MODEL-BASED PLANNING THROUGH CONSTRAINT AND CAUSAL ORDER DECOMPOSITION

by

Seung H. Chung

B.A.Sc., University of Washington (1999)
S.M., Massachusetts Institute of Technology (2003)

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

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2008

© 2008 Massachusetts Institute of Technology
All rights reserved.

Signature of Author ....
Department of Aeronautics and Astronautics
August 21, 2008

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

Certified by ....
Ari K. Jónsson
Dean of School of Computer Science at Reykjavik University, Iceland

Certified by ....
David W. Miller
Professor of Aeronautics and Astronautics

Certified by ....
Nicholas Roy
Assistant Professor of Aeronautics and Astronautics

Accepted by ....
David L. Darmofal
Chairman, Department Committee on Graduate Students

Model-based Planning through Constraint and Causal Order Decomposition

by

Seung H. Chung

Submitted to the Department of Aeronautics and Astronautics
on August 21, 2008 in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

One of the major challenges in autonomous planning and sequencing is the theoretical complexity of planning problems. Even a simple STRIPS planning problem is PSPACE-complete, and depending on the expressivity of the planning problem, the complexity of the problem can be EXPTIME-complete or worse. This thesis improves on current approaches to sequencing the engineering operations of a spacecraft or ground-based asset through the explicit use of verifiable models and a decomposition approach to planning. Based on specifications of system behavior, the planner generates control sequences of engineering operations that achieve mission objectives specified by an operator.

This work is novel in three ways. First, an innovative "divide-and-conquer" approach is used to assure efficiency and scalability of the planner. The key to the approach is in its combined use of *constraint decomposition* and *causal order decomposition*. This technique provides the means to decompose the problem into a set of subproblems and to identify the ordering by which each subproblem should be solved, thus reducing, and possibly eliminating, search. Second, the decomposed planning framework is able to solve complex planning problems with state constraints and temporally extended goals. Such complex system behavior is specified as concurrent, constraint automata (CCA) that provide the expressiveness necessary to model the behavior of the system components and their interactions. The mission objective is described as a desired evolution of goal states called a qualitative state plan (QSP), explicitly capturing the intent of the operators. Finally, the planner generates a partially-ordered plan called a qualitative control plan (QCP) that provides additional execution robustness through temporal flexibility.

We demonstrate the decomposed approach to Model-based planning on a scenario based on the ongoing Autonomous Sciencecraft Experiment, onboard EO-1 spacecraft. The EO-1 problem has a large state space with well over 660 quintillion states, 6.6×10^{20}. Despite the size and the complexity of the problem, the time performance is linear in the length of the plan and the memory usage is linear in the number of components.

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

Acknowledgments

I would like to express my gratitude to all who have supported me in completion of this thesis.

First, to my advisor Brian Williams: You have taught me the importance of precise verbal and written communication. You have given me the opportunity to explore the field of automated reasoning. But more importantly, you have given me the opportunity to experience many aspects of research that most students never get to experience. Over many years, I have gained broader perspective of research. And thank you for your guidance and inspiration.

I thank my thesis committee members, Ari Jónsson, David Miller and Nick Roy for their support throughout the process.

I especially thank Mitch Ingham and Paul Robertson for their support and encouragement. You have given me hope for completion when I was discouraged. I cannot imagine how I could have completed the thesis without your moral support.

I would like to thank Kathryn Fischer and Ed Gazarian for their advice and support. I thank all of my friends from MERS group and Space Systems Laboratory. Especially, Mark, thank you for your prayers.

Foremost, I thank my parents Shin-Kwan and Young-Soon Chung. You are the source of all knowledge I deem most invaluable. You taught me of faith and love. You will remain the greatest teachers of my life, and I will eternally be grateful to you. I also thank my sisters, Hee-Won Chung and Hee-Sun Chung, for their love and friendship.

Thank You for Your grace and blessings.

