CORE
Metadata, citation and similar papers at core.ac.uk
Provided by DSpace@MIT

massachusetts institute of technology — artificial intelligence laboratory

Safe Distributed Coordination of Heterogeneous Robots through Dynamic Simple Temporal Networks

Andreas F. Wehowsky

AI Technical Report 2003-012 May 2003

© 2003 massachusetts institute of technology, cambridge, ma 02139 usa — www.ai.mit.edu

Safe Distributed Coordination of Heterogeneous Robots through Dynamic Simple Temporal Networks

by

Andreas Frederik Wehowsky

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

Masters of Science in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2003

© Massachusetts Institute of Technology 2003. All rights reserved.

Certified by: Brian C. Williams
Associate Professor
Thesis Supervisor

Accepted by: Edward M. Greitzer
H.N. Slater Professor of Aeronautics and Astronautics, Chair,
Committee on Graduate Students

Abstract

Research on autonomous intelligent systems has focused on how robots can robustly carry out missions in uncertain and harsh environments with very little or no human intervention. Robotic execution languages such as RAPs, ESL, and TDL improve robustness by managing functionally redundant procedures for achieving goals. The model-based programming approach extends this by guaranteeing correctness of execution through pre-planning of non-deterministic timed threads of activities. Executing model-based programs effectively on distributed autonomous platforms requires distributing this pre-planning process. This thesis presents a distributed planner for model-based programs whose planning and execution is distributed among agents with widely varying levels of processor power and memory resources. We make two key contributions. First, we *reformulate* a model-based program, which describes cooperative activities, into a *hierarchical dynamic simple temporal network*. This enables efficient distributed coordination of robots and supports deployment on heterogeneous robots. Second, we introduce a distributed temporal planner, called DTP, which solves hierarchical dynamic simple temporal networks with the assistance of the distributed Bellman-Ford shortest path algorithm. The implementation of DTP has been demonstrated successfully on a wide range of randomly generated examples and on a pursuer-evader challenge problem in simulation.

Acknowledgments

This thesis could not have been completed without the help from several people. First and most important of all, I would like to thank Professor Brian Williams for supervising and for technical guidance of the research througout the entire process. I would like to thank Research Scientist Greg Sullivan from the AI-lab for giving me constant feedback, for helping me with the formal definitions and proofs and last but not least for teaching me Latex. I also deeply appreciate the assistance of Stanislav Funiak for helping with the design and verification of the distributed algorithm presented in this thesis. John Stedl and Aisha Walcott have provided invaluable technical assistance, as well as all other members of the Model-based Robotic and Embedded Systems Group at MIT. Finally, this work could never be carried out without the love and support from my family and friends.

This thesis also could not have been completed without the sponsorship of the DARPA NEST program under contract F33615-01-C-1896.

Contents

Contents **4**

1 Introduction **10**
1.1 Motivation 10
1.2 Problem Statement 11
1.3 The Pursuer-evader Scenario 13
1.4 Thesis Layout 15

2 Background **16**
2.1 Overview 16
2.2 Robotic Execution and Model-based Programming 16
2.2.1 Robotic Execution Languages 16
2.2.2 Model-based Programming 18
2.3 Distributed Planning 20
2.4 Simple Temporal Networks 21
2.5 Dynamic Constraint Satisfaction Problems 23
2.6 Distributed Constraint Satisfaction Problems 24
2.6.1 Background 24
2.6.2 The basic distributed CSP formulation 25
2.6.3 Fundamental distributed CSP algorithms 25

3 Model-based Programming for Heterogeneous Robots **28**
3.1 Overview 28
3.2 The TinyRMPL Language Specification 28
3.2.1 Primitive Commands 29
3.2.2 Simple Temporal Constraints 29
3.2.3 Basic Combinators 30
3.3 Scenario Encoded in TinyRMPL 31

4 Hierarchical Dynamic Simple Temporal Networks **33**
4.1 Overview . . . 33
4.2 An Introduction to HDSTNs . . . 33
4.3 Mapping TinyRMPL to HDSTNs . . . 38
4.4 Solving HDSTNs . . . 40

5 Distributing HDSTNs **43**
5.1 Overview . . . 43
5.2 Simple Distribution in a Processor Network . . . 43
5.2.1 Local Knowledge of Processors . . . 44
5.3 Mapping HDSTNs to Processor Networks . . . 45
5.3.1 Motivation . . . 46
5.3.2 Leader Election . . . 47
5.3.3 Distribution of an HDSTN in Ad-hoc Networks . . . 48

6 The Distributed Temporal Planning Algorithm **53**
6.1 Overview . . . 53
6.2 Solving Distributed HDSTNs . . . 53
6.2.1 Introducing the Distributed Temporal Planner . . . 54
6.2.2 Message Communication Model . . . 55
6.2.3 The DTP Algorithm . . . 56
6.2.4 Soundness and Completeness . . . 67
6.2.5 Checking Active STN Consistency in a Distributed Fashion . . . 68
6.2.6 Running DTP on the Persuer-evader Scenario . . . 72
6.3 Summary . . . 75

7 Conclusions **76**
7.1 Overview . . . 76
7.2 Implementation . . . 76
7.2.1 TinyRMPL to HDSTN Compiler . . . 76
7.2.2 Software Simulator for the Distributed Temporal Planner . . . 77
7.2.3 Porting the Code to Other Systems . . . 78
7.3 Experiments and Discussion . . . 78
7.4 Future work . . . 80
7.5 Summary . . . 81

Bibliography **83**

Appendix **86**

A Pseudo-code for HDSTN-solver **86**

B DTP Pseudo-code **87**

C XML format specification of HDSTN files **94**

List of Figures

1.1 Model-based Distributed Executive Architecture 13
1.2 Pursuer-evader scenario (Source: DARPA/NEST and UC Berkeley) 14

2.1 RMPL data downlink program. 19
2.2 a) A two-node STN, and b) the equivalent distance graph . . 22
2.3 Example of a negative weight cycle 22

3.1 TinyRMPL grammar. 29
3.2 Pursuer-evader scenario represented in TinyRMPL. 32

4.1 An HDSTN with parallel threads and a choice of methods. . 34
4.2 Example of HDSTN with activity constraints. 36
4.3 The function activityConstraint(N) 36
4.4 The pursuer-evader *Strategy* activity represented as an HDSTN. 37
4.5 Different sets of active edges within a DHSTN. 38
4.6 Graphical representation of the TinyRMPL to HDSTN mapping 40
4.7 HDSTN-solver animation of processing the *strategy* scenario. 41

5.1 DHDSTN processor node. 44
5.2 TinyRMPL program to be distributed among robots. 47
5.3 TinyRMPL program to be distributed within a sensor network. 47
5.4 Amorphous Computer group formation algorithm. 48
5.5 HDSTN Distribution procedure on amorphous computers . . 50
5.6 A three-level tree-hierarchy formed by Amorphous leader election. 51
5.7 TinyRMPL example for distribution on a amorphous computers network 51

6.1 DHDSTN networks 56
6.2 DTP search on a simple temporal constraint 57
6.3 Levels of nodes in a parallel hierarchical network. 58

6.4 Findfirst search method for start nodes of parallel networks. . 59
6.5 Findfirst and findnext search method for end nodes of parallel networks. 59
6.6 Findfirst and findnext search method for end nodes of a decision networks. 60
6.7 Findfirst search method for start nodes of decision networks. 61
6.8 Decision network with an activity constraint. 62
6.9 Findnext search method for start nodes of decision networks. 62
6.10 Sequential network example. 63
6.11 Example of a DHDSTN sequential network. 64
6.12 Extended findnext pseudo-code of parallel start nodes. . . . 65
6.13 Extended findnext pseudo-code of decision start nodes. . . . 66
6.14 a) Simple distance graph b) Same graph with a phantom node. 69
6.15 The pursuer-evader *Strategy* activity problem solved by DTP. 72
6.16 Snapshots of DTP searching the DHDSTN of the pursuer-evader problem. 73
6.17 Snapshots of DTP searching the DHDSTN of the pursuer-evader problem. 73
6.18 Snapshot of DTP searching the DHDSTN of the pursuer-evader problem. 74
6.19 Final consistent STN of the pursuer-evader problem selected by DTP. 74

7.1 Graphical depiction of empirical results, cycles vs. nodes. . . 79

A.1 HDSTN-solver pseudo-code. 86

B.1 a) DTP pseudo-code for processors with the parallel-start flag set. 88
B.2 b) DTP pseudo-code for processors with the parallel-start flag set. 89
B.3 a) DTP pseudo-code for processors with the decision-start flag set. 90
B.4 b) DTP pseudo-code for processors with the decision-start flag set. 91
B.5 DTP pseudo-code for processors with the flag set to parallel-end or decision-end. 92
B.6 DTP pseudo-code for processor with the flag set to primitive flag set. Pseudo-code for the distributed Bellman-Ford consistency check. 93

C.1 XML HDSTN file format specification. 95

List of Tables

7.1 Empirical results. 78

Chapter 1

Introduction

1.1 Motivation

Research on autonomous intelligent systems has focused on how robots can carry out missions with very little or no human intervention. This research area is receiving significant attention due to recent mission successes, such as Deep Space One and Mars Pathfinder. Autonomous robotics offers many benefits, particularly in dangerous environments, where human lives would be jeopardized, and in remote places, either unreachable by humans or where communication delays render remote controlled robot missions unfeasible. Examples include planetary rover missions, unmanned combat aerial vehicles (UCAV) in hostile environments, and search and rescue missions in emergency areas. Additionally, intelligent sensor networks that perform a variety of measurements can aid robots in the above scenarios. These robots must be able to autonomously plan cooperative activities, execute these activities, monitor execution, discover execution failures, and replan quickly with minimal interruption. Moreover, many of these missions are time-critical, demanding systems that react in real-time.

Distributing these autonomous robots within intelligent embedded networks of tiny processors raises a range of issues, such as how to efficiently and robustly coordinate activities in a distributed fashion while requiring minimum power, memory and communication. Robotic execution languages, such as RAPs [13], ESL [14], and TDL [34], have been used to coordinate activities on robots and to improve robustness by choosing between functionally redundant methods for achieving goals. These languages support complex procedural constructs, including concurrent activities and actions with specified durations.

Allowing an executive, which dispatches robot commands, to choose among

functionally redundant methods on-the-fly may introduce a temporal inconsistency that renders the selected methods un-executable [8]. For example, imagine two robots grabbing and lifting an object in collaboration. Each robot is allowed to select among a set of functionally redundant methods of slightly varying duration to achieve their goals. The constraint is imposed that they stop lifting the object simultaneously. Given this constraint, methods must be selected that allow the robots to stop the activities simultaneously, otherwise the execution of the activities fails. The model-based programming approach [36] guarantees correctness of execution by *pre-planning* temporally flexible threads of activities immediately before execution. In the pre-planning process, a series of methods are selected that are confirmed to satisfy temporal consistency. For example, in the example with the two robots, the model-based approach will search for methods to be executed by the robots that when combined allow the activities to be stopped simultaneously. In [18], selection is made efficient by framing the problem as a search through a temporal graph called a Temporal Plan Network, which encodes all possible executions, and the distributed temporal planning algorithm presented in this thesis builds upon this idea among others.

1.2 Problem Statement

While past research has concentrated largely on robots with a centralized executive on board, current research has identified many benefits from *distributed* robotic systems. One example is a spacecraft mission, such as NASA's Starlight or Terrestrial Planet Finder, which uses multiple spacecraft to form a distributed interferometer for imaging planets around other stars. Another example is NASA's Spacecraft Mobile Robot (SMR) [1], micro satellites for inside the International Space Station, which include features like video conferencing, measurement and repair. Distributed systems are inherently more complex than single systems, introducing new challenges such as synchronizing the distributed set of processors and providing communication.

Executing model-based programs effectively on distributed platforms requires distributing the pre-planning process. Centralized solutions introduce a single point of failure, do not scale well, and face the problem of high degrees of communication network congestion. Furthermore, centralized solutions often require substantial computational resources and cannot be deployed on robots with limited capabilities.

This thesis presents a *distributed temporal planner* called DTP that performs the pre-planning phase of model-based program execution. DTP enables robust coordination of activities between heterogeneous multi-agent

[1] SMR/Person Satellite Assistant website: http://ic.arc.nasa.gov/projects/psa/

systems, such as those described above. Furthermore, this thesis introduces *Hierarchical Dynamic Simple Temporal Networks* (*HDSTN*) and distributed HDSTNs *(DHDSTN)* as abstractions of centralized and distributed model-based program execution, respectively.

Hierarchical Dynamic Simple Temporal Networks extend Simple Temporal Networks (STN) [8] that have been widely used within planning and execution research, because they provide an efficient way of modeling and analyzing temporal constraints among concurrent and sequential actions, while allowing temporal uncertainty to be expressed in terms of interval bounds. HDSTNs extend Simple Temporal Networks by using dynamic variables to allow choices among simple temporal constraints. The choices are between functionally redundant methods [24]. Additionally, distributed HDSTNs provide a formalism for distributed pre-planning and execution across robots with varying computational resources.

To perform fast pre-planning, DTP uses a hierarchical form of distributed dynamic variable assignment to generate candidate plans, and uses the distributed Bellman-Ford shortest path algorithm to check for schedulability. Technically, the two most significant contributions of DTP are 1) a distributed algorithm for the pre-planning component of model-based program execution and 2) the ability to operate on heterogeneous robots, from computationally impoverished tiny embedded processors within sensor networks to much more capable processors inside rovers and satellites.

The contribution of distributed pre-planning is two-fold: First, DTP performs parallel graph search on DHDSTNs when possible. Recall that DHDSTNs are abstractions of model-based program execution. Model-based programs are inherently hierarchical and support complex parallel and sequential expressions of methods and primitive commands. DTP exploits the hierarchical property of model-based programs to achieve parallel distributed processing while synchronizing the distributed computation using only local interactions between robots. Second, DTP runs multiple isolated instances of the Bellman-Ford consistency check algorithm simultaneously. Bellman-Ford has the advantage of linear complexity and the need for only local coordination. Distributed Bellman-Ford is typically ran on the complete graph. DTP exploits the fact that DHDSTNS are hierarchical, enabling several Bellman-Ford consistency tests to be run simultaneously in a DHDSTN at different levels within the DHDSTN hierarchy.

In addition to parallel processing, DTP also effectively distributes and synchronizes the pre-planning process among robots with varying levels of processor power. Distribution is performed by first grouping processors in hierarchies, then dividing DHDSTNs into sub-networks of varying size and finally mapping sub-networks into groups of processors.

The pre-planning process is one of three interacting components of a dis-

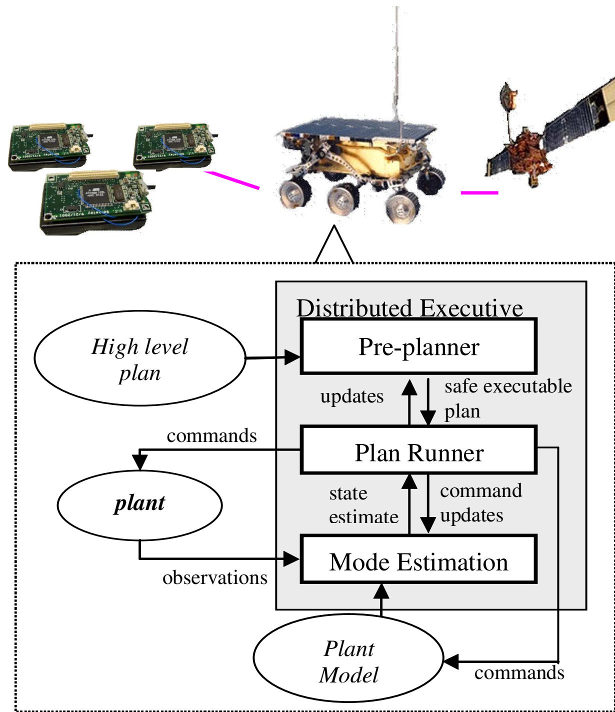

Figure 1.1: Model-based Distributed Executive Architecture

tributed model-based executive (Figure 1.1). The pre-planner selects from among alternative methods to produce threads of execution that satisfy all temporal constraints. This thesis focuses solely on the Pre-planning component. The Plan Runner executes these threads while scheduling activity execution times dynamically in order to adapt to execution uncertainties. Execution monitoring and failure diagnosis is performed by Mode Estimation, and involves monitoring state trajectories over time by searching for likely state transitions, given observations.

1.3 The Pursuer-evader Scenario

As an example, consider a problem in which a set of rovers in a field are pursuing a robot that is attempting to evade them. The pursuer rovers are assisted by helicopters that use visual tracking, and a wireless sensor network, which is distributed on the ground of the field. The scenario is depicted in

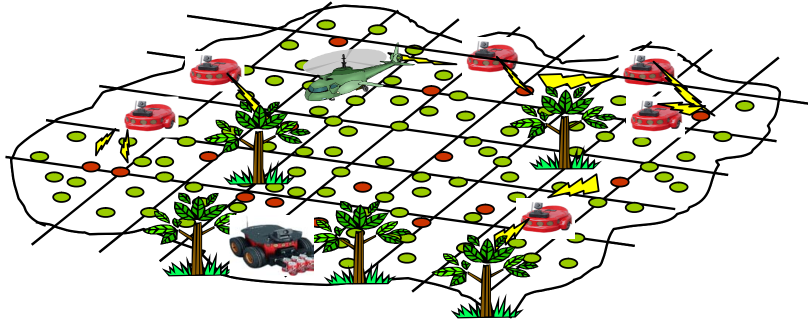

Figure 1.2: Pursuer-evader scenario (Source: DARPA/NEST and UC Berkeley)

Figure 1.2. In the particular scenario of this thesis we assume that the pursuer team consists of two rovers (Rover1, Rover2), one helicopter (Helicopter) and a group of sensors (SensorGroup). The helicopter's visual tracking of the evader is a computationally expensive process and takes longer than when the wireless sensor network performs sensing action, such as sensing light, sound or electromagnetic EM fields. The two rovers have different capabilities. Both rovers can analyze the feedback transmitted by the helicopter or the sensor network and can generate a path that it traverses. Rover1 can move faster than Rover2; however Rover1 is slower at computing a path. Moreover, Rover1 has the choice of generating a simple path or a detailed path. The latter offers more detailed driving information and makes it easier to traverse a path. The disadvantage is that it takes longer to compute the detailed path. Rover2 is only capable of computing and traversing a simple path.

Suppose that at a certain point in time, this heterogeneous set of pursuer robots must execute a strategy to get to the position of the evader. Our example strategy for the pursuer-evader problem is to first perform tracking, using either the helicopter or the sensor network for a user specified period of time. Next the rovers receive the tracking information. Third, either Rover1 or Rover2 must compute a path to the evader given the tracking information, and finally, one must traverse the path to get to the evader position. The strategy includes choices between functionally redundant methods with varying duration for achieving the goal of moving a rover to the evader position.

Applying DTP to this strategy, DTP first distributes the scenario's tasks among the robots by employing leader election and group formation algorithms to the robots. DTP then selects temporally consistent methods for execution in a distributed fashion by choosing among functionally redundant methods, performing this selection systematically and in parallel. One com-

bination of temporally consistent methods found is to perform sensor tracking first, and then to let Rover1 generate a simple path to the evader and to traverse that path.

To summarize, the research presented in this thesis makes the following contributions: 1) a reformulation of model-based programs into HDSTNs, enabling efficient distribution and pre-planning, 2) a method for distributing model-based programs among heterogeneous robots, 3) a distributed temporal pre-planning algorithm that ensures safe execution, and 4) the processing of model-based programs on heterogeneous robots including those that are severely constrained with respect to computational resources. The research builds upon previous work on model-based programming [18], simple temporal networks [8], dynamic CSPs [24] and distributed CSP algorithms [23].

1.4 Thesis Layout

The remainder of the thesis is organized as follows. Chapter 2 gives background on previous research related to this thesis. Chapter 3 introduces the TinyRMPL robot execution language. Chapter 4 defines HDSTN, illustrates how TinyRMPL is reformulated as an HDSTN, and develops an algorithm for solving HDSTNs in a centralized fashion. Chapter 5 first introduces distributed HDSTNs (DHDSTNs) and explains how they are used to enable distributed processing. It then describes a more advanced method for distributing processing within ad-hoc networks. Chapter 6 describes how to solve DHDSTNs in a distributed fashion and the distributed temporal planner DTP. The final chapter, Chapter 7, concludes with a description of the implementation, experimental results, a summary of the research presented, and suggestions for future work.

Chapter 2

Background

2.1 Overview

The research presented in this thesis is based on research in several areas: robotic execution languages and model-based programming, distributed planning techniques, simple temporal networks, dynamic constraint satisfaction problems, and distributed constraint satisfaction algorithms. This background chapter briefly summarizes relevant components of the above mentioned research areas.

2.2 Robotic Execution and Model-based Programming

2.2.1 Robotic Execution Languages

When robots perform cooperative activities in harsh, uncertain environments, such as search and rescue missions, robust planning and execution are key. Actions will sometimes fail to produce their desired effects and unexpected events will sometimes demand that robots shift their attention from one task to another; hence, plans must be structured to cope effectively with the unpredictable events that occur during execution.

