$
    for each $o \in O$
        $U$ = get Relevant ControlActions $(s, o, HTA)$
        for each $u \in U$
            $s'$ = takeTransitions $(s, o, u, HTA)$        // where $s' = \langle L', v' \rangle$
            $p$ = computeJointProbability$(s, o, u, s', MDP_{prims})$
            $P(s, u, s') = p$                      // add transition to the MDP
            if $\neg$Visited$(s')$
                $Q$.push$(s')$, VisitedList $\leftarrow s'$
for each transition in the MDP
    if s' = program end state, $R(s, s') = 1$       // reward transitions into the end state
return MDP
```

Algorithm Description:

First, an MDP is constructed for each primitive activity from its stochastic activity model. The set of MDPs for each primitive are referred to collectively as MDP_{prims}.

Then, each state in the HTA is explored in a depth-first manner to simulate all HTA transitions in order to infer all transitions between the end of one activity and the start of another. Each transition is simulated as follows. First, *getRelevantObservations(s,HTA)* (Step 7) returns all observable variables that will influence execution when transitioning from a state, *s*. Then for each combination of relevant observations, *getRelevantControlActions(s,o,HTA)* (Step 9) returns all control variables that will influence execution when transitioning from a state, *s*, given the observations, *o*.

Next, given a set of relevant observations, *o*, and relevant control actions, *u*, which affect execution from a state, *s*, the resulting state, *s'*, is computed via the function *takeTransitions(s,o,u,HTA)*. This function is the same as Step 6 in the HTA Execution Algorithm (Algorithm 3.1) from Chapter 3, which traverses enabled transitions marking and unmarking locations until no more transitions are enabled. Finally, the probability of the resulting transition is computed via *computeJointProbability(s,o,u,s',*MDP_{prims}*)*, and the transition is added to the MDP. This function determines the probability of each transition taken in the HTA between two states *s* and *s'*, and then returns their product, which represents the joint probability of the overall transition from state *s* to *s'*.

The last step of the algorithm computes a reward function for the MDP, $R(s,s')$, which rewards successful program execution. A reward of one is assigned to transitions that end in the program end state, while all other transitions are assigned a reward of zero.

The resulting MDP is then solved using value iteration [Bellman 1960], resulting in a minimum risk (i.e. optimal) decision policy.

Line 2: Computing an MDP from a Stochastic Activity Model

Next, we demonstrate line 2 in the HTA to MDP algorithm; how to compute an MDP from a stochastic activity model. This function call, *ComputePrimitiveMDPs()*, occurs once for each primitive activity in an HTA.

As depicted in Figure 4.5a, a stochastic activity model consists of a clock that varies from 0 to *n* (where *n* is the maximum possible duration of the activity), and two observation variables, $\mathit{primitive}_{dur}.\mathit{finished}$ and $\mathit{primitive}_{dur}.\mathit{exception}$. Taking the Cartesian product over these elements leads to the set of states in a stochastic activity model, shown in Figure 4.5b. Note the shorthand denotation for the two observations, $\mathit{primitive}_{dur}.\mathit{finished}$ and $\mathit{primitive}_{dur}.\mathit{exception}$.

Next, the transition probabilities that are provided in a stochastic activity model are depicted graphically in Figure 4.6b. They provide the *a priori* probability that an activity will finish successfully or throw an exception at each timestep, Δt. These probabilities are convenient to record experimentally, and are natural to estimate. However, to encode them as an MDP, they need to be reformulated as a Markovian transition relation. The probabilities we need in order to construct an MDP are depicted graphically in Figure 4.7a.

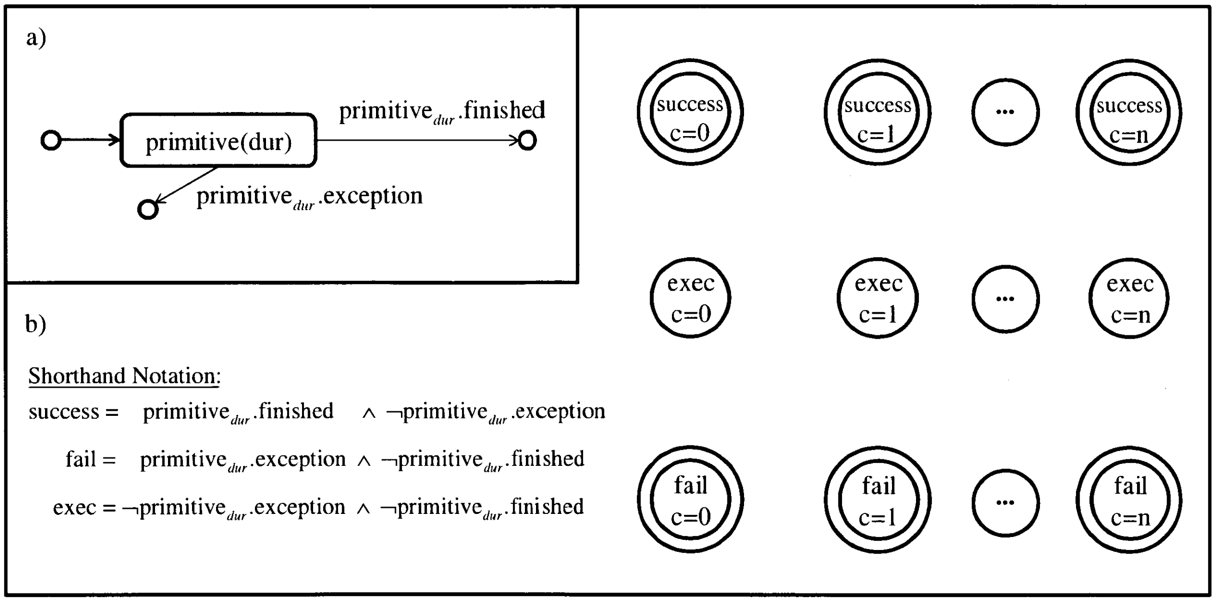

a)
primitive(dur)
primitive_{dur}.finished
primitive_{dur}.exception
b)
Shorthand Notation:
success = $\text{primitive}_{dur}.\text{finished} \wedge \neg\text{primitive}_{dur}.\text{exception}$
fail = $\text{primitive}_{dur}.\text{exception} \wedge \neg\text{primitive}_{dur}.\text{finished}$
exec = $\neg\text{primitive}_{dur}.\text{exception} \wedge \neg\text{primitive}_{dur}.\text{finished}$
success c=0
success c=1
...
success c=n
exec c=0
exec c=1
...
exec c=n
fail c=0
fail c=1
...
fail c=n

Figure 4.5 a) The HTA for a primitive activity, b) MDP states consisting of the Cartesian product of a single HTA location, clock values 0…*n*, and two observations, *finished* and *exception.*

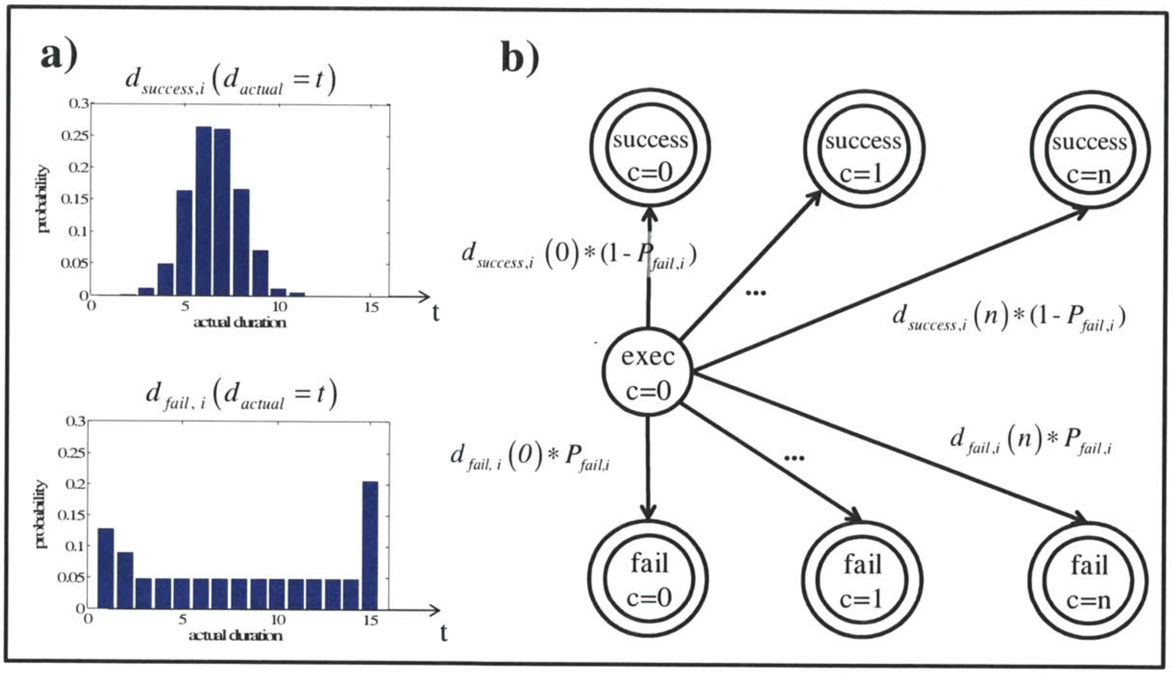
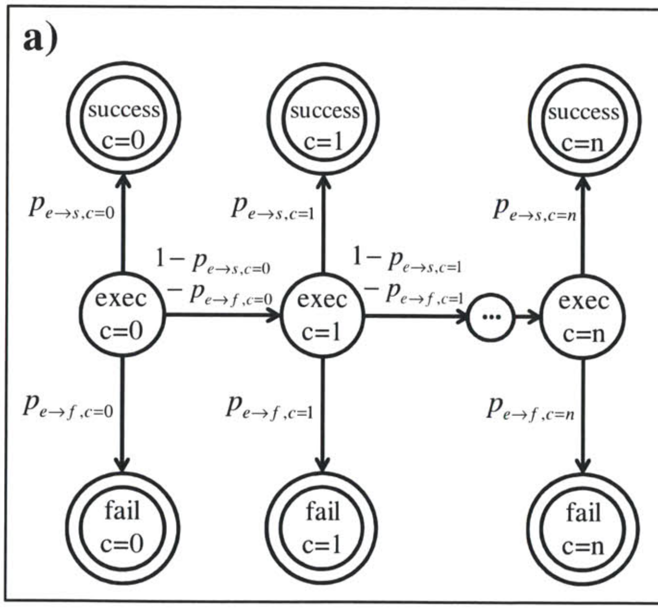


Figure 4.6 a) An example of a stochastic activity model, $d_{success,i}$ and $d_{fail,i}$, with intended duration, $i=7$, b) $d_{success,i}$ and $d_{fail,i}$, depicted graphically



b) for $c=0$, $i=$ intended duration

$$p_{e\to s,c=0}=d_{success,i}(0)*(1-P_{fail,i})$$

$$p_{e\to f,c=0}=d_{fail,i}(0)*P_{fail,i}$$

for $c>0$, $i=$ intended duration

$$p_{e\to s,c=t}=\frac{d_{success,i}(t)*(1-P_{fail,i})}{\prod_{j=1}^{t-1}\left(1-p_{e\to s,c=j}-p_{e\to f,c=j}\right)}$$

$$p_{e\to f,c=t}=\frac{d_{fail,i}(t)*(P_{fail,i})}{\prod_{j=0}^{t-1}\left(1-p_{e\to s,c=j}-p_{e\to f,c=j}\right)}$$

Figure 4.7 a) Markovian transition probabilities for a primitive activity, $p_{e\to s,c=0\cdots n}$ and $p_{e\to f,c=0\cdots n}$, in contrast to the stochastic activity model probabilities depicted in Figure 4.6. b) Equations that translate from the stochastic activity model probabilities to MDP transition probabilities.

The key distinction between the transition probabilities in Figure 4.6 and those in Figure 4.7, is the assumption on the current clock value. A stochastic activity model, Figure 4.6, tells us the *a priori* probability of success and failure at each timestep before an activity begins to execute, i.e. when $c = 0$. As the activity begins to execute, however, as depicted in Figure 4.7, we must account for the information learned at runtime, namely, that the probability of success and failure prior to the current clock value, $c = t$, have dropped to zero. The equations that account for this learned information are presented in Figure 4.7b. The probabilities we compute between each pair of states in Figure 4.7 are useful because they satisfy the Markov property at each timestep, enabling us to interleave the MDPs of concurrently executing primitive activities into a single MDP, and to compute their joint probabilities.

Line 7: Get Relevant Observations

The $\text{getRelevantObservations}(s, HTA)$ function computes the cross product over observable variables, $O = \{o_1 \times \cdots \times o_d\}$, that affect execution from a state, s. Examples of relevant observable variables from a state, s, include any *finished* and *exception* observable variables associated with marked primitive locations in state s, as well as any observable variables, *obsv*, associated with if(*obsv*){ }else{ } constructs reachable from state s.

Line 9: Get Relevant Control Actions

The $\text{getRelevantControlActions}(s, o, HTA)$ function computes the cross product over control variables that affect execution for a state s, given a specific set of relevant observations, o. The assignments to relevant control variables and observable variables determines the next state, s'.

Line 17: Create Rewards

Transitions from any state into the program end state are assigned a reward of one, while all other transitions are assigned a reward of zero. No discount factor is assumed. This choice of reward function results in a value function that computes the likelihood of success from each program state. This statement can be proven inductively as follows. The MDP has two absorbing states, the program end state and the uncaught exception state. Consider a state, s_1, with an outgoing transition to the program end state with probability 1. Value iteration will compute a value of 1 for state s_1, since the reward of 1 for transitioning to the end state is guaranteed. Next, consider a second state, s_2, which transitions stochastically with probability p to state s_1, and with probability $1\text{-}p$ to the uncaught exception state. Value iteration will compute a value of $1 * p + 0 * (1 - p) = p$ for this state, which is precisely the probability of reaching the program end state. This argument can be repeated inductively.

Solving the MDP using Value Iteration, resulting in a Minimum Risk Decision Policy

After the MDP is constructed, it can be solved using Value Iteration, resulting in a Minimum Risk Decision Policy (Definition 4.3). This decision policy is then used to make decisions at runtime that minimize risk. Because the MDP is computed over discrete intervals of Δt, the values of each location's clock variable are rounded down to the nearest multiple of Δt at runtime, in order to find the minimum risk decision.

4.5 Examples of the Algorithm

Next, we give an example of how the algorithm formulates each type of construct into an MDP.

How to Formulate a Choose Location into an MDP

First we demonstrate how the HTA to MDP algorithm formulates a choose location into an MDP. Each choice in an HTA is encoded as an action in the MDP. The choose location depicted in Figure 4.8 schedules the intended duration of a primitive activity, *name*( ). A choice, u_c, exists in which the subprogram, *name*( ), is invoked with intended durations between [lb,ub], at increments of Δt. To encode this choose location as an MDP, a start state, s_{choose}, is added to the MDP, and then an action is added for each intended duration, $u_c = \{lb, lb+\Delta t, \cdots, ub\}$. Each action results in a transition into an MDP consisting of the primitive activity's Markovian transition probabilities as computed in the Figure 4.7.

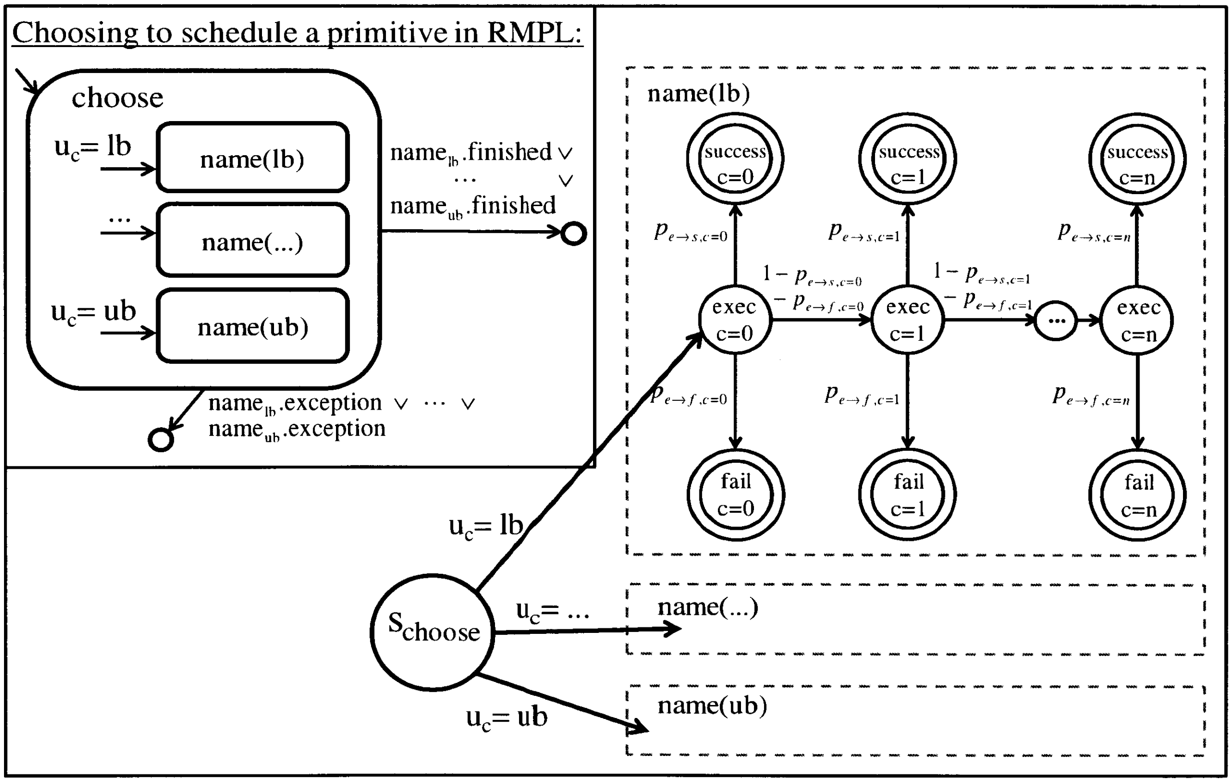


Figure 4.8: Formulating a choose location into MDP.

How to Formulate a Try-Catch Location into an MDP

Here, we demonstrate how the HTA to MDP algorithm formulates a try-catch location into an MDP. First, the try-catch start state is connected to the Markovian transition probabilities for the nominal activity.

Successful execution of the nominal MDP indicates program success, while exception states in the nominal MDP transition to the handler MDP start states, depending on the handler conditions, e_i. Successful executions of the handler MDPs also indicate program success, while an exception in a handler state results in failure.

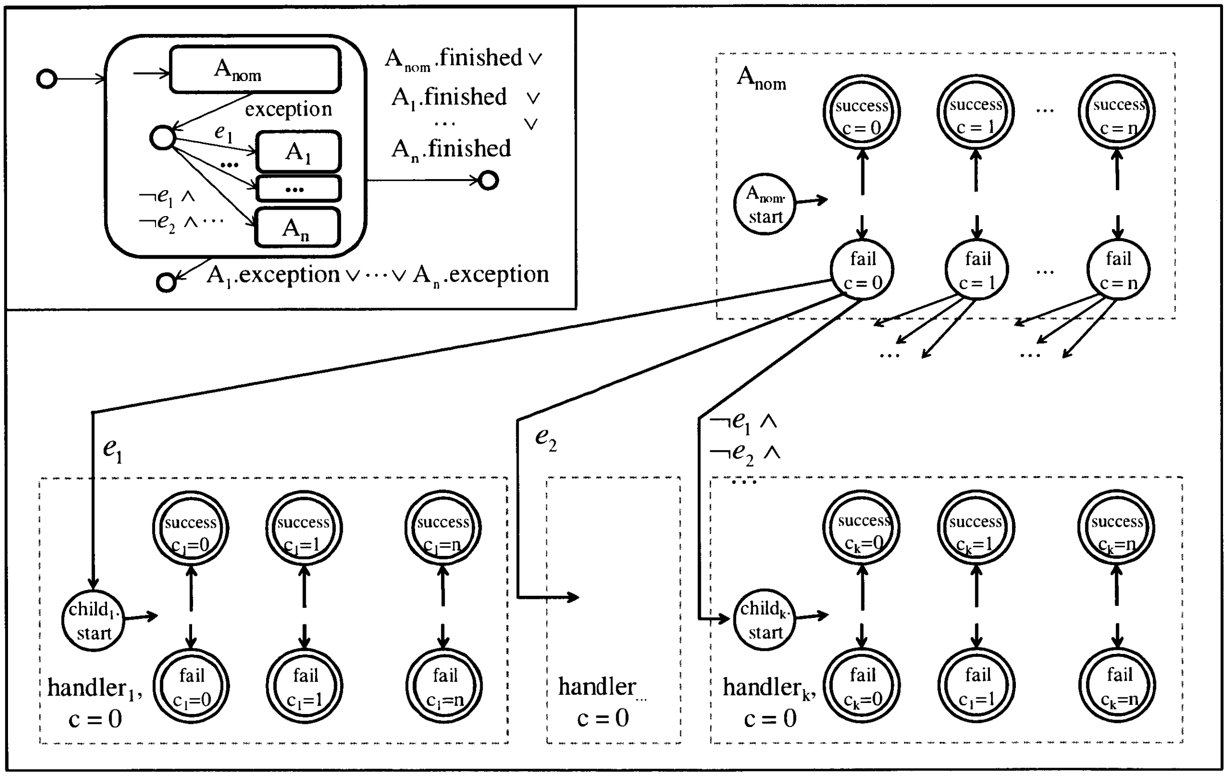


Figure 4.9: Formulating a Try-catch location into an MDP.

How to Formulate an If-Else Location into an MDP

Next, we demonstrate how the HTA to MDP algorithm formulates an if-else location, if(*obsv*){ }else{ }, into an MDP. The runtime observation, *obsv*, is encoded as a stochastic transition in the MDP. The transition to the "if" MDP is taken with probability, p_{obsv}, and the transition to the "else" branch MDP is taken with probability, $1-p_{obsv}$. An example is depicted in Figure 4.10.

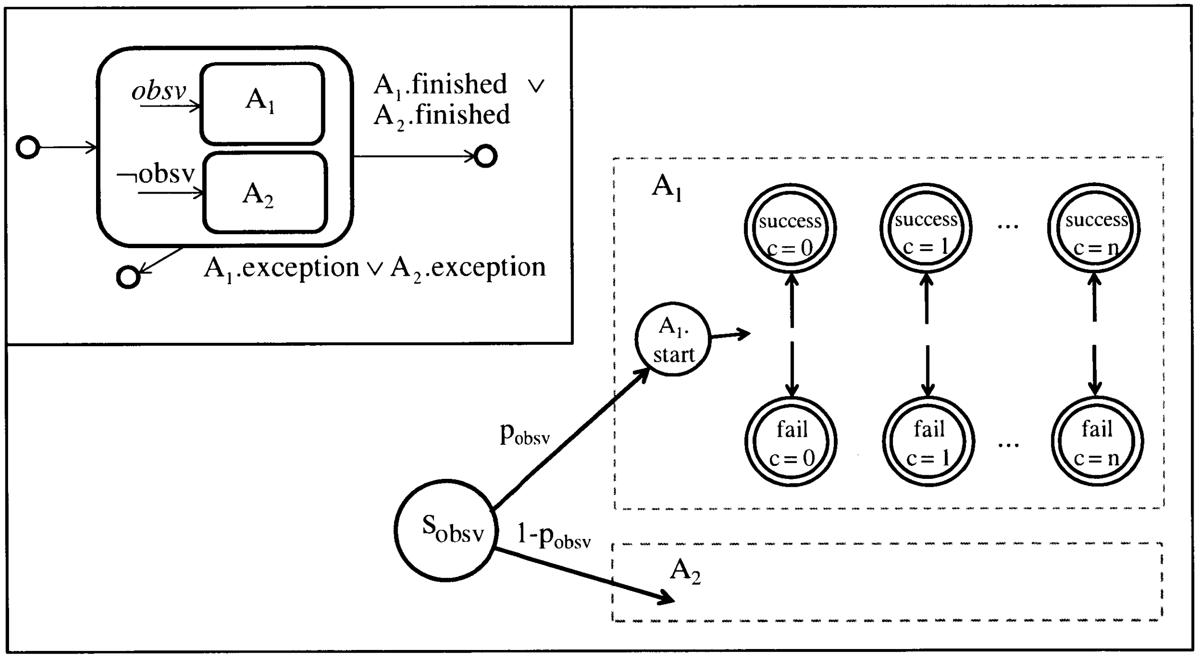


Figure 4.10: Formulating an If-Else Location into an MDP.

How to Formulate a Sequence Location into an MDP

A sequence location is reformulated into an MDP as depicted in Figure 4.11. Successful execution of each child MDP results in a transition to the start state of the next child MDP. Successful execution of the last child MDP results in a transition to a program success state. If any child MDP transitions into an exception state, then program execution has failed. In a larger program, a transition into an exception state must be to either the program uncaught exception state, or to the handler MDP of a try-catch location that exists further up the HTA hierarchy.

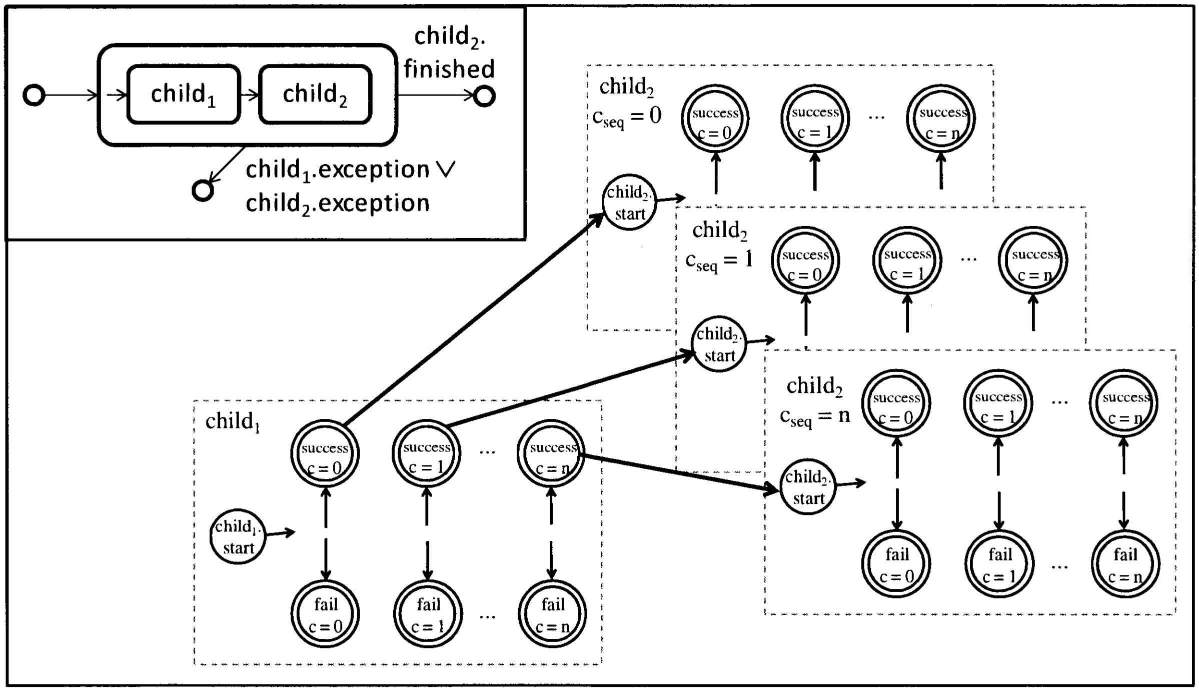


Figure 4.11: Formulating a sequence location into an MDP.

How to Formulate a Lb-Ub Location into an MDP

Next, we demonstrate how the HTA to MDP algorithm encodes a metric timing constraint, [lb,ub], along with an encapsulated subprogram into an MDP. Depicted in Figure 4.12, a successful execution of the child MDP that ends before the minimum duration, *lb*, transitions into a failure state in the time-constrained MDP. Similarly, a successful execution of the child MDP that ends after the maximum duration, *ub*, transitions to a failed state in the time-constrained MDP.

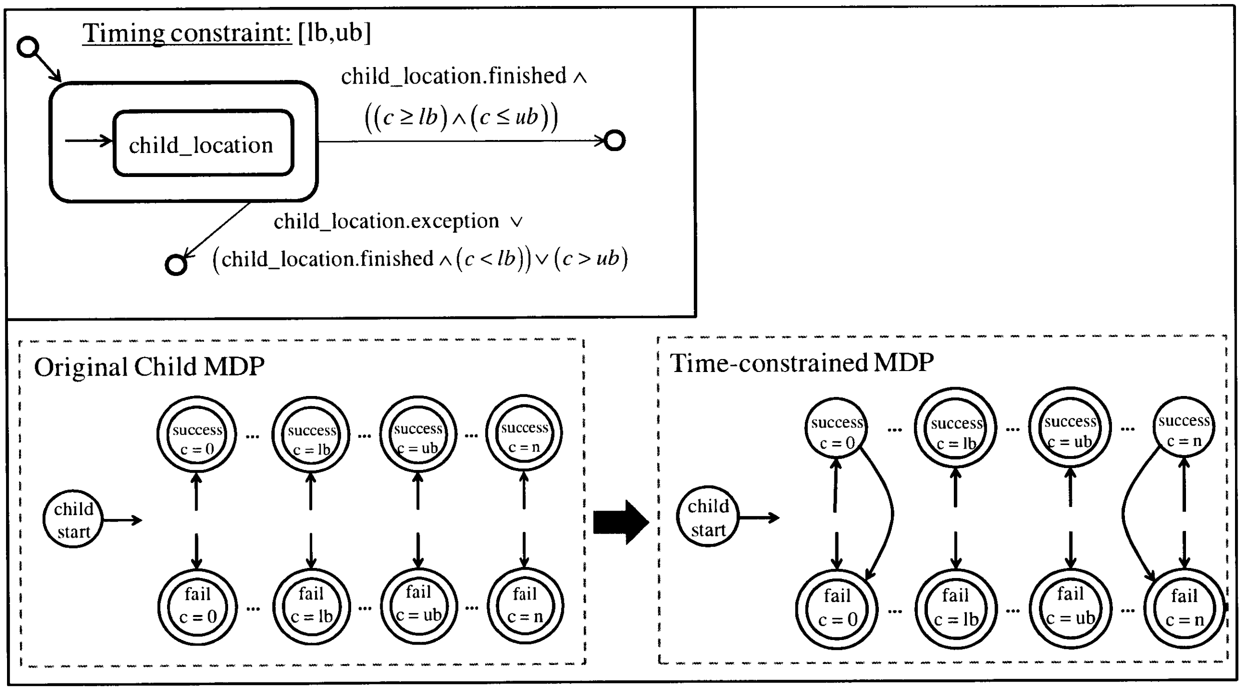


Figure 4.12: Formulating an Lb-Ub location into an MDP.

How to Formulate a Parallel Location into an MDP

A Parallel location is formulated into an MDP as depicted in Figure 4.13. First, a parallel start state is created, $\{s, c = i\}$. For each state in an MDP, *s*, denotes the set of marked descendant states within each child location, as well as the clock value of each marked state, while $c = i$ denotes the clock value of the parallel location itself. Transitions are constructed in two phases at each timestep. First, the MDP branches over the cross product of sensory observations, $j \in \{o_1 \times \cdots \times o_d\}$, available in each state $\{s, c = i\}$. Then, for each possible combination of sensory observations, *j*, the MDP branches over the cross product of control actions, $k \in \{u_1 \times \cdots \times u_d\}$, available in each state $\{s, c = i, j\}$. A parallel location finishes in success when all children finish in success, and alternatively, finishes in failure when any of the children fail.

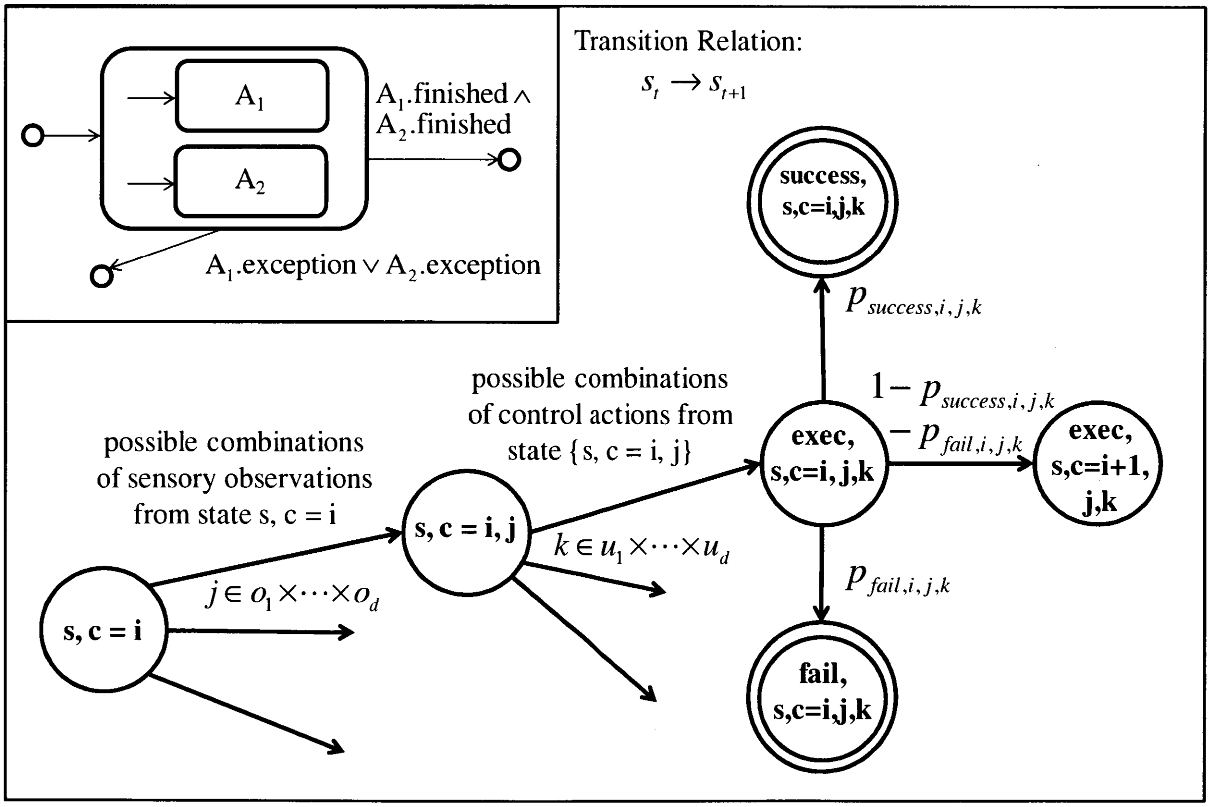
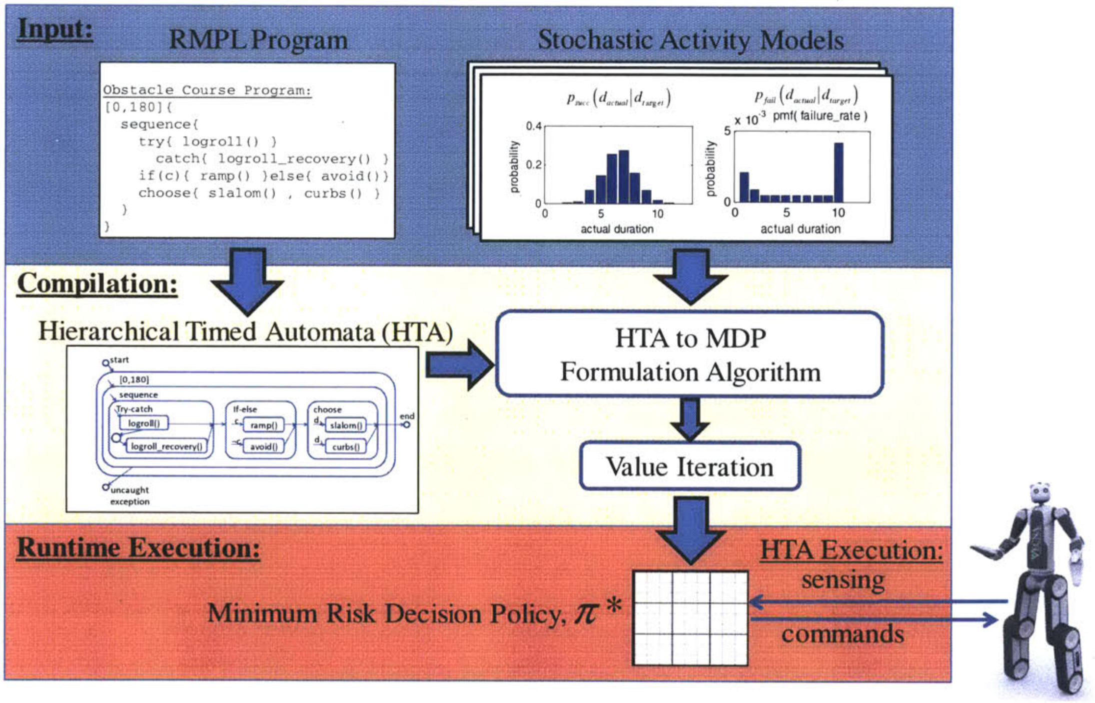


Figure 4.13: Formulating a parallel location into an MDP.

4.6 Chapter Summary - Minimum-risk Execution of RMPL Programs

We summarize the minimum-risk execution approach as follows: An RMPL program and a set of stochastic activity models are formulated into an MDP, and then a minimum risk decision policy is computed offline. The HTA execution algorithm then uses the minimum risk decision policy to make decisions at runtime that maximize the probability of successful program execution. The primary drawback of the exact probabilistic approach is its poor scalability. In Chapters 5 and 6, we explore two methods to improve algorithm scalability; set-bounded dynamic execution and risk-minimizing program execution with limited coupling, respectively.