Contents

1 Introduction **15**

1.1 Motivation for Autonomous Model-based Planning 15

1.2 Objectives 18

1.3 Approach 18

1.3.1 Formal Model Specification 19

1.3.2 Capturing Operator's Intent through Goal-directed Plans 20

1.3.3 Automating Sequence Generation through Model-based Temporal Planning 20

1.4 Innovations 21

1.4.1 Constraint Decomposition 22

1.4.2 Causal Order Decomposition 22

1.4.3 Unifying Constraint & Causal Order Decomposition 23

2 Approach **25**

2.1 Example Problem 26

2.2 Reducing the Size of the Search Space through Decomposition 27

2.3 Planning using the Decomposed Reachability Graph 34

3 Concurrent Constraint Automata **37**

3.1 A Simple System 38

3.1.1 Formal Definition of a Constraint Automaton 38

3.1.2 Execution of a Constraint Automaton 42

3.1.3 Feasible Trajectory of a Constraint Automaton 42

3.2 Concurrent Constraint Automata 43

3.2.1 Formal Definition of CCA 43
3.2.2 Execution of a CCA 45
3.2.3 Feasible Trajectory of a CCA 47
3.3 Related Work 47

4 Planning Problem for CCA **49**
4.1 Planning Problem for CCA 50
4.2 Qualitative Time QSP 50
4.2.1 Qualitative Time in QSP 52
4.2.2 Formal Definition of Qualitative Time QSP 52
4.3 Qualitative Time QCP 57
4.3.1 Partially Ordered Plan 58

5 Planning for CCA & Qualitative Time QSP **63**
5.1 Compiled Bus Controller and Device Example 63
5.2 Solving the Planning Problem 64
5.2.1 Plan Space: A Trellis Diagram 65
5.2.2 Solution within a Trellis Diagram 67
5.3 Decomposed Planning 68
5.3.1 Decomposing the Trellis Diagram 69
5.3.2 Searching through the Decomposed Trellis Diagram 75
5.3.3 Extending to Temporal Planning 81
5.4 Related Work 81
5.4.1 State Constraint 82

6 Conclusion **83**
6.1 Implementation 83
6.2 Results on EO-1 Mission Ops Scenario 83
6.2.1 CCA of EO-1 Mission Ops System 84
6.2.2 Initial State and QSP 86

6.3 Experimental Results on EO-1 Scenario 86
6.4 Approach & Innovation 88
6.4.1 The Importance of Formal Model Specification 90
6.4.2 Capturing Operator's Intent through Goal-directed Plan 90
6.4.3 The Decomposition Approach 91
6.5 Future Work 92
6.5.1 Decision Theoretic Planning with the Decomposed Planner . . . 92
6.5.2 Distributed Planning with the Decomposed Planner 92

List of Figures

2-1 Simple Concurrent Constraint Automata 27
2-2 Standard reachability graph 29
2-3 Decoupled state reachability graph 29
2-4 Bus controller reachability graph with redundant states 30
2-5 Bus controller reachability graph without redundant states 31
2-6 Bus controller reachability graph without redundancies 31
2-7 Device reachability graph with redundant states 32
2-8 Device reachability graph without redundant states 32
2-9 Device reachability graph without redundancies 33
2-10 Rearranged device reachability graph without redundancies 33
2-11 Bus-Device reachability graph without redundancies 33
2-12 Decomposed Bus-Device reachability graph 34
2-13 Reachability graph comparison 34
2-14 Planning problem for reachability graph 35
2-15 Planning problem for decomposed reachability graph 36

4-1 QT-QSP with Point Algebra 51
4-2 QT-QSP with Simple Temporal Constraints 52
4-3 Simple Temporal Constraint of a QSP 55
4-4 Concurrent episodes 57
4-5 Example of a QCP 60
4-6 Example of an elaborated QCP 61

5-1 Compiled CCA of *CompiledBus1Dev1* 64

5-2 Compiled CA of Bus Controller & Device 64
5-3 Trellis Diagram of *CompiledBus1Dev1* 66
5-4 QSP for *CompiledBus1Dev1* 67
5-5 Solution within a Trellis Diagram 68
5-6 Constraint Decomposition of *CompiledBus1Dev1* 72
5-7 Causal Constraint Decomposition of *CompiledBus1Dev1* 73
5-8 Decomposed Trellis Diagram of *CompiledBus1Dev1* 76
5-9 eQCP with Initial State 77
5-10 Closed Goal 78
5-11 Causal Graph 78
5-12 Temporal Dominance 78
5-13 State Trajectory 79
5-14 Subgoal Extraction 79
5-15 eQCP with Subgoals 80
5-16 Example of an elaborated QCP 80
5-17 eQCP Update 81