Robotic execution languages address the above challenges by providing reactive planning in the execution cycle to cope with unexpected events and achieve plan goals. The languages typically support complex procedural constructs, including concurrent activities and actions with specified durations. They improve robustness by choosing between functionally redundant methods for achieving goals and by reacting to unpredicted events in uncertain

environments. In the following we briefly describe three important robotic execution languages, RAPS [13], ESL [14], and TDL [34], which have influenced the design of the reactive model-based programming language, described in Section 2.2.2.

The RAP system by James Firby [13] is an executive that provides reactive hierarchical task decomposition planning. A robot in a realistic environment cannot expect pre-compiled plans to succeed due to undesired changes and unexpected events in the environment. Planning involves the ordering of primitive actions that will achieve a goal. Reactive planning is situation-driven, meaning that the state of the world determines the order of actions chosen. Furthermore, for reactive planning, actions are not selected in advance but are chosen opportunistically as execution takes place, hence, there is no need for explicit replanning on failures. A RAP is a *reactive action package*, which is a program-like representation of an action that can be taken in the environment. A RAP consists of the goal or sub-goal it will achieve and a variety of methods that can be attempted in order to achieve the goal. As in the real world, there is often a multitude of ways to achieve a goal. A RAP, when executed, may call on other RAPs until the task is decomposed to primitive skills. At the same time, the system monitors its own execution as well as changes in the environment.

Note that the reactive action packages (RAPs) are selected during execution. Since the RAPs have fixed durations and the execution cycle selects one RAP command to be executed at a time, unsafe execution is unlikely to occur. In the contrary, model-based programs, as described below, allow lower and upper bounds on durations of actions. However, choosing arbitrarily among functionally redundant methods with flexible durations can cause unsafe execution. The pre-planner is different from the execution cycle of RAPS, because the pre-planner, before execution, selects methods that are temporally consistent and can be safely executed.

Several other languages and planning and execution systems are based on or influenced by the RAPS system. One of them is *Execution Support Language* (ESL) [14]. ESL is a language used within the Deep Space 1 remote agent for encoding execution knowledge. ESL is a language extension to Lisp, containing common features of the RAPS system. Relative to RAPS, ESL aims for a "more utilitarian point in the design space" (E.Gat [14]). ESL consists of several independent sets of features, including constructs for contingency handling, task management, goal achievement, and logical database management that all can be composed in arbitrary ways.

The robotic execution language *Task Description Language*, TDL (Simmons [34]), is an extension of C++ that includes syntax to support task-level control, such as task decomposition, task synchronization, execution monitoring and exception handling. TDL is a layer on top of the Task Control Ar-

chitecture (TCA) [20], a general-purpose architecture to support distributed planning, execution, error recovery, and task management for autonomous systems. TDL is ideally suited for event-driven architectures, in which events occur asynchronously during real-world situations, such as in robotics or satellite systems. Recently, TDL has been generalized to a distributed version called MTDL for Multi-TDL[1].

2.2.2 Model-based Programming

Robot missions are becoming increasingly more complex. Programmers make common-sense mistakes when designing and implementing missions and control software, such as planners and executives. Examples of mistakes are designing activities that cannot be scheduled correctly, or reasoning about hidden states, i.e., plant states that are not directly observable or controllable. The objective of model-based programming is to provide embedded languages that think from common-sense models in order to robustly estimate, plan, schedule, command, monitor, diagnose, and repair collections of robotic explorers. The embedded languages help programmers avoid common programming mistakes by reasoning about hidden states automatically.

The *Reactive Model-based Programming Language* (RMPL) [36] is a high-level object-oriented embedded language used to describe models of reactive systems. The models specify the behaviors of a system in terms of its nominal behavior and also its possible actions and their effects on the system. RMPL serves several purposes at both the planning level and execution level of a multi-layered architecture, as described below.

The RMPL language provides expressions for timing of actions, full concurrency, preemption (when-donext), conditional execution (if-thennext), maintenance conditions (do-watching), and constraint assertion using *ask* and *tell* constraints for forward and backward chaining and threat-resolution [33]. At the execution level, RMPL is used to describe both probabilistic plant models, such as hardware component interaction, and control programs. The control program specifies the desired state trajectory to achieve state goals, and the plant model is used to deduce a command sequence that tracks this trajectory. The model-based executive executes command sequences, while monitoring states, diagnosing faults and reactively planning new commands to achieve state goals by reasoning about hidden states automatically. At the planning level, RMPL is designed to describe complex strategies for robot teams, including temporal coordination and functionally redundant threads of execution with lower and upper time bounds on actions.

RMPL inherits features from RAP, ESL and TDL. The model-based executive shares key features with the RAPS system, by supporting pre-conditions,

[1]For more information see http://www-2.cs.cmu.edu/~tdl/

control programs, and effects when reactively planning and executing commands to achieve state goals. RMPL also supports the rich set of expressions found in ESL and TDL. For example, contingency handling in ESL is handled by RMPL's preemption constructs with the hidden state diagnosis. ESL's and TDL's task management capabilities, such as spawning new concurrent tasks and setting up task networks, are handled by RMPL's parallel composition and preemption constructs. Synchronization features, such as handling events and signaling, are supported by RMPL as long as the events can be represented as changes to system states. RMPL also has full support of time-keeping, both at the executive layer and at the planner layer.

```
(downlink ()
    (sequence
        (choose_orientation []
            (choose
                ;; two cases where reorientation is necessary
                (if-thennext (AND (antenna = omniA) (pos = posB))
                    (move_to_A (reorient_sc_to_A()) [5,10])
                )
                (if-thennext (AND (antenna = omniB) (pos = posA))
                    (move_to_B (reorient_sc_to_B()) [5,10])
                )
            )
        )
        ;; When packet is ready and a comm window is open send packet
        ( when-donext (comm = window_ok)  ;;wait till window is open
            (do-watching (NOT(comm = window_ok))  ;;download data
                ((download_data_block()) [3,5])
            )
        )
    )
)
```

Figure 2.1: RMPL data downlink program.

Figure 2.1 shows an RMPL program at the planning level that describes the download activity of data from a satellite to a ground station. The satellite is either in position A or B and has two antennas A and B used for transmission. If, for example, the satellite is in position A and is using antenna B, the satellite must rotate to position B first. After the satellite is reoriented to a correct position, it must wait for a communication window to open in order to transmit data, and the communication window must be open during the entire transmission of data. The *antenna* and *pos* are internal state variables, and *comm* is an external state variable that tells when the communication window is open. The RMPL interpreter performs temporal planning at the planning level by selecting methods that achieve temporal consistency and by performing threat-resolution and backward and forward chaining. For example, in Figure 2.1, the temporal planner (RMPL interpreter) ensures that the spacecraft is reoriented to the correct position and that the data is downloaded

when the communication window is open while satisfying the temporal constraints on the activities.

The distributed temporal planner presented in this thesis uses a subset of RMPL to describe complex cooperative activities for heterogeneous robots. The reasons for using a subset of RMPL are to enable pre-planning and execution operations in highly distributed contexts and to enable deployment on severely computationally constrained robots by reducing the most computationally extensive tasks, as detailed in Chapter 3.

2.3 Distributed Planning

Executing model-based programs effectively on distributed autonomous platforms requires distributing the pre-planning process, which guarantees correctness of execution through pre-processing of non-deterministic timed threads of activities. Furthermore, the high demand of processing power and memory for centralized planners is a problem that arises when deploying planning systems in resource-constrained robots. In most cases, it is simply not possible to deploy existing centralized systems, because the systems require orders of magnitude more memory than available. In order for a planning and execution system to be deployable and still exploit the resources of a particular robot, the system must be designed to solve problems with largely varying complexities. The pre-planner and its underlying distributed data structures presented in this thesis are designed to be robust yet deployable on robots with varying computational resources. This thesis focuses on pre-planning to enable safe executions of programs. Constraints of time and space prevent focus on handling physical failures, such as communication failures or processor unreliability.

Several surveys present distributed planning; for example, [9] gives a high level overview of distributed planning and presents many references. In general, there currently is a large variety of distributed planning and execution systems that serve different purposes, such as Robocup rescue simulation [19], Robotic Soccer [31], and planetary exploration with cooperative rovers (ASPEN/CASPER) [12]. Planning and execution modules are most often coupled in a multi-layered architecture. However, a majority of distributed systems perform centralized planning or partial centralized planning, and distributed execution, such as ASPEN/CASPER and Robotic Soccer. The architecture of many distributed planners is often based on hierarchical task networks [33]. The ideas on distributed hierarchical planning were founded in the 1970s by, Corkill [5] among others.

Distributed planning and execution systems are often tailored to solve specific tasks. Examples are the distributed cooperative robotic system AS-

PEN and CASPER [12] and similar systems, such as Multi-Rover Integrated Science Understanding System (MISUS) [11], and CAMPOUT [29]. These systems are tailored to coordinate multiple rover behavior for science missions. They perform centralized high level planning and centralized data analysis, but distribute lower level science goals to individual rovers. One disadvantage of systems, such as ASPEN and CASPER is that they require substantial computational resources and cannot be deployed on small robots, such as tiny rovers or robotic spiders.

The objective of the distributed temporal planner (DTP) presented in this thesis differs from the other designs of distributed planning and execution systems. The purpose of DTP is to provide safe distributed coordination of activities on heterogeneous robots, while supporting deployment of the planner in robots that are severely limited with respect to computational resources. Furthermore, DTP does not require centralized coordination at any level.

2.4 Simple Temporal Networks

Temporal constraints are used to describe requirements concerning the times of different events, where an event is defined as something that occurs at a single point in time. For example, the activity *driving to the shop* is not an event because it the occurs over an interval of time, but *starting the drive* and *completing the drive* are both events because they correspond to instants of time. To specify a temporal constraint between two events, binary constraints are used. For example, to express that the drive takes between 10 and 20 time units, the binary constraint [10,20] is introduced between the *starting the drive* and *completing the drive*, hence, the binary constraint determines the lower and upper bound on the duration of the activity *driving to the shop*.

A Temporal Constraint Network [8] is a formal framework for representing and reasoning about systems of temporal constraints. The description in this section concentrates on Simple Temporal Networks (STN), a simple class of temporal constraint networks that support binary constraints between pairs of time events, which enables polynomial time algorithms to check if a temporal constraint system is consistent. STNs have been widely used in planning systems for representing bounds on duration among threads of activities [12, 31]. The model-based programs used by the pre-planner in this thesis use a hierarchical dynamic STN representation, based on STNs, to encode temporal information, and the pre-planner uses a temporal consistency checking technique as a part of the pre-planning process.

A Simple Temporal Network (STN) consists of nodes that represent time events and directed edges with interval labels that represent binary temporal constraints over time events. The binary temporal constraints are also known

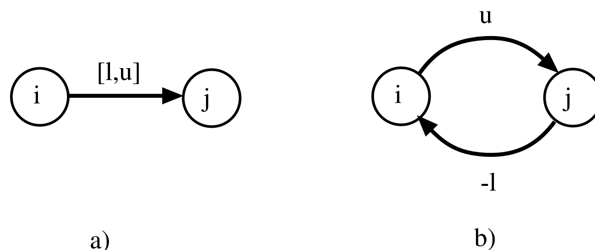

Figure 2.2: a) A two-node STN, and b) the equivalent distance graph

as simple temporal constraints. A simple temporal constraints, represented by an edge $\langle x_i, x_j, lb, ub \rangle$ between nodes x_i and x_j, says that the time event x_i must precede the time of event x_j by at least lb time units and at most ub time units. Figure 2.2a depicts a two-node STN with a single temporal constraint.

Simple Temporal Networks have an equivalent graph representation called *distance graphs*, which enable efficient temporal consistency checking using polynomial-time shortest path algorithms. An STN, and hence, a distance graph is temporally consistent if there exist times that can be assigned to each time event such that all temporal constraints are satisfied.

A *distance graph* of an STN is an equivalent weighted directed graph $G = (V, E)$. The vertices in G correspond to the vertices (nodes) in the STN. An edge $\langle x_i, x_j, lb, ub \rangle$ in the STN induces two edges in E, where the first edge goes from x_i to x_j with the weight ub, and the other goes from x_j to x_i with the weight $(-lb)$, such that $x_i - x_j \leq -lb \wedge x_j - x_i \leq ub$; see [8] for details. Figure 2.2b depicts a distance graph that corresponds to the two-node STN with a single temporal constraint in Figure 2.2a.

Rina Dechter et. al. [8] prove that an STN is *temporally consistent* if its corresponding distance graph has no negative weight cycles. Figure 2.3

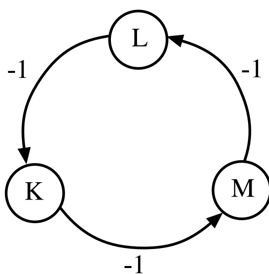

Figure 2.3: Example of a negative weight cycle

shows a network with a negative weight cycle, in which time event K is exactly one time unit before time event M, and M is one time unit before L, and L is one time unit before K, and hence, the network is impossible to execute, in other words temporally inconsistent. Negative weight cycles can be detected by any shortest path algorithm that allow negative weights, such as the Bellman-Ford single source shortest path algorithm [6]. Checking for temporal consistency is critical to the pre-planner, as described in Chapter 6. Since the pre-planner is distributed, it uses a distributed shortest path algorithm to check for consistency, namely the distributed Bellman-Ford algorithm.

2.5 Dynamic Constraint Satisfaction Problems

Constraint satisfaction problems (CSPs) have been used widely, because they provide a powerful and efficient framework for describing state space search problems. A CSP is typically defined as the problem of finding a consistent assignment of values to a fixed set of variables given some constraints over these variables. However, for many tasks, including pre-planning, the set of variables that are relevant to a solution and must be assigned values changes dynamically in response to decisions made during the course of problem solving.

Mittal and Falkenhainer provide a formulation of Dynamic Constraint Satisfaction Problems (Dynamic CSPs) in [24]. This formulation has been used widely, and the data structure for pre-planning presented in this thesis uses dynamic CSPs as well. In the dynamic CSP formulation, two types of constraints are used. *Compatibility constraints* are the constraints over the values of variables and correspond to those traditionally found in CSPs. *Activity constraints* describe conditions under which a variable may or may not be actively considered as a part of a final solution. When a variable is *active*, it must have an assigned value and be included in the solution. By expressing the conditions under which variables are and are not active, standard CSP methods can be extended to make inferences about variable activity as well as their possible value assignments.

In [24] four types of activity constraints that are closely related to traditional constraints are introduced. The most fundamental type of an activity constraint is the *require variable* activity constraint, which establishes a variable's activity based on an assignments of values to a set of active variables. For example, $x = 5 \Rightarrow y$, says that if the active variable x is assigned the value 5, then y becomes active. The data structure used for distributed pre-planning presented in this thesis uses this fundamental type of activity constraint. The three other types of activity constraints are *always require*, *require not* and *always require not*, and are used to express other types of

conditions in which variables activate or deactivate, as described in further detail in [24].

Mittal and Falkenhainer implements the dynamic CSP framework as a specialized problem solver integrated with an *assumption-based truth maintenance system (ATMS)* [7]. Furthermore, they implement a subset of the dynamic CSP framework by extending a conventional backtrack-search CSP algorithm [30]. This thesis presents a similar centralized backtrack-search algorithm that solves a dynamic CSP combined with simple temporal constraints for the pre-planning problem. The distributed pre-planner also presented in this thesis uses a distributed graph-based search to solve the same problem.

2.6 Distributed Constraint Satisfaction Problems

The distributed temporal planner (DTP) presented in this thesis leverage distributed constraint satisfaction problems (distributed CSPs), because DTP utilizes dynamic constraint satisfaction problems (dynamic CSPs) [24] in a distributed fashion. This section gives an overview of distributed CSPs and distributed CSP algorithms.

2.6.1 Background

Yokoo et.al. [23] provides an extensive review of distributed CSP algorithms. This section briefly describes the most common distributed CSP algorithms.

CSPs have been used to solve a large range of AI problems, such as planning, resource allocation, and fault diagnosis. Distributed CSPs are useful when the problem to be solved is inherently distributed. In many situations, researches have realized the benefits of reformulating multi-agent systems with inter-agent constraints to distributed CSPs, and using distributed CSP algorithms to solve the problems. A wide range of problems have been mapped to distributed CSPs. Examples are distributed resource allocation [25], distributed scheduling [21], and distributed truth-maintenance systems [17]. Furthermore, Pragnesh et.al. [26] describes an asynchronous complete method for general distributed constraint optimization, which can be applied to several areas, including optimal distributed planning. Modi et.al.[25] have presented a mapping from distributed resource allocation to a dynamic distributed constraint satisfaction problem (DDSCP) applied to large-scale sensor networks for tracking moving targets [25]. The dynamic properties of the formulation enable the constraint problem to change during run-time, which is essential when tracking a moving object within a sensor network field. Dynamic constraints distributed among agents are continuously activated or

deactivated at run-time according to sensor input from the sensor network. To keep the DDCSP consistent, they use a distributed CSP algorithm that runs in a loop, reacting to dynamic changes of the CSP. Although this solution is inspired the author of this thesis, there are a few fundamental differences: 1) the research in this thesis (DTP) focuses on pre-planning prior to execution in contrast to the DDCSP system, which focuses on resource allocation, 2) DTP focuses on coordination of temporally flexible activities using an STN graph-representation, whereas the DDCSP system uses a distributed CSP algorithm for solution extraction. Nevertheless, the DDCSP system has some valuable properties that could be applied to distributed execution and monitoring.

2.6.2 The basic distributed CSP formulation

The basic distributed CSP formulation is defined as a set of m agents (processors) $p_1 \dots p_m$, where each processor p_i has one variable x_i with an associated domain. The constraints among agents are binary, for example, $x_1 \neq x_2$. It is assumed that every agent p_i knows about all the constraints which are related to p_i, and no global knowledge is assumed. However, these assumptions can be relaxed. A distributed CSP is solved if all variables are assigned and all constraints are satisfied.

Agents communicate with neighbors using messages when a neighbor has a shared constraint. The basic assumption for a distributed CSP algorithm is that message delivery is finite, though random, and that messages are received in the order in which they were sent. Underlying communication protocols are assumed to handle communication. The communication protocols used depend on the type of the distributed network of processors. Computers connected to the Internet typically use the TCP/IP protocol, and ad-hoc wireless networks use adaptive routing algorithms. [32] provides an extensive review of routing protocols in wireless ad-hoc networks, and [28] presents a highly adaptive distributed routing algorithm for mobile wireless networks.

2.6.3 Fundamental distributed CSP algorithms

Distributed CSP algorithms can be classified as backtracking, iterative improvement or hybrid. Furthermore, distributed CSP algorithms can be divided into three groups. Algorithms in the first group are used for problems with a single local variable per processor, the second group supports multiple local variables, and the third group represents distributed partial CSP, see [23] for details. In the following we briefly describe the properties of three classes of algorithms in the single local variable group to give the reader a sense of the properties of the algorithms and how they relate to the distributed pre-planning algorithm presented in this thesis. In the following we assume the

basic distributed CSP formulation with the three properties: 1) every agents owns one variable, 2) all constraints are binary, and 3) each agent knows all constraints relevant to its variable.

The first algorithm is *asynchronous backtracking* (AB), which is derived from centralized backtracking [30]. In AB, every agent maintains an *agent view*, which is an agent's current belief about variable assignments of other agents. Agents can send *(ok?*$x_j = d_j$*)* messages to other agents to check if a particular assignment is consistent with agent views of other agents, and they can send *nogoods(constraint)* which specifies a new constraint that contains violating assignments. Agents are ordered alphabetically using agent IDs, and the order decides the priority of the variable assignments of agents. Lower prioritized agents try to resolve conflicts first before higher prioritized agents resolve conflicts. The AB algorithm is complete.

The main inefficiency of asynchronous backtracking is that agent and value ordering is statically determined. This forces lower priority agents to perform an exhaustive search in order to revise bad decisions made by higher priority agents. The *asynchronous weak-commitment* (AW) search addresses this inefficiency by 1) introducing a minimum conflict heuristic to reduce the risk of making bad decisions, and 2) enabling dynamic agent ordering. For a particular agent, the minimum conflict heuristic selects the assignment that will minimize the number of violated constraints. The agents use *priority values* to dynamically change the order in which agents make assignments. If an agent cannot make a variable assignment that is consistent with the agent view, the agent creates a nogood constraint and increases its priority value to change its priority. The AW algorithm is complete.

The last algorithm, *distributed breakout* (DB), is based on the iterative repair method, which starts with an initial, flawed solution and performs repairs in an iterative manner to find a consistent solution. DB defines a weight variable for each constraint and uses an evaluation function, the sum of weights of violating constraints, as a breakout mechanism to escape local minima. Weights are increased when neighboring agents detect that they are in a local minimum with respect to their value assignments, and the agent that can improve the evaluation value the most, changes its value. The algorithm, however, is not complete.