```
Obstacle Course Program:
[0,180]{
  sequence{
    try{ logroll() }
      catch{ logroll_recovery() }
    if(c){ ramp() }else{ avoid()}
    choose{ slalom() , curbs() }
  }
}
```

Figure 4.14: Minimum-risk Execution of RMPL Programs

Chapter 5: Set-bounded Dynamic Execution of RMPL Programs

In this chapter, we develop a set-bounded dynamic execution algorithm for RMPL programs. This algorithm achieves efficiency over the minimum-risk execution algorithm (Chapter 4) by considering only program states from which execution is guaranteed to succeed. The primary limitation of the set-bounded approach is that often within a program, there will be many program states from which execution is likely to succeed, but not guaranteed. Since a set-bounded approach has no notion of probability, proving correct execution is an all or nothing proposition; either all executions end in the program end state, or no guarantee can be made.

Set-bounded dynamic execution leverages prior work in temporal planning; building upon a technique called *dynamic controllability* [Morris et. al., 2001]. Dynamic controllability is a guarantee that program execution will succeed. Section 5.1 gives an overview of the set-bounded approac6h. In Section 5.2 we present the set-bounded activity model. In Section 5.3 we introduce a novel set-bounded encoding, called a Temporal Plan Network with Uncertainty (TPNU), which enables us to reason over nondeterministic timed RMPL programs with set-bounded activities. In Section 5.4, we describe TPNU execution, and in Section 5.5 we present the mapping from RMPL to TPNU. Section 5.6 develops a dynamic controllability algorithm for RMPL programs. Finally, Section 5.7 discusses related work on dynamic controllability.

5.1 Overview of the Set-bounded Dynamic Execution Approach

In this chapter we develop a set-bounded algorithm to determine dynamic controllability of RMPL programs. Our approach is inspired by the approach taken in [Tsamardinos, et., al., 1998] to determine dynamic controllability of Conditional Temporal Problems (CTPs). However, our definition of *successful execution* differs from prior work. In RMPL, constraint violations (which result in a thrown exception) are allowed as long as the resulting exception is 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 temporal consistency with the rest of the program. Each handler must be guaranteed to have time to complete in response to caught exceptions, or the program is identified as uncontrollable. This approach tends to overconstrain programs 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.

5.2 Set-bounded Activity Model

In this section, we present a set-bounded activity model which is then used by the set-bounded execution algorithm to guarantee successful program execution. A set-bounded activity model specifies a lower and upper bound on task duration, lb and ub, respectively. An activity is modeled either as controllable or uncontrollable, denoted respectively, $[lb,ub]$ and $[lb?ub]$. The $[lb,ub]$ notation represents a controllable time-window between the lower bound, lb, and upper bound, ub, within which the executive may choose to schedule the duration of an activity. Alternatively, the $[lb?ub]$ notation represents an uncontrollable time-window, within which the executive must observe the activity duration at runtime, but has no control over the outcome.

Definition 5.1: Set-bounded Activity Model

A *Set-bounded activity* is a 4-tuple $\langle id, n_i, n_j, STC, e \rangle$.

- ***id*** is a unique identifier to name the activity

- $\boldsymbol{n_i}$ is a time event representing the start of the activity

- $\boldsymbol{n_j}$ is a time event representing the end of the activity

- ***STC*** enforces a flexible timing constraint bounding the duration of the activity:

  - $[lb, ub]$ - A controllable metric timing constraint

  - $[lb?ub]$ – An uncontrollable metric timing constraint

- ***e*** is a set $\{e_{lb}, e_{ub}, e_1, e_2, \cdots, e_n\}$, of exceptions throwable by a task. By default these are:

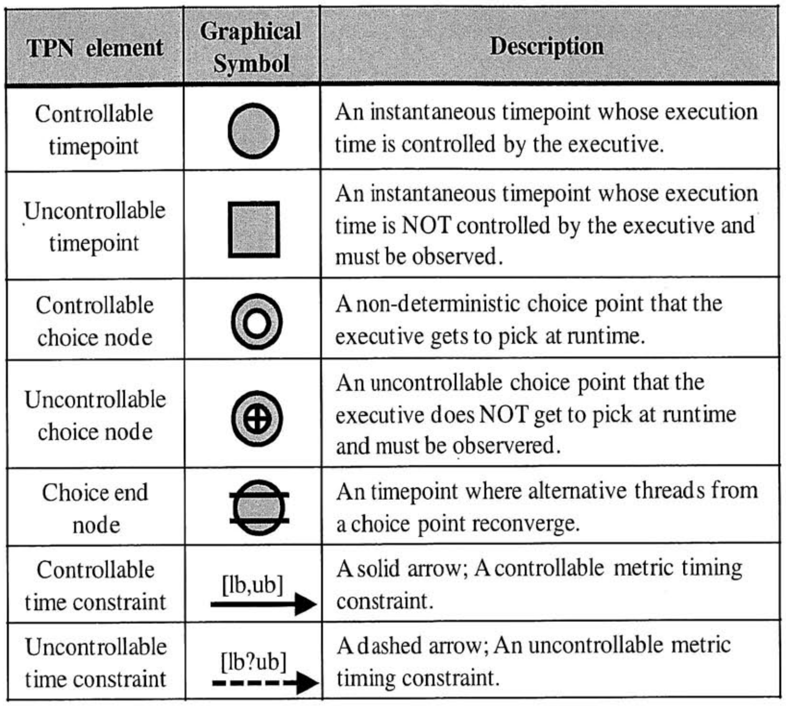
  - e_{lb} – task completes before *lb*

  - e_{ub} – task completes after *ub*

5.3 Temporal Plan Network with Uncertainty (TPNU)

In this section, we develop a set-bounded semantics for RMPL in terms of a graphical constraint representation called a Temporal Plan Network with Uncertainty (TPNU). The TPNU generalizes upon a Temporal Plan Network (TPN) introduced in [Kim 2001] as the semantics of a timed subset of a timed RMPL in which all durations are controllable.

A TPNU is comprised of five node types and two edge types, as described in Figure 5.1. Vertices represent instantaneous time points, while edges represent flexible temporal durations. A solid edge represents a flexible duration controlled by the executive, while a dashed edge represents a duration that is controlled by the environment.



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 observed.
Choice end node	⊖	An timepoint where alternative threads from a choice point reconverge.
Controllable time constraint	[lb,ub] →	A solid arrow; A controllable metric timing constraint.
Uncontrollable time constraint	[lb?ub] ⇢	A dashed arrow; An uncontrollable metric timing constraint.

Figure 5.1: A description of TPNU elements.

5.4 TPNU Execution

In this section we define TPNU execution. We start with three supporting definitions for TPNU execution. Then we explain TPNU execution using the wounded soldier example.

Definition 5.2: Candidate Execution

A *candidate execution* of a TPNU is an assignment of choices to controllable and uncontrollable choice nodes in a TPNU. A candidate execution is constructed as follows:

(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) arbitrarily. Stop if the program end node is reached.

For example, two candidate executions for the obstacle course example are presented in Figure 5.2. Note that candidate executions may be partial executions. Step 2 above may stop after any arbitrary number of iterations, including zero. The highlighted nodes and arcs in Figure 5.2 are those marked for timing analysis. Only one outarc from controllable and uncontrollable choice nodes may be highlighted. Runtime analysis of the timing of candidate executions enables robust execution within a program's flexible constraint boundaries.

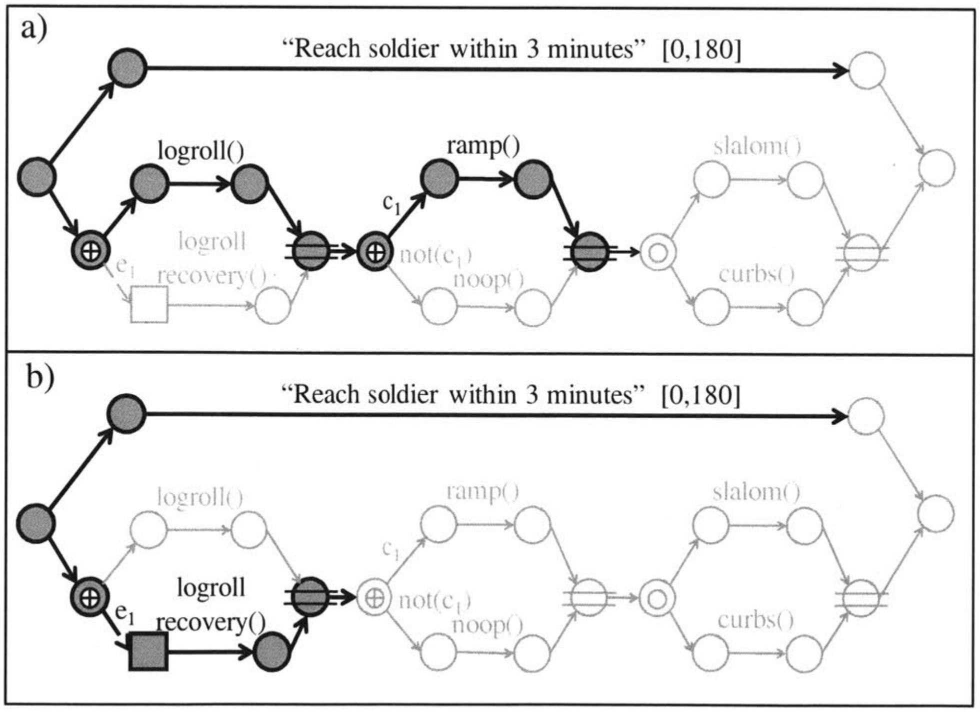


Figure 5.2: a) TPNU candidate execution with logroll and ramp obstacles selected for analysis.

b) TPNU candidate execution with a logroll exception and recovery selected for analysis.

Definition 5.3: Complete Execution

A *complete execution* is a candidate execution 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.

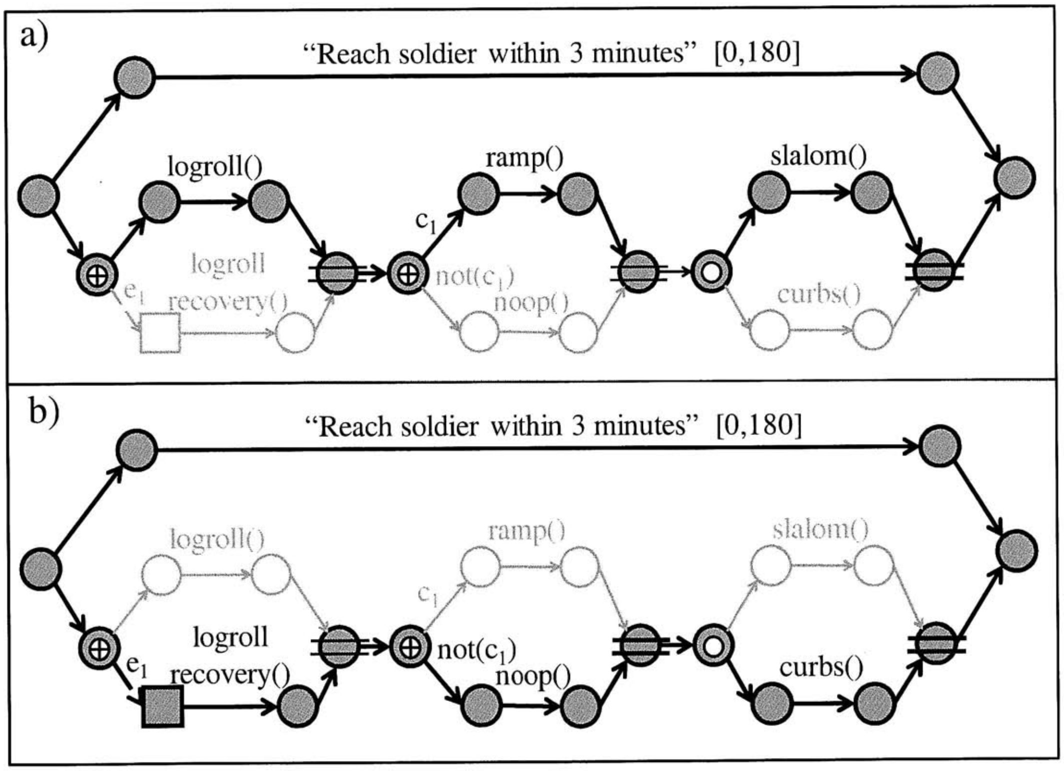


Figure 5.3: a) Complete execution which traverses the logroll, ramp, and slalom obstacles.

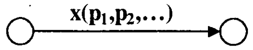
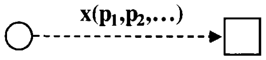
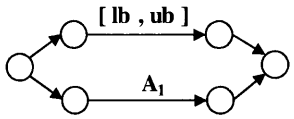
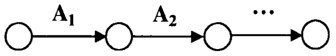
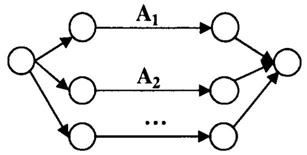
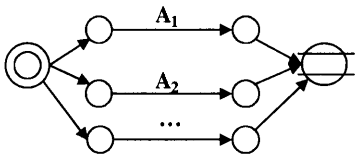
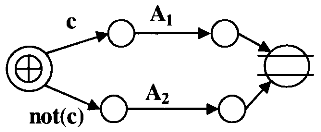
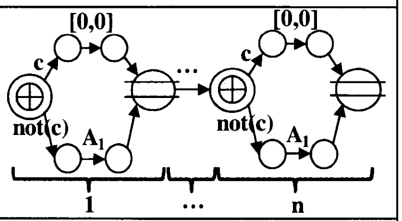
b) Complete execution which encounters an exception and executes a recovery during the logroll task, skips the ramp, and then traverses the curbs obstacle.

Definition 5.4 - TPNU Fringe

The *TPN fringe* is defined as the current state of execution progress in a TPNU and is used to distinguish between already executed nodes and nodes that are next in line for execution. The TPNU fringe consists of exactly the set of nodes that are capable of being executed next during TPNU 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.

5.5 Mapping from RMPL to TPNU

The mapping from RMPL constructs to TPNU elements is presented in Figure 5.4. Arcs with no annotation are assumed to have bounds [0,0]. The *try-catch* construct consists of an uncontrollable choice between the nominal thread of execution and a block of contingencies. 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. The *until* construct consists of an uncontrollable choice between each number of possible iterations, bounded by *n*. This construct can be implemented in practice through repeated use of the if-else construct.

$x(p_1,p_2,...)$ [lb , ub]	
$x(p_1,p_2,...)$ [lb ? ub]	
[lb , ub] { A_1 }	
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 }	 [0,0] c not(c) A_1 ... [0,0] c not(c) A_1 1 ... n

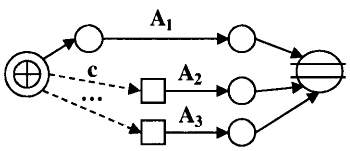
try { A_1 } catch (c) { A_2 }
catch(...) { A_3 }
	

Figure 5.4: Mapping from RMPL to TPNU

5.6 Dynamic Controllability of the Temporal Plan Network with Uncertainty (TPNU)

In this section, we define dynamic controllability for TPNUs. Then, we focus on developing an algorithm that determines dynamic controllability. Our approach is inspired by the approach taken in [Tsamardinos, et. al., 2003] to determine dynamic controllability of CTPs. However, our definition of *successful execution* differs from prior work. In TPNUs, constraint violations (which cause an exception to be thrown) 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 temporal consistency with the rest of the program. Each handler must be guaranteed to have time to complete a recovery in response to caught exceptions, or the TPNU is identified as uncontrollable. This approach tends to overconstrain TPNUs with many exception handler blocks, since they all must be consistent with the nominal thread to guarantee successful execution. In response, we develop a more reasonable approach called *N-fault dynamic controllability*, which ensures robustness to at most *n* exceptions at runtime.

Definition 5.5: Dynamic Controllability of TPNUs

Dynamic controllability of a TPNU 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. Next, we summarize AND/OR search, which is utilized by the dynamic controllability algorithm.

5.6.1 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*, $S_G \subseteq S$ 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 5.5. 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 5.6. In Figure 5.6a, the AND/OR tree is depicted for the case where the villain must choose first. In Figure 5.6b, 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 5.6a but not for case 5.6b. 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 5.5: Hero/Villain Example

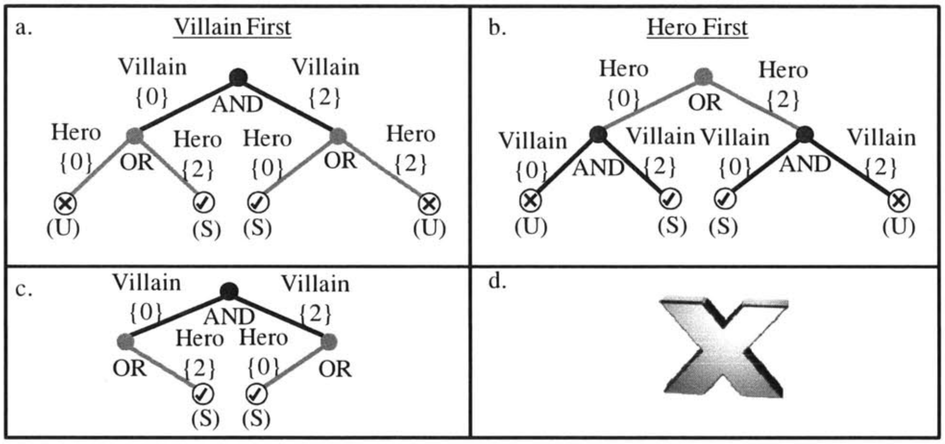


Figure 5.6: a. And/Or search tree when Villain goes first.

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.

5.6.2 Overview of Dynamic Controllability Algorithm

An algorithm is developed which frames the dynamic controllability problem for TPNUs as an AND/OR search tree over candidate program executions. To guarantee dynamic controllability, the algorithm starts from the beginning of the TPNU and branches generatively over the *TPNU fringe* (Definition 5.4) 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 5.6: *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 5.7: AND/OR Search Tree Encoding of a TPNU

An AND/OR search tree encoding of a TPNU 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 set-bounded executable activities (Definition 5.1), $s = \{x_1, x_2, \cdots, x_j\}$.

- ***O*** is a set of operators used to construct the child states for each state in S. (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 comprising the subtree must form a temporally consistent STP.

Definition 5.8 – 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 must form a temporally consistent STP.

5.6.3 Defining the Operators, *O*

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