6-1 EO-1 Scenario CCA 84
6-2 Time vs. Number of Components 88
6-3 Time vs. Number of Components^2 89

List of Tables

6.1 Test Case Parameters 87

6.2 Time Performance 87

6.3 Memory Performance 87

Chapter 1

Introduction

Contents

1.1 Motivation for Autonomous Model-based Planning 15

1.2 Objectives 18

1.3 Approach 18

1.3.1 Formal Model Specification 19

1.3.2 Capturing Operator's Intent through Goal-directed Plans . . . 20

1.3.3 Automating Sequence Generation through Model-based Temporal Planning 20

1.4 Innovations 21

1.4.1 Constraint Decomposition 22

1.4.2 Causal Order Decomposition 22

1.4.3 Unifying Constraint & Causal Order Decomposition 23

1.1 Motivation for Autonomous Model-based Planning

As the operational complexity of NASA's increasingly ambitious missions increases, the capabilities of our current operations processes and tools are becoming increasingly strained. The tremendously successful Mars Exploration Rover (MER) mission has provided important lessons learned in what it takes to efficiently operate a long-duration planetary mission. Among these lessons learned is the need to improve the level of integration across the various planning and sequencing tools employed by the operations team, and reduce the number of overlapping models used by these disparate tools in generating and validating tactical sequences. Furthermore, the sheer number of tools

and ad-hoc scripts, along with many distinct data representations used among them, introduces unnecessary brittleness into the operations planning process.

Operational Efficiency through Automation

Operational efficiency is a driving concern for our ground-based embedded systems, as well. As mentioned before, NASA MER mission is a great example in which operational efficiency was crucial to the successes of the mission. To maximize the science return, the scientists and the engineers had to identify the mission objective, generate command sequence, and verify the safety of the sequence in a less than 24 hour cycle. This was possible in part with the help from an autonomous scheduler called MAPGEN.

Another example is the Deep Space Network (DSN), which is under constant pressure to reduce its operations budget, while continuing to provide high quality-of-service telecommunications support to an increasing number of spacecraft. This dilemma can be at least partially addressed by increasing the level of automation of its Monitor and Control (M&C) functions, thus enabling operators to work more efficiently, taking on supervisory responsibility for multiple concurrent spacecraft tracking activities.

Planning using the Design Specification

The blossoming area of goal-based operations has been identified as providing potential breakthrough capabilities that can directly address these issues. By advancing the state-of-the-art in this field, we can improve on the current approach to sequencing the engineering operations of a spacecraft or ground-based asset, through the explicit use of verifiable models and state-of-the-art goal-directed planning algorithms.

One well accepted problem with NASA's development methodology in flight and ground operations software design and development is that software is developed at the last minute specifically for that mission. The problem is that the time is insufficient for developing safe software and to transfer all necessary knowledge from the engineers to software developers. This creates the possibility for bugs in software and possible inconsistency between hardware design and requirements vs. assumptions on which software is designed.

The ability to leverage existing engineering specifications for direct use in sequencing will reduce the risk of errors in translating the understanding of system behavior into operational sequences, and will mitigate the proliferation of multiple potentially inconsistent models used for different purposes across the system.

Model-based Planning

The aforementioned problems can be reduced if an automated planner is designed to be capable enough to operate based on the specification of the system provided. For estimation and reactive planning, this approach has been shown to be very promising within the framework of Model-based Programming [23]. A reactive planner called Burton [25, 3] has provided a great foundation to model-based planning and great insight and innovation that allows it to plan efficiently enough to be reactive in real-time, while assuring scalability.

The benefit of scalability and efficiency comes with some limitations, however. Firstly, Burton is not able to plan on models that have been written with the full expressivity of the Reactive Model-based Programming Language (RMPL). Some of the information must be compiled away the *state constraints* through a method that can potentially result in an intractably large model. Secondly, Burton cannot plan on *time-evolved goals* described using RMPL. Time-evoloved goals, or also called *temporally extended goals*, specify goals over time. Such expressivity is necessary for command sequence planning and scheduling. While, Burton provides guarantees on it's ability to achieve future goals, it relies on an external sequencer to provide one goal at a time, and real-time feedback of the progress via an estimator. The desire is to extend the foundation set by Burton to a more capable planner that can plan on more complex engineering systems and mission objectives, i.e., temporally extended goals.