The experimental results in [23] show that for a graph coloring problem with n agents and $m = 2n$ constraints, the asynchronous weak commitment search outperforms both the asynchronous backtracking and distributed breakout algorithms. However, when the number of constraints are increased to $m = 2.7n$, an interesting phase transition occurs, and the distributed breakout algorithm starts to outperform the other algorithms. Several other distributed CSP algorithms have been developed. Distributed CSP algorithms often perform well on a particular problem, but do not generalize well [23].

While past research on distributed CSPs inspired the author, the distributed temporal planning algorithm, presented in this thesis, takes a different graph-based search approach to solve the distributed problem of pre-planning activities to ensure safe execution. The main reason is that our representation of a model-based program is based on a hybrid of simple temporal constraints (STC) [8] and dynamic CSPs [24]. Checking for consistency in this context requires running a distributed shortest path algorithm on a graph that represents the STCs, which prevents the pre-planning problem from being modeled as a classical distributed CSP.

Chapter 3

Model-based Programming for Heterogeneous Robots

3.1 Overview

We introduce TinyRMPL (Tiny Reactive Model-based Programming Language), which is used for robust multi-agent coordination and execution. To support deployment on processors with very constrained computational resources, TinyRMPL uses a subset of the features of RMPL (Chapter 2). This reduction of features relieves processors from computationally extensive tasks.

3.2 The TinyRMPL Language Specification

In TinyRMPL, robustness is accomplished by specifying multiple redundant methods to achieve each task and temporally flexible metric time constraints on activities. Since the flexible time constraints on methods vary, the preplanner (Chapter 6) is able to select methods that satisfy time constraints, thereby achieving robustness by not being dependent on a single method with certain time constraints that always have to be satisfied.

While RMPL is designed to support a complex set of model-based features, such as mode estimation, fault diagnosis and repair [36], the purpose of TinyRMPL is solely to describe cooperative activities. It does not support types of constraints other than temporal constraints, and it does not support parameterless recursion. This simplification decreases the workload and requirements of the processors, which is crucial when deployed on very constrained processors. However, ongoing research will fold features of RMPL

into the distributed framework. The grammar of TinyRMPL is shown in Figure 3.1.

```
A ::= ((c[lb,ub])
  |   ((parallel (A) (A+)) [lb,ub])
  |   ((sequence (A) (A+)) [lb,ub])
  |   (choose (A) (A+))

c ::= target.action(parameter list)
target ::= single robot | robot team
```

Figure 3.1: TinyRMPL grammar.

The following sections explain the components of TinyRMPL in further detail.

3.2.1 Primitive Commands

A primitive command c is defined as c = *target.action(parameters)*, where *target* is either the name of a single robot or a team of robots defined by the executive. The *action* is the command to be executed on the target, and the action has an argument list, *parameter list*. The pre-planner does not interpret primitive commands. The commands are used for distribution of tasks and for distributed execution.

For example, *R.drive-to(50 70)* describes the motion command for robot R to drive to location $(x, y) = (50, 70)$. Suppose instead that there exists a team called Team1, which consists of three robots (R,S,T). *Team1.drive-to(50 70)* specifies that Team1 must drive to that same location. When this command is dispatched, all three robots will drive to that location. The pre-planner treats the Team.drive-to command as a single command, and the executive interprets the command and ensures that the drive-to command is executed on all members of the team.

3.2.2 Simple Temporal Constraints

TinyRMPL is a timed language in which activities and structures of activities have temporal constraints. TinyRMPL provides temporal constraints in the form of lower and upper time bounds on actions and compositions of actions. Lower bound and upper bound (lb and ub) are specified as positive integers. The units of the temporal constraints are defined by the dispatching algorithm, to which the TinyRMPL programmer must adhere. If lb and ub have not been specified, which is the case for the above *drive-to* examples, they are

assumed to be $(lb, ub) = (0, \infty)$, implying an unlimited upper bound on the duration. This unlimited duration, however, can be restricted by other actions, as described below.

To specify a time constraint on the duration of, for example, activity *R.drive-to(50 70)* with a lower bound of 20 time units and an upper bound of 30 time units, in TinyRMPL it is specified as *R.drive-to(50 70)[20,30]*.

The temporal constraints of commands are interpreted during the pre-planning phase described in Chapter 6 to ensure consistency, but the semantics of primitive commands are not interpreted in this phase.

3.2.3 Basic Combinators

TinyRMPL provides three combinators. The two combinators, *sequential* and *parallel*, are used to create hierarchical sets of concurrent actions. The *choose* combinator is used to select among multiple methods. These constructs can be combined recursively to describe arbitrarily complex behaviors.

An example of TinyRMPL code illustrating the sequence combinator is listed below. In this example, robot R must first drive to location W within 10 to 12 time units and then immediately afterwards broadcast a message M within 1 to 2 time units.

```
(sequence
  ((R.drive-to(W)[10,20])
  ((R.transmit(M)[1,2])
)
```

Since the two actions are performed sequentially, the overall lower bound and upper bound on the above example are $lb = lb(driveto) + lb(transmit) = 10 + 1 = 11$ and $ub = ub(driveto) + ub(transmit) = 20 + 2 = 22$, respectively.

The choose combinator is used to model the selection of functionally redundant methods and can be used in various contexts. For example, if the TinyRMPL programmer wants to specify that either robot R or S drives to location W, the corresponding code is:

```
(choose
  ((R.drive-to(W)[lb,ub])
  ((S.drive-to(W)[lb,ub])
)
```

Explicit specification of lb and ub for parallel and sequential structures is optional and is used to set absolute limits on the time of execution of the structures. For example, if two rovers are driving simultaneously, but the programmer wants to specify an overall time constraint on this activity, the TinyRMPL code is:

```
((parallel
  ((R.drive-to(W)[10,25])
  ((S.drive-to(Y)[10,25])
)[12,22])
```

Here the time constraints are [12,22], which will tighten the time bounds on the concurrent behavior, such that the lower bound is 12 and the upper bound is 22 and the rovers are required to start and stop the driving activity simultaneously. They are not allowed to wait for each other. In real life uncertain environments, however, there is a nearby 0 probability that the two rovers actually reach their goals simultaneously, because driving activities have uncontrollable durations. To model uncontrollability the rovers must wait for each other. To accomplish this, the programmer turns the driving command into a sequence consisting of first the driving command followed by a waiting command.

3.3 Scenario Encoded in TinyRMPL

The TinyRMPL program for the scenario outlined in the introduction is shown in Figure 3.2. The first half of the program describes the tracking activities of the helicopter and sensor network, followed by communication of the tracking information between the helicopter, sensor group and the rovers. The second half describes the path generation and path traversal activities of the two rovers.

The [0,40] at the end of the *strategy* procedure denotes tightening of the time bound of execution time of the top-level procedure. The minimum and maximum duration is 0 and 40 time units, respectively. Not all combinations of parallel methods render a temporally consistent execution. For example, as described in Chapter 4, an execution is unsafe when two parallel threads of execution need to end at the same time, but the temporal constraints imposed on the threads prevents the threads from ending simultaneously. Two parallel threads can never end simultaneously, if the lower bound on the first thread is higher than the upper bound on the other thread or vice versa.

For example, executing *Rover1.compute-advanced-path* is inconsistent with the surrounding *sequence* (S1 in Figure 3.2) of the command, since the lower time bound on action *Rover1.compute-advanced-path* $lb = 30$, is higher than the upper bound of the sequence, $ub = 20$. Also, if the total lower bound of the tracking activity and a rover path traversal activity was higher than 40 (the overall upper bound), it would yield an unsafe execution, since the maximum allowed time is 40 time units. An example of a temporally consistent execution is to first execute the sensor network tracking and then *Rover1.compute-simple-path*, see Figure 3.2.

```
;; pursuer - evader strategy
(strategy [0,INF]
  ((sequence
    ;; perform evader tracking and communicate with rovers
    (parallel
      (choose
        (sequence
          ((SensorGroup.sensor-tracking(LIGHT SOUND EM_FIELDS))[5,6])
          ((SensorGroup.transmit-info(TO_ROVERS)) [1,2])
        )
        (sequence
          ((Helicopter1.vision-tracking(EVADER1)) [10,20])
          ((Helicopter1.transmit-info(TO_ROVERS)) [1,2])
        )
      )
      ;; wait and receive tracking information
      ((Rover1.wait-receive-info())[0,8])
      ((Rover2.wait-receive-info())[0,8])
    )
    ;; move Rover1 or Rover2 to evader position
    (choose
      ((sequence ; S1
        (choose
          ((Rover1.compute-advanced-path())[30,40])
          ((Rover1.compute-simple-path())[10,15])
        )
        ((Rover1.fast-path-traversal()) [10,20])
      )[20,35])
      (sequence
        ((Rover2.compute-simple-path())[5,10])
        ((Rover2.path-traversal()) [20,30])
      )
    )
  )[0,40])
)
```

Figure 3.2: Pursuer-evader scenario represented in TinyRMPL.

Before identifying a safe execution, we first map TinyRMPL programs to a data structure that the pre-planner can process. The following section presents a Hierarchical Dynamic Simple Temporal Network used as an abstraction of TinyRMPL program that is sufficient for pre-planning.

Chapter 4

Hierarchical Dynamic Simple Temporal Networks

4.1 Overview

This thesis focuses on the process of selecting a safe execution of a TinyRMPL program in a distributed fashion, prior to dispatching the plan for execution. For this purpose, we map the TinyRMPL program into a data representation that satisfies the following criteria: 1) supports efficient distribution of TinyRMPL, 2) supports parallel processing, and 3) supports deployment on heterogeneous robots with varying computational resources. To support deployment on robots with extremely limited computational resources, the robots must make modest use of memory. The local knowledge per robot must be minimized and the communication among robots must be minimized. These criteria are satisfied by reframing the decision problem as a hierarchical dynamic simple temporal network (HDSTN), which enables efficient distribution, parallel processing and solution extraction of a TinyRMPL program. This chapter introduces HDSTNs, shows how HDSTNs are mapped from TinyRMPL, and how they are solved with a centralized algorithm. Chapter 5 describes how HDSTNs are distributed among a set of processors, and Chapter 6 details a solution to distributed HDSTN problems.

4.2 An Introduction to HDSTNs

HDSTNs extend STNs [8] with two key properties: they are *hierarchical* and *dynamic*. The hierarchy is inherited from TinyRMPL and its *parallel* and *choose* combinators. The hierarchical property of an HDSTN enables key

features of the distributed temporal planner, introduced in Chapter 6 : efficient parallel search and parallel consistency checks. In HDSTNs, *dynamic* variables are used to encode choices among alternative threads of execution, while constraints are restricted to simple temporal constraints and activity constraints.

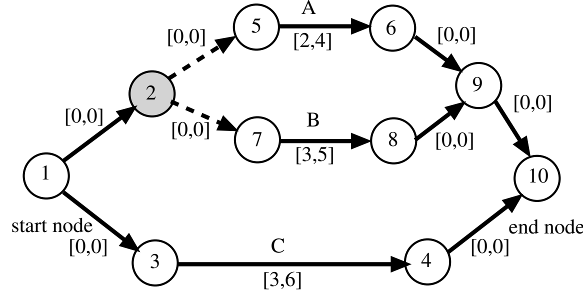

Figure 4.1: An HDSTN with parallel threads and a choice of methods.

Figure 4.1 shows an HDSTN with parallel activities and a choice between two methods. The nodes represent time events. A solid edge constraint labeled [lb,ub] denotes a simple temporal constraint. Edges annotated with commands, such as *A*, correspond to the execution of TinyRMPL commands, where the arrows show the forward direction of execution in time. Note that commands are not interpreted during pre-planning and, hence, are not a part of the formal definition of an HDSTN, introduced below. Nodes 1 and 10 are the start and end events, respectively, and also denote the start and end of parallel threads of commands. The gray solid node, 2, denotes a time event with an associated dynamic variable and two associated choices (5 and 7). A choice is graphically represented as dotted edges, one of which must be chosen. In this case either node 5 or 7 must be chosen. Commands have non-zero duration. For example, command A is represented by the edge between nodes 5 and 6, with a lower and upper bound of 2 and 4 time units, respectively.

Definition 4.1 *An* HDSTN *is a 5-tuple* $N = \langle V, E, \delta, s, e \rangle$*.* V *denotes time event variables partitioned into three mutually disjoint finite sets,* V_{simple}*,*$V_{decision}$*, and* $V_{parallel}$*.* V_{simple} *denotes simple time events for actions.* $V_{decision}$ *denotes time events that form the start and end of decision threads, and* $V_{parallel}$ *denotes the same for parallel threads. The edge set E contains 4-tuples* $\langle x_i, x_j, lb, ub \rangle$*. An edge* $\langle x_i, x_j, lb, ub \rangle$ *represents a simple temporal constraint on the values for time events* x_i *and* x_j *such that* $x_i - x_j \leq -lb \wedge x_j - x_i \leq ub$*. This is a lower and upper bound* (lb, ub) *on the temporal distance between* x_i *and* x_j*, where* lb *and* ub *are positive inte-*

gers. The dynamic variable set δ contains 4-tuples $\langle x, \delta_x, initial?, Dom_x \rangle$. For every decision time event, with time event variable $x \in V_{decision}$, there is an element in δ, $\langle x, \delta_x, initial?, Dom_x \rangle$, where δ_x is a dynamic variable *at time event x, and Dom_x is the set of target time event variables of which one must be chosen. The time events s and e represent the start and end events of N, respectively.*

The following section describes the properties of an HDSTN in further detail. By convention, time event variables are named by integers. Dynamic variables are named δ_i and are attached to time event variables $i, i = 0 \ldots |V_{decision}|$. A dynamic variable can be labeled active or inactive as determined by an *activity constraint*. Only active variables are assigned values during pre-planning. They denote the solution of an HDSTN, which is an STN, as defined below. The values of time event variables V of an HDSTN are assigned during execution. Dynamic variables with the *initial?* flag set to true are by definition always active. For example, if an HDSTN solely consists of a choice between action A or B, the initial flag of the corresponding dynamic variable is true, because the decision is not dependent on other decisions - in all cases the decision must be made.

HDSTN networks are designed to be created from TinyRMPL and inherit the hierarchical properties of TinyRMPL, as described in further detail in Section 4.3. The set E of edges in a DHSTN, $N = \langle V, E, \delta, s, e \rangle$, is referred to as $edges(N)$, the set of variables V is referred to as $vars(N)$ and the set of dynamic variables δ is referred to as $dynvars(N)$. Any given DHSTN network N created from a TinyRMPL expression always has a start and end node s, e. We refer to them as $start(N), end(N) \in vars(N)$.

The set of *activity constraints* are induced from dynamic variables and simple temporal constraints. An *activity constraint* α of an HDSTN N is a tuple $\alpha = \langle \delta_i, d_i, \delta_j \rangle$ denoting constraint $\{\delta_i = d_i \Rightarrow activate(\delta_j)\}$, equivalent to a variable assignment $\delta_i = d_i$ that activates the dynamic variable δ_j. An activity constraint $\langle \delta_i, d_i, \delta_j \rangle$ is *enabled* if the constraint $\delta_i = d_i$ is satisfied. If the initial? flag of a dynamic variable is false, or the variable is not activated by an activity constraint, it is always inactive. An activity constraint, $\delta_i = d_i \Rightarrow activate(\delta_j)$, is stored with its antecedent variable δ_i.

Figure 4.2 shows an HDSTN $N = \langle V, E, \delta, s, e \rangle$ with two dynamic variables (δ_1, δ_2) and associated decision time events (v_1, v_2), which are denoted in the figure as two grey circles (1,2). The bounds on the simple temporal constraints are omitted in the example for simplicity. The start and end time events s and e are 1 and 10. The HDSTN in the example contains an activity constraint $(\delta_1 = 2 \Rightarrow \delta_2)$, which models the activation of the dynamic variable δ_2, given the assignment $\delta_1 = 2$, i.e., a choice between A or B is only needed if C has not been chosen. The time events $\{1, 2, 9, 10\} = V_{decision}$

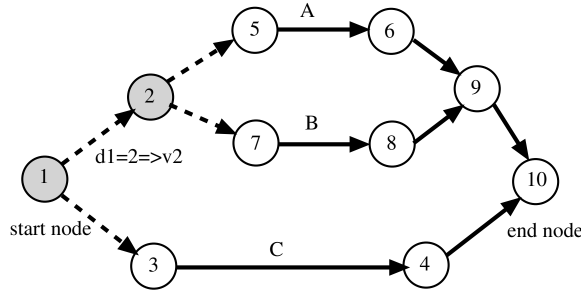

Figure 4.2: Example of HDSTN with activity constraints.

are the start and end time events of functionally redundant methods (A or B, or C), the remainder of the nodes in the figure are simple time events from V_{simple}.

activityConstraints(N) $= f(start(N), -, -, N)$ where
$f(x, var, val, N) =$
if $x \in V_{decision}$
let $\langle x, \delta_x, b, Dom_x \rangle \in \delta$
let $\alpha = \bigcup f(d, \delta_x, d, N) \quad \forall d \in Dom_x$
if $var \neq -$,
$\{(var = val \Rightarrow x)\} \bigcup \alpha$
else
α
else
$\bigcup f(y, var, val, N) \quad \forall \langle x, y, -, - \rangle \in E$

Figure 4.3: The function activityConstraint(N)

The function *activityConstraints(N)=f(x, var, val, N)*, shown in Figure 4.3, returns a set of activity constraints, given an HDSTN, N. $f(N)$ performs a search on the network of N. For a dynamic variable x, it performs recursive calls with $x = val$ as the enabling assignment, for each subnetwork reachable from x. For each dynamic variable y encountered in a subnetwork reached from x, an activity constraint $x = val \Rightarrow y$ is added.

For example in Figure 4.2, $f(\ldots)$ starts from $start(N) = v_1$, i.e. $f(v_1, -, -, N)$. The search branches out to $f(v_2, v_1, 2, N)$ and $f(v_3, v_1, 3, N)$. $v_2 \in V_{decision}$ has a corresponding dynamic variable δ_2, and the activity constraint $\delta_1 = 2 \Rightarrow \delta_2$ is created. The remainder of the search on the network will not create additional activity constraints.

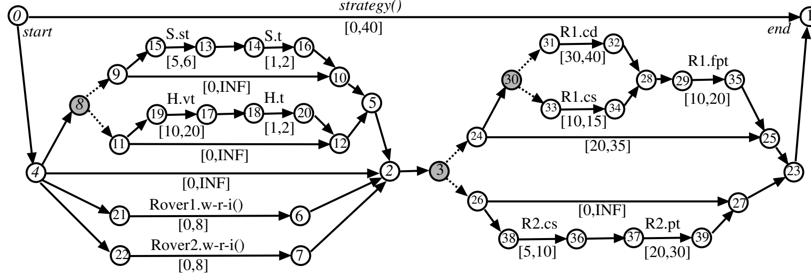

Figure 4.4: The pursuer-evader *Strategy* activity represented as an HDSTN.

Figure 4.4 depicts the HDSTN corresponding to the TinyRMPL example, *strategy*, from Figure 3.2. Again, the nodes represent time event variables. The nodes are grouped into the three classes: simple, parallel, and decision. The formal representation of the HDSTN in Figure 4.4 is $\langle V, E, \delta, s, e \rangle$, where:

$V_{simple} = \{6, 7, 13, 14, 15, 16, 17, 18, 19, 20, 21,$
$22, 29, 31, 32, 33, 34, 35, 36, 37, 38, 39\}$
$V_{parallel} = \{0, 1, 2, 4, 5, 9, 10, 11, 12, 24, 25, 26, 27\}$
$V_{decision} = \{8, 3, 30\}$
$E = \{\langle x_0, x_4, 0, 0\rangle, \langle x_{15}, x_{13}, 5, 6\rangle, \langle x_{14}, x_{16}, 1, 2\rangle, \ldots\}$
$\delta = \{\langle \delta_8, \{9, 11\}\rangle, \langle \delta_3, \{24, 26\}\rangle, \langle \delta_{30}, \{31, 33\}\rangle\}$
$s = 0, e = 1$

Unlabeled edges implicitly correspond to zero duration simple temporal constraints. Nodes 0 and 1 are the start and end events of *strategy*, respectively. The gray solid nodes, $(8, 3, 30)$, denote time events in $V_{decision}$ with associated dynamic variables from δ. The domain Dom_i of a variable $\delta_i \in \delta$ is graphically represented as the nodes pointed to by the dashed edges, of which one branch must be chosen. Nodes $4, 9, 11, 24, 26$ are the start nodes of parallel networks of actions from $V_{parallel}$. Again, commands have non-zero duration. Commands are shown in the figure to clarify the relationship to the *strategy* scenario, but commands are not interpreted in the pre-planning phase. Command H.vt, for example, denoting Helicopter vision tracking, is represented by the edge between node 19 and node 17, with a lower and upper bound of 10 and 20 time units, respectively.

In the following we present the notion of active edges in a HDSTN, which leads to the definition of a feasible solution of an HDSTN.