```
function N-Fault Dynamically Controllable TPNU ( startnode , N )
TPNUfringe = startnode; i = 0; //fault count
s_0 = null; parent(s_0) = null; //andor start state
ConvertUncontrollableTimepoints( );
AND-OR-tree = RecursivelyExpand( startnode, s_0, TPNUfringe,i, N )
if SolutionSubtreeExists( AND-OR-tree ) return TRUE
else return FALSE
```

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

```
function ExpandFringe ( s , TPNUfringe , i , N)
Create an (OR) constraint with parent s, and create a child state:
for each node p in TPNUfringe
    c = null; parent(c) = s; //Creates the new OR state c with parent s
    TPNUfringe_copy = TPNUfringe; // Creates a copy of TPNUfringe
    for each node a in TPNUfringe and its copy, denoted $a_{copy}$
        e = [0,0]; start(e) = a; end(e)=$a_{copy}$;   //Creates [0,0] edge, e,
                                                     // between each node and its copy
        c += e;                                      // puts edge e into the andor state c
    if node p is a controllable timepoint node
        ExpandNode( $p_{copy}$ , c , TPNUfringe_copy , i , N)
    else if node p is an uncontrollable choice node
        ExpandUncontrollableChoice( $p_{copy}$ ,c ,TPNUfringe_copy, i, N)
    else if node p is a controllable choice node
        ExpandControllableChoice( $p_{copy}$ , c , TPNUfringe_copy , i , N)
```

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

```
function ExpandUncontrollableChoice( p , s , TPNUfringe, i , N )
if node p comes from a try-catch statement, i = i + 1;
Create an (AND) constraint with parent s, and create a child state c:
for each uncontrollable choice out-edge, $e_{out}$, emanating from node p
    if i >= N and node p is from a try-catch statement   //N exceeded,
        if $e_{out}$ is not the nominal thread, continue; //skip contingencies
    c = null; parent(c) = s; //Creates a new AND state c with parent s
```

```
TPNUfringe_copy = TPNUfringe; // Creates a copy of TPNUfringe
for each node $a$ in TPNUfringe and its copy, denoted $a_{copy}$
    e = [0,0];  start(e) = a;  end(e)=$a_{copy}$;   //Creates [0,0] edge, e,
                                             // between each node and its copy
    c += e;                                  // puts edge $e$ into new andor state $c$
for each node $c_{copy}$ in TPNUfringe_copy
    if $c_{copy}$ is $p_{copy}$, continue;//skip it
    e = [0,$inf$]; start(e) = $p_{copy}$; end(e) = $a_{copy}$; //Constrains node $p_{copy}$
    c +=  e;                //to occur before all other TPNUfringe nodes
Remove node $p_{copy}$ from TPNUfringe_copy
TPNUfringe_copy += end($e_{out}$)      //Add chosen out-edge endnode
c += $e_{out}$;          // Add chosen out-edge $e_{out}$ into new andor state $c$
RecursivelyExpand( c , TPNUfringe_copy , i , N )
```

```
function ExpandControllableChoice ( p , s , TPNUfringe , i , N )
Create an (OR) constraint with parent s, and create a child state c:
for each controllable choice out-edge, $e_{out}$, emanating from node p
    c = $null$; parent(c) = s; //Creates a new OR state $c$ with parent s
    TPNUfringe_copy = TPNUfringe; // Creates a copy of TPNUfringe
    for each node $a$ in TPNUfringe and its copy, denoted $a_{copy}$
        e = [0,0];  start(e) = a;  end(e)=$a_{copy}$;   //Creates [0,0] edge, e,
                                             // between each node and its copy
        c += e;                              // puts edge $e$ into the andor state $c$
    for each node $c_{copy}$ in TPNUfringe_copy except $p_{copy}$
        if $c_{copy}$ is $p_{copy}$, continue;//skip it
        e = [0,$inf$]; start(e) = $p_{copy}$; end(e) = $a_{copy}$; //Constrains node $p_{copy}$
        c += e;                 //to occur before all other TPNUfringe nodes
    Remove node $p_{copy}$ from TPNUfringe_copy
    TPNUfringe_copy += end($e_{out}$)      //Add chosen out-edge endnode
    c +=  $e_{out}$;          // Add chosen out-edge $e_{out}$ into the andor state $s$
    RecursivelyExpand( c , TPNUfringe_copy , i , N )
```

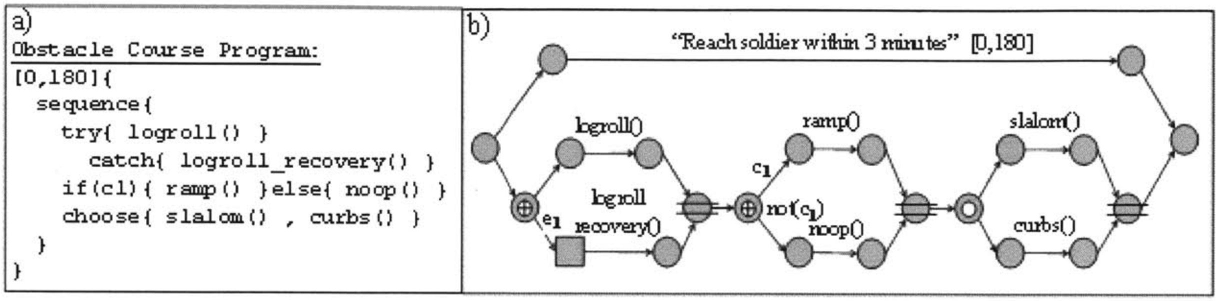
```
function BumpFringe ( s , TPNUfringe)
Find all nodes, n, in or extending from TPNUfringe that are guaranteed
  to execute before or simultaneously with the next possible choice or
  uncontrollable choice node.
for each node in n
    for each out-edge e emanating from n
        if endnode(e) is also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

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

```
function IntermediateTemporalConsistencyCheck ( s , TPNUfringe )
Find all nodes, n, and edges, e, between the startnode and TPNUfringe.
if TemporallyConsistent( startnode + n + TPNUfringe , e )
  return true;
else
  return false;
```

5.6.4 A Simple Walkthrough on the Wounded Soldier Example

The obstacle course example is presented again in Figure 5.7 for convenience. The dynamic controllability algorithm starts by placing the TPNU 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.



a)

```
Obstacle Course Program:
[0,180]{
  sequence{
    try{ logroll() }
      catch{ logroll_recovery() }
    if(c1){ ramp() }else{ noop() }
    choose{ slalom() , curbs() }
  }
}
```

Figure 5.7: RMPL program and TPNU for the Wounded Soldier Example.

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 TPNU_fringe, and creates a new OR state which consists of a copy of the TPNUfringe. This is depicted graphically in Figure 5.8 for the first call to *ExpandFringe*() for the Obstacle Course 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 Bear 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 calls is depicted in Figure 5.9. 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.

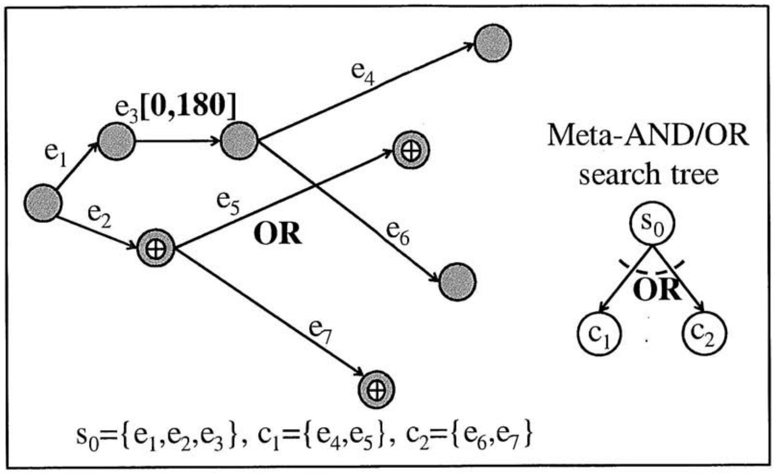


Figure 5.8: First call to *ExpandFringe*() in the wounded soldier example

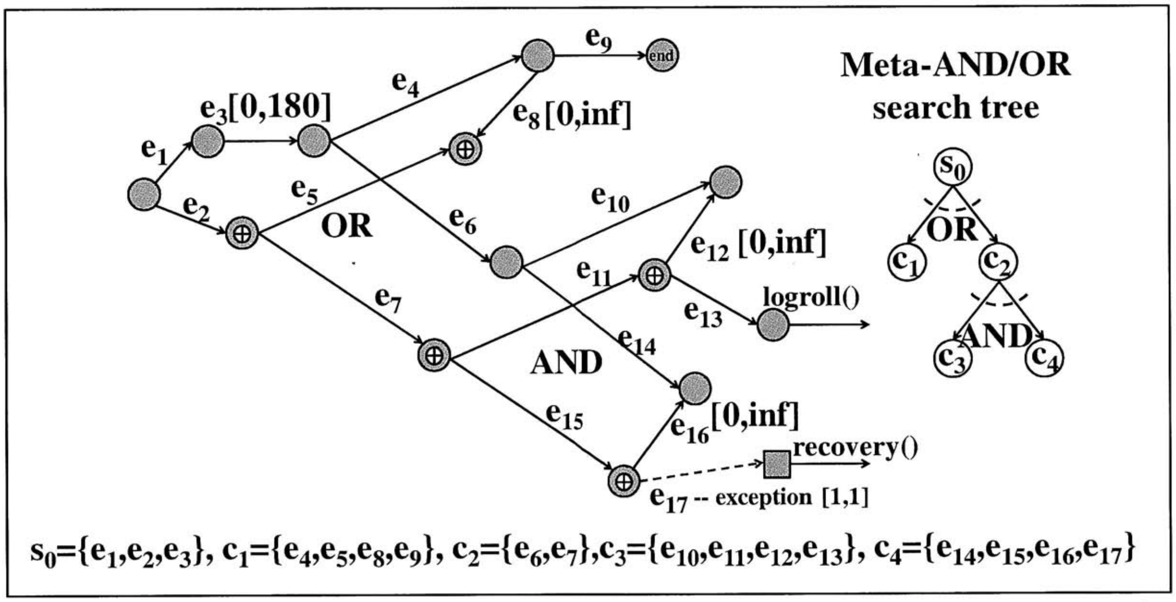


Figure 5.9: Function subcalls to *ExpandNode*() and *ExpandUncontrollableChoice*() in the wounded soldier example.