The new model-based planning capability has the potential for significant impact on the operations of future space missions and related ground infrastructure (e.g., EO-1, DSN), in the form of improved efficiency for ground-based operations in the near term, and in the form of greater onboard autonomy in the longer term. It will particularly

benefit highly complex MSL-class missions, where our experience on MER points to ground-based planning of engineering operations as a potentially time-consuming and error-prone process. The ability to leverage systems engineering models for direct use in sequencing will reduce the risk of errors in translating the understanding of system behavior into operational sequences, and will mitigate the proliferation of multiple potentially inconsistent models used for different purposes across the system.

1.2 Objectives

We improve on the current approach to sequencing the engineering operations of a spacecraft or ground-based asset through the explicit use of verifiable models and state-of-the-art goal-directed planning algorithms. Our objective is to develop a model-based temporal planner that generates an executable sequence, based on behavior specifications of the components of a system. We leverage lessons learned from current operations of systems like MER and the DSN Monitor & Control (M&C) system, in order to significantly improve the operations efficiency of future JPL missions, reduce costs and increase the likelihood of mission success.

1.3 Approach

We leverage model-based programming formalism to specify the system specification and mission objectives. We use a formal modeling formalism called *Concurrent Constraint Automata* to specify the models of system behavior that will then be directly used by our model-based planner to produce sequences of engineering operations that achieve mission objectives. A mission objective is specified as a *qualitative state plan* (QSP) that is capable of describing temporally extended goals. More specifically, we adopt the following step-by-step approach:

1. Specify the model of system behavior as concurrent constraint automata that provide the expressiveness necessary to model the behavior of the system components, including operational modes with uncertain mode transitions.

2. Describe the mission objective as a desired evolution of goal states, explicitly capturing the intent of the operators, rather than implicitly capturing it in a sequence of commands and procedures that achieve the desired goals.

3. Using a "divide-and-conquer" approach, apply an offline reasoning algorithm to synthesize a set of modular, reusable, and compact partial plans for achieving goal states, thus only requiring a simple composition of partial plans at plan time, minimizing the amount of computationally expensive online search.

We demonstrate our new planning technology and approach by applying it to EO-1 scenarios, and evaluate its benefits in comparison with existing tools and processes.

1.3.1 Formal Model Specification

The conventional approaches to systems and software engineering inadvertently create a fundamental gap between the requirements on software specified by systems engineers and the implementation of these requirements by software engineers. Software engineers must perform the translation of requirements into software code, hoping to accurately capture the systems engineer's understanding of the system behavior, which is not always explicitly specified. This gap opens up the possibility for misinterpretation of the systems engineer's intent, potentially leading to software errors.

Specifying the system and software requirements as a formally verifiable specification called conccurent constraint automata (CCA) allow the system behavior specification to be directly used to automatically generate provably correct sequences. CCA provides the expressiveness necessary to model the behavior of the system components, including various interactions between components. The concurrency and the use of state constraints allow the model-based planner to reason about the system interactions. The model of the uncertain duration allows the model-based temporal planner to account for the execution time uncertainty.

1.3.2 Capturing Operator's Intent through Goal-directed Plans

The ability to explicitly capture the intent of the operators, rather than implicitly capturing it in a sequence of commands and procedures that achieve the desired goals, is crucial for sequence verification. To enable verifiability of automatically generated sequences, the model-based temporal planner assures that each sequence can be traced directly to the mission objective and/or operator's intent. Generally, mission objective and operator intent can be described explicitly as an evolution of goal-states. Thus, for model-based temporal planning, the mission objective and operator's intent is described as a desired evolution of goal states. Then, the model-based temporal planner uses a goal-directed planning method to elaborate each goal state into a sequence, while explicitly associating each sequence to a goal state, that is the intent of the operator and mission objectives.