The HDSTN solution algorithm, presented in Section 4.4, identifies the $E' \subseteq E$, *active edges* of a given HDSTN. E' is determined by the dynamic variable assignments. When an HDSTN's edge set E is restricted to its active edges E', it reduces to an STN. Figure 4.5 shows the HDSTN given

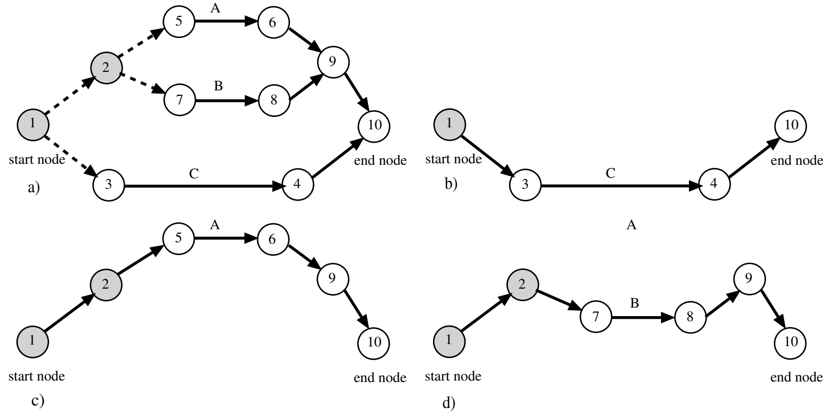

Figure 4.5: Different sets of active edges within a DHSTN.

earlier in Figure 4.2 with three different sets of active edges determined by the dynamic variables δ_1 and δ_2. Figure 4.5a shows the HDSTN before pre-planning. Figure 4.5b shows the active edges of the HDSTN determined by the assignments $\delta_1 = 3$ and $\delta_2 = inactive$. Figure 4.5c and d reflect the assignments $\{\delta_1 = 2, \delta_2 = 5\}$ and $\{\delta_1 = 2, \delta_2 = 7\}$, respectively.

Definition 4.2 *A* feasible solution *of an HDSTN is an assignment γ to all active variables such that 1) every variable mentioned in the consequent of an enabled activity constraint has an assignment, 2) every variable mentioned in the consequent of a disabled activity constraint is unassigned, and 3) the corresponding STN derived by considering only the active edges is temporally consistent.*

One feasible solution for the HDSTN in Figure 4.4 is represented by the variable assignments $\delta_8 = 9$, $\delta_3 = 24$, and $\delta_{30} = 33$. In the figure there is one activity constraint α at the dynamic variable δ_3 because it can enable δ_{30}, i.e., $\alpha = \{\delta_3 = 24 \Rightarrow \delta_{30}\}$. In the next section we present the mapping between TinyRMPL and HDSTNs.

4.3 Mapping TinyRMPL to HDSTNs

TinyRMPL code is translated to an HDSTN, where the signature of the translation function $[\![\text{-}]\!]$ is $(A, \text{HDSTN}) \rightarrow \text{HDSTN}$. For example, if a TinyRMPL construct A is mapped and added to a DHSTN N, the translation function is $[\![\text{A}]\!]N = N'$, where N' is the resulting DHSTN.

The formal translation rules for the four constructs in TinyRMPL are:

- $[\![$c[lb,ub]$]\!]N = N'$, where
  $vars(N') = vars(N) \cup \{x_s, x_e\}, x_s, x_e \in V_{simple}$,
  $x_s, x_e \notin vars(N)$,
  $edges(N') = edges(N) \cup \{\langle x_s, x_e, lb, ub\rangle\}$
- $[\![$parallel A B [lb,ub]$]\!]N = N'$, where
  $vars(N') = vars(N) \cup \{x_s, x_e\}, x_s, x_e \in V_{parallel}$,
  $x_s, x_e \notin vars(N)$,
  $[\![$A$]\!]N = N_A, [\![$B$]\!]N_A = N_B$,
  $edges(N') = edges(N_B) \cup$
  $\{\langle x_s, start(N_A), 0, 0\rangle, \langle x_s, start(N_B), 0, 0\rangle,$
  $\langle end(N_A), x_e, 0, 0\rangle, \langle end(N_B), x_e, 0, 0\rangle, \langle x_s, x_e, lb, ub\rangle\}$
- $[\![$sequence A B [lb,ub]$]\!]N = N'$, where
  $vars(N') = vars(N) \cup \{x_s, x_e\}, x_s, x_e \in V_{parallel}$,
  $x_s, x_e \notin vars(N)$,
  $[\![$A$]\!]N = N_A, [\![$B$]\!]N_A = N_B$,
  $edges(N') = edges(N_B) \cup$
  $\{\langle x_s, start(N_A), 0, 0\rangle, \langle end(N_A), start(N_B), 0, 0\rangle,$
  $\langle end(N_B), x_e, 0, 0\rangle, \langle x_s, x_e, lb, ub\rangle\}$
- $[\![$choose A B$]\!]N = N'$, where
  $vars(N') = vars(N) \cup \{x_s, x_e\}, x_s, x_e \in V_{decision}$,
  $x_s, x_e \notin vars(N)$,
  $[\![$A$]\!]N = N_A, [\![$B$]\!]N_A = N_B$,
  $edges(N') = edges(N_B) \cup$
  $\{\langle x_s, start(N_A), 0, 0\rangle, \langle x_s, start(N_B), 0, 0\rangle,$
  $\langle end(N_A), x_e, 0, 0\rangle, \langle end(N_B), x_e, 0, 0\rangle\}$
  $dynvars(N') = dynvars(N_B) \cup \langle x_s, \delta_{x_s}, b, Dom_{x_s}\rangle$,
  $\delta_{x_s} \notin dynvars(N)$, where
  $Dom_{x_s} = \{start(N_A), start(N_B)\}$

The above rules show the translation of constructs that consists of binary sub-constructs, for example *choose A B*, where *choose* is the construct and A and B are the sub-constructs. The translation rules generalize to constructs with arbitrary numbers of sub-constructs. Note that the mapping of a *sequence* construct creates a start and end variable that is added to $V_{parallel}$, because the sequence is a special case of a parallel construct with one sequence of activities and a simple temporal constraint between the start and end node. The initial flags of dynamic variables are set in a second pass of the resulting DHSTN; this pass can be incorporated into the extraction of activity constraints. The rules are depicted graphically in Figure 4.6. Once a TinyRMPL program has been mapped to an HDSTN, it can be solved using an HDSTN-solver, introduced in the following section.

4.4 Solving HDSTNs

HDSTNs can be solved using a dynamic constraint satisfaction backtrack search algorithm, which extends the generic constraint satisfaction chronological backtracking algorithm in [30]. In an HDSTN, an inconsistency corresponds to a negative weight cycle. We use the Bellman-Ford single source shortest path algorithm to detect such cycles. We only run Bellman-Ford on *active* edges. The pseudo-code for this centralized *HDSTN-solver* is presented in Appendix, Section A.

The HDSTN-solver (referred to as the solver from now on) assumes access to all elements of an HDSTN, $N = \langle V, E, \delta, s, e \rangle$, that is being solved, and takes as input the size of the vector of dynamic variables $n = |\delta|$. The solver accesses dynamic variables $\delta_i \in \delta$ using the operator $v[i]$.

The solver is similar to the chronological backtracking algorithm, because it systematically labels (assigns) values to variables while checking for consistency. The solver searches for a solution in the dynamic CSP of an HDSTN by labeling only active dynamic variables. Each time a dynamic variable is labeled or the solver backtracks and unlabels a variable, the dynamic variables are updated based on the activity constraints; the update involves activation or deactivation of variables. The algorithm returns when all active dynamic variables are labeled and no activity constraints activate other new, unlabeled variables. It returns *success* if the STN determined by the active edges is

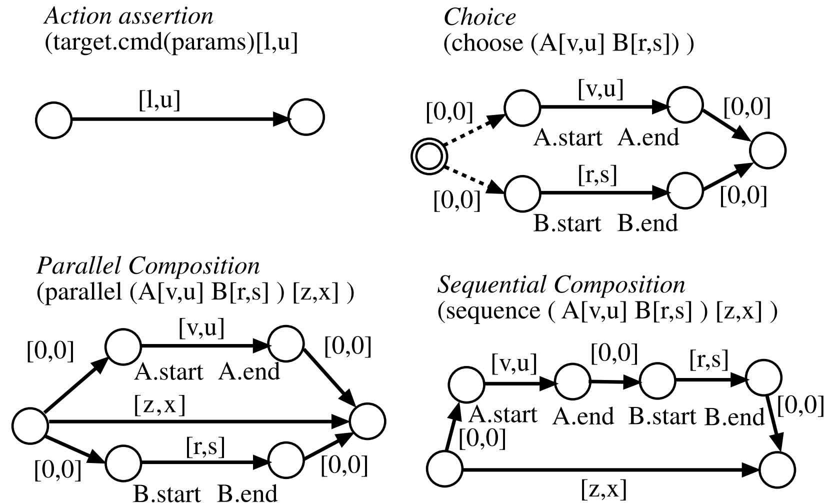

Figure 4.6: Graphical representation of the TinyRMPL to HDSTN mapping

temporally consistent, or *failure* if no consistent STN could be found.

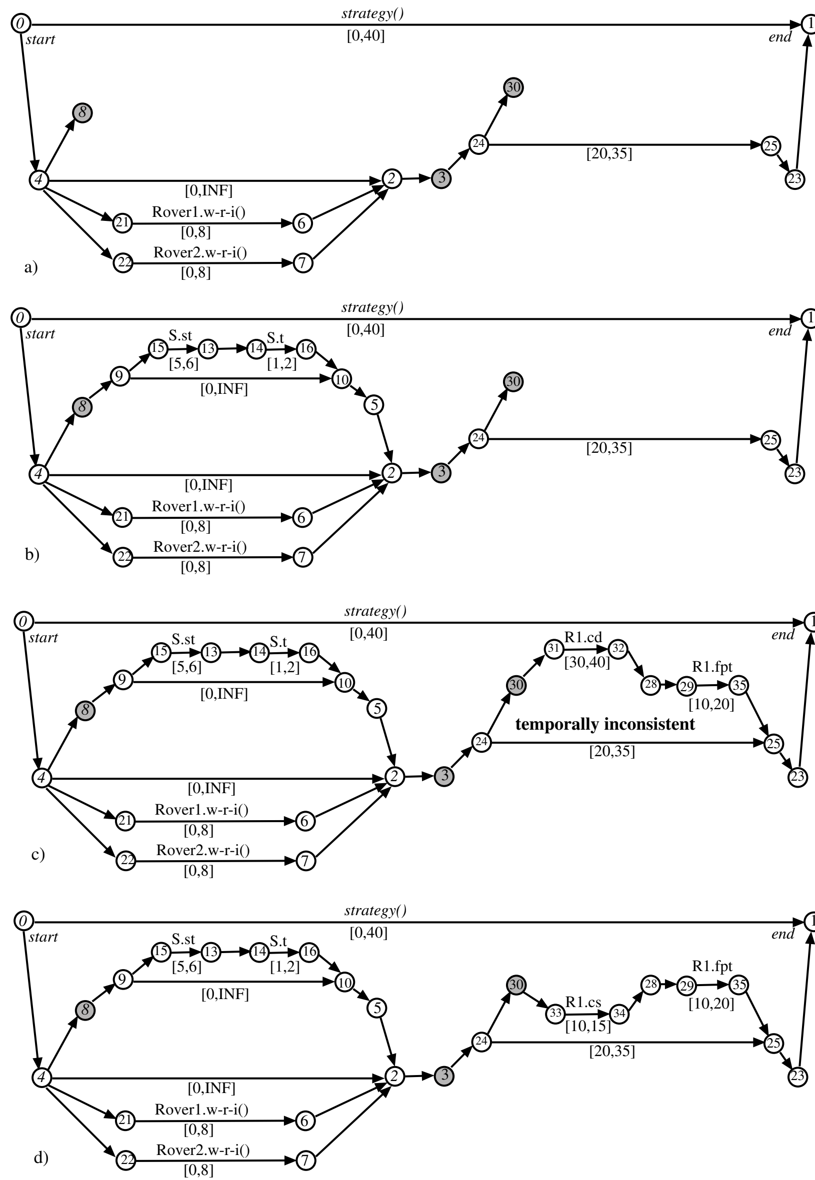

Figure 4.7: HDSTN-solver animation of processing the *strategy* scenario.

We demonstrate the solver on the pursuer-evader scenario HDSTN in Figure 4.4 by walking through the solution process while illustrating the network changes (Figure 4.7). The total number of dynamic variables supplied to the solver is 3, $\{\delta_3, \delta_8, \delta_{30}\}$. The *initial?* flag of the dynamic variables δ_3, δ_8 are true. The first variable assignment is $\delta_3 = 24$. Next comes a consistency check on the active edges (Figure 4.7a). Since no negative weight cycles are induced, the current assignment is consistent. The next dynamic variable in the array is v_8, and the variable assignments becomes $\delta_8 = 9$. Another consistency check is performed on the active edges (Figure 4.7b), and the network is consistent. Next the activity constraint $\delta_3 = 24 \Rightarrow \delta_{30}$ activates δ_{30}, which makes the assignment $\delta_{30} = 31$. However, this creates an inconsistency (Figure 4.7c), since the activities R1.cd and R1.fpt combined have a lower bound (40) greater than the upper bound of the surrounding simple temporal constraints $(lb, ub) = (20, 35)$. Instead, the variable assignment $\delta_{30} = 33$ is made, which yields a temporally consistent network (Figure 4.7d), and the solver returns *success*. The final variable assignments are $\delta_8 = 9$, $\delta_3 = 24$, and $\delta_{30} = 33$.

Chapter 5

Distributing HDSTNs

5.1 Overview

The objective is to support fine grained distributed execution on processors with severely limited computational resources. As a result, we allow distribution to go down to the level of each robot handling a single variable or constraint. In addition, our robotic systems *vary* substantially in their computational capabilities, from wireless sensors to rovers, with more capable systems being able to handle large collections of constraints. This heterogeneous case is handled using the same fine grained distributed algorithm, by simply having each robot execute the distributed algorithm on all of the constraints it owns. We frame this problem as a distributed HDSTN. This chapter first describes a simple distribution of an HDSTN in a processor network. The second section describes more general cases, in which leader election is necessary to determine the distribution.

5.2 Simple Distribution in a Processor Network

Definition 5.1 *A* processor *is defined as an independent computer that communicates with other processors using messages. A* processor network *is an array of N processors, $p_i = 1, 2, ..., N$, where any pair of processors can communicate with each other either directly or by message routing.*

Definition 5.2 *A Distributed Hierarchical Dynamic Simple Temporal Network,* DHDSTN*, is a Hierarchical Dynamic STN (HDSTN), N, where every time event variable $v \in vars(N)$ is assigned a specific processor p_i in a processor network. A processor can own one or more HDSTN nodes. The*

simple temporal constraints, $edges(N)$ are distributed such that every processor p_i only has local knowledge on the topology, entailing knowledge on simple temporal constraints that are directly connected to a node $v \in p_i$.

Initially, for simplification, every time event $v \in vars(N)$ in an HDSTN N is assigned to a unique processor. It is straight forward to extend this to the general case, where every processor owns an arbitrary number of nodes in an HDSTN. Section 5.3 explains the process by which elements of an HDSTN are assigned to processors in ad-hoc networks by using leader election and group formation algorithms.

5.2.1 Local Knowledge of Processors

Figure 5.1 shows a processor representing a single node from an HDSTN. The processor maintains a set of local attributes used by the DTP planner and the simple temporal constraints between its neighbors $\{s, p, r\}$, where d_{ij} denote upper time bounds and denote the forward direction in execution time, and d_{ji} denote lower bounds.

Node *a* from the processor point of view

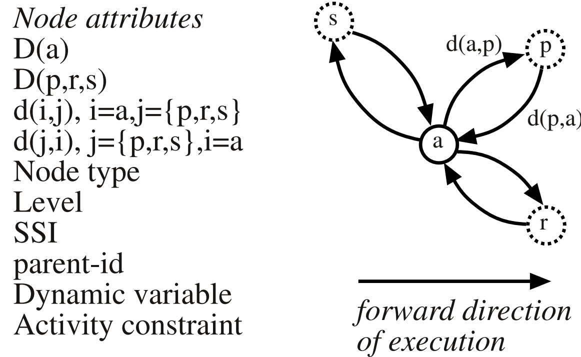

Figure 5.1: DHDSTN processor node.

The local attributes of processors are:

- D and D_i are estimates of temporal distance to the target for a node and its neighbors, respectively, and are used for consistency checks together with the simple temporal constraints d_{ij} between neighbors.

- The *node type* indicates the type of node this processor represents. The behavior of DTP depends on the node type. The node type is either *decision-start, decision-end, parallel-start, parallel-end, primitive*, which is directly related to the variable type $v \in V$ of an HDSTN.

- The *level* variable denotes the "level of nesting" of a node within parallel networks (defined in Section 6.2.3) and is used for isolating concurrent consistency checks at different levels in the network.

- The Sequential Network Id reference, SNI, speeds up search by enabling parallel search in cases were sequential search otherwise is the basic solution. SNI references are found using simple forward search on the HDSTN during the distribution phase.

- The *parent-id* variable is used by DTP to send feedback messages backward toward the origin. They are enabled during DTP runtime and are described in Section 6.2.2.

- *dynamic variables* and *activity constraints* are preserved from the HDSTN. During distribution, every activity constraint $\alpha = \{\delta_i = val \Rightarrow \delta_j\}$ is copied to the same processor as δ_i.

5.3 Mapping HDSTNs to Processor Networks

This section describes a method for mapping a Hierarchical Dynamic Simple Temporal Network (HDSTN) to a set of distributed processors. The distribution can take place in several contexts. One example is the pursuer-evader scenario with rovers, a helicopter and a sensor network. Another example is ad-hoc networks, such as amorphous computer networks [16, 1]. An amorphous computer network consists of an array of processors, where there is no a priori organization, and there are no leaders or leader hierarchies to organize computation a priori. The first objective of amorphous computing is to obtain coherent behavior from the cooperation of large numbers of these processors that are interconnected in unknown, irregular, and time-varying ways. The second objective is to find methods for instructing the arrays of processors to cooperate and achieve goals.

The first sub-section motivates the reader by giving examples. The following sub-section describes a type of leader election that can be applied in

general. The last sub-section describes a distribution method of an HDSTN over an ad-hoc network.

5.3.1 Motivation

When distributing an HDSTN over a robotic network, tasks associated with particular robots are typically distributed to that particular robot. Moreover, hierarchical subnetworks of an HDSTN can be mapped to hierarchies within the robotic network. However, there is still a question of determining which robot must initiate the search, and how to coordinate the solution, and how to ensure that robots are within communication range of each other. For ad-hoc networks, such as wireless sensor networks of small processors that are randomly distributed on a surface, there may be no mapping of tasks to particular robots, and the question is how to determine the distribution of tasks given the layout and structure of the network. This is usually unknown beforehand.

Section 5.2 introduced a simple distribution of an HDSTN, where each variable $v \in V$ of an HDSTN was assigned a unique processor in a sensor network, simply by copying each variable to a unique processor while assuming full communication. This simple form of distribution can be generalized to other types of distribution. For heterogeneous robots with varying computational resources, the most constrained robots can handle one constraint each, and more capable robots can handle large collections of constraints. Hence, the HDSTN should be distributed *unevenly* among robots to maximize the use of the resources of each robot. For this uneven type of distribution, every robot still runs DTP, where each robot simulates DTP on all of the constraints it owns. In addition, communication is reduced by having collections of variables and constraints within each robot. For example, if two DHDSTN nodes reside in the same physical robot, they do not need to communicate, but can merely exchange data directly, because they are running on the same machine.

Only in some cases can the mapping between tasks and robots be determined beforehand. In general it is not clear *how* to distribute the HDSTN nodes and constraints among processors. Therefore two main challenges for performing distribution occur: 1) which processor is the lead at each level in the hierarchy and which leader initiates DTP, and 2) ensuring that communication between pairs of processors can occur either directly or using message routing. When full communication is non-existent, the distribution method of an HDSTN must ensure that the processors that need to communicate with each other when running DTP actually can communicate with each other.

Figure 5.2 shows an example with two (or more) robots that are performing cooperative activities. In this example, the HDSTN time events that represent drive-to tasks for a particular robot can be assigned to that robot during

```
(parallel
  ((Robot1.drive-to(RockA))[10,20])
  ((Robot2.drive-to(RockB))[15,25])
  ...
)
```

Figure 5.2: TinyRMPL program to be distributed among robots.

execution. However, there is a question about what robot takes the responsibility of the parallel start and end nodes that represent the start and end of the cooperative activities. In this situation it is useful to run a leader election algorithm [22, 4] prior to distribution and let the leader take the responsibility of parallel start and end nodes. Sub-section 5.3.2 describes one form of leader election.

```
(parallel
  ((Sensor1.measure-light())[10,20])
  ((Sensor2.measure-sound())[15,25])
  ((Sensor3.measure-EM-fields())[15,25])
)
```

Figure 5.3: TinyRMPL program to be distributed within a sensor network.

Figure 5.3 shows another motivating example for a more advanced form of distribution in an ad-hoc sensor network, in which the TinyRMPL program must be distributed to three sensor processors. Assume that the three processors are not synchronized or organized, so it is unknown how or if the processors can communicate with each other. To enable pre-planning and later the dispatching of commands, groups have to be formed to determine communication routing and a leader needs to be selected, which can initiate the pre-planning. We show how these tasks are performed in sub-section 5.3.2.