5.7 Background Material

We briefly review some related work in dynamic controllability:

- Consistency for STPs, DTPs, and TPNs
- Weak, strong, dynamic controllability for STNU
- Weak, strong, dynamic consistency for CTP

5.7.1 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 5.10. In an STP, each edge, $i \rightarrow j$, is labeled with an interval, $[lb,ub]$, which represents a metric timing constraint, $lb \le X_j - X_i \le ub$. 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], respectively.

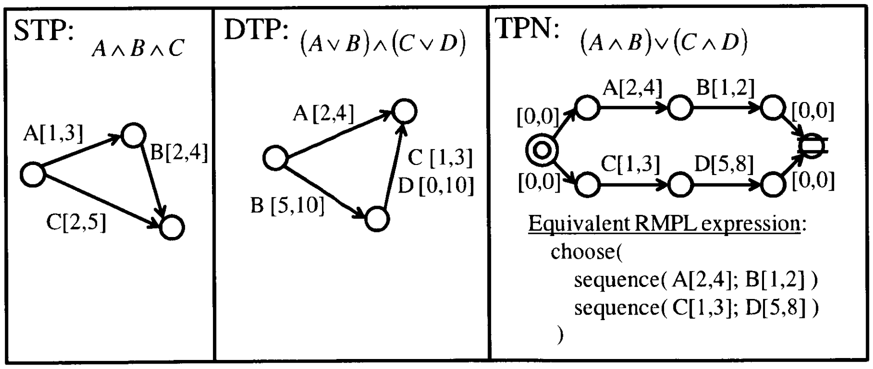

STP: $A \wedge B \wedge C$
A[1,3]
B[2,4]
C[2,5]
DTP: $(A \vee B) \wedge (C \vee D)$
A [2,4]
B [5,10]
C [1,3]
D [0,10]
TPN: $(A \wedge B) \vee (C \wedge D)$
[0,0]
A[2,4]
B[1,2]
[0,0]
[0,0]
C[1,3]
D[5,8]
[0,0]
Equivalent RMPL expression:
choose(
sequence(A[2,4]; B[1,2])
sequence(C[1,3]; D[5,8])
)

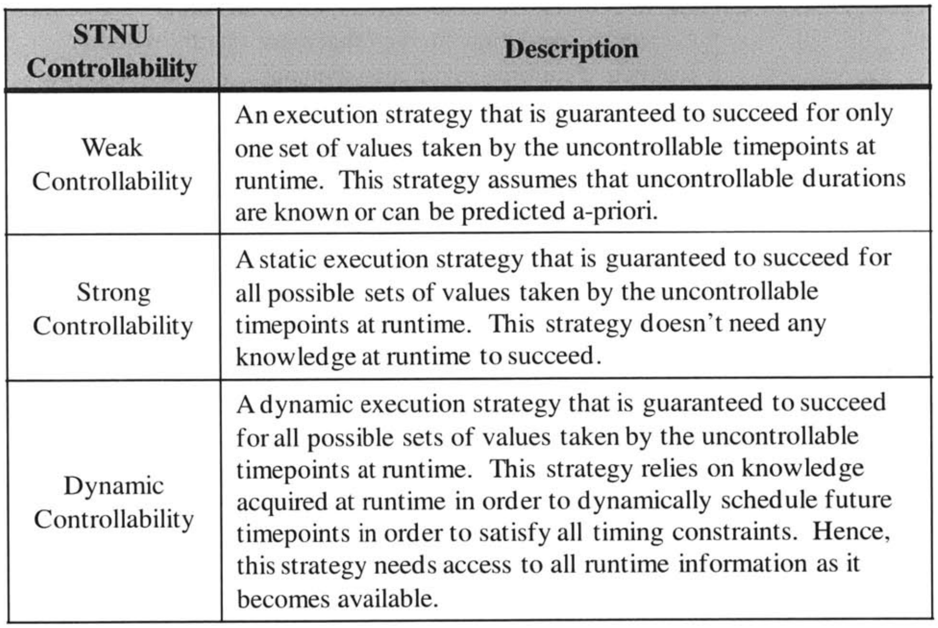
Figure 5.10: An Example STP, DTP, and TPN

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, et. al., 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].

5.7.2 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 5.11, based on the information available to the agent at runtime.

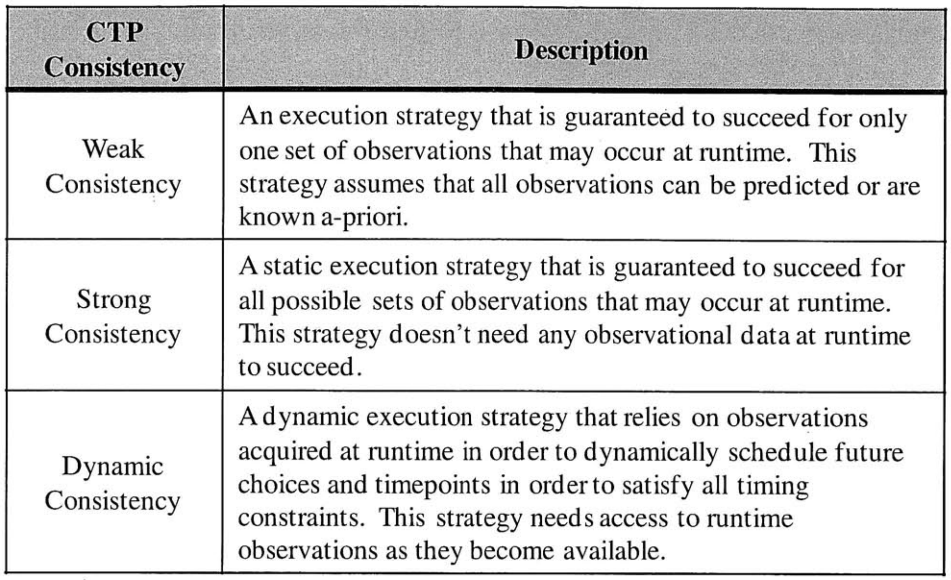


STNU Controllability	Description
Weak Controllability	An execution strategy that is guaranteed to succeed for only one set of values taken by the uncontrollable timepoints at runtime. This strategy assumes that uncontrollable durations are known or can be predicted a-priori.
Strong Controllability	A 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 5.11: STNU Controllability Classifications

5.7.3 Weak, Strong, Dynamic Consistency for CTP

The Conditional Temporal Problem (CTP) [Tsamardinos, et. al., 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 5.11, three notions of consistency are defined for the CTP; *weak, strong,* and *dynamic consistency.* The term consistency is used instead of controllability because all 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 5.12: CTP Consistency Classifications

The CTP dynamic consistency checking algorithm developed in [Tsamardinos, et. al., 2003] 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 existing DTP dispatching techniques.

This concludes Chapter 5, which develops a set-bounded dynamic execution algorithm for RMPL programs. Next, in Chapter 6, we develop an approximate probabilistic dynamic execution algorithm for RMPL programs.

Chapter 6: Risk-Minimizing Execution with Limited Coupling

In this chapter, we develop an algorithm for executing RMPL programs that improves upon the tractability results of the minimum-risk execution algorithm (Chapter 4) by limiting the amount of scheduling information that is communicated between subprograms that execute in parallel.

We start in Section 6.1, by pinpointing the leading source of complexity in the minimum-risk execution algorithm, the parallel operator. We discover that the parallel operator adds an exponential number of states to the MDP, while the other operators add only a linear or polynomial number of states. We identify this source of complexity as a state explosion produced by considering all possible couplings between locations executing in parallel in the HTA.

In Section 6.2, we introduce an alternative execution algorithm that reduces this explosion by reducing the coupling between locations to neighbors in the child/parent graph, and by maintaining the value function and policy in a factored form, one for each location in the HTA. To eliminate coupling between parallel threads, the limited coupling algorithm requires that parallel threads decide ahead of time on a shared target duration, which they then execute towards independently. This approach eliminates the primary computational bottleneck of the minimum-risk execution approach. However, some flexibility is lost. The resulting value function and decision policy will in general be suboptimal, implying a lower likelihood of successful program execution than is achievable with the minimum-risk approach.

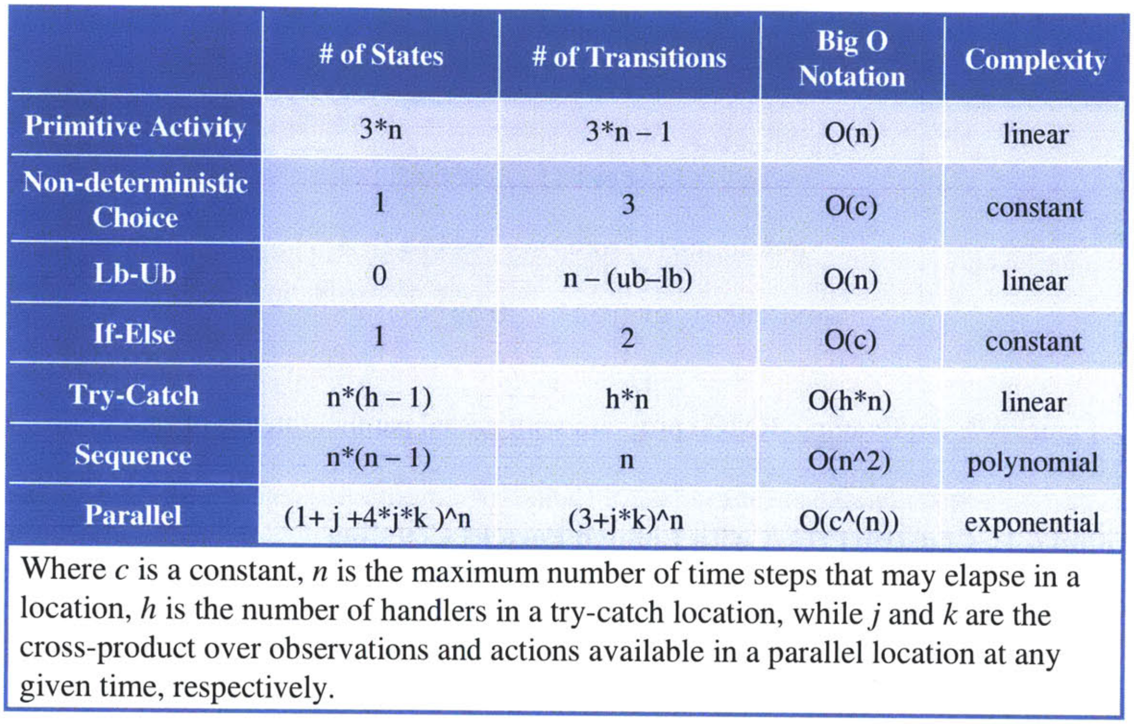
In Sections 6.3 and 6.4, we introduce the details of the limited coupling algorithm, which involves the development of two new concepts, *macro activity* and *policy-based temporal decomposition*, respectively. A macro activity decouples program execution by invoking a subprogram as a separate HTA that executes independently. Meanwhile, the subprogram is interpreted from the parent HTA as a primitive activity. In order to interpret a subprogram as a primitive activity, we introduce a new technique called *temporal decomposition*, which allows an HTA to be scheduled just prior to its execution, just like a primitive activity. Our decomposition approach leverages prior work in the hierarchical decomposition of MDPs [Wang and Mahadevan, 1999]. Finally, in Section 6.5 we discuss how the limited coupling algorithm can be augmented to apply to activities executing in sequence, as well as activities executing in parallel.

In the experimental results chapter (Chapter 7), we quantify the improvement in memory and runtime performance of the limited coupling algorithm compared to the minimum-risk algorithm on a set of representative examples. Finally, in Chapter 8, we present our conclusions and discuss promising directions for future work.

6.1 Computational Bottleneck of the Minimum-Risk Execution Algorithm

In Figure 6.1, we summarize the number of states and transitions added to the MDP for each location type. For reference, a visual depiction of the states and transitions added to the MDP for each location type are presented in Figures 4.7 through 4.13, in Chapter 4. The number of states and transitions added to the MDP for each parallel location grows exponentially in the number of observations and actions available at each time step, while the other constructs add only a linear or polynomial number. The exponential complexity of the parallel construct is caused by the need to account for an exponential number of ways that two interleaved threads of execution may be scheduled. In Figure 6.1, c is a constant, *n* is the maximum number of time steps that may elapse in a location, *h* is the number of handlers in a try-catch location, while *j* and *k* are the cross-product over observations and actions available in a parallel location at any given time, $j = j_1 \times j_2 \times \cdots \times j_d$ and $k = k_1 \times k_2 \times \cdots \times k_m$, respectively.

This analysis suggests that the key to reducing the complexity of the reformulation algorithm is to mitigate the complexity of representing parallel threads as a single state sequence.



	# of States	# of Transitions	Big O Notation	Complexity
Primitive Activity	3*n	3*n – 1	O(n)	linear
Non-deterministic Choice	1	3	O(c)	constant
Lb-Ub	0	n – (ub–lb)	O(n)	linear
If-Else	1	2	O(c)	constant
Try-Catch	n*(h – 1)	h*n	O(h*n)	linear
Sequence	n*(n – 1)	n	O(n^2)	polynomial
Parallel	(1+ j +4*j*k)^n	(3+j*k)^n	O(c^(n))	exponential

Where *c* is a constant, *n* is the maximum number of time steps that may elapse in a location, *h* is the number of handlers in a try-catch location, while *j* and *k* are the cross-product over observations and actions available in a parallel location at any given time, respectively.

Figure 6.1: Number of MDP States and Transitions added to the MDP per Location Type in the Minimum-Risk Execution Algorithm

6.2 Overview of the Limited Coupling Execution Algorithm

In this section, we develop an execution algorithm that eliminates coupling between parallel threads of execution. This is accomplished by requiring that parallel threads decide ahead of time on a shared target duration, which they then execute towards independently.

The high-level pseudocode for the algorithm is presented below, in Algorithm 6.1. The algorithm takes as input an HTA, and provides as output an HTA augmented with *macro activities*. A macro activity allows an HTA location to be scheduled and executed just like a primitive activity. A macro activity, therefore, must specify the same information as a primitive activity's stochastic activity model. In this section, we demonstrate how to compute an overall probability of success for an HTA location, as well as to compute distributions on execution time for an HTA location, allowing the child to be scheduled just like a primitive activity.

```
Example Program:
[0,120]
  parallel{
    if(cond){
      parallel{ text() , drive() }
    }else{
      sequence{ text() , drive() }
    },
    sequence{ stop() , drop() , roll()}
}
```

Figure 6.2: An Example RMPL program with nested parallel threads of execution.

Algorithm 6.1: Construct HTA with Limited Parallel Coupling

Inputs: Hierarchical Timed Automata (HTA), time discretization, Δt,
$\eta_{parallel}$, which denotes a hierarchy of HTA parallel locations
Outputs: An augmented HTA for execution, *HTA'*, along with its execution policy, π'.

```
function Construct_HTA_with_Limited_Parallel_Coupling$(HTA, \Delta t)$
$\eta_{parallel} = Construct\_Eta\_Parallel(HTA)$   // denotes a hierarchy of parallel locations
while $\exists l \in \eta_{parallel}$ such that $\eta_{parallel}(l) = \varnothing$   // choose any leaf remaining in $\eta_{parallel}$
    if $l$ is a parallel location
        Let $c = \eta(l)$, be the children of location $l$
        for each child $i \in c$,
            $\text{macroAct}_{\text{i}} = Temporal\_Decomposition(HTA, i, \Delta t)$
        $l' = Limit\_Parallel\_Coupling(\text{macroAct}_1, \cdots, \text{macroAct}_{\text{N}})$
        $\eta(\eta^{-1}(l)) = l'$   // Replace location $l$ in the $HTA$ with $l'$
    $\eta_{parallel}(\eta_{parallel}{}^{-1}(l)) = \varnothing$   // Remove location $l$ from $\eta_{parallel}$
$HTA' = HTA$
$MDP' = HTA\_to\_MDP(HTA', \Delta t)$   // call the exact algorithm (Alg. 4.1) on HTA'
$\pi' = DynamicProgramming(MDP')$
return $\{HTA', \pi'\}$
```

The first step of the algorithm constructs a tree, called $\eta_{parallel}$, which represents the hierarchal relationship between parallel locations in an HTA. For example, consider the RMPL program shown in Figure 6.2. Recall from the definition for HTA (Definition 4.1) that η represents the parent-child relationship between locations in an HTA, as shown in Figure 6.3a. Analogously, $\eta_{parallel}$ represents the hierarchical relationship between only parallel locations in an HTA, as shown in Figure 6.3b. To ensure a well-formed tree, $\eta_{parallel}$ also includes the root location of the HTA, regardless of whether it is a parallel location or not.

The limited coupling algorithm employs $\eta_{parallel}$ to convert each child of a parallel location into a macro activity in a recursive, depth-first manner. In order to construct a macro activity, the child location is *temporally decomposed.* The result of the temporal decomposition is a *macro activity,* which executes independently from the others. Finally, joint scheduling between macro activities in parallel is enabled by an algorithm, *Limit_Parallel_Coupling( ),* which enables a joint scheduling approach between macro activities executing in parallel. This algorithm is described in detail in Section 6.4.

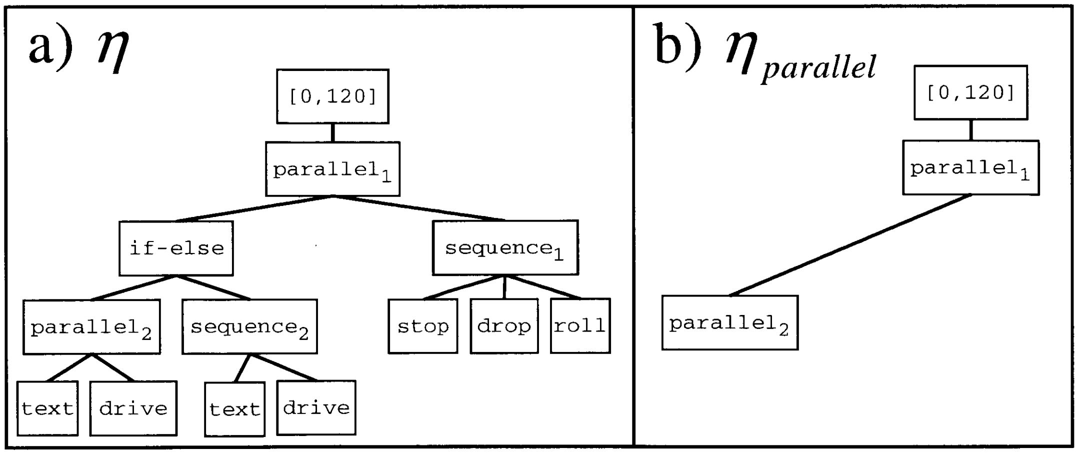


Figure 6.3: a) η represents the parent-child relationship between HTA locations, and b) represents the hierarchal relationship between parallel locations.

6.3 Temporal Decomposition Algorithm

In this section, we explain Step 6 of Algorithm 6.1, *temporal decomposition*, which allows an HTA location to be decoupled from its siblings and scheduled just like a primitive activity. The algorithm builds upon an idea developed previously in the MDP literature, called *policy-based decomposition*, by [Wang and Mahadevan, 1999]. First, in Section 6.3.1, we summarize the prior work by Wang and Mahadevan. Then, in Section 6.3.2 we define a new concept called a *macro activity*, which allows for HTA locations to be scheduled and executed just like a primitive activity. Finally, in Section 6.3.3 we present the pseudocode for the *temporal decomposition* algorithm and walk through some of its details.

6.3.1 Prior Work by Wang and Mahadevan (1999)

Hierarchical decomposition methods for solving large MDPs [Wang and Mahadevan, 1999], are explained in detail in Chapter 2 – Markov Decision Processes. In this thesis, we leverage an MDP decomposition technique called *policy decomposition*, which is explained in Section 2.3.2.

To summarize briefly, policy decomposition allows for the state occupied by one sub-MDP to affect the transition probabilities and rewards in another sub-MDP. Policy decomposition first identifies for each sub-MDP, the states in other sub-MDPs that affect its transition probabilities and rewards. These are called the sub-MDP's *parameter states*. Then, each sub-MDP is solved independently, once for each possible combination of parameter states, resulting in a parameterized decision policy for each sub-MDP. Please see Section 2.3.2 for more details.