1.3.3 Automating Sequence Generation through Model-based Temporal Planning

For the given mission objective and operator's intent, the model-based temporal planner automatically plan a robust sequence in two steps:

First, using a "divide-and-conquer" approach, an offline reasoning algorithm precompiles a set of modular, reusable, and compact partial plans for achieving goal states. During this offline step, a model, that is system specification, is first decomposed and partially ordered into a set of decomposed CCA (DCCA) based on the structure of the component interactions, that is dependency graph. Then, a set of partial plans, called decomposed goal-directed plan (DGDP), are generated for each DCCA. Pre-compiling DGDPs offline not only reduces the online planning time, but also provides a means to verify the plans before they are used online.

Finally, during the plan time, the desired mission objective, i.e., sequence of goal-states, are elaborated into an executable sequence by simply composing the appropriate DGDPs. The resulting conditional sequence is guaranteed to achieve the desired goal-states in the most robust manner. Again, the efficiency of the online planning is realized

by pre-compiling the DGDPs offline, and pre-compiling DGDPs for all potential mission objectives is possible due to the use of the "divide-and-conquer" method. With this effective method, the model-based temporal planner can not only be applied to automate the ground operation, but also onboard to provide an onboard automation and reactivity.

1.4 Innovations

Unlike other planning and sequencing systems, our approach directly exploits engineering models of system component behavior to compose the plan, validate its robustness under both nominal and failure situations, and, when required, synthesize novel procedures from first principles. Our planner is complementary to mission activity planners, such as the CASPER planner used in the Autonomous Sciencecraft Experiment on EO-1 and the EUROPA planner used in MAPGEN in MER operation. These activity planners generate a high-level mission plan, which our planner will elaborate into an executable lower-level sequence by reasoning about the model. Our approach is novel in several ways:

1. The "divide-and-conquer" approach that leverages the structure of the component interactions to simplify the planning problem ensures the tractability of planning, even during time-critical situations. This approach is innovative in that it unifies the existing decomposition techniques used in constraint satisfaction problem and reactive planning.

2. The new model-based planner is able to solve a planning problem with state constraints. Addition of state constraints increases the complexity of the problem. The use of constraint and causal order decomposition allows us to solve such complex problems efficiently.

3. The new model-base planner extends the existing model-based reactive planning capability to partially-ordered sequence generation that provides additional execution robustness through temporal flexibility.

To mission operations, the new planner provides the benefits of model-based programming. The planner incorporates the ability to generate modular, reusable, and compact partial plans that can be automatically verified for correct execution. This extends the existing model-based reactive planning capability to sequence generation. Furthermore, the goal-directed and model-based planning approach ensures traceability of the executable plan back to the mission intent and system specification, increasing the reliability and reviewability of the automatically generated plan.

1.4.1 Constraint Decomposition

The main technical innovation is in its combined use of constraint decomposition and causal order decomposition. Constraint decomposition has been widely used in solving constraint satisfaction problems (CSP). Constraint decomposition takes advantage of the structure of the constraints to minimize search. If a CSP can be decomposed into a tree of subproblems, the decomposed problem can be solved in polynomial time with respect to the size of the tree. Solving a decomposed problem is only exponential in the size of the subproblems, known as the *tree width*. Thus, if a CSP can be decomposed into small subproblems, we can solve the CSP quite efficiently. Framing a planning problem as a CSP, and taking advantage of CSP solving techniques is quite common and has been quite successful . Such methods, however, requires one to guess the number of steps required to solve the planning problem and encode the n-step model of the planning problem as a CSP. The size of such a CSP can be quite large depending on the planning problem. Furthermore, such methods do not easily extend to more complex goals, such as temporally extended goals.

1.4.2 Causal Order Decomposition

To address the issues that are specific to planning problems, specialized planning algorithms have been developed. As in all general search problems, the efficiency of planning depends on how well one can identify which path is most promising. Because of the large search space of a planning problem, many techniques resort to heuristic-based search methods. Many of the heuristics attempt to relax the original planning problem into

an easier problem, whose solution can be used to guide the search. Another common method is to simplify the search space as a disjunctive set of reachable state spaces, which then can be searched more efficiently for the solution.