5.3.2 Leader Election

This section describes the leader election algorithm [4] to form groups in the processor network. We briefly summarize the basic leader election algorithm in Figure 5.4 that creates a two-level hierarchy of leaders and followers. I encourage the reader to read [4], which describes network algorithms for amorphous computing more thoroughly.

The leader election algorithm is based on performing a countdown from a randomly selected integer, within a given range for every processor. In each step of the algorithm, every processor decrements its integer. If the integer of a processor reaches zero, the processor broadcasts a "followMe" message and becomes a leader. Otherwise, if a processor receives a "followMe" when

For every processor:

$r = randomInteger(1..R)$
$while(r > 0)$
  $r \leftarrow r - 1$
  $if(r = 0)$
    $broadcast("followMe")$
    processor becomes leader
  $if\ received("followMe")$
    processor becomes follower of sender
    $return$

Figure 5.4: Amorphous Computer group formation algorithm.

its integer is greater than zero, it becomes a follower of the sender of the message. The result of the algorithm is that all processors within the network are divided into groups. In each group there is one leader and a number of followers of the leader. The followers are within direct communication range of the leader.

Given the basic two-level hierarchy with leaders and followers, a tree-hierarchy can be created with a higher number of levels. To extend tree-hierarchy from level n to $n + 1$, a slightly different leader election is performed among the leaders at level n, and the groups are connected in a tree-like fashion. By specifying a max number (depth) of followers, the tree can be balanced. The tree-hierarchy formation algorithm is described in [4].

The leader election algorithm can be extended to allow followers to continue to listen for other leaders, instead of returning once a leader has recruited the follower. The purpose is to enable processors to act as communication hops between groups, as illustrated in the following section. The current leader of the follower becomes the primary leader, and the follower then allows for secondary leaders (and so forth).

5.3.3 Distribution of an HDSTN in Ad-hoc Networks

This section describes a distribution method for ad-hoc networks that addresses the issue of ensuring that processors that perform the pre-planning are within communication range of each other. The distribution method assumes that the leader election procedure in Figure 5.4 has been performed beforehand, and furthermore that tree-hierarchies are formed on top of the basic two-level hierarchy.

In many cases it is important to address the problem of distributing the

actual computations *evenly* on processors according to available processing power. This is key to networks of processors with limited power, for example for processors that run on batteries. It is undesirable to have one processor performing all computations and potentially running out of power before the others do. The method of distribution described here maximizes the distribution of computation by assigning followers of leaders subnetworks whenever possible. The averaging of computations is possible because the tree-hierarchy of groups formed by the leader election and group formation algorithm [4] can be mapped to the hierarchical network of a DHDSTN derived from TinyRMPL, as described next.

We assume that the processors have no prior knowledge of the HDSTN but are programmed by receiving the HDSTN from an external source. The objective is to enable distribution of an HDSTN among groups of processors in a network where communication between all pairs is possible using direct communication or routing. To accomplish this, a tree-hierarchy of a number of levels is built until one top leader in the hierarchy can be selected for the point of contact with the external source. The top leader can be selected, for example, by comparing the number of followers of leaders at the highest level of the tree-hierarchy and selecting the leader with the highest number of followers as the top leader. This requires the leaders to be within communication range of each other.

To enable efficient pre-planning by minimizing the size of the DHDSTN, robot teams are interpreted as single units, and the tasks are decomposed to represent an action for every team member at execution time. For example, if *Team1* is defined as robots *R,S,T*, then the pre-planner interprets *Team1.take-picture()[lb,ub]* as a start and end time event connected with a simple temporal constraint. Afterwards, the distributed executive decomposes the command to three parallel commands, one for each robot in *Team1*.

We introduce a HDSTN distribution method, shown in Figure 5.5, which enables parallel processing of parallel and sequential networks of HDSTNs by assigning subnetworks to followers and co-leaders, i.e. a leader at the same level as the distributor. The distribution method runs on every processor and takes in an HDSTN as input. The key property of the distribution method is to maximize the distribution of HDSTN subnetworks to followers, thereby maximizing the amount of parallel computations.

The HDSTN distribution procedure works as follows. Every processor is ready to receive an HDSTN network. Assume that a processor p with n followers receives an HDSTN N. The distribution method follows these rules:

- If N is a primitive command, then p assigns N to itself.

- If N is network with k subnetworks, then first p assigns the start and

```
HDSTN-Distribution-Procedure()
For every processor p
n = number of followers(p)
if received(HDSTN)
  if HDSTN = command with simple temporal constraint C
    assign C to p
  if HDSTN = choose or sequence or parallel network L
    with k subnetworks A-1....A-k
    assign start and end node of L to p
    if (n = 0) // no followers
      if p is leader and has a neighbor leader v
        send k/2 subnetworks to v
        assign k/2 subnetworks to p // the rest
      else
        assign k subnetworks to p // the rest
    else if (n > k)
      for k subnetworks
        send(subnetwork) to a follower of p
    else if (n < k)
      for n subnetworks
        send(subnetwork) to a follower of p
      assign (k-n) subnetworks to p
```

Figure 5.5: HDSTN Distribution procedure on amorphous computers

end node of N to itself. The distribution of the subnetworks depends on whether p has any followers and if there are more subnetworks than followers $(k > n)$:

- *No followers*. There are two possible distributions: 1) if p is a leader and has a neighbor leader v, it attempts to even out the computations by sending $k/2$ subnetworks to v and assigning $k/2$ subnetworks to itself. 2) there are no neighbor leaders, so p assigns all k subnetworks to itself.
- *More followers than subnetworks*. Assign k subnetworks to k followers of p randomly.
- *Less followers than subnetworks*. Assign n of the k subnetworks to the n followers randomly. Assign the rest $(k - n)$ subnetworks to p itself.

Figure 5.6 shows a tree-hierarchy of three levels with a top leader, depicted in the figure as a square. The top leader enables the external source to connect with the processor network and is the point of contact for distribution of an HDSTN. The white circles denote followers at the lowest level, level 0. The black circles are followers at level 0 that have both a primary and secondary leader, and the arrows point to their primary leader. The gray circles denote leaders at level 0 and the dotted circles denote their communication range. The lines between the gray circles and the square represent the tree-hierarchy at level 1. Note that if a processor is a leader at level j it is also a

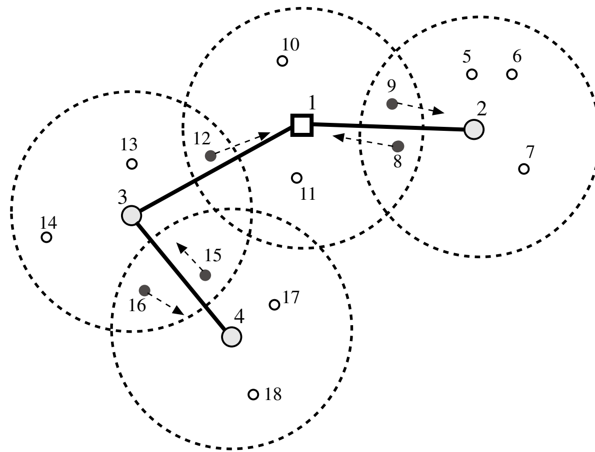

Figure 5.6: A three-level tree-hierarchy formed by Amorphous leader election.

leader at all lower levels $0...j-1$, because a tree-hierarchy at level n is based on the leaders at level $n-1$. For example, in Figure 5.6, processor 1 is leader at both level 1 and 0, while processors 2, 3, and 4 are leaders at level 0.

```
(parallel ; P1
    (sequence ; S1
        ((A())[1,2])
        ((B())[1,2])
    )
    (sequence ; S2
        ((C())[1,2])
        ((D())[1,2])
    )
    (choose ; C1
        ((E())[2,4])
        ((F())[2,4])
    )
)
```

Figure 5.7: TinyRMPL example for distribution on a amorphous computers network

The distribution method in Figure 5.5 is demonstrated as follows. Assume that the TinyRMPL program in Figure 5.7 must be distributed on the amorphous computer network in Figure 5.6. In the figure, each processor has an ID, and we now refer to processor p with ID i as p_i. Since TinyRMPL and HDSTN have an equivalent hierarchical property, walking through the TinyRMPL program simplifies the explanation.

Initially, the network is unstructured, so the network executes the leader

election algorithm first and creates a three level tree-hierarchy. Figure 5.6 shows the resulting hierarchy. It is assumed that the TinyRMPL program comes from an external source, which establishes a connection with the top leader p_1 (see Figure 5.6) and sends the TinyRMPL in form of an HDSTN to p_1. At the top level, the HDSTN has three subnetworks. p_1 assigns the start and end node of the network to itself. Furthermore, p_1 assigns the subnetworks $\{S1, S2, C1\}$ to the followers $\{p_2, p_3, p_4\}$ at level 1 with one subnetwork for each follower. Now each follower in $\{p_2, p_3, p_4\}$ processes its subnetwork in parallel with the others. p_2 processes $S1$ and since it is a sequence, it assigns the node pairs and simple temporal constraints of the primitive commands $A()$ and $B()$ to two of its followers p_5, p_6. Since primitive commands cannot be decomposed, p_5 and p_6 assign the commands to themselves. p_3 processes $S2$, assigns $C()$ to p_{13} and $D()$ to p_{14}. p_4 processes $C1$, but has no followers. p_4 has a neighboring leader (p_3), however, and sends half of the subnetworks (E, F) to p_3. Since $E()$ is a primitive command, p_3 assigns $E()$ to itself. p_4 keeps the other half, $F()$, and assigns that command to itself.

Even though the distribution method described above attempts to balance the workload on the processors and exploit parallelism, it can be optimized in a number of ways. This is discussed in Section 7.4, which is on future work.

Chapter 6

The Distributed Temporal Planning Algorithm

6.1 Overview

This chapter describes how Distributed Hierarchical Dynamic Simple Temporal Networks (DHDSTNs) are solved by the Distributed Temporal Planner (DTP). The description of DTP covers a high-level overview, the communication model, the algorithm, proof of soundness and completeness, a description of distributed consistency checking of DHDSTNs, and a walk-through of DTP running on the pursuer-evader scenario.

6.2 Solving Distributed HDSTNs

A temporally consistent execution of a TinyRMPL program is found in a distributed manner by solving its corresponding DHDSTN. The TinyRMPL program is reformulated as a Hierarchical Dynamic Simple Temporal Network (HDSTN) (Section 4.2) and distributed among the processors (Section 5.2), forming a DHDSTN. The DHDSTN is solved by searching the network in a hierarchical manner and assigning values to dynamic variables, while ensuring consistency of the active simple temporal constraints induced by the value assignments. DTP returns success when it has found a feasible solution. The activities corresponding to the active STN are then executed using a distributed version of the dynamic dispatching algorithm, introduced in [27, 35].

6.2.1 Introducing the Distributed Temporal Planner

DTP is a distributed algorithm in which every processor of the network runs an instance of the DTP algorithm. In collaboration, these processors find a *correct* plan, given a TinyRMPL program mapped to a DHDSTN as input.

Definition 6.1 *A plan is* correct *if and only if a solution to the DHDSTN exists, and the dynamic variables of the DHDSTN solution are assigned values such that the activity constraints are satisfied, and there are no negative weight cycles induced by the active simple temporal constraints.*

The significant advantages of DTP over centralized solutions are that 1) DTP performs parallel DHDSTN search when possible and 2) DTP simultaneously runs multiple isolated instances of the Bellman-Ford consistency check on different groups of processors in the DHDSTN. The remainder of this chapter details exactly how these advantages are achieved.

Distributed algorithms are inherently more complex than centralized algorithms. It is a greater challenge to solve the DHDSTN than HDSTNs, because the processing is distributed and asynchronous. In distributed processor networks, processors communicate with each other using messages. To search a DHDSTN, messages have to be propagated through the network and instantaneous message delivery is not guaranteed.

The technique for solving DHDSTNs is different from solving HDSTNs. The HDSTN-solver finds a solution to the dynamic CSP by performing chronological backtracking and by using the Bellman-Ford shortest path algorithm for consistency checks. DTP is a divide-and-conquer search method that exploits the fact that only parallel networks, described below, can create negative cycles and cause inconsistency [1]. Analogous to the HDSTN solver, DTP uses a distributed version of the Bellman-Ford shortest path algorithm to check for consistency. Moreover, DTP exploits the fact that the hierarchical network of TinyRMPL, and therefore DHDSTNs, enables both parallel search and concurrent isolated consistency checks. This speeds up the solution extraction tremendously, as the experimental results in Section 7.3 demonstrate.

Synchronization is used to ensure consensus in the network, i.e., that any processor at any given time is solving just one task, either searching the network and making variable assignments or checking for consistency, but *not* both simultaneously. An example of lack of consensus is the case in which a processor is waiting for a response of a search, but before it gets a response, it is asked by another processor to perform consistency checking. Synchronization prevents those situations from occuring. Synchronization is achieved by

[1] Kirk [18] exploited a similar property when pre-planning RMPL programs in the centralized case.

propagating *findfirst* and *findnext* search messages forward along the simple temporal constraints in the network of a DHDSTN to find consistent dynamic variable assignments and by waiting for responses in the form of *fail* or *ack* messages, for failure or success (acknowledge), respectively.

DTP performs a parallel recursive depth-first network search on a DHDSTN to make dynamic variable assignments and checks the simple temporal constraints for temporal consistency. During network search, dynamic variables are processed and assigned values when their associated decision nodes, $V_{decision}$, in the DHDSTN are visited. The algorithm ensures consistency at the deepest levels in the hierarchy and gradually moves up to higher levels until reaching the top level of the DHDSTN. This search method has two advantages: it automatically synchronizes the processors and it enables parallel search and consistency checks. However, to ensure completeness, DTP performs a systematic exhaustive search on the dynamic variables.

To understand the behavior of DTP, we first describe the message communication among processors when solving a DHDSTN. The next section describes how the DTP algorithm searches networks for consistent assignments and abstracts the distributed consistency check as a function. After that, soundness and completeness is proven. Finally, we describe distributed consistency checking in detail.

6.2.2 Message Communication Model

Messages in the DTP algorithm are sent between processor pairs. The DTP algorithm assumes an underlying communication protocol that provides seamless message exchange between any pair of processors in the network. For simplicity all messages *msg* sent by DTP have four data fields: *(SenderID, RecipientID, Type, Data)*. The $Data$ field is used only when performing consistency checks. DTP uses 6 different message types, which are shown below in the form *message-type(data)* :

- *findfirst()* is propagated in the forward direction of execution in time to search a subnetwork for consistent variable assignments. Whenever a node receives a *findfirst*, the node's parent-id is set to the parent of the sender to enable a feedback response later to that parent. The parent-id is necessary, since the parent of a processor may not be its neighbor and parents change when new variable assignments are made.

- *findnext()* is propagated in the forward direction of execution in time to search for the next consistent set of variable assignments in a subnetwork. The *findnext* message is used when a subnetwork was consistent by itself, but when combined with other subnetworks it is inconsistent. In that case, a systematic search is performed using *findnext*.

- *BF-init(level)* initializes a Bellman-Ford consistency check in a subnetwork at *level* and above. Consistency checking is described in Section 6.2.5.
- *BF-update(distance)* is used by the Bellman-Ford algorithm's update cycle.
- *fail()* indicates that the subnetwork is inconsistent with the current set of assignments within the subnetwork.
- *ack()* (acknowledge) indicates that a subnetwork has a consistent set of variable assignments.

The *fail* and *ack* messages are sent backwards in the opposite direction of execution in time towards the start node of a network. The Bellman-Ford messages are sent both forward and backward.

6.2.3 The DTP Algorithm

This section describes the Distributed Temporal Planner (DTP) algorithm. DTP exploits the hierarchical network of a DHDSTN mapped from TinyRMPL. The hierarchical network of TinyRMPL is created using *parallel, sequence*, and *choose* combinators recursively on top of primitive commands, i.e., *simple temporal constraints*.

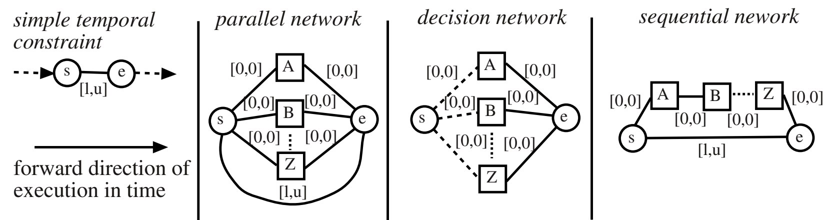

Figure 6.1: DHDSTN networks

These four types of DHDSTN networks are shown in Figure 6.1. Each network type consists of a start and end time event (proccesor node), denoted as circles in the figure, and DHDSTN subnetworks, denoted as squares. The edges represent simple temporal constraints between the start and end time events and subnetworks. DTP searches these networks to find consistent variable assignments. DTP uses the *findfirst* and *findnext* messages to find the first consistent variable assignments and next consistent variable assignments of subnetworks, in case the first were inconsistent. DTP checks for consistency at the deepest level of hierarchical parallel networks (see Figure 6.1)

and gradually moves up to the top level of a DHDSTN. If the top level of a DHDSTN N rooted at $start(N)$ is consistent, DTP returns success (*ack*). Analogous to the HDSTN solver, distributed STN consistency (Section 6.2.5) automatically considers only the active edges of the DHDSTN. The choices among methods are represented as dynamic variables; thus, backtracking in the context of the DTP algorithm involves undoing variable assignments and trying new assignments.

The next four sub-sections further describe DTP for the four types of sub-networks in Figure 6.1. The first three sub-sections, on searching simple temporal constraints, parallel networks and decision networks, concentrate on finding the first and next consistent variable assignment in networks, assuming that there are no sequential networks. The last sub-section describes searching sequential networks and finding the next consistent variable assignments, when the first were inconsistent, which completes the description of the search method of DTP.

Simple Temporal Constraint

In a DHDSTN, N, a simple temporal constraint has a start time event s and an end time event e, where $(s, e) \in V_{simple}$ of N (see Figure 6.2). Simple temporal constraints pass search requests forward and search responses backward during search. They cannot themselves induce negative cycles.

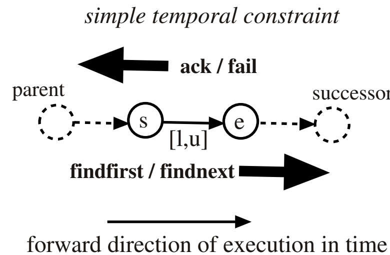

Figure 6.2: DTP search on a simple temporal constraint

During a search, node s receives either a *findfirst* or *findnext* from its parent and propagates it forward to the e node. The e propagates *findfirst* and *findnext* messages to its successor. When e receives a *fail* or *ack* from its successor, it propagates it backwards by sending it to s, which sends it to its parent.

Parallel Networks

Recall that DTP ensures consistency at the deepest levels in the hierarchy and gradually moves up to higher levels until it reaches the top level of the DHD-STN. For simplicity, assume for now that the function *check-consistency()* checks if the variable assignments of a subnetwork are temporally consistent and returns true if the subnetwork is consistent, or false otherwise. Check-consistency() is initialized by the start node s of a parallel network (see Figure 6.1). Consistency checking is explained in details in Section 6.2.5.

DTP searches a parallel network for temporally consistent choices by first sending a *findfirst* to all the subnetworks, before checking for temporal consistency of the entire parallel network. For example, in Figure 6.1, DTP first searches for consistent assignments in the subnetworks A $\ldots$Z, and checks that each of them is consistent before checking the entire network including s and e for consistency. The search of subnetworks is performed in parallel.

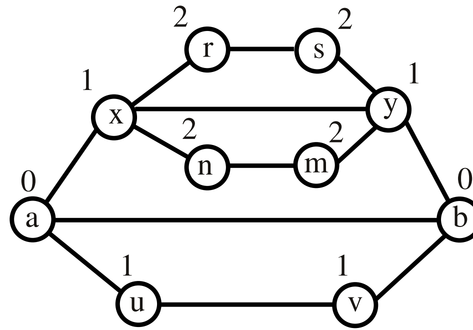

Figure 6.3: Levels of nodes in a parallel hierarchical network.

The start node $start(N)$ sends a *findfirst* to each child, where a child is defined as the start node of a subnetwork, $start(S_i)$. Figure 6.3 shows an example of a hierarchical parallel network with nodes and levels of nodes within the hierarchy. Let N be the entire network in Figure 6.3. Then $start(N) = a$ and $end(N) = b$. Let S_1 be the subnetwork that consists of the nodes $\{x, r, s, n, m, y\}$, then $start(S_1) = x$. In Figure 6.3, the nodes $\{a, b\}$ are at the top-level 0, the nodes $\{x, y, u, v\}$ are at level 1, and nodes $\{r, s, n, m\}$ are at level 2. The *findfirst* sent by $start(N)$ will eventually reach $end(N)$, which will reply with an *ack* message in the opposite search direction towards $start(N)$. If all subnetworks of N are consistent individually, $start(N)$ initializes a consistency check of the entire parallel network. Otherwise $start(N)$ sends a *fail* to its parent.