6.3.2 Macro Activity

Next, we define a *macro activity*, which enables an HTA location to be decoupled from its siblings, and to be scheduled just like a primitive activity. Within the macro is an encapsulated HTA, and executing the macro activity corresponds to executing the encapsulated HTA. The details of the encapsulated HTA are hidden from the rest of the program, and instead are interpreted by the rest of the program as a primitive activity, the behavior of which is described by a stochastic activity model (Definition 4.8). A graphical depiction of a macro activity is shown in Figure 6.4 below. Next, we give a formal definition for macro activity.

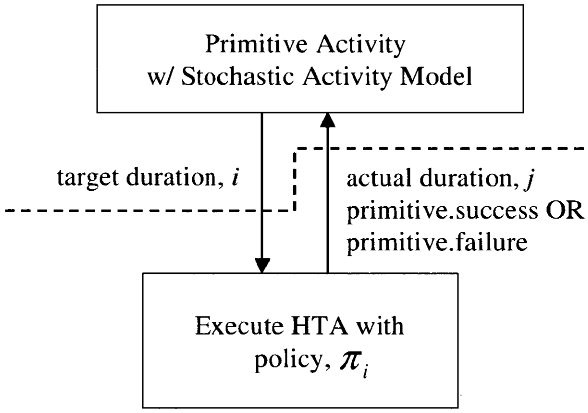


Figure 6.4: A *macro activity*, interpreted from above as primitive activity with a target duration *i*, and interpreted from below as an HTA executing a policy, π_i.

Definition 6.1: Macro Activity, $\langle HTA, \Pi, sam \rangle$

A macro activity is a 3-tuple consisting of an HTA, a set of policies, $\Pi = \{\pi_{\text{lb}}, \cdots, \pi_{\text{ub}}\}$, and a stochastic activity model, sam. Each policy, $\pi_i \in \Pi$, is intended to achieve the target duration *i*. Analogously, each element of the stochastic activity model, $sam(i)$, characterizes the timing behavior of policy π_i in achieving that target duration.

6.3.3 Pseudocode for Temporal Decomposition Algorithm

The temporal decomposition algorithm accepts three inputs, an HTA, a location to decompose, *l*, and a time discretization, Δt. The algorithm provides as output a macro activity as defined in the previous section (Definition 6.1). The algorithm proceeds in several steps:

1. First, the algorithm converts the HTA into an MDP using our existing HTA to MDP Algorithm (Algorithm 4.1). Then, a copy of the MDP is constructed for each possible target duration for the location, *l*, that we wish to temporally decompose.

2. Then, the algorithm modifies the reward function for each MDP_i so that it prefers to finish at target duration *i*, and solves for the associated policy, π_i, using dynamic programming. The reward function is modified to reward HTA executions that end near the target duration, via the reward function $r = \lambda e^{-\lambda|actual\_dur - target\_dur|}$. λ is a free parameter that determines the shape of the reward curve, as depicted in Figure 6.5, and must be determined for a given application.

3. Finally, the limiting distribution for each MDP_i as it follows policy π_i is computed, via the Chapman-Komolgorov equation, described below. The set of limiting distributions serves as the stochastic activity model, $sam(i)$, for the macro activity.

The pseudocode is included below, in Algorithm 6.3, along with a graphical depiction of temporal decomposition, in Figure 6.5.

Algorithm 6.3: Temporal Decomposition

Inputs : Hierarchical Timed Automata (HTA),
a location, l,
time discretization, Δt

Outputs : A Macro Activity, $macroAct = \langle HTA, \Pi, sam \rangle$ (Definition. 6.1)

```
function Temporally_Decompose( $HTA$ , $l$ , $\Delta t$ )
1. $MDP = HTA\_to\_MDP(HTA, l, \Delta t)$        // construct an MDP via Algorithm 4.1
2. for each $i \in \{lb,\ lb+\Delta t,\ \cdots\ ,\ ub\}$,        // For each target duration of location $l$
3.   $MDP_i = MDP$ // Create a copy of the MDP
4.   for each transition, $t$, in $MDP_i$
5.     Let $\lambda = 1/\Delta t$, and Let $c$ = clock time of the target location for each transition, $t$
6.       $r = r * \lambda e^{-\lambda|i-c|}$ // weight the rewards, $r$, to prefer target duration, $i$
7.       $\pi_i = DynamicProgramming(MDP_i)$ // Solve each MDP with weighted rewards
8.       $sam(i) = ChapmanKomolgorov(MDP_i, \pi_i, n)$   // Get the limiting distribution
9. Let $\Pi = \{\pi_1, \cdots, \pi_N\}$
10. return $macroAct = \{HTA,\ \Pi,\ sam\}$
```

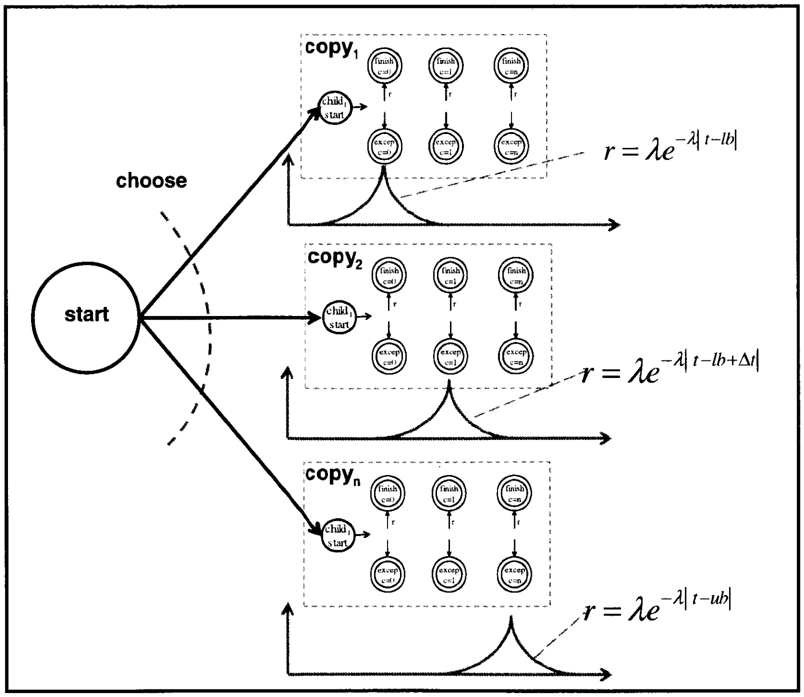


Figure 6.5: A graphical depiction of temporal decomposition.

Next, we describe the Chapman-Komolgorov equation in Step 8 of Algorithm 6.3, which computes the limiting distributions for the stochastic activity model. Recall from Chapter 4 that a Markov decision process coupled with a policy reduces to a Markov chain. A Markov chain can be classified into transient states and absorbing states. The probability distribution over absorbing states is called the *limiting distribution,* and is calculated via the Chapman-Kolmogorov Equation:

$$r_{ij}(n) = \sum_{k=1}^{m} r_{ik}(n-1)\, p_{kj}, \quad \text{for } n > 1, \text{ and all } i, j, \text{ with } r_{ij}(1) = p_{ij}.$$

This equation is simply the *n*th power of the transition probability matrix, p_{ij}, and converges to a limiting value as $n \to \infty$. For an initial state distribution, s_{init}, the limiting distribution is:

$$s_{\lim} = s_{init} * r_{ij}(n), \text{ for sufficiently large } n.$$

Practically speaking, all of the Markov chains develop in Chapters 4 and 6 are acyclic, so the limiting value is well-defined to be at most the number of transitions in the Markov chain:

$$s_{\lim} = s_{init} * r_{ij}(k), \text{ for } k \geq \text{ \# of non-zero entries in } p_{ij}.$$

This concludes our explanation of Step 6 of Algorithm 6.1, the temporal decomposition algorithm. Next, we explain Step 7 of Algorithm 6.1, the *limit parallel coupling algorithm,* which utilizes the macro activities constructed in Step 6 to decouple the scheduling of threads that execute in parallel.

6.4 Limited Parallel Coupling Algorithm

In this section, we explain Step 7 of Algorithm 6.1, the *limit parallel coupling algorithm.* The algorithm accepts as input a set of macro activities (Definition 6.1), one for each child location belonging to a parallel location, and constructs as output, a single HTA augmented with macro activities that limit the coupling between parallel threads. The algorithm works by constructing a choose location, with the alternatives being to execute each macroAct_i in parallel with the same target duration, as depicted below in Figure 6.6.

```
Limited Coupling Parallel:
choose{
  parallel{macroAct1(lb)    , ... , macroActN(lb)     },
  parallel{macroAct1(lb+dt) , ... , macroActN(lb+dt)  },
    ...
  parallel{macroAct1(ub)    , ... , macroActN(ub)     }
}
```

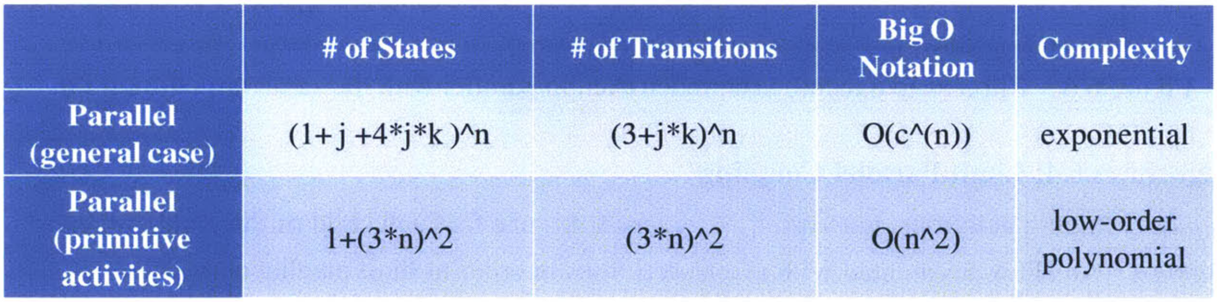
Figure 6.6: Choices to execute each macroAct$_i$ in parallel with the same target duration, *i*.

Algorithm 6.4: Limit Parallel Coupling

Inputs: Macro activities, $macroAct_1,\cdots,macroAct_N$, one for each child of the parallel location
Outputs: An HTA augmented with macro activities in order to limit parallel coupling

```
function Limit_Parallel_Coupling( $macroAct_1,\cdots,macroAct_N$ )
Let $HTA$ be a new Hierarchical Timed Automata, with parent function, $\eta$
Let $c$ be a new choose location
$\eta(c)=\varnothing,\ \eta^{-1}(c)=\varnothing$                    // Make c the HTA root location
for each $i\in\{min\_lb,\ min\_lb+\Delta t,\ \cdots,\ max\_ub\}$ // From min_lb to max_ub
Let $p$ be a new parallel location
    $\eta(c)=\eta(c)\cup p$                          // Make p a child of c
    for each $macroAct,\ i=\{1,\cdots,N\}$,          // For each macroAct
        Let $h=macroAct_i(j)$                  // A copy of $macroAct_i$ with target $j$
$\eta(p)=\eta(p)\cup h$                              // Make h a child of p
return $HTA$
```

To see why this algorithm is useful, consider the primary source of complexity in the minimum-risk execution algorithm, the parallel operator. The number of states and transitions added to the MDP for each parallel location grows exponentially in the number of observations and actions available at each time step, because of the combinatorial number of ways that two threads can be scheduled in parallel. Let us analyze separately, however, the case in which only macro activities are being scheduled in parallel, which is the case above. To jointly schedule hierarchical primitive activities, identical target durations are chosen for each activity prior to execution. In this simplified case, the number of states added to the MDP is much smaller, since each activity executes independently from the others. Complexity results for the general and simplified cases are given in Figure 6.7, below.



	# of States	# of Transitions	Big O Notation	Complexity
Parallel (general case)	(1+j +4*j*k)^n	(3+j*k)^n	O(c^(n))	exponential
Parallel (primitive activites)	1+(3*n)^2	(3*n)^2	O(n^2)	low-order polynomial

Figure 6.7: The complexity of scheduling two parallel threads vs. two macro activities.

Each time an augmented HTA is returned from Step 7 in Algorithm 6.1, the augmented HTA replaces the original, thus reducing complexity. The process is repeated in a recursive, depth-first manner until $\eta_{parallel}$ contains no more parallel locations. This concludes our development of the limited parallel coupling algorithm. Next, we discuss how a similar approach can be taken to limit coupling in sequentially executing activities as well.

6.5 Limited Sequential Coupling

Step 5 in Algorithm 6.4 can be augmented to construct a sequence location instead of a parallel location, resulting in an HTA which can be represented by the RMPL fragment depicted in Figure 6.8. This results in a sequential execution in which activities are scheduled "proportionally". Either all activities are scheduled to complete early, or all are scheduled to complete late.

We can do better than this open-loop scheduling approach by recording how early or late each activity in the sequence finishes, and then augmenting the executive with a simple differencing rule that schedules subsequent activities earlier or later in order to pick up the slack, so to speak, from the earlier activities. The performance of this limited coupling approach to sequential execution is compared to the minimum-risk approach in Chapter 7.

```
Limited_Coupling_Sequence:
choose{
  sequence{ macroAct1(lb)    , ... , macroActN(lb)     },
  sequence{ macroAct1(lb+dt), ... , macroActN(lb+dt) },
    ...
  sequence{ macroAct1(ub)    , ... , macroActN(ub)     }
}
```

Figure 6.8: Choices to execute each macroAct_i in sequence with the same target duration, *i*.

Chapter 6 Summary:

In this chapter we developed a limited coupling algorithm for executing RMPL programs that improves tractability while admitting a lesser likelihood of successful program execution. We introduced two new concepts, *macro activity* and *temporal decomposition*, which together enable us to decouple RMPL program execution. Finally, we develop a novel technique for scheduling parallel threads of execution that leverage the macro activities improve complexity over the minimum-risk approach.

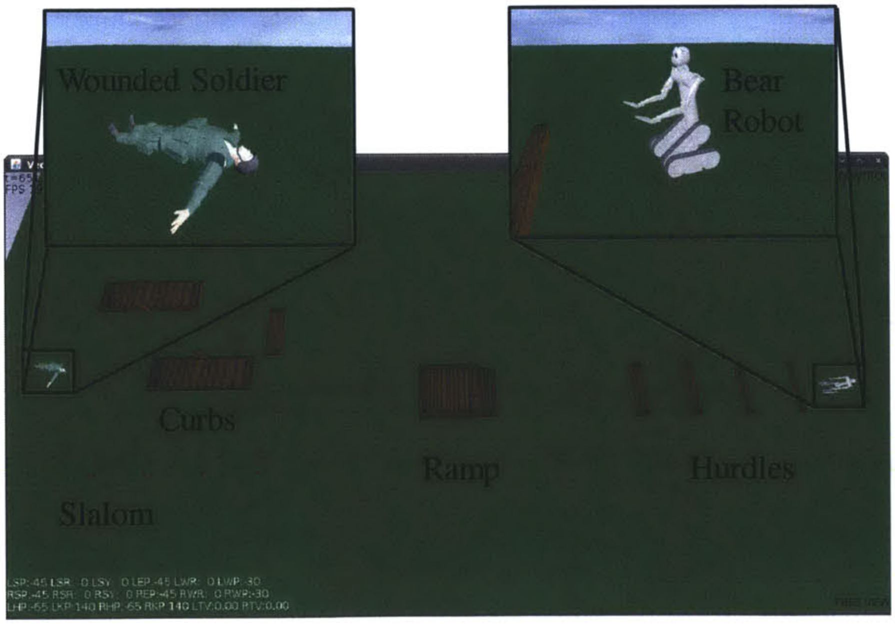
Next, in the experimental results chapter, Chapter 7, we quantify the improvement in tractability, and associated loss in optimality, of the limited coupling algorithm compared to the minimum-risk algorithm on a set of representative programs. Finally, in Chapter 8, we discuss interesting directions for future work.

Chapter 7 – Experimental Results

In this chapter we summarize our experimental results. First, we demonstrate risk-minimizing execution on a humanoid robot simulator. Then, we perform an empirical evaluation of the complexity of our three risk-minimizing execution algorithms on a set of representative programs.

7.1 Experimental Validation on a Humanoid Robot Simulator

In this section, we demonstrate risk-minimizing execution on a humanoid robot simulator, built by Vecna Technologies, which simulates the Battlefield Extraction-Assist Robot (BEAR). In the simulator, we have built a wounded soldier rescue scenario in-which the robot must traverse multiple obstacles, as shown in Figure 7.1, in order to rescue a wounded soldier. In this scenario, it is assumed that due to the proximity of enemy forces, it is desirable for the robot to attempt the



```
LSP:-45 LSR:  0 LSY:  0 LEP:-45 LWR:  0 LWP:-30
RSP:-45 RSR:  0 RSY:  0 REP:-45 RWR:  0 RWP:-30
LHP:-65 LKP:140 RHP:-65 RKP:140 LTV:0.00 RTV:0.00
```

Figure 7.1: Wounded soldier obstacle course.

rescue autonomously in order to prevent additional casualties. If the robot will be unable to reach the wounded soldier in 4 minutes, then additional soldiers should be dispatched to assist in the rescue.

The obstacle course consists of four stations: 1) a series of hurdles, 2) a 45 degree inclined ramp followed by a 1 meter vertical drop, 3) a slalom course, and 4) sidewalk curbs. Additionally, the robot is told to traverse the stations in the following order:

- First, traverse the hurdles.
    - If a fall occurs while traversing the hurdles, go to the first hurdle and start over.
    - If a second fall occurs, halt execution and ask for help.
- Second, traverse the ramp.
    - Skip the ramp if it is blocked at runtime.
- Third, choose between traversing the slalom course or the sidewalk curbs. The robot should choose the option that maximizes probability of success.
    - The slalom activity is less risky, but it takes more time to execute.
    - The curbs activity is fast to execute, but it has a higher risk of failure.

Since this scenario has only a single deadline constraint, the activity scheduling is trivial; the robot should always choose the fastest available duration for each activity. In addition, this scenario is constructed such that the robot should choose the slalom activity if and only if the hurdles activity has not thrown an exception. If an exception is thrown, then the robot should choose the curbs activity. Finally, if at any point it becomes unlikely that the robot will reach the wounded soldier within 4 minutes (<30% likelihood), then the robot should halt execution and alert a commanding officer to help.

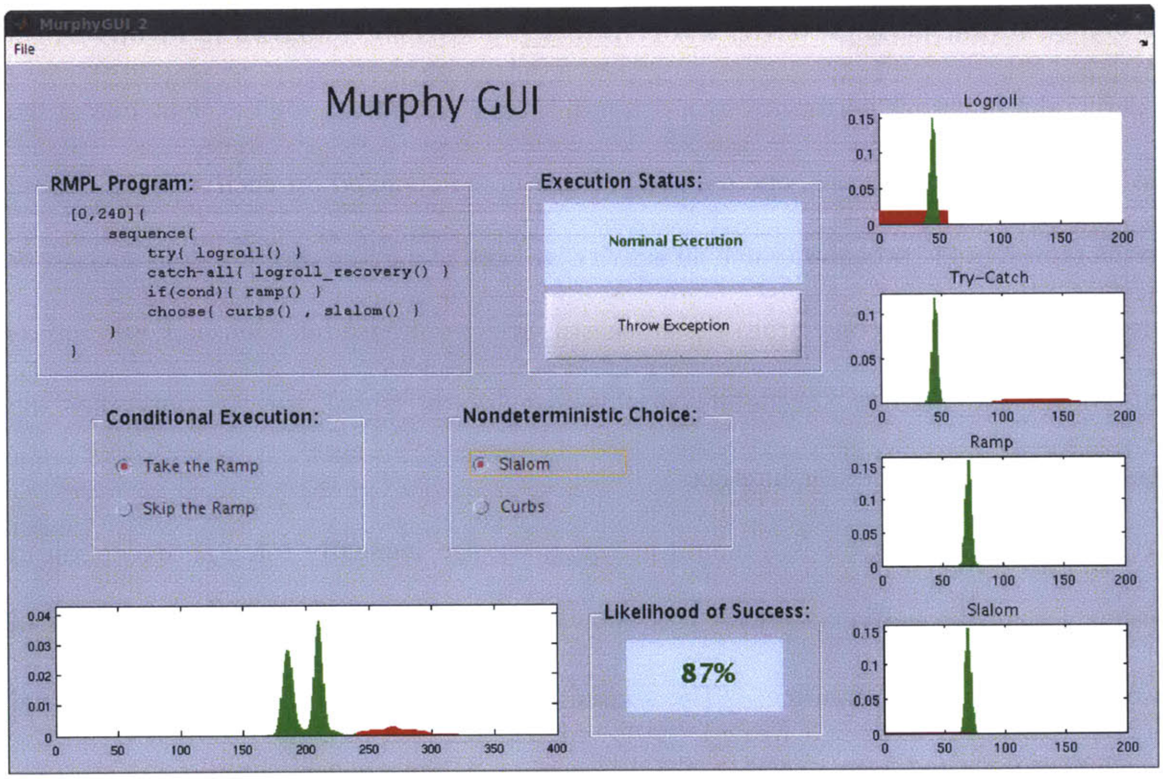
The control interface to the simulator accepts joint and tread velocities as input, and provides joint angles and absolute position of the center of mass as output. We built an interface from the BEAR simulator to Matlab, and for each activity we have constructed a set of feedback controllers in Matlab to perform the desired motion. Also, a stochastic activity model was developed for each activity, which consists of a set of probability distributions that represent actual activity duration, both in success and failure, for a range of target durations.

A graphical user interface (GUI) was constructed in Matlab, as shown in Figure 7.2, in order to depict the functions performed by a risk-minimizing executive. On the right side of the GUI, two probability distributions, one red and one green, are depicted for each activity. These distributions represent the probability that an activity will end at a given time either in success or failure, respectively. They come from the stochastic activity model for each activity, and as discussed above, the activity scheduling in this scenario is trivial, so we only plot the distributions for the fastest activity duration.