Another method is the causal order decomposition technique . Unlike most heuristic methods that try to estimate how likely it is for a search path to lead to the solution, A causal order decomposition determines how a single planning problem can be decomposed into a set of subproblem. When a problem is decomposed, the ordering in which each subproblem is solved becomes crucial. As many readers may have experienced from solving puzzles, depending on the ordering in which each piece of a puzzle is solved, one may quickly find the solution or one may run in circles trying to repair what was solved before. Causal ordering decomposition uses the structure of the problem, that is, the cause and effect relationship among concurrent automata, to determine the proper decomposition as well as the ordering. As shown by Burton, if a problem is decomposable into a directed acyclic graph, the planning problem can be solved without search.

1.4.3 Unifying Constraint & Causal Order Decomposition

Intuitively, causal order decomposition provides a means to identify the ordering in which one should evolve the state over time to quickly come to the solution. In contrast, constraint decomposition itself is indifferent to time evolution of state. In fact, the size of the decomposition of a planning problem encoded as a CSP grows linearly with the number of planning time steps encoded. This is expected since the relationship between variables do not change with the increase in the number of time steps encoded as a CSP. Thus, for planning, constraint decomposition does not provide any benefit in guiding the search over time space.

These two techniques are, however, are synergistic. We can use constraint decomposition to efficiently search the state-space defined by the state constraints of a planning problem and use the causal order decomposition to efficiently search the time-space defined by state transitions.

Chapter 2

Approach

Contents

2.1 Example Problem 26

2.2 Reducing the Size of the Search Space through Decomposition 27

2.3 Planning using the Decomposed Reachability Graph 34

One of the major challenges in autonomous planning and sequencing is the theoretical complexity of planning problems. Even a simple STRIPS Planning problem is PSPACE-complete [2], and depending on the expressivity of the planning problem, the complexity of the problem can become even worse [8]. As an example, [21] has shown that allowing derived predicates, that is, indirect effect or state constraints, increases the theoretical complexity of a planning problem to EXPTIME-complete.

Using a traditional approach, planning for missions with multitude of complex spacecraft and ground assets with hundreds and thousands of components will most certainly be intractable. Though many state-of-the-art techniques, with their restricted domain assumptions, enable planners to find solutions remarkably fast, in many cases, it is not clear if those techniques can easily be extended to more complex domain such as spacecraft and ground systems. Moreover, while incredible advancements in automated planning owe much to new heuristic and stochastic search methods, such methods are not well received for many space missions that require planning and execution time guarantees.

This chapter introduces a new approach. Instead of relying on the heuristics and chances (i.e. stochasticity), the new approach takes advantage of the nature of the problem, that is, a structure inherent to the problem. Not all problems are tightly

coupled. Many large problems can be decomposed into a set of subproblems with weak or no interactions. If such decomposition is known, the tractability of the problem can be predetermined, with tighter bounds on the time requirement for planning and execution.

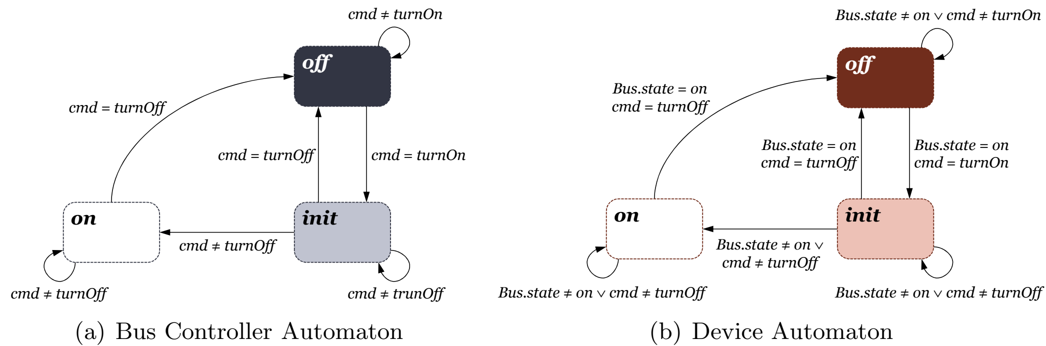
In summary, the idea is to exploit decomposition to represent the space of possible trajectories compactly, and to generate a feasible or optimal set of trajectories within the decomposed space. The decomposition is achieved by using a *constraint graph decomposition* technique, and searching through the decomposed space is achieved by utilizing the *causal ordering* of the decomposition. The main innovation of the approach is in the infusion of constraint graph decomposition [6, 12, 4] and causal graph decomposition [25, 3] techniques.