Figure 6.4 shows the pseudo-code of a start node $start(N)$ of a parallel

procedure parallelStartNode() //node v

```
1   wait for message msg
2   if msg = (findfirst)
3     set parent of v to msg.from
4     for each child
5       send(findfirst) to w
6     wait for all responses from children
7     if any of the responses is fail
8       send(fail) to parent of v
9     else // all ok
10      if check-consistency(v)?
11        send(ack) to parent
12      else
13        // search systematically
14        for w = child-0 to child-n //last child
15          send(findnext) to w
16          wait for response
17          if response = ack then
18            if check-consistency(v)?
19              send(ack) to parent
20              return
21            else // not consistent
22              w = child-0 // reset w
23          else // response is NOT ok
24            send (findfirst) to w
25            wait for response // it is ok
26        end-for
27        send(fail) to parent
```

Figure 6.4: Findfirst search method for start nodes of parallel networks.

network N for finding the first consistent assignment. The pseudo-code of end nodes of parallel networks is shown in Figure 6.5. DTP checks the network in the example in Figure 6.3 as follows. Node a is the parallel start node to which the pseudo-code in Figure 6.4 applies. Node a receives a *findfirst* from its parent (lines 1-2) and records its parent (line 3). Node a initiates the search by sending a *findfirst* to nodes $\{x, b, u\}$ in that order (lines 4-5).

procedure DTP-parallelEndNode()

```
1   wait for message msg
2   if msg = (findfirst) OR msg = (findnext)
3     if msg.from is parallel start node?
4       set parent of v to msg.from
5     send(ack) to parent
```

Figure 6.5: Findfirst and findnext search method for end nodes of parallel networks.

Three searches occur in parallel: 1) node x is also a parallel start node and receives *findfirst*, sets a to be its parent, and sends out a *findfirst* message to nodes $\{r, n, y\}$ (lines 1-5). 2) Meanwhile the parallel-end node $end(N) =$

b receives the *findfirst* message. It checks with its local knowledge if the sender is its corresponding start node of a parallel structure (Figure 6.5, line 3), which is the case, so it records the sender a as its parent and sends back an *ack* message to a. 3) Node u receives the *findfirst* from a and forwards it to v. v receives the message and forwards it to b, which sends back an *ack* message to v. v receives the *ack* and sends it back to u and finally to a. a still needs an *ack* message from x. In the meantime, once nodes r and n receive the *findfirst* messages from x, they perform a parallel search towards y, which sends back one *ack* message for each branch. Node y also receives a *findfirst* directly from x and records x as its parent and sends an *ack* message back to x. Node x receives the *ack* messages and starts a consistency check at level 1 and above (lines 6-10). Here it is assumed that the subnetwork is consistent, so node x sends an *ack* message to a (line 11). Node a finally initiates a consistency check at level 0 and above (lines 6-10), and assuming that the network is consistent, the search of the parallel network is successful.

In some cases, each subnetwork $S_i \in N$ is consistent, but combined with other subnetworks, N is inconsistent (line 12). In that case, DTP performs an exhaustive search to find variable assignments that make N consistent. $start(N)$ systematically sends a *findnext* to one child at a time to find the next consistent variable assignment (Figure 6.4, lines 14-27).

The pseudo-code for a parallel start node handling a *findnext* message is in its basic form identical to the systematic search in Figure 6.4, lines 14-27 when handling *findfirst* messages. However, the pseudo-code for *findnext* messages is extended to handling sequential networks, as described later.

Decision Networks

Making consistent decisions in networks is the core of DTP. The search mechanism for a decision network N, rooted at $start(N)$, makes one assignment to the dynamic variable $\delta_i \in \delta$ of $start(N)$ at a time until it finds a consistent assignment. For the decision network in Figure 6.1, for example, node s tries one assignment (A . . . Z) at a time.

procedure DTP-decisionEndNode()

```
1  wait for message msg
2  if msg = (findfirst) OR msg = (findnext)
3    set parent of v to sender of msg
4    send(ack) to parent
```

Figure 6.6: Findfirst and findnext search method for end nodes of a decision networks.

Figure 6.7 shows the pseudo-code of a decision start node $start(N)$ of a

decision network N for finding the first consistent assignment, and Figure 6.6 shows pseudo-code of end nodes of decision networks. When an end node of decision network receives a *findfirst* or *findnext* message, it records the sender as its parent and sends an *ack* back to the parent (Figure 6.6, lines 1-4).

procedure decisionStartNode()

```
1   wait for message msg
2   if msg = (findfirst)
3     parent = msg.from
4     for w = child-0 to child-n //last child
5       value assignment = w
6       send(findfirst) to w
7       wait for response from child w
8       if response = ack then
9         send(ack) to parent
10        return
11      else // fail
12        remove w from child list
13    end-for
14    // no more assignments (children) exist
15    send(fail) to parent
```

Figure 6.7: Findfirst search method for start nodes of decision networks.

When a start node $start(N)$ of a decision network receives a *findfirst*, $start(N)$ makes one assignment at a time and sends a *findfirst* message to the corresponding start node of a subnetwork of N (Figure 6.7, lines 5-6). When $start(N)$ receives an *ack* from the selected subnetwork, it sends an *ack* message to its parent, signifying that it found a consistent assignment for the entire network N (lines 8-10). In case of a *fail*, $start(N)$ removes the current value of the assignment from the domain, because no consistent assignment exists for that particular subnetwork (lines 11-12). $start(N)$ then continues with a new assignment to δ_i, until all values in domain Dom_i of δ_i have been examined. If all assignments fail, then $start(N)$ returns *fail* to its parent (line 15).

For example, in Figure 6.8, the first assignment of the start node of the decision network N is $a = 1$, which points to node x. Node x is a dynamic variable activated by $a = 1$. Node x makes the assignment $x = 1$ and propagates *findfirst* to r which is then forwarded to s, then y, then b. Node b replies to y with an *ack*, which is propagated backwards to x. x returns *ack* to a, which returns *ack* to its parent. Now suppose that x returns *fail*, then there are no valid assignments rooted at x. Next a makes a new variable assignment $a = 2$ and sends a *findfirst* message to u.

After the start node of a decision network N sends an *ack* message to its parent, it may later receive a *findnext* message from its parent, because N is inconsistent with some other parallel network. The pseudo-code of a

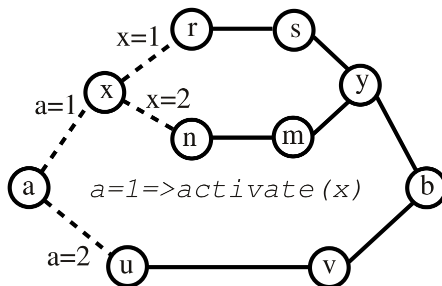

Figure 6.8: Decision network with an activity constraint.

procedure decisionStartNode()

```
1   if msg = (findnext)
2     w = current assignment (child)
3     // search on subnetwork
4     if w enables activity constraint
5       send(findnext) to w
6       wait for response
7       if response = ack
8         send(ack) to parent
9         return
10    while w < last child do
11      w = next assignment
12      send(firstfirst) to w
13      wait for response
14      if response = ack
15        send(ack) to parent
16        return
17      else // fail
18        remove w from child list
19    end-while
```

Figure 6.9: Findnext search method for start nodes of decision networks.

decision start node when a *findnext* is received is shown in Figure 6.9. Since DTP performs a depth-first search, $start(N)$ will check whether the current variable assignment enables any activity constraints. If that is true, a different variable assignment in the currently selected subnetwork can be made first before moving on to the next child (lines 4-5).

For the example in Figure 6.8, if the previous assignment of node a is $a = 1$, a will send a *findnext* to x, because the activity constraint $\{a = 1 \Rightarrow activate(x)\}$ is enabled. Node x does not have any activity constraints, and therefore makes a new assignment, $x = 2$ and returns *ack*.

If there was no enabled activity constraint or the *findnext* sent out to the currently selected child failed, DTP performs a search starting from the next child and searches the remaining children (lines 10-19) to find a consistent

variable assignment. For example, in Figure 6.8, if there was no enabled activity constraint at a, node a would ignore node x and immediately make a new assignment $a = 2$.

The $start(N)$ node returns *ack* as soon as a subnetwork returns *ack*. If no *findnext* of subnetworks of N were successful, $start(N)$ returns *fail*.

Sequential Networks

A sequential network consists of a series of interconnected subnetworks (Figure 6.1). Subnetworks can be simple temporal constraints, parallel networks, decision networks or sequential networks. A sequential network N also has a simple temporal constraint between $start(N)$ and $end(N)$, which requires a consistency check with the entire network. To accomplish this check, DTP views a sequential network as a parallel network case with only one compound subnetwork. When searching for valid assignments in a sequence of subnetworks A . . . Z, each subnetwork must be consistent and the overall network must be consistent as well.

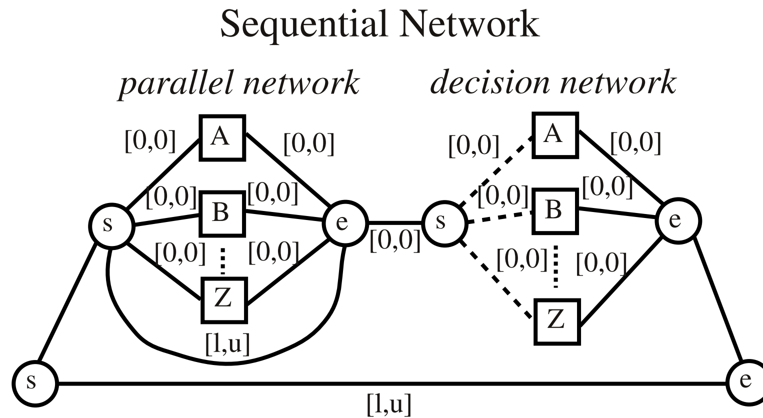

Figure 6.10: Sequential network example.

Figure 6.10 shows a sequential network with two subnetworks: a parallel network and a decision network. When searching for assignments, DTP checks the two subnetworks first. Then DTP checks the entire sequential network. DTP systematically searches subnetworks of a sequential network in parallel, and each subnetwork in the sequence is searched independently. For example, consider a sequence of three subnetworks, where each subnetwork represents a choice between two dynamic assignments (0 or 1). DTP performs a systematic search of up to $2^3 = 8$ assignments (000,100,010,. . . 111)

to identify a consistent assignment, if one exists. The systematic search of DTP is achieved by using *findfirst* and *findnext* messages on the subnetworks of sequential networks. The section below describes how parallel and decision start nodes are modified to perform systematic search in case there are sequences of subnetworks.

To simplify the communication between subnetworks during systematic search and to improve efficiency, we introduce the *SNI* (Sequential Network Id) reference pointer. SNI is used for parallel and decision subnetworks embedded within sequential networks. Start nodes of parallel and decision subnetworks use SNI to instantiate parallel search on their succeeding subnetwork and to wait for a response by communicating directly with the start node of their succeeding subnetwork. For example, if two subnetworks, A and B, are sequentially connected, then the SNI variable of start(A) is a pointer to start(B). For the sequential network A ... Z in Figure 6.1, $SNI(start(A)) = start(B)$, $SNI(start(B)) = start(C)$ and so forth. DTP does not perform parallel search on a sequence of simple temporal constraints, because it does not speed up search in that case.

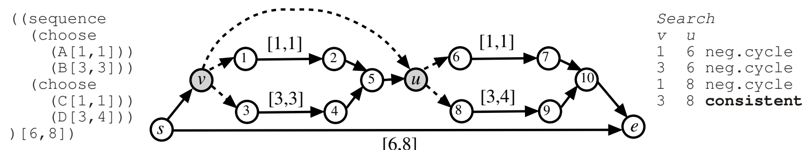

Figure 6.11: Example of a DHDSTN sequential network.

Figure 6.11 shows an example of two connected sequential networks with a simple temporal constraint, $(lb, ub) = (6, 8)$, on the entire graph. The SNI reference is represented as the dotted arc from v to u. In the figure, the TinyRMPL code is shown to the left and the corresponding DHDSTN is in the middle. Note that the command names are irrelevant to the preplanning processing. There are two connected decision networks, each with two options, hence a total of four combinations. Only the assignments $v = 3$ and $u = 8$ are consistent. In that case the upper time bound on the two decisions is $3 + 4 = 7$, which is greater than the overall lower time bound (6). However, three inconsistent combinations, shown to the right in Figure 6.11, are attempted first before the consistent assignments are found.

The DTP algorithm running on start nodes of decision and parallel networks is extended to find first and succeeding consistent assignments within a sequential network as follows. To find the first consistent assignments to the subnetworks of a sequential network, a *findfirst* message is sent to the start node of a subnetwork using SNI. For example, if $SNI(start(A)) =$

$start(B)$, then $start(A)$ searches subnetwork A and concurrently begins a search in subnetwork B by sending a *findfirst* message to B using the SNI. Then $start(A)$ synchronizes by waiting for the result of the search of the network A and the search of sequential network B. Next $start(B)$ receives the search request from $start(A)$ and sets its parent-id to $start(A)$ to enable a dreict response later to $start(A)$, thus jumping over the entire network A. If B has a sequential network C, $start(B)$ does the same as $start(A)$ and so forth. This method of parallel search generalizes to sequential networks of arbitrary length.

The pseudo-code of parallel start nodes for finding first consistent assignments is extended to enable search on sequential subnetworks by inserting the following two lines between lines 5 and 6 in Figure 6.4 :

```
if sequel B exists
    send(findfirst) to B
```

The parallel start node waits for a response from the sequence when waiting for responses from the children. To extend the pseudo-code of a decision start node for finding first consistent assignments, the same two lines are inserted between lines 3 and 4 in Figure 6.7, and "`wait for sequel B (if it exists)` is inserted" between lines 14 and 15 to synchronize with the sequential subnetwork. The resulting pseudo-code is shown in the Appendix, Section B.

procedure parallelStartNode() //node v

```
1  if msg = (findnext)
2    for w = child-0 to child-n //last child
3      send(findnext) to w
4      wait for response
5      if response = ack then
6        if check-consistency(v)?
7          send(ack) to parent
8          return
9        else // not consistent
10         w = child-0 // reset w
11     else // response is NOT ok
12       send (findfirst) to w
13       wait for response // it is ok
14   end-for
15   // no next configuration exists
16   if sequel B exists
17     send(findnext) to B
18     wait for response
19     send response (ack/fail) to parent
20   else
21     //no combinations are ok,
22     //sequential network fails
23     send(fail) to parent
```

Figure 6.12: Extended findnext pseudo-code of parallel start nodes.

The search gets more complex when a sequential network is inconsistent and the next valid assignment must be found. In this case, a sequential network must be searched systematically by trying remaining combinations of assignments to the subnetworks. The pseudo-code in Figure 6.12 shows how a parallel start node v processes a *findnext*. First, v systematically tries to find a consistent assignment locally, i.e., on its own subnetwork (lines 2-14). If there is no next consistent assignment to v's subnetwork, v checks if there is a sequential subnetwork (line 16). If there is, v sends a *findnext* to the start node of the immediate neighbor network $SNI(v)$ in order to try all combinations (lines 17-19). If there is no sequential subnetwork, v sends a *fail* to its parent (line 23). This method also generalizes to sequential networks of arbitrary lengths.

procedure decisionStartNode()

```
1   if msg = (findnext)
2     (first search for consistent assignment locally)
3     // search on sequel if it exists
4     if sequel B exists
5       // search on sequel
6       send(findnext) to B
7       // reset subnetwork
8       for w = child-0 to child-n //last child
9         value assignment = w
10        send(findfirst) to w
11        wait for response from child w
12        if response = ack then
13          break
14      end-for
15      // subnetwork will be ok
16      wait for response from B
17      send response (ack/fail) to parent
18    else
19      //no combinations are ok
20      send(fail) to parent
```

Figure 6.13: Extended findnext pseudo-code of decision start nodes.

The psuedo-code in Figure 6.13 shows how a decision start node processes a *findnext* and communicates with sequential subnetworks. First, the start node systematically searches for a consistent assignment on its own subnetwork (line 2) - the pseudo-code for this search is identical to the pseudo-code for processing *findnext* in Figure 6.9. If the search in line 2 fails, but a sequential subnetwork exists (line 4), the start node searches for the next consistent assignment to the sequential subnetwork (line 6) and resets its own subnetwork to the first consistent assignment, identical to processing a *findfirst* message, except for that the start node does not send a *findfirst* to the sequential network. If there is no sequential network, a *fail* is sent to the parent (line 20).

When searching for a consistent assignment to the network in Figure 6.11, for example, DTP first makes the assignments $v = 1$ and $u = 6$. Processor s initializes a consistency check, and determines that the assignments are inconsistent with the simple temporal constraint of the network ([6,8]). s sends a *findnext* to v, which still has unexamined values left in its domain. v makes the assignment $v = 3$ and immediately returns with an *ack* to s. This assignment is also inconsistent. In the following *findnext*, v is forced to reset its assignment to the first value in its domain ($v = 1$). v also sends a *findnext* to u using the SNI reference. u makes a new assignment $u = 8$. The combination $v = 1$ and $u = 8$ is inconsistent. Finally, the consistent combination of assignments $v = 3$ and $u = 8$ are found.

The pseudo-code of the DTP algorithm that runs on every processor is shown in the Appendix (Section B page 87).

6.2.4 Soundness and Completeness

Definition 6.2 *A solution to a DHDSTN is* feasible *if and only if the solution to the corresponding HDSTN is feasible. For HDSTN feasibility, see Definition 4.2.*

Proposition 6.1 Soundness of DTP. *If a given DHDSTN, N, has a feasible solution, $DTP(N)$ will return* ack*, otherwise it will return* fail*.*

Proof of Proposition 6.1: We prove that
$DTP(N) = ack \Rightarrow$
$\{\exists STN \subseteq N \mid STN$ is temporally consistent$\}$
by proving that $DTP(S)$ on any given DHDSTN network, S, (Figure 6.1) is *sound*, i.e., S will return *ack* iff S is temporally consistent. There are three cases to prove: 1) *trivial case*: A time variable event $x \in V_{simple}$ cannot itself create inconsistency. Let a be a simple time event variable, and B be an arbitrarily complex DHDSTN network. Assume that B is temporally consistent. A network S of a connected to B, creates a simple temporal constraint edge $\langle a, start(B), 0, 0\rangle$. Since this edge does not introduce new cycles in the corresponding distance graph, S is temporally consistent and always returns *ack*. 2) *parallel network case*: the start node of a parallel network P with arbitrarily complex subnetworks $\{A, B, ..., Z\}$ returns *ack* if and only if all subnetworks of P return *ack* AND P is temporally consistent. 3) *decision network case*: the start node of a decision network D with subnetworks $\{A, B, ..., Z\}$ returns *ack* iff there exists one subnetwork of D that returns *ack*. Any given DHDSTN mapped from a TinyRMPL program always consists of zero or more recursive combinations of DHDSTN networks. Since $DTP(S)$ is sound, where S is any of the three cases just described, DTP is *sound* when solving any DHDSTN derived from TinyRMPL. □

Proposition 6.2 Completeness of DTP. *If there exists a feasible solution of a given DHDSTN, N, $DTP(N) = ack$, i.e., DTP will find a feasible solution and return* ack.

Proof of Proposition 6.2: Because DTP in the worst case performs an exhaustive search on the graph of a given DHDSTN, N, trying all combinations of dynamic variable assignments, it will find a feasible solution if one exists, by the definition of exhaustive search.□

So far the search behavior of DTP has been described, and we mentioned that a parallel start node checks for consistency when it has synchronized with its children, before sending feedback to its parent. The following section explains how consistency checks are performed in a distributed fashion using the synchronized Bellman-Ford algorithm.

6.2.5 Checking Active STN Consistency in a Distributed Fashion

At any given point during the DHDSTN search, the current dynamic variable assignment forms an active subnetwork, analogous to an active subnetwork of an HDSTN. Temporal consistency is determined by running a single source shortest path (SSSP) algorithm on the distance graph corresponding to the active subnetwork, and by checking if there are any negative weight cycles [8]. The weights correspond to upper and lower time bounds (distances) between the nodes in the network. If there is a negative cycle, the HDSTN is not temporally consistent, i.e., it cannot be executed safely.

We use the distributed Bellman-Ford SSSP algorithm to check for negative cycles [22]. This has three major features: 1) the algorithm requires only local knowledge of the network at every processor, 2) it does not exhibit exponential behavior in its synchronized version when running on DHDSTNs, and 3) in general it runs in time linear in the number of processors in the network.

The processors in many networked intelligent systems run asynchronously. The asynchronous version of the Bellman-Ford shortest path algorithm is worst case exponential, with respect to both computation and communication. However, if one synchronizes the processors, then the shortest path can be determined in linear time [22]. The extra computation required for synchronization does not have a significant impact on the overall runtime of the algorithm.