Also depicted in the GUI is a button in which we can manually force an exception to occur during the hurdles activity, as well as a radio button in which we can force the robot to traverse or skip the ramp at runtime. The second radio button is set by the executive, and indicates whether the robot has decided to take the curbs or slalom.

In addition to selecting the program path that minimizes risk, the executive continuously computes a probability distribution over likely program completion times and a corresponding measure of risk, as shown in the lower left of the GUI in Figure 7.2. Note that any probability mass beyond 240 seconds on the x-axis indicates a violation of the deadline and is colored red. The ratio of probability mass within the constraint boundary to the total is a measure of the probability of program success, which is 87%.

The probability distributions displayed in the GUI are updated continuously as the program executes and new information is learned at runtime, as is the probability of success. The GUI snapshot depicted in Figure 7.2 was taken before the executive has begun to execute the program. Next, we will depict the executive at various stages of program execution, and comment on the changes that occur in the GUI.



Murphy GUI

RMPL Program:

```
[0,240]{
    sequence{
        try{ logroll() }
        catch-all{ logroll_recovery() }
        if(cond){ ramp() }
        choose{ curbs() , slalom() }
    }
}
```

Execution Status: Nominal Execution

Conditional Execution:

● Take the Ramp
○ Skip the Ramp

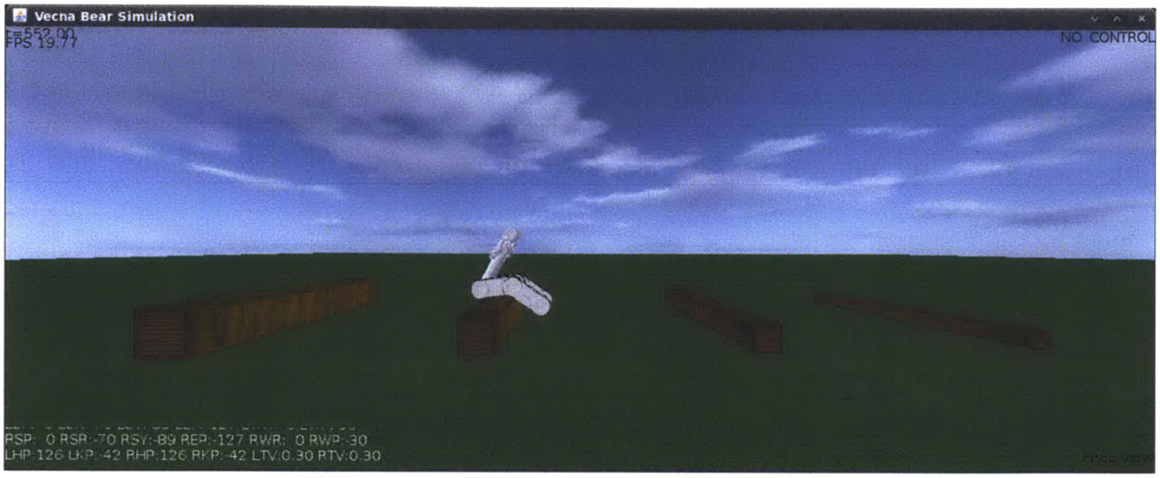
Nondeterministic Choice:

● Slalom
○ Curbs

Likelihood of Success: 87%

Figure 7.2: A GUI demonstrating the functions performed by a risk-minimizing executive

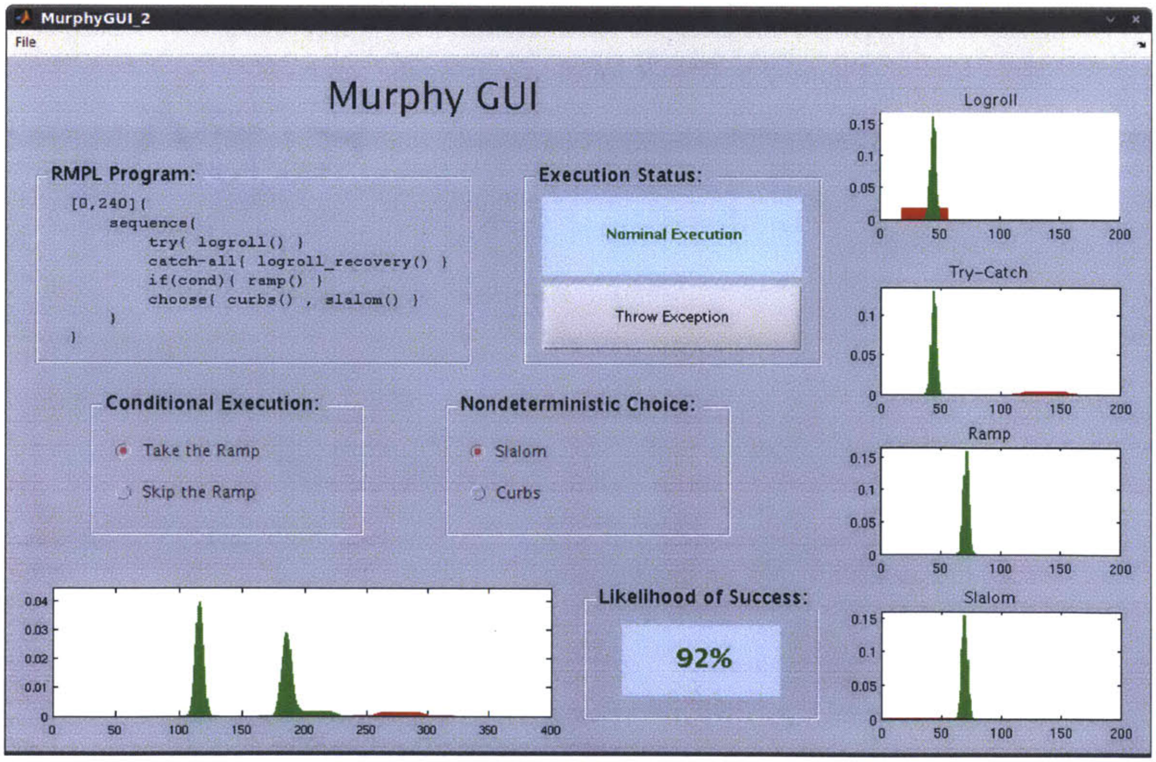
Bear robot traversing the hurdles: Figure 7.3 shows the BEAR robot traversing the hurdles obstacle. As shown in Figure 7.4, the executive updates the probability distributions for the hurdles activity at each timestep accounting for new information that the activity has not yet failed and is still executing. Note how this new information affects the likelihood of success; it has increased slightly from 87% in Figure 7.2 to 93%.



```
t=552.00
FPS 19.77
```

```
RSP:  0 RSR:-70 RSY:-89 REP:-127 RWR:  0 RWP:-30
LHP:126 LKP:-42 RHP:126 RKP:-42 LTV:0.30 RTV:0.30
```

Figure 7.3: BEAR robot traversing the *hurdles* activity.



RMPL Program:

```
[0,240]{
    sequence{
        try{ logroll() }
        catch-all{ logroll_recovery() }
        if(cond){ ramp() }
        choose{ curbs() , slalom() }
    }
}
```

Conditional Execution:

● Take the Ramp

○ Skip the Ramp

Nondeterministic Choice:

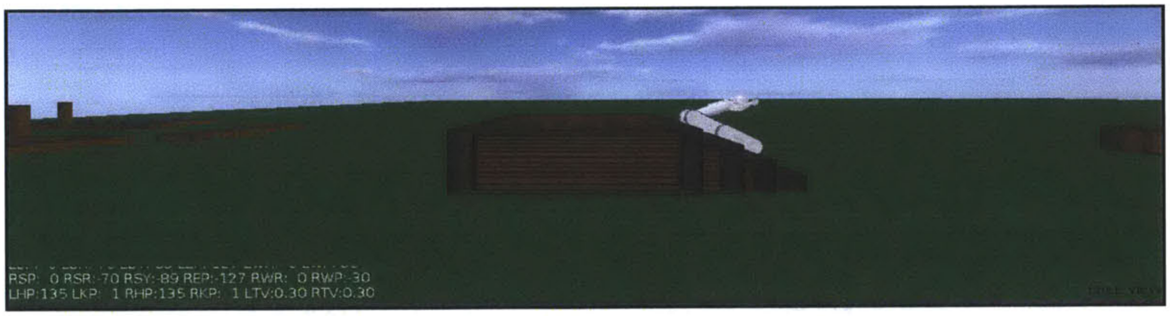
● Slalom

○ Curbs

Likelihood of Success: 92%

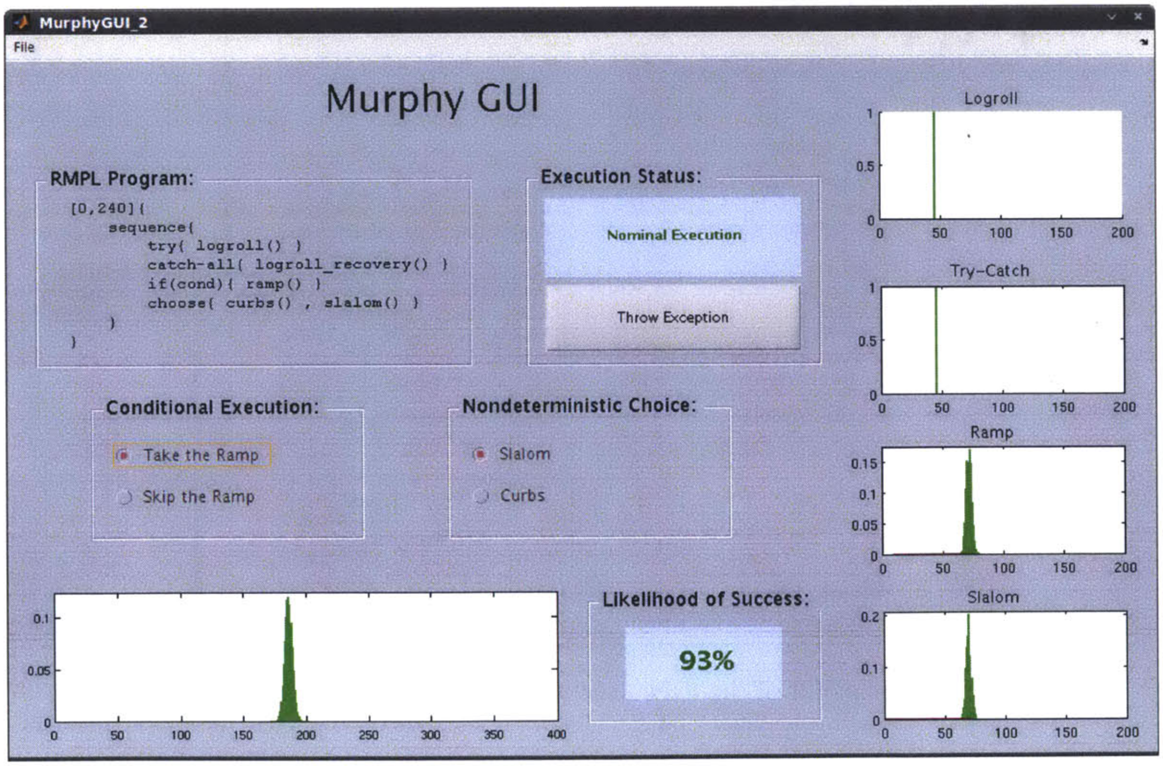
Figure 7.4: Snapshot of the executive while executing the hurdles activity.

Bear robot traversing the ramp: We notice several changes in the GUI as the ramp activity begins execution. First, the hurdles activity has finished, so now it is a unit step function. Second, the robot observes the radio button on the GUI and has to traverse the ramp. Finally, the distribution on program duration converges to a narrow range as the executive learns two new and important pieces of information; 1) no exception occurred during the hurdles, and 2) the ramp must be traversed.



```
RSP:  0 RSR:-70 RSY:-89 REP:-127 RWR:  0 RWP:-30
LHP:135 LKP:  1 RHP:135 RKP:  1 LTV:0.30 RTV:0.30
```

Figure 7.5: BEAR robot traversing the ramp.



RMPL Program:

```
[0,240]{
    sequence{
        try{ logroll() }
        catch-all{ logroll_recovery() }
        if(cond){ ramp() }
        choose{ curbs() , slalom() }
    }
}
```

Execution Status:

Conditional Execution:

● Take the Ramp

○ Skip the Ramp

Nondeterministic Choice:

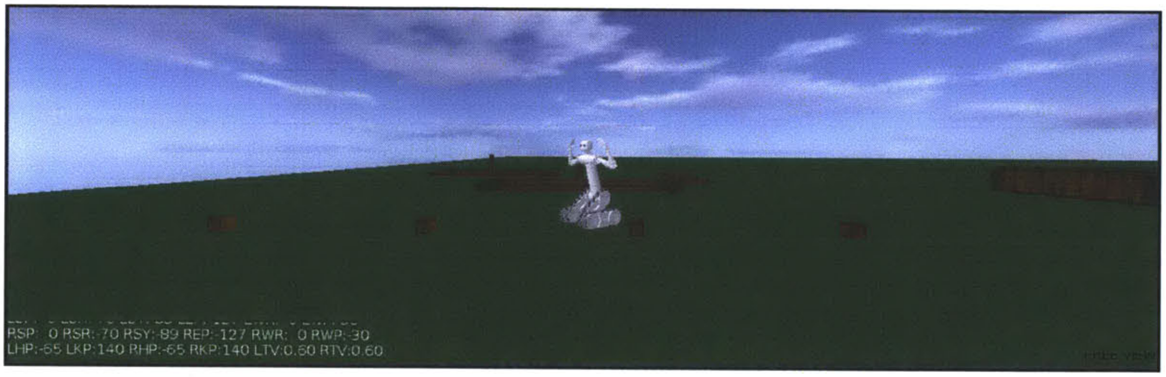
● Slalom

○ Curbs

Likelihood of Success: 93%

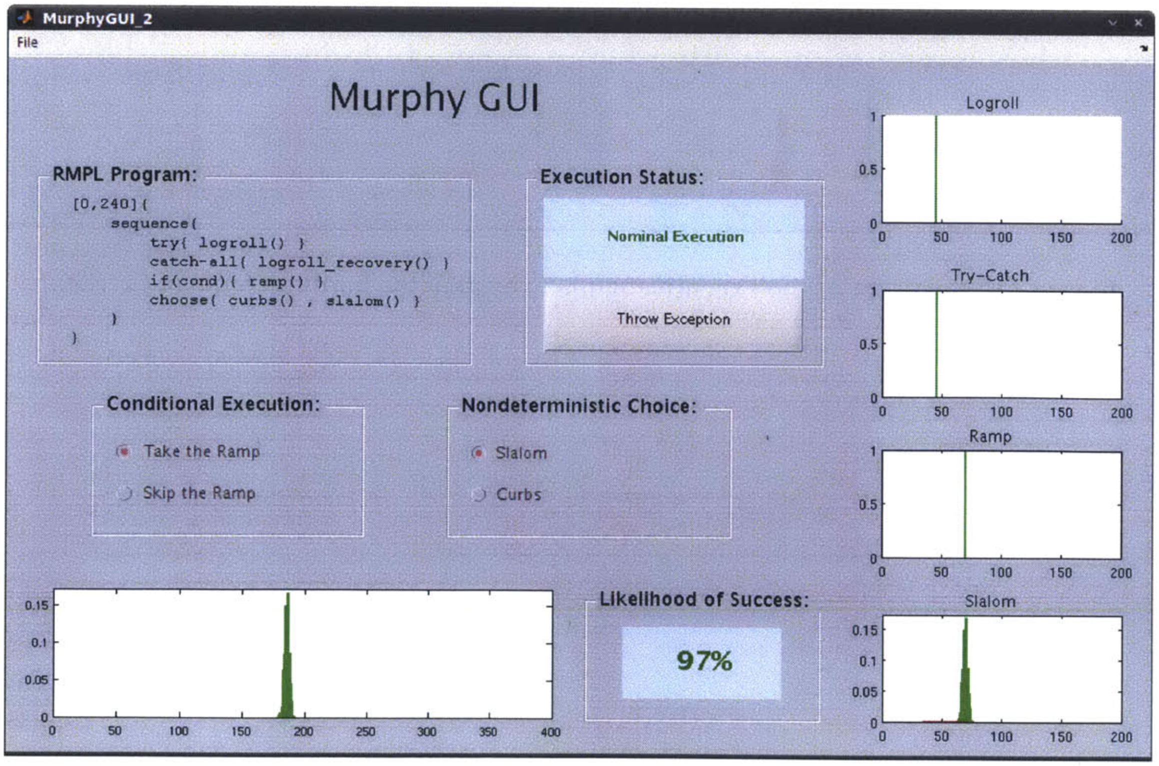
Figure 7.6: Snapshot of the executive while executing the ramp activity.

Bear robot traversing the slalom: After the robot finishes the ramp activity, it decides whether to traverse the slalom or curbs. In this scenario, the curbs activity was given a higher likelihood of failure, so the executive chooses to go with the slalom. The slalom has an overall likelihood of failure of 3%, and it is the last activity to execute before reaching the wounded soldier, hence a overall likelihood of success of 97%.



```
RSP:  0 RSR:-70 RSY:-89 REP:-127 RWR:  0 RWP:-30
LHP:-65 LKP:140 RHP:-65 RKP:140 LTV:0.60 RTV:0.60
```

Figure 7.7: BEAR robot traversing the slalom.



RMPL Program:

```
[0,240]{
    sequence(
        try{ logroll() }
        catch-all( logroll_recovery() )
        if(cond){ ramp() }
        choose( curbs() , slalom() )
    )
}
```

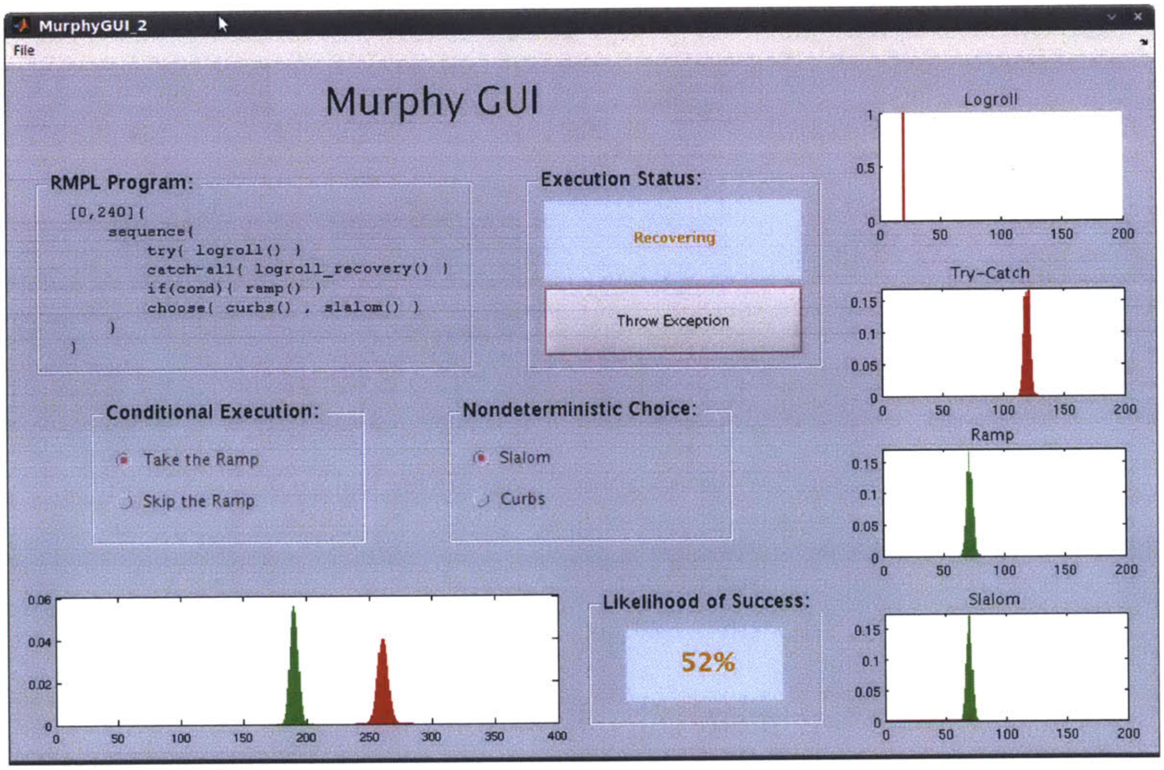
Conditional Execution:
● Take the Ramp
○ Skip the Ramp

Nondeterministic Choice:
● Slalom
○ Curbs

Likelihood of Success: 97%

Figure 7.8: Snapshot of the executive while executing the slalom activity.

An alternative scenario, an exception occurs during the hurdles: If we press the exception button on the GUI during the hurdles activity, the robot must go back and start over. In Figure 7.9 we show the effect this has on the robot's probability of success. Now, the robot will fail unless it is able to skip ramp, which we assume occurs with 50% probability.



```
[0,240]{
    sequence{
        try{ logroll() }
        catch-all{ logroll_recovery() }
        if(cond){ ramp() }
        choose{ curbs() , slalom() }
    }
}
```

Execution Status: Recovering

Conditional Execution: ● Take the Ramp ○ Skip the Ramp

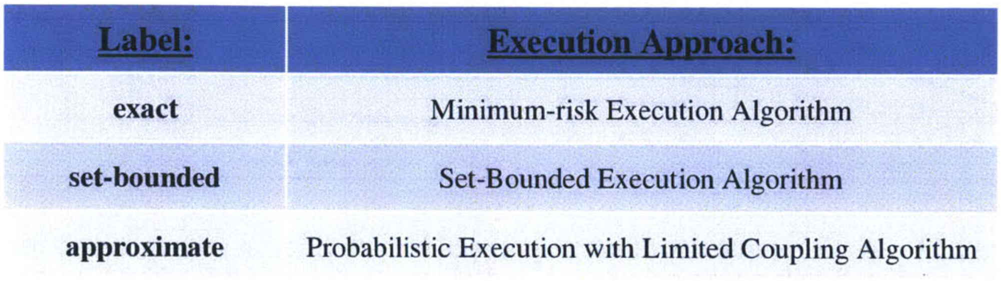
Nondeterministic Choice: ● Slalom ○ Curbs

Likelihood of Success: 52%

Figure 7.9: Snapshot of the executive while executing the hurdles recovery activity.