The Bellman-Ford shortest path algorithm, which has also been widely used as a data network routing algorithm, works as follows: every node $i = 0..N$ in the network maintains an estimated distance, D_i, to a single predetermined target node $:= 0$. The algorithm is the reverse of a single source

shortest path algorithm because it finds *the shortest paths from all sources to a single target*. Initially D_i = 0 if i = 0 or $D_i = \infty$, otherwise. Every node i also maintains a table D_{ij} containing estimates of distance to target from neighboring nodes (j = 1,2,3...). A node j is a neighbor to node i if there is a simple temporal constraint between them. Every node i runs an update procedure which monotonically decreases D_i by comparing distance estimates, D_{ij}, going through neighboring nodes until it reaches the actual shortest distance to the target, given that there are no negative weight cycles. The update procedure does the following:

1. Update the table D_{ij} with message updates from neighboring nodes.

2. Update the distance estimate D_i :
   $D_i = min_j$ ($D_{ij} + d_{ij}$) where d_{ij} is the distance on the link from node i to j. It sets its distance to the value that minimizes the cost of going to the target through one of its neighbors.

3. Broadcast updated D_i to its neighbors.

DTP extends the Bellman-Ford algorithm in the following two ways to support consistency testing of dynamic variable assignments:

1. While the centralized Bellman-Ford shortest path algorithm [6] can detect negative weight cycles in a graph, the original *distributed* Bellman-Ford shortest path algorithm is an all sources single destination algorithm, and does not allow visiting the destination node, i.e., the node that initialized the consistency check, multiple times. Recall that the distributed Bellman-Ford algorithm monotonically decreases $D_i, i = 1 \ldots n$ until they converge to the shortest path distances. However, for the destination node (node 0), $D_0 = 0$, and D_0 cannot be decreased further, since it is the destination. There is no shorter path to itself than a path with zero distance. This restriction prevents the algorithm from searching cycles in graphs, which is needed to detect negative weight cycles.

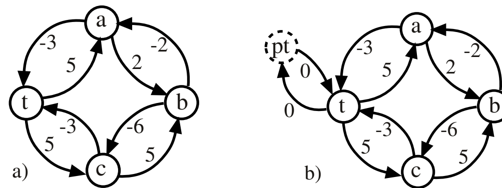

Figure 6.14: a) Simple distance graph b) Same graph with a phantom node.

For example, consider the distance graph in Figure 6.14a with four nodes $\{t, a, b, c\}$ and simple temporal constraints between the nodes. Node t is the

destination, so $D_t = 0$ and $D_i = \infty, i \neq t$. The graph contains a negative weight cycle ($t \to a \to b \to c \to t$). However, D_t cannot update its distance estimate, because it is *the* destination.

We introduce a *phantom-node* with a zero weight link to the destination node of the network. The phantom node becomes the destination, and the original destination can then be visited multiple times, permitting the detection of negative cycles. Figure 6.14b shows the graph in Figure 6.14a extended with a phantom node pt. During runtime of the Bellman-Ford algorithm, the original destination t may be updated with a negative distance estimate, but it will always reset its estimate to zero, satisfying the zero distance temporal constraint to the new destination. Nevertheless, negative weight cycles can be detected by other nodes, as explained below.

2. The other extension to the distributed Bellman-Ford for DTP is synchronization. The processors are synchronized to ensure a linear runtime proportional to the number of processors in the network (nodes in the DHD-STN). Linear runtime is a substantial improvement of the otherwise exponential asynchronous Bellman-Ford algorithm. To implement synchronization, the processors run N (number of processors) rounds. Each processor cannot increase its iteration counter during each round until it has received updates from all its processor neighbors. For every processor at round N, the distance estimate will have converged or not (see the proof in [2]). The distance estimate of a processor has converged if the estimate of round N equals the estimate in round N-1,i.e., $D_i^N = D_i^{N-1}$ for all $i = 0..N$. If D_i for one or more nodes has not converged, there is a negative weight cycle.

It is always true that during a particular consistency check, a subset of the processor network is running distributed Bellman-Ford in parallel. The significant advantage with the hierarchical network of the HDSTN and DHD-STN is that multiple concurrent checks can run on various subnetworks concurrently and independently from each other. This speeds up the solution time; see the experimental results in Section 7.3. Concurrent instances of consistency checks on subnetworks are isolated from each other using the level variables, defined in Section 5.2.1, as follows.

Only parallel start nodes (this includes sequential networks as well) initiate Bellman-Ford consistency checks. At the consistency check initialization, the *Bellman-Ford init* message is used to inform nodes about the bottom level, hence the level of the parallel start node, of the instance of that consistency check. Only nodes with a level greater or equal to the bottom level will be a part of the consistency check, i.e., only nodes within the parallel subnetwork rooted at the start node run the consistency check. For example in Figure 6.3, if node x initializes a consistency check at level 1 at start node x it will only affect nodes $S = \{x, y, r, s, n, m\}$. Even though the nodes $\{u, v\}$ are at level 1, they are not affected, because they are not directly connected to any nodes

in S and are not located within the parallel subnetwork.

After round N of the Bellman-Ford algorithm has been executed, the parallel start node that initialized the consistency check needs to be informed whether the subnetwork is temporally consistent. Since an undetermined number of processors in the subnetwork may indicate a negative consistency results, i.e., inconsistency ($D_i^N \neq D_i^{(N-1)}$), a converge cast is needed to gather this information and propagate it back to the parallel start node. A converge cast is a backward propagation of consistency results through the subnetwork and is initiated by the parallel end node at the lowest level of the consistency check. For example, in Figure 6.3, if node a initiated the consistency check, node b initiates a converge cast. The parallel start node that initiated the consistency check eventually receives consistency results from all its children and determines if the subnetwork is consistent.

The parallel end node sends an *ack* or *fail*, depending if it is consistent, to all its incoming neighbors to initialize the converge cast towards the parallel start node. Due to synchronization, all processors have executed N rounds and are waiting for the converge cast. When a processor receives a converge cast message, it performs a logical *AND* with its own consistency view, since *all* processors must be consistent. It then propagates the result to its parent. A parallel start node in the subnetwork synchronizes with all its n children performing n logical *AND*s before relaying the result to its parent. Eventually the consistency check initiator receives feedback from the converge cast.

For example, in Figure 6.3, if node x initiated a consistency check at level 1, then after N rounds, node y initiates a converge cast, since the level of y is 1. First y checks with its own distance estimate. Assume that it is consistent. Then y sends an *ack* message to s and m. Assuming that all nodes are consistent, the two *ack* messages are propagated backwards to node x, which determines that the subnetwork is consistent.

Consider now a consistency check initiated at level 0 in Figure 6.3. Then at round N, node b initializes a converge cast, since the level of b is 0. Assume that for all nodes i except for node x, $D_i^N = D_i^{N-1}$, indicating consistency. During the converge cast, b sends an *ack* to $\{y, a, v\}$. v relays the *ack* to u, which relays it to a. Now a need only synchronize with x. y relays the *ack* to $\{s, m\}$ and eventually x receives two *ack*s. However, since x is inconsistent, it sends a *fail* to a, and a determines that the subnetwork is inconsistent.

DTP in general always runs a number of Bellman-Ford iterations equivalent to the total number of nodes in the network, even though only a sub-set of the processors are running the Bellman-Ford algorithm. The reason is that the size of a subnetwork is dynamic and depends on the variable assignments, and a processor does not have real-time knowledge about a given size of a subnetwork when running Bellman-Ford. Section 7.4 on future work describes how to address the problem with this limitation.

The pseudo-code of distributed consistency checking and its integration into the search methods of the DTP algorithm is shown in the Appendix (Section B page 87).

6.2.6 Running DTP on the Persuer-evader Scenario

This sub-section walks DTP through the persuer-evader scenario, see the corresponding HDSTN in Figure 6.15. DTP makes the same variable assignments as the centralized HDSTN solver does on the same example (see Section 4.4). Additionally, DTP searches the network in parallel, as described below.

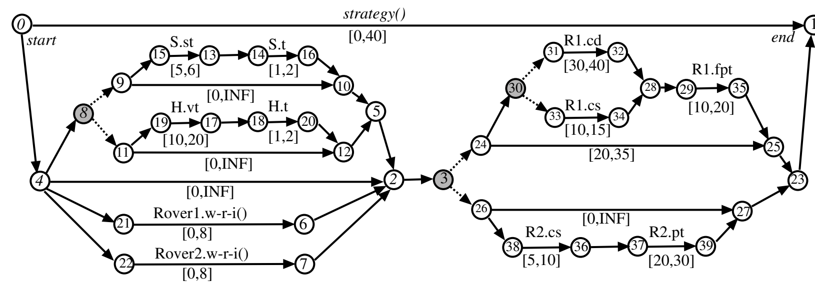

Figure 6.15: The pursuer-evader *Strategy* activity problem solved by DTP.

When DTP starts running on the processors of a DHDSTN N, they are in an idle-mode, i.e., they are not performing any computations but listening for messages. A *findfirst* message injected into the network to the origin node, $start(N)$, will initialize the solution process. The injection can take place by own initiative in the network or through an external interface to other computers such as robots, a base station or ground control.

The start node (0) sends out a *findfirst* message to nodes 1 and 4. Node 1 sends back an *ack* message to node 0. Node 4 sends out five *findfirst* messages to processors $\{2, 8, 21, 22, 3\}$; the *findfirst* message to 3 is sent using the SNI reference pointer. The process of node 4 sending out *findfirst* messages is illustrated in Figure 6.16a. In the figures in this sub-section, *findfirst* and *findnext* are abbreviated as *FF* and *FN*, respectively.

The search now becomes parallel as the *findfirst* messages propagate along the five paths simultaneously. Node 4 will wait for feedback from all nodes before returning a result to its parent, node 0. Recall that node 2 is a parallel end node and will just return *ack* messages back to message senders. Since the parent of node 3 becomes node 4, node 3 will report directly back to 4. The search on the nodes $\{21, 6, 22, 7\}$ is a propagation of *findfirst* messages

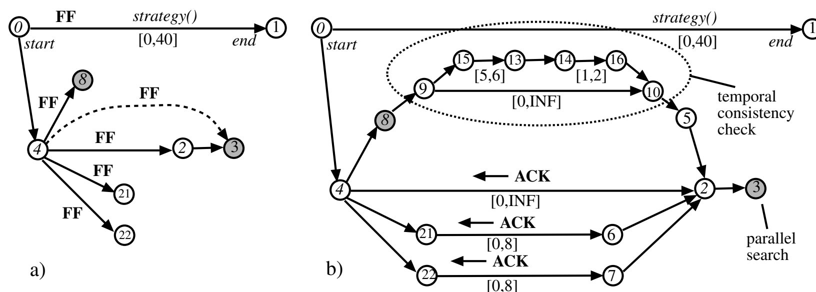

Figure 6.16: Snapshots of DTP searching the DHDSTN of the pursuer-evader problem.

towards 2 and *ack* messages backwards towards 4. The decision start node 8 makes the assignment $\delta_8 = 9$ and sends a *findfirst* to 9, from where the search propagates to nodes $\{9, 15, 13, 14, 16, 10, 5\}$. Figure 6.16b is a snapshot of the search, where *ack* messages are propagated backwards to node 4 and the subnetwork of nodes $\{9, 15, 13, 14, 16, 10\}$ performs a consistency check, while the subnetwork rooted at node 3 performs a parallel search, as described next.

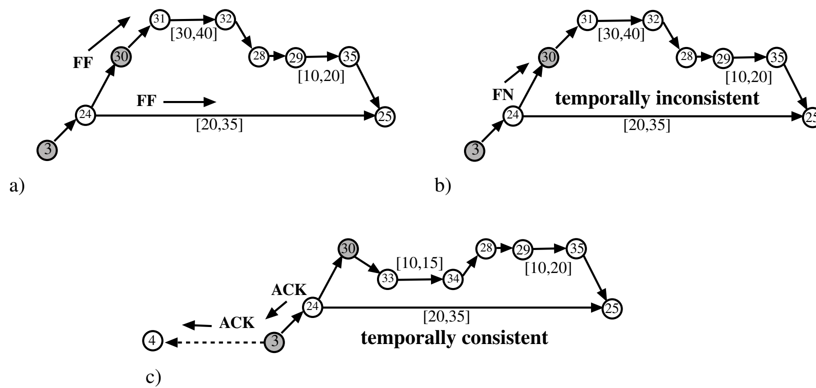

Figure 6.17: Snapshots of DTP searching the DHDSTN of the pursuer-evader problem.

Simultaneously, node 3 makes the variable assignment $\delta_3 = 24$, and sends a *findfirst* message to node 24, which activates the dynamic variable δ_{30}. δ_{30} makes the assignment $\delta_{30} = 31$ and sends a *findfirst* message to 31 (Figure 6.17a). Eventually the parallel start node 24 will initiate a consistency

check on nodes between itself and node 25. An inconsistency is detected, so node 24 sends a *findnext* to node 30 (Figure 6.17b). Node 30 makes a new assignment $\delta_{30} = 33$. This assignment is consistent at the level of 24, so an *ack* message is propagated to node 3 and then node 4 (Figure 6.17c).

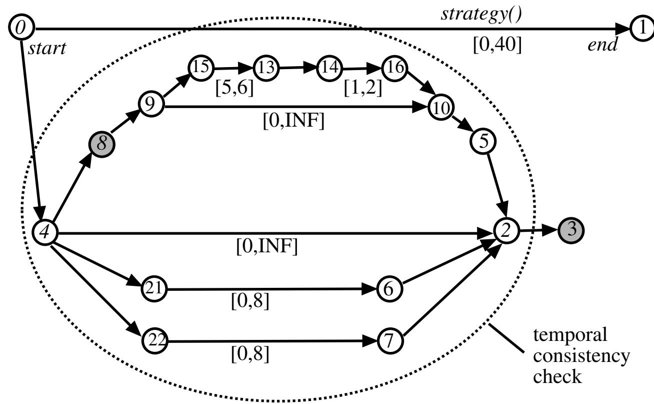

Figure 6.18: Snapshot of DTP searching the DHDSTN of the pursuer-evader problem.

A consistency check is performed on the parallel network between nodes 4 and 2 (Figure 6.18), which is consistent with the current assignments. Since the assignments are consistent at node 4, node 4 sends an *ack* to the start node 0, which checks that the entire network is consistent. The network is consistent and node 0 returns *success*. The final variable assignments are $\delta_8 = 9$, $\delta_3 = 24$, and $\delta_{30} = 33$. Figure 6.19 shows the final consistent STN selected by DTP.

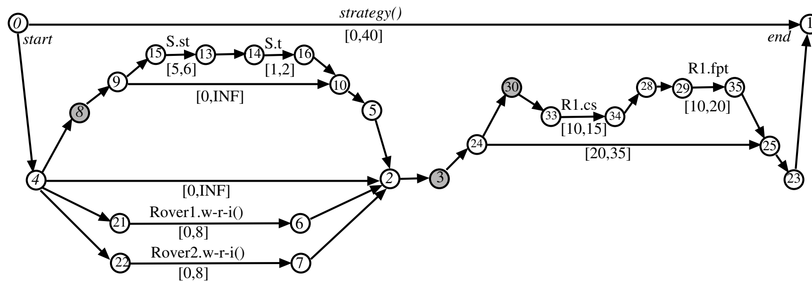

Figure 6.19: Final consistent STN of the pursuer-evader problem selected by DTP.

6.3 Summary

This chapter introduces the Distributed Temporal Planner (DTP) for the preplanning of a Distributed Hierarchical Dynamic Simple Temporal Network (DHDSTN). DTP identifies a temporally consistent selection of funtionally redundant methods prior to dispatching to ensure a safe execution. This chapter describes the DTP algorithm, proves soundness and completeness, explains the distributed temporal consistency checking and illustrates the behavior of DTP by walking through the solution extraction of the pursuer-evader strategy scenario. Chapter 7 concludes the thesis by covering the description of the implementation of DTP, experimental results, future work and a final summary.

Chapter 7

Conclusions

7.1 Overview

The first section of this chapter describes the implementation of the distributed temporal planner, including the compiler and simulator. The following section focuses on experimental results and discussion. The chapter ends with a section on future work and a summary.

7.2 Implementation

The implementation consists of the *TinyRMPL to HDSTN compiler* and a *software simulator* for the HDSTN distribution algorithm and DTP. The entire implementation is written in ISO/ANSI-compliant C++. The TinyRMPL to HDSTN compiler outputs the HDSTN file in XML format [10]. XML is today's standard media for document exchange across networks and between various computer platforms. The simulator is a batch program, which uses the Xerces-C XML library [3] for parsing the XML file.

7.2.1 TinyRMPL to HDSTN Compiler

The compiler translates TinyRMPL code into HDSTN by performing the mapping described in Chapter 4. Additionally, the compiler performs a search over the HDSTN in order to extract activity constraints and sequential network IDs (SNIs). It saves the final HDSTN in XML. The compiler generates a graphical representation of the HDSTN and saves it as a Postscript file using the GraphViz Dot program [15]. Although GraphViz is very useful for illustration of HDSTNs, it becomes impractical to read the generated graphs

when the number of nodes exceeds 30 to 40. The XML format that represents an HDSTN is specified in the Appendix, Section C.

7.2.2 Software Simulator for the Distributed Temporal Planner

The simulator distributes the HDSTN by creating robot agents that represent real robots. Each robot agent owns a number of virtual agents. Virtual agents have simple temporal constraints between them. A simple temporal constraint points from a virtual agent to another virtual agent inside the same robot agent or to a virtual agent in an external robot agent. The simulator takes care of message communication using a routing table, which determines how messages are routed within the network.

The simulator runs the temporal planner in cycles. The DTP procedures for processors are implemented as finite state machines, enabling execution of multiple processors in one thread. The implementation of the DTP algorithm (namely the while loops) for the different types of processors is divided into states. In each state a few lines of code is executed on each processor at a time, simulating a truly multi-threaded environment with one thread. During runtime, the user can:

- execute one DTP cycle at a time,
- jump ahead to a certain planning cycle,
- let the simulator run until the planning process finishes,
- view state and local knowledge information of the processors,
- view the current assignments of the dynamic variables, and

The implementation of the DTP simulator consists of the following C++ classes:

- *AgentSimulator.cpp* distributes an HDSTN, simulates robot agents, and enables message communication between robot agents.
- *PhysicalAgent.cpp* contains one or more virtual agents and simulates all of its virtual agents. It also handles data exchange among virtual agents that reside in the same robot agent.
- *VirtualAgent.cpp* represents a DHDSTN node (see Figure 5.1) and runs DTP where the actual DTP code depends on the node type flag. The pseudo-code is shown in the Appendix, Section B.

- *AgentLink.cpp* represents a simple temporal constraint between two virtual agents.
- *AgentMessage.h* defines the message structure used for communication.

7.2.3 Porting the Code to Other Systems

The code can ported to any other system that has a C++ compiler. The Xerces-C XML library exists for most common platforms. For porting DTP to a distributed platform, such as a rover test-bed, the PhysicalAgent and its classes must be running on each robot. To port the code to the tiny processors, the PhysicalAgent, VirtualAgent and AgentLink classes must be converted to C code compatible with the Tiny OS[1], but the core algorithm implementation does not need to be changed.

7.3 Experiments and Discussion

The implementation of DTP has been tested on a 1.133GHz PC with 384MB of RAM. In brief, the system simulates an array of processors solving a DHD-STN in rounds. In every round, every processor performs a *listen-act-respond* cycle. We have implemented a random TinyRMPL code generator in Java to test DTP. The TinyRMPL generator takes in three parameters *(C,D,N)*, where C denotes the number of desired TinyRMPL combinators, D denotes the maximum recursive depth, and N denotes the desired number of corresponding DHDSTN processors. The generator creates TinyRMPL code while attempting to fulfill the parameters.

Range	*Trials*	*Nodes*	*Cycles*	*RunT/ms*	*Assignmt.*	*Checks*	*Backtrack*	*Messages*	*Success*
0-10	5	6.00	10.00	6.00	0.50	0.50	0.00	69.00	1.00
11-20	5	16.38	38.94	39.88	1.38	1.19	0.38	558.31	0.75
21-30	5	24.83	36.08	61.13	0.88	0.71	0.13	828.08	0.63
31-40	5	32.00	41.73	72.73	2.09	0.82	0.09	1030.36	0.91
41-50	5	44.67	54.07	138.87	3.27	0.73	0.27	2087.67	0.40
51-60	5	53.62	64.69	180.31	3.00	0.81	0.31	2342.85	0.54
61-70	5	63.00	101.13	162.63	3.50	1.63	1.13	2251.75	0.50
71-80	5	72.86	73.43	175.86	4.14	1.14	0.57	2288.71	0.43
81-90	5	83.00	106.50	239.00	3.50	1.33	1.00	3238.17	0.50
91-100	5	92.36	125.27	297.82	3.45	1.45	0.64	4222.73	0.82

Table 7.1: Empirical results.

For each TinyRMPL program, we compiled it to an HDSTN, distributed it among the processors, and ran 5 trials to average small runtime fluctuations.