7.2 Performance Benchmarks

In this section we characterize and compare the performance of each execution approach. The three execution approaches are labeled and referred to as follows:



Label:	Execution Approach:
exact	Minimum-risk Execution Algorithm
set-bounded	Set-Bounded Execution Algorithm
approximate	Probabilistic Execution with Limited Coupling Algorithm

Figure 7.10: Labels for each execution approach.

In this section, we first compare the time and space complexity of the sequence and parallel operators for each approach. Recall that these two operators are identified in Chapter 6 as the primary sources of complexity in the exact approach. Next, we benchmark compilation time as a function of the temporal resolution at which activities are scheduled. We record program compilation times for each execution approach at scheduling resolutions from 1 to 20 seconds. Then, we demonstrate by example, the limitations of the set-bounded and approximate approaches relative to the exact approach.

We characterize the time and space complexity of the sequence and parallel operators for each approach in Figures 7.11 and 7.12. The experimental results for the exact approach match the complexity analysis performed in Chapter 6. The sequence operator is polynomial in both time and space, while the parallel operator is exponential in both time and space. In Figures 7.11 and 7.12 we also characterize the complexity of the set-bounded and approximate approaches. For the set-bounded approach, the sequence operator is polynomial in both time and space, and

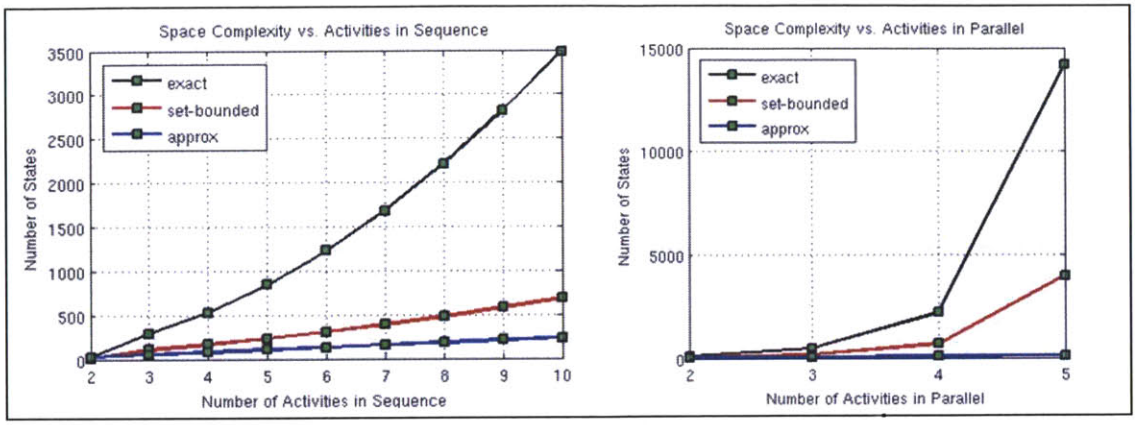


Figure 7.11: Space complexity of each approach for activities in sequence and parallel.

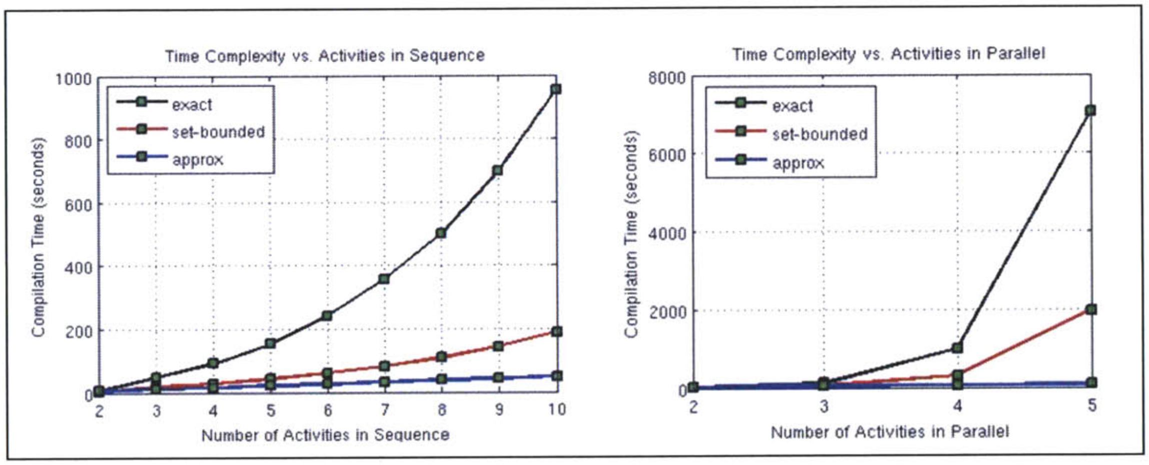


Figure 7.12: Time complexity of each approach for activities in sequence and parallel.

the parallel operator is exponential in both time and space, just as in the exact approach. The set-bounded approach outperforms the exact approach, however, by neglecting to explore program states from which execution is not guaranteed to succeed. Finally, the sequence and parallel operators in the approximate approach are linear in both time and space. This improvement in complexity comes at the cost of optimality in risk avoidance, however. We characterize this tradeoff later in the section.

Next, we benchmark compilation time as a function of the temporal resolution of activity scheduling. As shown in Figure 7.13, the approximate approach significantly outperforms both the set-bounded and exact approaches. For example, the set-bounded and exact approaches take approximately 1 minute and 10 minutes, respectively, to compile the wounded soldier example program at a temporal scheduling resolution of approximately 10 seconds. The approximate approach, on the other hand, compiles in a number of seconds, and can schedule activities at a resolution of several seconds.

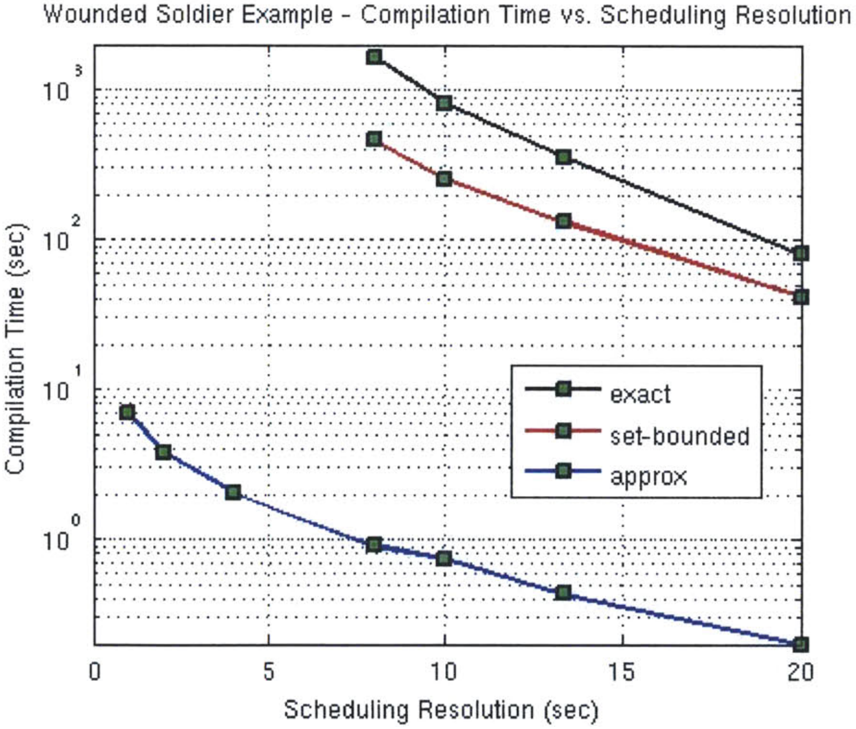


Figure 7.13: Compilation Time vs. Scheduling Resolution for the Wounded Soldier Example.

Next, we characterize the limitation of the set-bounded approach. In order to achieve greater tractability than the exact approach, the set-bounded approach only computes the decision policy for program states from which execution is guaranteed to succeed. The reason for this limitation is that the set-bounded approach does not reason over the stochastic activity models provided as input, and therefore has no knowledge of which execution paths are more likely to occur than others. Therefore, proving successful execution is an all or nothing proposition; either all executions end in success or no guarantee can be made at all. This compromise results in the set-bounded algorithm ignoring a number of program states from which execution might succeed.

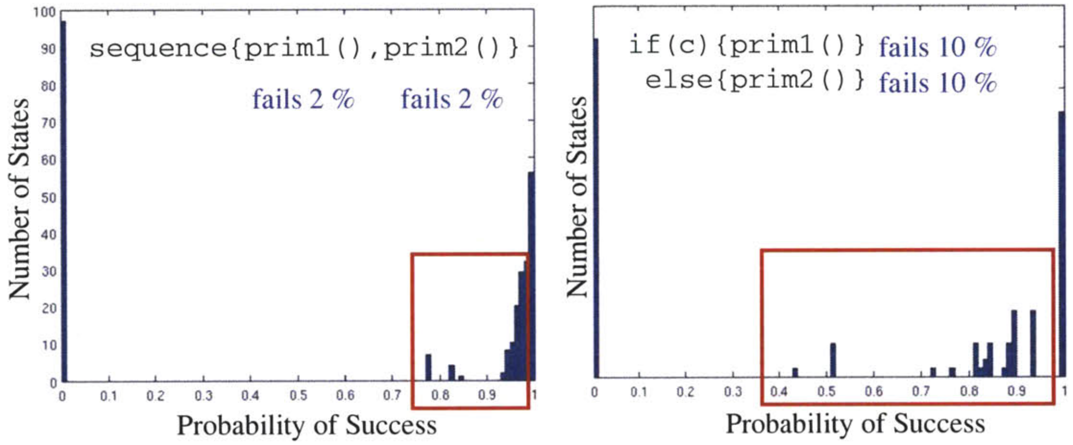


Figure 7.14: States not considered for execution by the set-bounded approach.

Consider two simple examples depicted in Figure 7.14. On the left, two activities are executed in sequence, *prim1()* and *prim2()*. Each activity has a probability of failure of 2%. A histogram is plotted (obtained by running the exact algorithm) which counts the number of states in the program with a given probability of success. In this example, there are many program states from which the probability of successful execution is within 80% to 99%, as indicated by the red box. The set-bounded execution approach is not able to consider these program states, which in-fact do lead to successful program execution with high probability, because it is not able to compute such probabilities. The example on the right in Figure 7.14 depicts another such example.

Next, we discuss the limitations of the approximate execution approach, which takes a simplified approach to sequential and parallel execution. The benefit of the approximate approach is that it improves tractability by intentionally not considering all combinations of available choices in the construction of a decision policy. First, we briefly summarize the simplified approach to sequential and parallel execution. Then we discuss two specific counter-examples, each of which gets the approximate algorithm into trouble. Finally, we show that the approximate algorithm performs well on two classes of stochastic activity models found commonly in robotic domains.

The approximate sequence operator improves tractability by scheduling activities proportionally, meaning that each activity in sequence is scheduled to end early if the sequence is scheduled to end early, and vice versa. This simplified approach neglects combinations of scheduling choices in which some activities are scheduled to end early and others late. In general, such combinations of scheduling choices are not fruitful, however, there are counter-examples, one of which we describe below. Similarly, the parallel operator schedules activities proportionally, meaning that a common end time is agreed upon by all activities prior to execution, and then each thread executes toward that deadline independently. In general, it makes sense to schedule each activity to complete at the same time, since a parallel construct finishes when all activities are likewise finished. However, there are counter-examples, one of which we describe below.

The argument to make in favor of the above described simplified approaches to sequential and parallel execution is that they work well for the majority of stochastic activity models that are encountered in robotic domains. While it is possible to construct counter-examples that break each simplified approach, such examples are counter-intuitive or pathological, and they are easy to check for a-priori. To prove or refute the above claims, generally, for robotic domains would require an exhaustive amount of test examples and is a difficult empirical task. Here, we present two counter-examples, one which breaks each simplified approach, and we discuss the pathological nature of each counter-example. Then, we demonstrate two common classes of stochastic activity models for which both simplified approaches work well.

Now, we consider a counter-example which breaks the simplified approach to sequential execution. Consider two primitive activities that execute in sequence. The stochastic activity model for each activity is depicted in Figure 7.15. The first activity has a 50% chance of failure if it is scheduled to complete early, and has a 0% chance of failure if it is scheduled to complete late. The second activity has a 0% chance of failure if it is scheduled to complete early, and a 50% chance of failure if it is scheduled to complete late. The exact algorithm will try all combinations of scheduling

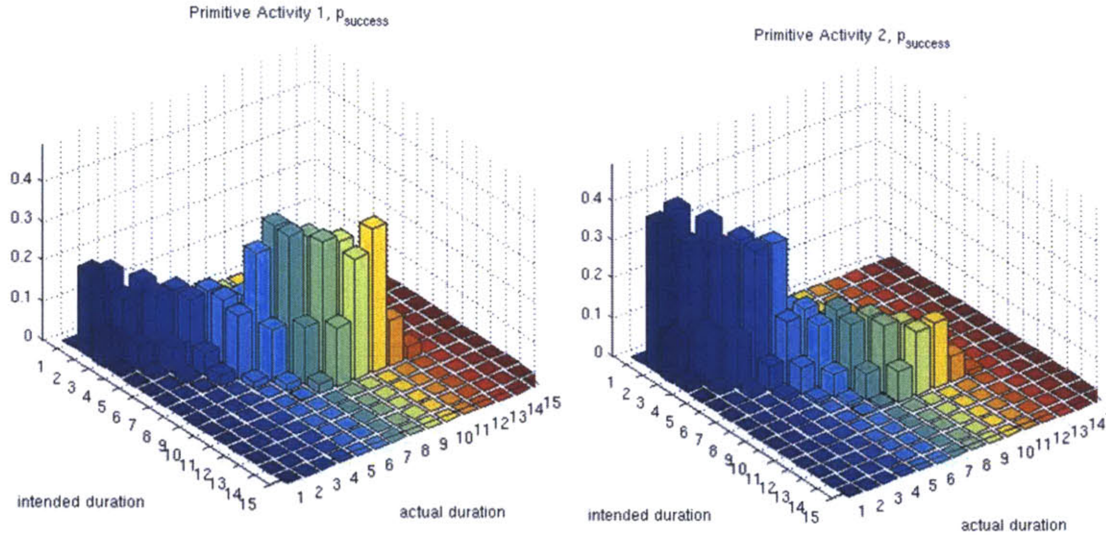


Figure 7.15: Two tricky activities in sequence.

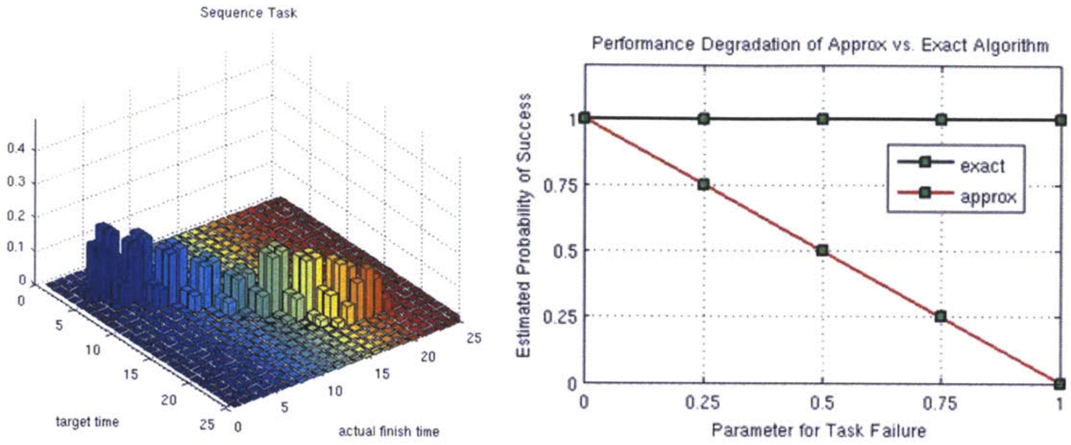


Figure 7.16: Performance degradation of approximate vs. exact probabilistic algorithms for two tricky activities in sequence.

decisions and deduce that the minimum-risk decision policy is to schedule the first activity to end late and the second activity to end early. The resulting minimum-risk decision policy incurs zero risk. The approximate approach, however, will schedule these activities proportionally. In order to avoid reasoning over all possible combinations of decisions, the simplified sequential operator will either schedule both of the activities early or both of the activities late. With this simplified approach, the resulting probability of success is always 50% as depicted by the plot on the left in Figure 7.16. For this specific example, we can make the simplified approach perform arbitrarily worse than the exact approach, as demonstrated in Figure 7.16, by increasing the discrepancy between the low-risk and high-risk probabilities in the stochastic activity models of Figure 7.15.

It is useful to note, however, the pathological nature of this counter example. If the low-risk distributions overlapped just slightly in Figure 7.15, the simplified approach would find a zero risk policy. Also, a stochastic activity model with a sharp change in probability of success, with the next activity in sequence having the exact opposite sharp change in probability of success seems contrived and uncharacteristic of activities that would be modeled and executed in practice by a robotic system.

Finally, we consider a similar counter-example, in order to break the simplified approach to parallel execution. Consider two primitive activities that execute in parallel with the same stochastic activity distributions presented in Figure 7.15. The exact algorithm will try all combinations of scheduling decisions in parallel in order to deduce that the minimum-risk decision policy is to schedule the first activity to end late and the second activity to end early, despite the fact that they will be executing in parallel. The resulting minimum-risk decision policy incurs zero risk. The simplified approach to parallel execution, on the other hand, requires the scheduled duration for each activity to be the same, in order to avoid reasoning over all possible combinations of scheduling decisions. Similarly to the simplified sequential operator, the simplified parallel operator will either schedule both of the activities early or both of the activities late. With this simplified approach, the resulting probability of success is again at most 50%.

We can generalize from the two prior examples a useful trend. Stochastic activity models with a failure probability that varies with target duration can fool both the simplified approach to sequential and parallel execution that are developed in Chapter 6. However, the counter-examples studied here are quite specific, and could easily be checked for with a pre-processing step. Most realistic scenarios will have failure probabilities that are only somewhat correlated with activity duration, are similarly distributed over the duration, and don't have failure distributions that are extreme opposites.

Next, consider two activities executing in sequence, each with failure probabilities of 40% that are independent of target duration. As shown in Figure 7.17 below, this example shows that when an activity's failure probability is independent of activity duration, the ability of the approximate approach to minimize risk mirrors that of the exact approach with a much reduced compilation time.

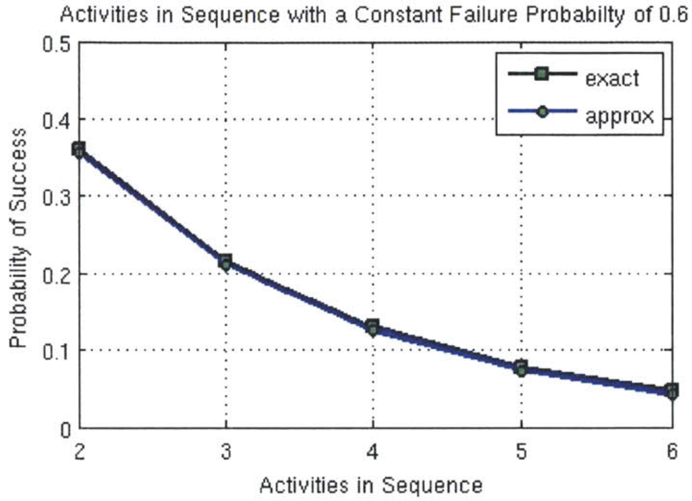


Figure 7.17: Performance degradation of approximate vs. exact probabilistic algorithms for two activities with failure probabilities independent of target duration.

This concludes the Experimental Results chapter.

Chapter 8 – Conclusions, Discussion and Future Work

This chapter concludes the thesis with a summary of our key accomplishments, a discussion of the benefits, drawbacks, and limitations in scope of our current approach, and outlines concepts for future work.

8.1 Summary of Accomplishments

In this thesis, we argue that autonomous robots can improve robustness by computing and reasoning explicitly about risk. We define robustness as the avoidance of program failure. We develop a novel, risk-minimizing program executive, called Murphy, which utilizes program flexibility and estimation of risk in order to make runtime decisions that minimize the probability of program failure. Murphy utilizes two forms of program flexibility in order to minimize two forms of program risk. The two forms of program flexibility are 1) flexible scheduling of activity timing and 2) redundant choice between subprocedures, while the two forms of program risk are 1) exceptions arising from activity failures and 2) exceptions arising from timing constraint violations in a program. Functionally, Murphy takes two inputs and computes two outputs. The two inputs are 1) a program written in a nondeterministic variant of the Reactive Model-based Programming Language (RMPL) and 2) a set of stochastic activity failure models, while the two outputs are, 1) a risk-minimizing decision policy and 2) value function. In order to execute with low latency, Murphy computes the decision policy and value function offline, as a compilation step prior to program execution.

Our approach makes two key technical contributions, a precise syntax and semantics for a nondeterministic variant of RMPL, and three risk-minimizing execution approaches, namely, Minimum-risk Program Execution (Chapter 4), Set-bounded Dynamic Execution (Chapter 5), and Risk-minimizing Execution with Limited Coupling (Chapter 6). We validate the approach on a humanoid simulator, and we perform complexity experiments in order to determine the tractability of each approach. The minimum-risk approach requires time and space that is worst-case exponential in the number of activities in a program, and is therefore only feasible for small programs. The set-bounded approach, although worst-case exponential, achieves better performance than the minimum-risk approach by ignoring program states from which execution is not guaranteed to succeed. The approximate, limited-coupling approach achieves linear complexity in time and space, at the cost of being less risk-adverse than the minimum-risk approach. We demonstrate the approximate approach performing as well as the minimum-risk approach on some example programs, and we also discover examples where the approximate approach performs significantly worse than the minimum-risk approach.

8.2 Discussion and Future Work

We break up this section into a number of interesting discussion topics, each of which leads to one or more suggestions for future work. The topics discussed are as follows:

1. Infusing machine learning into embedded programming for robotic systems
2. Defining program failure as an uncaught program exception
3. Other types of program flexibility
4. Other types of program risk
5. Conditioning execution based on the current risk estimate
6. Modeling conditions
7. Set-bounded activity model is physically unrealizable in robotic domains
8. Framing set-bounded execution as a restricted instance of minimum-risk execution

1. Infusing machine learning into embedded programming for robotic systems:

In this section, we think more broadly about the implications of this thesis work. More specifically, we discuss this thesis in the context of infusing machine learning into embedded programming for automated robotic systems.

Several recent successes in AI have been the result of machine learning algorithms, for example, speech and vision recognition, and game playing, such as chess and jeopardy. These problems are framed as sequential decision making problems, for which learning algorithms are well-suited. Extending machine learning into the domain of automated control for robotic systems has proven difficult, however, due to several complicating factors, such as time, dynamics, resources, and uncertain operating environments. These factors are difficult to encode in existing machine learning formalisms, and simultaneously they are unable to be abstracted away in a meaningful manner.

This thesis takes a small step towards infusing machine learning into the embedded programs we currently use to control automated robotic systems. We show that control programs including time, multithreading, sensing, and exception handling can be modeled compactly using hierarchical timed automata, and that machine learning can be performed over such programs in order to minimize an important metric, *risk of failure*. Alternatively, a robotic system could learn to optimize against a variety of other interesting metrics, such as, minimizing project completion time, minimizing fuel or energy use, or maximizing physical distance between robots and workers.

In any context where a control program needs to be run, and some metric needs to be optimized, machine learning could be employed in order to optimize the robotic system's behavior. To enable effective learning, the correct type and degree of flexibility (i.e.

nondeterminism) must be afforded to the robot's control program. Too little flexibility leaves no room for optimization, while too much flexibility does not allow effective control over the system and hinders tractability in the underlying learning problem.

In this thesis, we have explored two types of program flexibility, redundant method selection and flexible activity scheduling. We show that control programs modeling time, sensing, and exception handling, in addition to flexible activity scheduling and redundant method selection, can be reformulated efficiently into Markov decision processes, enabling machine learning in the context of robotic control.

To summarize, this thesis demonstrates that infusing machine learning into embedded programming for automated robotic systems is possible through a series of steps, 1) add in a restricted form of flexibility to the control program to enable learning, 2) define a metric against which to optimize behavior, 3) develop a compact reformulation of the control program into an established machine learning formalism, and 4) devise the architecture in which the executive pre-computes, or otherwise iteratively solves for, the decisions it needs to make in order to best utilize the program flexibility at runtime.

2. Defining program failure as an uncaught program exception:

In this subsection, we discuss our justification for defining program failure as an uncaught program exception. Then, we discuss other options for defining failure, and we conjecture how our execution approach might be modified to handle them. In this discussion it is helpful to distinguish between three types of failure: failure in the plant being controlled (such as a flat tire, or a broken robot arm), failure in an individual activity or subprocedure (which is represented by a thrown exception in the control program), and a failure in the control program (which is represented by an uncaught exception), beyond which execution cannot be correctly defined by the control program.

In this thesis, we define failure as an uncaught exception in the control program. Success, on the other hand, is defined as reaching the end of the control program, which given our definition of the problem space, will occur in all instances where there is no uncaught program exception. Therefore, risk is defined as the probability of ending up in the uncaught exception state instead of the program end state. This is a useful interpretation of risk in robotic applications where there is no notion of partial success, and a program must be executed to completion in order to succeed, such as the wounded soldier example. Either the soldier is rescued or not.

More generally, however, we can discuss what it means to achieve partial success in executing a program. Consider, for example, a delivery robot with more than one package. In this case, how do we define success, or risk? One can also consider robotic applications in which accomplishing a goal is more important than avoiding risk. In such cases, minimizing risk may not be the primary objective, but a secondary one. In this thesis, we purposefully limit the scope of our focus to risk-minimizing behavior within the context of a flexible program. Thinking beyond the scope of this thesis, however, one way to represent partial success is to assign utility to an activity or a subprogram. Then, instead of solely minimizing risk, an executive can decide to optimize against a combined metric of risk and utility. In this way, an executive could execute across a spectrum from being solely risk-adverse (ignoring utility) to solely greedy (ignoring risk).

To support a combined risk-utility execution approach, one could define a syntax construct that assigns utility to completing an activity or sub-program. To attain the specified utility, a subprogram or activity must execute successfully (i.e. without failing due to a thrown exception). Handling utility maximization in our existing execution approach is straightforward; one just needs to augment the reward function when computing the MDP. In our current approach, a reward of 1 is assigned for entering the program end state. All other transitions have zero reward. To account for utility, one can assign rewards to transitions leaving specified program states.

3. Other types of program flexibility:

In this subsection, we discuss several other types of nondeterminism that could help to improve robustness. First, we consider nondeterministic variable assignment. A language construct that allows variables to be assigned nondeterministically by the executive could be used to model resource allocation, and task assignment problems. Consider, for example, a group of construction robots that must decide how to share a limited set of tools, or a fleet of robots deciding how to allocate a required set of tasks.

Another type of nondeterminism is thread synchronization and information sharing. In the variant of RMPL considered in this thesis, there are no constructs which allow for user-defined information to flow between threads. If this type of interaction is allowed, then the executive must account for a number of scenarios in which each parallel thread interacts with other threads. This type of interaction is necessary in resource planning and cooperative construction tasks, for example.

Finally, we mention potentially infinite control programs. In this thesis, we carefully define the syntax such that infinite loops are not possible. However, in programming robotic systems it is often helpful to define loops for which there may be no well defined stopping condition. For an executive to support this type of nondeterminism, more advanced techniques from model-checking and first-order logic might be useful.

4. Other types of program risk:

In this thesis, we only consider two types of risk, exceptions from failed activities and exceptions from failed timing constraints in a program. There are many other types of risk that would be useful to monitor and avoid in robotic systems. For example, one could constrain the relative motion of two x-DOF factory robots from performing operations that could potentially create human crush zones. It also might be useful for a robot to distinguish between and track several types of risk at once. For example, a surveillance robot might weigh the risk of detection vs. the risk of making it to a transmission location within a specified window of time in order to uplink valuable data.

5. Conditioning execution based on the current risk estimate:

One interesting idea is to condition program execution based on the current risk estimate in a subprogram. Currently, our executive makes decisions that minimize risk overall, but a more flexible and general approach to risk-aware programming could be to label subprograms, and then allow the user to reference the runtime probability of success of the labeled subprogram. For example, the execution in one subprogram could proceed conditionally based on the probability of success in another subprogram, *if(label.P_success > 0.90){ }else{ }*.

6. Modeling conditions:

Another related topic is how to accurately model the probability that a conditional statement will evaluate to true at runtime, *if(condition){ }else{ }*. In this thesis, we assume a static probability value over the life of a program. This is a gross approximation, however, as the probability a condition will evaluate to true at runtime may vary based on many program parameters. For example, if a branch in robotic control program depends on a sensor reading, then that sensor could be modeled in more detail, in order to provide a better estimate of the which branch might occur at runtime. Also, the likelihood of specific program branches being taken at runtime can be learned as a program executes over time. This type of predictive behavior is already performed in high-speed processors in order to load data from memory that might be needed by an executing process.

7. Set-bounded activity model is physically unrealizable in robotic domains:

In this subsection, we discuss an important issue with the set-bounded activity model, which prompts us to develop the stochastic activity model in Chapter 4. A set-bounded temporal constraint, [lb,ub], is defined as either controllable or uncontrollable. Activities performed by robotic systems, however, are often not fully controllable or fully uncontrollable, but rather somewhere in-between. Consider, for example, a robot driving from point A to point B with a scheduled duration of 1 minute. To perform this activity, the robot can estimate the distance to travel and then divide by the scheduled duration in order to determine its average speed. Subsequently, the robot will arrive at the destination close to 1 minute. However, there will always be some uncertainty in how close to the 1 minute target duration the robot is able to arrive at the destination. This will depend on the robot's environment and the physical characteristics of the robot. This causes a problem with the set-bounded approach of reasoning over a network of fully controllable and fully uncontrollable timing durations in order to construct a temporal plan, since the resulting network is not physically realizable by the robot.

This leads to a discrepancy between the temporal plan output by a planner, and the execution approach used at runtime. This discrepancy has yet to be resolved formally in the literature.

In this thesis, we develop the stochastic activity model in order to account for such uncertainty, and to sidestep the issue with set-bounded temporal planning raised above. In performing temporal planning for robotic systems, the basic activity model used to reason about activity timing, should at a minimum, be physically realizable by the robotic system being modeled. Either the set-bounded execution approach should be extended to support some uncertainty in otherwise controllable set-bounded activities, or the temporal planning community should adopt a physically realizable activity representation similar in spirit to the stochastic activity model developed in this thesis.

8. Framing set-bounded execution as a restricted instance of minimum-risk execution:

In this subsection, we discuss how set-bounded dynamic execution (Chapter 5) can be viewed as a restricted instance of minimum-risk execution (Chapter 4). Recall that the set-bounded approach only considers states from which execution is guaranteed to succeed. Also recall that the minimum-risk algorithm outputs a minimum-risk value function, $V^*(s)$, which computes the probability of successful program execution from each state in an HTA. Therefore, for each program state, s, such that $V^*(s)=1$, successful program execution is guaranteed. Therefore, ignoring the additional computational burden entailed, a set-bounded decision policy can be obtained from the minimum-risk decision policy by simply omitting any program states from which successful program execution is not guaranteed. Mathematically, this corresponds flooring the minimum risk value function, $floor(V^*(s))$. By flooring the minimum risk value function, $floor(V^*(s))$, we retain only states, s, from which successful execution is guaranteed.

Bibliography

[Alberts and Hayes 2006] Alberts, D., and Hayes, R., "Understanding Command and Control", The Command and Control Research Program Publication Series, ISBN 1-893723-17-8, 2006.

[Alur and Dill, 1990] R. Alur and D. L. Dill, "Automata for modeling real-time systems," in Lecture Notes in Computer Science, Automata, Languages and Programming. Heidelberg, Germany: Springer-Verlag, 1990, vol. 443, pp. 322–335.

[Alur and Dill, 1994] , "A theory of timed automata," Theor. Comput. Sci., vol. 126, pp. 183–235, 1994.

[Alur, 1999] Rajeev Alur, Timed Automata, Theoretical Computer Science, 1999.

[Ambrose, et. al., 2000] Ambrose, et. al., "Robonaut: NASA's Space Humanoid." IEEE Intelligent Systems Journal, Volume 15, p57-63, 2000.

[Andre and Russel, 2002] D. Andre and S. Russell. State Abstraction for Programmable Reinforcement Learning Agents. in Proceedings of the 19th Conference on Artificial Intelligence (AAAI-02). AAAI Press.

[Andre and Russel, 2001] Andre, D. and Russel, S. Programmable reinforcement learning agents. In Dietterich, T. G., Tresp, V., and Leen, T. K., editors, Proceedings of Advances in Neural Information Processing Systems 13, pages 1019{1025, Cambridge, MA. MIT Press.

[Atwood and Klein, 2007] Atwood and Klein, "Vecna's Battlefield Extraction-Assist Robot BEAR". Robot Magazine. April, 25, 2007.

[Baier, et. al. 2007] Jorge A. Baier, Christian Fritz, Sheila A. McIlraith: Exploiting Procedural Domain Control Knowledge in State-of-the-Art Planners. ICAPS 2007: 26-33

[Bellman 1957] R. Bellman. *A Markovian Decision Process.* Journal of Mathematics and Mechanics 6, 1957.

[Bellman 1960] R. E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957. Dover paperback edition (2003), ISBN 0486428095.

[Benveniste, et. al. 2003] Benveniste, Caspi, Edwards, Halbwachs, Quernic, Simone. The Synchronous Languages 12 Years Later. Proceedings of the IEEE, Vol. 91, No. 1, January 2003.

[Berry and Gontheir, 1992] Gérard Berry and Georges Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation." *Science of Computer Programming*, 19(2):pp. 87--152, 1992.

[Bertsekas and Tsitsiklis, 2008] Dimitri Bertsekas and John Tsitsiklis, Introduction to Probability, 2nd Edition., Athena Scientific, ISBN: 978-1-886529-23-6. July 2008.

[Blackmore, et. al., 2007] Blackmore, L., S. Gil, S. Chung and B. C. Williams, "Model Learning for Switching Linear Systems with Autonomous Mode Transitions," Proceedings of the Control and Decision Conference, New Orleans, December, 2007.

[Boutilier, et. al., 2000] Craig Boutilier, Ray Reiter, Mikhail Soutchanski, and Sebastian Thrun, ‘Decision-theoretic, high-level agent programming in the situation calculus’, in Proceedings of the 17th Conference on Artificial Intelligence (AAAI-00), pp. 355–362, Austin, TX, (July 30–3 2000). AAAI Press.

[Caspi, et. al., 1987] Caspi, Pilaud, Halbwachs, and Plaice, “Lustre, a Declarative Language for Real-Time Programming, In Proceedings of the Conference on Principles of Programming Langauges, Munich, 1987.

[David, 2003] A. David. Hierarchical Modeling and Analysis of Timed Systems. PhD Thesis, Uppsala University, 2003.

[Dean and Lin, 1995] Dean, and Shieu-Hong Lin, Techniques for Planning in Stochastic Domains, in Proceedings of IJCAI’95, Montreal, 1995.

[Dechter, et. al., 1991] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. Artificial Intelligence, 49(1-3):61--95, 1991.

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

[Dietterich, 2000] Dietterich, T. G. 2000. Hierarchical reinforcement learning with the maxq value function decomposition. Journal of Artificial Intelligence Research 13:227–303.

[Doucet, et. al. 2001] Doucet, N, de Freitas, and Gordon, N. Sequential Monte Carlo Methods in Practice. Springer Verlag. 2001.

[Effinger, 2006] Effinger, R. "Optimal Temporal Planning at Reactive Time Scales via Dynamic Backtracking Branch and Bound," S.M. Thesis, Massachusetts Institute of Technology, September 2006.

[Ferrein, et. al., 2003] A. Ferrein, Ch. Fritz, G. Lakemeyer. Extending DT-Golog with Options. In Proc of the 18th International Joint Conference on Artificial Intelligence.

[Firby, 1987] Firby, R. "An investigation into reactive planning in complex domains." *Proceedings of the 6th National Conference on AI, Seattle, WA, July 1987,* 1987.

[Fritz 2003] Christian Fritz, "Integrating Decision-Theoretic Planning and Programming for Robot Control in Hightly Dynamic Domains", M.Sc. Thesis, RWTC Aachen University, Germany.

[Gat, 1996] Gat, E. "Esl: A language for supporting robust plan execution in embedded autonomous agents." *AAAI Fall Symposium: Issues in Plan Execution*, Cambridge, MA, 1996.

[Guernic, el. al., 1991] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with Signal. *Proceedings of the IEEE*, 79(9):1321-1336, 1991.

[Halbwachs, et. al., 1991] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous programming language Lustre. Proceedings of the IEEE, 79(9):1305-1320, 1991.

[Halbwachs, 1993] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Series in Engineering and Computer Science. Kluwer Academic, 1993.

[Hamon and Rushby, 2004] G. Hamon and J. Rushby. An Operational Semantics for Stateflow. Fundamental Approaches to Software Engineering, Barcelona, Spain, March 2004. Springer Verlag LNCS 2984, pp. 229-243.

[Harel, 1987] Statecharts: A visual formulation for complex systems. Science of Computer Programming, 8(3):231-274, 1987.

[Hofmann and Williams, 2006] Andreas G. Hofmann and Brian C. Williams, "Robust Execution of Temporally Flexible Plans for Bipedal Walking Devices," *Proceedings of the International Conference on Automated Planning and Scheduling*, Cumbria, UK, June 2006.

[Hunt, 2000] J Hunt. The Unified Process for Practitioners: Object-oriented Design, UML and Java. Springer, 2000.

[Ingham, et. al., 2001] M. Ingham, R. Ragno and B. Williams, “A Reactive Model-based Programming Language for Robotic Space Explorers,” Proceedings of the Sixth International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey, Montreal, Canada, June 2001.

[Ingham, 2003] M. Ingham,"Timed Model-based Programming: Executable Specifications for Robust Mission-Critical Sequences," Doctoral Thesis, MIT, May 2003.

[Kaelbling, et. al., 1996] Kaelbling, L. P.; Littman, M. L.; and Moore, A. W. 1996. Reinforcement learning: A survey. Journal of Artificial Intelligence Research 4:237–285.

[Kim, et. al., 2001] P. Kim, B. Williams, and M. Abramson. Executing Reactive, Model-based Programs through Graph-based Temporal Planning. In *Proceedings of IJCAI-2001, Seattle, WA*, 2001.

[Lee, 2002] Lee, E. "Embedded Software," *Advances in Computers*. Vol. 56, Academic Press, London, 2002.

[Le Guernic, et. al., 1985] Le Guernic, Beneveniste, Bournai, and Gauthier, "Signal: A Data Flow Oriented Language for Signal Processing", IRISA Repor, IRISA, Rennes, France, 1985.

[Levesque, et. al., 1997] H.J. Levesque, R. Reiter, Y. Lesp´erance, F. Lin, and R. Scherl. GOLOG: a logic programming language for dynamic domains. J. of Logic Programming, Special Issue on Actions, 31(1-3):59–83, 1997.

[Lin and Makedon, 2000] Yong Lin and Fillia Makedon, Planning in Markov Stochastic Task Domains, International Journal of Artificial Intelligence and Expert Systems, Volume 1: Issue 3, 2000.

[McGovern, et. al. 1998] McGovern, Precup, Ravindran, Singh, Sutton, "Hierarchical Optimal Control of MDPs", In Proceedings of the 10th Yale Workshop on Adaptive and Learning Systems, 1998.

[Meuleau, et. al., 1998] Meuleau, N., Hauskrecht, M., Kim, K, Peskin, L. Kaebling, L. Dean, Tl, Boutilier, C. "Solving Very Large Weakly Coupled Markov Decision Processes", in Proceedings of the Converence on Uncertainty in AI, 1998.

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

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

[Parker, 2002] D. Parker. Implementation of Symbolic Model Checking for Probabilistic Systems. Ph.D. Thesis, School of Computer Science, University of Birmingham. August 2002.

[Parr, 1998] Parr, R. 1998. "Flexible Decomposition Algorithms for Weakly Coupled Markov Decision Problems" in Proceedings of the Conference on Uncertainty in AI, 1998.

[Parr and Russel 1998] Ronald Parr, and Stuart Russell, "Reinforcement learning with hierarchies of machines", Advances in Neural Information Processing Systems 10, 1998.

[Pettersson, 1999] P. Pettersson. Modelling and Verification of Real-Time Systems Using Timed Automata:Theory and Practice, Ph.D. Thesis, Technical Report DoCs 99/101, Department of Computer Systems, Uppsala University, February 1999.

[Precup and Sutton, 1998] Precup, D., and Sutton, R. 1998. Multi-time models for temporally abstract planning. In Kearns, M., ed., Advances in Neural Information Processing Systems 10. Cambridge, Massachusetts: MIT Press.

[Puterman, 1994] Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley.

[Schiex, et. al., 1995] Schiex, T., et. al. 1995. Valued constraint satisfaction problems. In *Proceedings of IJCAI'95*, 631-637.

[Simmons, 1998] R. Simmons. A task description language for robot control. In proceedings of the Conference on Intelligent Robots and Systems (IROS), Victoria Canada, 1998.

[Singh and Cohn, 1998] Sing S, Cohn D., "How to Dynamically Merge Markov Decision Processes", In NIPS 11, 1998.

[Slonneger 1995] Slonneger, K. "Formal Syntax and Semantics of Programming Languages.", Addison-Wesley Publishing Company, 1995.

[Sobol, 1975] Sobol, I.M. "The Monte Carlo Method", Mir Publishers, Moscow, 1975.

[Stergiou and Koubarakis, 1998] Stergiou, K. and Koubarakis, M., 2000. Backtracking algorithms for disjunctions of temporal constraints. AI 120, 81-117.

[Sutton et. al., 1999] Between MDPs and Semi-MDPs: A Framework for Termporal Abstraction in Reinforcement Learning, Artificial Intelligence 112:181-211, 1999.

[Tsamardinos, et. al., 1998] Tsamardinos, Muscettola, and Morris, "Fast Transformation of Temporal Plans for Efficient Execution" In Proceedings of the Thirteenth National Conference on Artificial Intelligence, 1998.

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

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

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

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

[Wang and Mahadevan, 1999] Gang Wang, Sridhar Mahadevan: Hierarchical Optimization of Policy-Coupled Semi-Markov Decision Processes. ICML 1999: 464-473.

[Williams, et. al., 2003] Brian C. Williams, Michel Ingham, Seung H. Chung, and Paul H. Elliott, "Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers," Invited Paper, *Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software*, vol. 91, no. 1, pp. 212-237, January 2003.

[Williams, et. al., 2004] Brian C. Williams, Michel Ingham, Seung Chung, Paul Elliott, and Michael Hofbaur,"Model-based Programming of Fault-Aware Systems," *AI Magazine*, vol. 24, no. 4, pp. 61-75, 2004.