[1]TinyOS Website: http://webs.cs.berkeley.edu/tos/

Each robot agent had exactly one virtual agent. The running time and number of cycles needed to terminate depended heavily on the amount of backtracking and consistency checks. In order to smooth out the effects of outlier results, we sorted the trials into buckets depending on the number of nodes of a DHDSTN, in increments of 10. The results are shown in Table 7.1 and graphically in Figure 7.1. The graph shows a linear increase in the number of cycles before completion as a function of the number of nodes in the DHDSTN. Some of the variations due to outliers were not fully averaged out. In

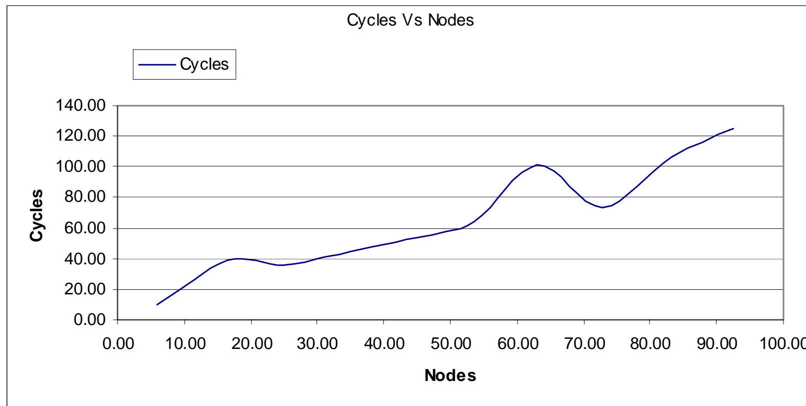

Figure 7.1: Graphical depiction of empirical results, cycles vs. nodes.

our tests, we generated several hundred randomly generated TinyRMPL programs with varying parameters in the ranges (C=[3,30],D=[4,10],N=[5,200]). Tests included running DTP with from tens of nodes to over two thousand nodes. The table shows the average test results for 10 different buckets and indicates a relatively steady, linear increase in all variables - cycles, runtime in ms, total number of assignments, number of consistency checks, number of backtracks, and processor-to-processor messages - as a function of the average number of nodes in a particular bucket. The robot example from Section 1.3 is relatively small and is solved in around 120 cycles by DTP.

The result is what was expected, for two reasons: 1) As the parameter values of the TinyRMPL generator increase, the random programs become increasingly complex and harder to solve; and 2) the increase in cycles as a function of nodes is linear or close to linear, because DTP performs parallel search and consistency checks *whenever* possible. The network search including variable assignments is of linear complexity and the distributed Bellman-Ford algorithm is of linear complexity. However, the runtime depends on the number of backtracks and consistency checks, proportional to

the complexity of the problem.

The pre-planning problem in general is worst-case exponential. The rightmost column of the table shows the average success rate (indicating whether a solution was found), where 0 is a failure and 1 a success. Given the success rates in the table, we can infer empirically that the random TinyRMPL generator produced a large variety of programs. It did not create programs that were extremely hard to solve, which would show polynomial or even exponential running time.

The experimental results show that the number of processor-to-processor messages increases significantly as a function of nodes and cycles. The high number of messages is mainly due to the distributed Bellman-Ford algorithm which is of quadratic communication complexity with respect to the number of nodes in a network. During consistency checking in a DHDSTN, each processor typically sends at least two Bellman-Ford updates to neighbors in a cycle; hence with n processor and n cycles, at least $2n^2$ messages will be sent during a Bellman-Ford consistency check on the entire graph. The numbers in the message column in Table 7.1 would be reduced by about 50% or more if local Bellman-Ford update broadcasts to neighbors within the communication range of a processor were counted instead of processor-to-processor messages, because HDSTN nodes on average have two to three neighbors. The other reason for the high number of messages is that the consistency check always runs $|V|$ Bellman-Ford iterations on any subnetwork independent of its level, since processors cannot know the dynamic size of a subnetwork during runtime and that size can also not be calculated at compile-time.

An efficient way to determine the number of Bellman-Ford iterations, which does not increase the computational complexity of DTP, is to count nodes within each subnetwork during search. When an end node of a parallel subnetwork receives a *findfirst* or *findnext*, it can send back an *ack* with a counter set to 0 using the *data* field of a message. Each node visited on the way towards the parallel-start node increases the counter. Parallel-start nodes return to their parents the sum of the counts of all their children. When the Bellman-Ford algorithm is initialized, the initialization message must include the counted number of nodes, i.e., the number of Bellman-Ford iterations, along with the level information.

7.4 Future work

Conflict-directed backtracking could improve the DTP search. This would require DTP to locate the negative weight cycles in order to prune. Negative cycles can be located with centralized algorithms. For example, the second half of the centralized Bellman-Ford shortest path algorithm [6] identifies

edges that are part of negative cycles. It is much harder to identify negative cycles in distributed asynchronous networks, even if the shortest path algorithm is synchronized. Additionally, a better backtrack search only helps in some cases, because an inconsistency is often induced at the top level, which makes it hard to identify the parallel threads that caused the inconsistency and prune.

Scaling DTP to run on large numbers of processors requires the reduction of message communication. Communication can be reduced by counting the number of nodes and hence iterations before each Bellman-Ford consistency check, as described above, and by allowing each processor to execute DTP on collections of variables and constraints. Nevertheless, our results indicate that DTP is an efficient distributed algorithm for ensuring safe execution on networked embedded processors with widely varying computational resources.

The ad-hoc computer network distribution algorithm for HDSTNs presented in section Section 5.3 takes the first steps towards the goal of applying model-based programming and distributed coordination to ad-hoc networks. Much more could be done to improve the distribution method. One could apply a more intelligent load-balancing distribution method that had more extensive knowledge about the topology of the group hierarchies and the resources available for each robot. Additionally, the current distribution method is static, but the environment of these networks is inherently dynamic. For example, processors fail from time to time. To adapt to a dynamic environment and to increase robustness, the tree-hierarchy formation algorithm and the HDSTN distribution method should run continuously to adapt to network topology changes and to reallocate pre-planning tasks as necessary.

Lastly, the TinyRMPL language ought to be extended with more features from RMPL, such as preemption and conditionals [36], to enable the description of more complex scenarios. Currently, TinyRMPL uses a subset of RMPL to support deployment on robots that are severely constrained with respect to computational power. However, more capable robots could and should support all the features of RMPL.

7.5 Summary

Robotic execution languages improve robustness by managing functionally redundant procedures for achieving goals. The model-based programming approach extends this by guaranteeing correctness of execution through preplanning of non-deterministic timed threads of activities. Executing model-based programs effectively on distributed autonomous platforms requires distributing this pre-planning process.

This thesis presents a distributed planner for model-based programs whose

planning and execution is distributed among agents with widely varying levels of processor power and memory resources. TinyRMPL is a compact language that leverages the hierarchical properties, functionally redundant methods, and flexible time bounds of RMPL and enables distribution of model-based programs among robots that are severely constrained with respect to power and memory. Hierarchical Dynamic Simple Temporal Networks (HDSTNs) map directly from TinyRMPL and enable 1) efficient task distribution because of the minimal local knowledge requirements, and 2) efficient parallel pre-planning including parallel consistency checks, enabled by the hierarchical properties of HDSTNs. These contributions are implemented within the Distributed Temporal Planner (DTP) system, which includes a TinyRMPL to HDSTN compiler, distribution of HDSTNs with robot and virtual agents, and distributed pre-planning. The initial, simple distribution method is extended through leader election and hierarchical group formation, and through a more advanced distribution method of HDSTNs within ad-hoc computer networks. Finally, the experimental results indicate that DTP is an efficient distributed algorithm for ensuring safe execution on heterogeneous robots with widely varying computational resources. This research presents a first step toward distributed model-based planning. Future research will be required to further develop the ideas of the distributed model-based programming paradigm.

Bibliography

[1] H. Abelson and D. Allen et.al. Amorphous computing. *Communications of the ACM*, 2000.

[2] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, 1992.

[3] Xerces c XML Parser by Apache. http://xml.apache.org.

[4] Daniel Coore, Rhadika Nagpal, and Ron Weiss. Paradigms for structure in a amorphous computer. *MIT AI Memo No. 1614*, 1997.

[5] Daniel D. Corkill. Hierarchical planning in a distributed environment. In *IJCAI-79*, 1979.

[6] Cormen, Leiserson, and Rivest. *Introduction to Algorithms*. MIT Press, 1995.

[7] J. de Kleer. An assumption-based tms. *Artificial Intelligence*, 28(2),March,1986.

[8] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence, 49:61-95, 1991*, 1990.

[9] M. desJardins, E. Durfee, C. Ortiz, and M. Wolverton. A survey of research in distributed, continual planning. *AI Magazine*, 1(4):13 – 22, 1999.

[10] Extensible Markup Language (XML) documentation. http://www.w3.org/XML.

[11] T. Estlin, A. Gray, T. Mann, G. Rabideau, R. Castano, S. Chien, and E. Mjolsness. An integrated system for multi-rover scientific exploration. In *AAAI-99*, 1999.

[12] T. Estlin, G. Rabideau, D. Mutz, and S. Chien. Using continuous planning techniques to coordinate multiple rovers. *Electronic Transactions on Artificial Intelligence, 4:45-57, 2000*, 2000.

[13] R. James Firby. An investigation into reactive planning in complex domains. In *In Proc. of the 6th National Conf. on AI, Seattle, WA, July 1987.*, 1987.

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

[15] AT&T GraphViz. http://www.graphviz.org/ or http://www.research.att.com/sw/tools/graphviz/.

[16] Amorphous Computing Homepage. http://www.swiss.ai.mit.edu/projects/amorphous.

[17] M. N. Huhns and D. M. Bridgeland. Multiagent truth maintenance. *IEEE Transactions on Systems, Man and Cybernetics 21(6),1437-1445.*, 1991.

[18] Phil Kim, Brian Williams, and Mark Abramson. Executing reactive, model-based programs through graph-based temporal planning. In *Proceedings of IJCAI-2001, Seattle, WA*, 2001.

[19] H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjoh, and S. Shimada. Robocup-rescue: Search and rescue for large scale disasters as a domain for multi-agent research. In *Proceedings of IEEE International Conference Systems, Man and Cybernetics*, 1999.

[20] Long-Ji Lin, Reid Simmons, and Christopher Fedor. Experience with a task control architecture for mobile robots. Technical report, Robotics Institute, Carnegie Mellon University, 1989.

[21] J.S. Liu and K.P.Sycara. Multiagent coordination in tightly coupled task scheduling. In *Proceedings of the Second International Conference on Multi-Agent Systems, pp.181-188*, 1996.

[22] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.

[23] K. Hirayama M. Yokoo. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-agent Systems, Vol.3.,No.2,pp.192-212*, 2000.

[24] Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *AAAI-1990*, 1990.

[25] J. Modi, H. Jung, M. Tambe, W. Shen, and S. Kulkarni. Dynamic distributed resource allocation: A distributed constraint satisfaction approach. In *Intelligent Agents VIII Proceedings of the International workshop on Agents, theories, architectures and languages (ATAL'2001)*, 2001.

[26] Pragnesh Jay Modi, Milind Tambe Wei-Min Shen, and Makoto Yokoo. An asynchronous complete method for general distributed constraint optimization. In *Proceedings of Autonomous Agents and Multi-Agent Systems Workshop on Distributed Constraint Reasoning*, 2002.

[27] P. Morris and N. Muscettola. Execution of temporal plans with uncertainty. In *AAAI-00*, 1999.

[28] Vincent Park and M. Scott Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *IEEE Proceedings of INFOCOM 1997, Kobe, Japan.*, 1997.

[29] P. Pirjanian, T. Huntsberger, A. Trebi-Ollennu, H. Aghazarian, H. Das, S. Joshi, and P. Schenker. Campout: a control architecture for multirobot planetary outposts. In *Proc. SPIE Conf. Sensor Fusion and Decentralized Control in Robotic Systems III, Boston, MA, Nov 2000.*, 2000.

[30] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence, Vol 9, Number 3*, 1993.

[31] Patrick Riley and Manuela Veloso. Planning for distributed execution through use of probabilistic opponent models. In *IJCAI-2001 Workshop PRO-2: Planning under Uncertainty and Incomplete Information*, 2001.

[32] Elizabeth Royer and Chai-Keong Toh. A review of current routing protocols for ad hoc mobile wireless networks. *IEEE Personal Communications*, April 1999.

[33] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

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

[35] I. Tsamardinos, N. Muscettola, and P. Morris. Fast transformation of temporal plans for efficient execution. In *AAAI-98*, 1998.

[36] B. C. Williams, M. Ingham, S. Chung, and P. Elliott. Model-based programming of intelligent embedded systems and robotic explorers. In *In IEEE Proceedings, Special Issue on Embedded Software*, 2003.

Appendix A

Pseudo-code for HDSTN-solver

```
procedure HDSTN_Solver(n)
  consistent = true
  i = 1
  while true
    if consistent?
      i = label(i,consistent)
    else
      i = unlabel(i,consistent)
    if i > n
      return success
    else if i = 0
      return failure

procedure label(i,consistent)
  while v[i] inactive and i<=n
    i = i + 1
  if i > n
    return i
  for v[i] = each element of currDom[i]
    consistent = Bellman-Ford(DSTN,0)
    if not consistent?
      currDom[i]=remove(v[i],currDom[i])
    else
      update set of variables based on
        activity constraints
  end-for
  if consistent
    return i+1
  else
    return i
```

```
procedure unlabel(i,consistent)
  h = i-1
  while v[i] inactive and i>0
    h = h - 1
  if h = 0 return h
  currDom[i]=domain[i]
  currDom[h]=remove(v[h],currDom[h])
  update set of variables based on
    activity constraints
  consistent = currDom[h] != nil
  return h


procedure Bellman-Ford(G,s)
initialize-single-source(G,s)
V[G] = V-simple + V-decision +
  V-parallel
for i=1 to |V[G]|-1
  for each active edge (u,v) in E[G]
    relax(u,v,w)
for each active edge(u,v) in E[G]
  if d[v] > d[u]+w(u,v)
    return false
return true


subprocedure relax(u,v,w)
if d[v] > d[u]+w(u,v)
  d[v] = d[u]+w(u,v)
```

Figure A.1: HDSTN-solver pseudo-code.

Appendix B

DTP Pseudo-code

Figure B.1 shows the DTP pseudo-code for processors with the parallel-start flag set. The check-consistency() procedure is a helper function used to initialize consistency check and to process the results. Figure B.3 shows the DTP pseudo-code for processors with the decision-start flag set. Figure B.5 shows the DTP pseudo-code for processors with the parallel-end flag, decision-end flag, or primitive flag set, and the pseudo-code for the distributed Bellman-Ford consistency check.

```
procedure check-consistency(x)
  BF-level = level // isolate check
  for (each neighbor w at BF-level
    and above)
    send (BF init) to w
    send (BF update) to w
  run distributed-Bellman-Ford()
    on subgraph rooted at x
  wait for BF responses
  if consistent?
    return true
  else
    return false

procedure parallelStartNode() //node v
wait for message msg
if msg = (findfirst)
  set parent of v to msg.from
  for each child
    send(findfirst) to w
  if sequel B exists
    send(findfirst) to B
  wait for all responses from children
  wait for response from B
  if any of the responses is fail
    send(fail) to parent of v
  else // all ok
    if check-consistency(v)?
      send(ack) to parent
    else
      // search systematically
      for w = child-0 to child-n //last
        send(findnext) to w
        wait for response
        if response = ack then
          if check-consistency(v)?
            send(ack) to parent
            return
          else // not consistent
            w = child-0 // reset w
        else // response is NOT ok
          send (findfirst) to w
          wait for response // it is ok
      end-for
      send(fail) to parent
```

Figure B.1: a) DTP pseudo-code for processors with the parallel-start flag set.

```
(continued)
else if msg = (findnext)
  for w = child-0 to child-n //last child
    send(findnext) to w
    wait for response
    if response = ack then
      if check-consistency(v)?
        send(ack) to parent
        return
      else // not consistent
        w = child-0 // reset w
    else // response is NOT ok
      send (findfirst) to w
      wait for response // it is ok
  end-for
  // no next configuration exists
  if sequel B exists
    send(findnext) to B
    wait for response
    send response (ack/fail) to parent
  else
    //no combinations are ok,
    //sequential network fails
    send(fail) to parent

else if msg = (BF init)
  BF-level = data(msg)
  for each neighbor w
    send (BF init) to w
    send (BF update) to w

else if msg = (BF update)
  run distributed-Bellman-Ford-update()
```

Figure B.2: b) DTP pseudo-code for processors with the parallel-start flag set.

```
procedure decisionStartNode()
wait for message msg
if msg = (findfirst)
  parent = msg.from
  if sequel B exists
    send(findfirst) to B
  for w = child-0 to child-n //last child
    value assignment = w
    send(findfirst) to w
    wait for response from child w
    if response = ack then
      wait for sequel B (if there is any)
      if (sequel B and sequel is OK)
        OR no sequel
        send(ack) to parent
      else
        send(fail) to parent
      return
    else // fail
      remove w from child list
  end-for
  // no more assignments (children) exist
  wait for sequel B
  send(fail) to parent
```

Figure B.3: a) DTP pseudo-code for processors with the decision-start flag set.

```
(continued)
else if msg = (findnext)
  w = current assignment (child)
  // search on subnetwork
  if w enables activity constraint
    send(findnext) to w
    wait for response
    if response = ack
      send(ack) to parent
      return
  while w < last child do
    w = next assignment
    send(firstfirst) to w
    wait for response
    if response = ack
      send(ack) to parent
      return
    else // fail
      remove w from child list
  end-while
  // search on sequel if it exists
  if sequel B exists
    // search on sequel
    send(findnext) to B
    // reset subnetwork
    for w = child-0 to child-n //last child
      value assignment = w
      send(findfirst) to w
      wait for response from child w
      if response = ack then
        break
    end-for
    // subnetwork will be ok
    wait for response from B
    send response (ack/fail) to parent
  else
    //no combinations are ok
    send(fail) to parent

else if msg = (BF init)
  BF-level = data(msg)
  for current selected child w
    send (BF init) to w
    send (BF update) to w
  send (BF init) to parent
  send (BF update) to parent

else if msg = (BF update)
  run distributed-Bellman-Ford-update()
```

Figure B.4: b) DTP pseudo-code for processors with the decision-start flag set.

```
procedure DTP-parallelEndNode()
wait for message msg
if msg = (findfirst) OR msg = (findnext)
  if msg.from is parstart?
    set parent of v to msg.from
  send(ack) to parent

else if msg = (ack) OR msg = (fail)
  send msg to parent

else if msg = (BF init)
  BF-level = data(msg)
  if BF-level = level
    BF-par-end? = true
  for (each neighbor w at BF-level
    and above)
    send (BF init) to w
    send (BF update) to w

else if msg = BF update
  run distributed-Bellman-Ford-update()


procedure DTP-decisionEndNode()
wait for message msg
if msg = (findfirst) OR msg = (findnext)
  set parent of v to sender of msg
  send(ack) to parent

else if msg = (ack) OR msg = (fail)
   send(msg) to parent

else if msg = (BF init)
  BF-level = data(msg)
  if (neighbor w in forward direction
    at BF-level and above)
    send (BF init) to w
    send (BF update) to w
  send (BF init) to parent
  send (BF update) to parent

else if = (BF update)
  run distributed-Bellman-Ford-update()
```

Figure B.5: DTP pseudo-code for processors with the flag set to parallel-end or decision-end.

```
procedure DTP-primitiveNode()
wait for message msg
if msg = (findfirst) OR msg = (findnext)
  relay message forward

else if msg =(fail) OR msg = (ack)
  relay message backwards

if msg = (BF init)
  BF-level = data(msg)
  for each neighbor w
    send (BF init) to w
    send (BF update) to w

if msg = (BF update)
  run distributed-Bellman-Ford-update()


procedure distributed-Bellman-Ford-update()
  update distance table
  if synchronized with all neighbors
    run BF update rule
    broadcast distance to neighbors
    increment iteration counter
  if iteration counter = N (finished)
    if (parallel-end node of structure
      (BF-par-end?))
      initialize converge cast
    else
      wait for feedback message(s)
    if (all feedback are ok
        AND locally consistent?)
      send(ack) to parent
    else
      send(fail) to parent
```

Figure B.6: DTP pseudo-code for processor with the flag set to primitive flag set. Pseudo-code for the distributed Bellman-Ford consistency check.

Appendix C

XML format specification of HDSTN files

The TinyRMPL to HDSTN compiler saves the output in XML. The format supports arbitrary numbers of dynamic variables, activity constraints, time events and commands. Every dynamic variable has an associated domain with an arbitrary number of values. The HDSTN file format specification is shown in Figure C.1. The Strategy scenario example compiled to an HDSTN XML file is shown next.

```
HDSTN XML file :=
  <hdstn>
    <domains>
      <domain name>
        <value value-name />
        ...
      </domain>
      ...
    </domains>
    <variables>
      <variable variable-name, time-event id,
        domain, initial?/>
      ...
    </variables>
   <activity_constraints>
        <activity_constraint variable-name,
          value-name, variable-name/>
        ...
    </activity_constraints>
    <nodes>
        <node time-event id,node type,ssi,level >
            <neighbors>
                <neighbor time-event id, level,
                  simple-temporal-constraint-dist,
                    forward? />
                ...
            </neighbors>
        </node>
        ...
    </nodes>
    <commands>
        <command start time-event id,
          end time-event id, command name>
            <parameters>
                <parameter name/>
                ...
            </parameters>
        </command>
    <commands>
  </hdstn>
```

Figure C.1: XML HDSTN file format specification.