In this chapter, an intuitive example of problem decomposition and its benefit is introduced, along with an illustration of a plan for the decomposed problem.

2.1 Example Problem

For a planning problem, the space of all possible trajectories can be graphically represented as a reachability graph. For a planner that uses forward search, a reachability graph rooted at the initial state represents the search space of the problem. In this section, we will illustrate how this space can be decomposed, and thus reduced, by taking advantage of the structure of the problem.

For this illustration, consider a simple system composed of two components, a bus controller and a device. The system is modeled as a Concurrent Constraint Automata (CCA) as described in Chapter 3. The CCA of the system is depicted in Figure 2-1. The device (Figure 2-1(b)) has three states, $\{off, initializing, on\}$. The device can be turned on by issuing the device command $devCmd = turnOn$ and turned off with $devCmd = turnOff$. Note that the device command $devCmd = turnOn$ indirectly turns on the device by first initializing it. Once the device is initializing, it may be commanded to be turned off. Otherwise, it will automatically transition into the *on* state. For any command to be routed to the device, the bus be *on*, thus the precondition $bus = on$



(a) Bus Controller Automaton (b) Device Automaton

Figure 2-1: 2-1(a) represents a simple bus controller automaton and 2-1(b) represents a simple device automaton, e.g. valve-driver.

on all commanded transitions. The bus controller, like the device, has three states, $\{off, initializing, on\}$, with the same set of commands $\{turnOn, turnOff\}$. Unlike the device, the transitions of the bus controller is not dependent on the state of any other components.

Problem 2.1 Given the bus controller and device system in Figure 2-1 and the initial state $(bus = off, dev = off)$, find a trajectory to the goal state $(bus = off, dev = on)$.

2.2 Reducing the Size of the Search Space through Decomposition

The size of the search space can be reduced through decomposition. In a decomposed reachability graph, a global state is decoupled into a set of individual or partial state assignments and sequence of transitions are parallelized into a set of concurrent transitions such that the size of the reachability graph is reduced. The feasibility of the decomposition depends on the property of the problem. For example, the two components, the bus controller and the device, are only weekly coupled, that is, via the transition constraints of the device. This property of the problem allows the reachability graph to be decomposed into two respective reachability graphs that are also weakly coupled.

Next, the notion of a decomposed reachability graph is described through a straightforward, step-by-step reduction of the reachability graph. While this is not the proposed

decomposition method, this example will provide the intuition behind what a decomposed reachability graph is, why a reachability graph may be decomposable, and how the decomposition reduces the search space.

Standard Reachability Graph

Figure 2-2 illustrates the reachability graph for the given Problem 2.1. Each outer circles represent a state, which is an assignment to a state vector. The each pair of circumscribed inner circles represents an assignment to a state vector. A blue inner circle represents state *bus* variable, that is, bus controller, and a maroon circle represents *dev* variable. The shades dark, light and white respectively represent the values $\{off, initializing, on\}$ of the state variables. For example, the left most outer circle, that is, the root of the reachability graph, represents the state $(bus = off, dev = off)$.

A dotted arrow represents a no-op transition, or equivalently no transition. The arrows that connect two subsequent states (from a state on the "left" with an outgoing arrow to a state to the "right" with an incoming arrow) indicate that the a state (right) is reachable from another state (left) through some state transition (depicted as a box). Thus, a set of subsequent states to the right represent a set of reachable states.

The reachability graph in Figure 2-2 can be mapped into an equivalent representation in which a state (outer circle) is decoupled into a set of state variable assignments (inner circles) and state transitions are transformed into a set of concurrent state variable transitions, or primitive transitions. A primitive transition explicitly specifies what partial state assignments are changed through the transition and what partial state assignments are required for the transition to occur. This representation is shown in Figure 2-3, where blue arrows correspond to primitive transitions that change the bus controller state and red arrows correspond to primitive transitions that change the device state. Main objective in transformation to this representation is to decompose the graph based on individual state variable transitions instead of global state transitions.

This reachability graph highlights the dependence between state variables: A red transition, a device state transition, may depend (preconditions) on both variables red,

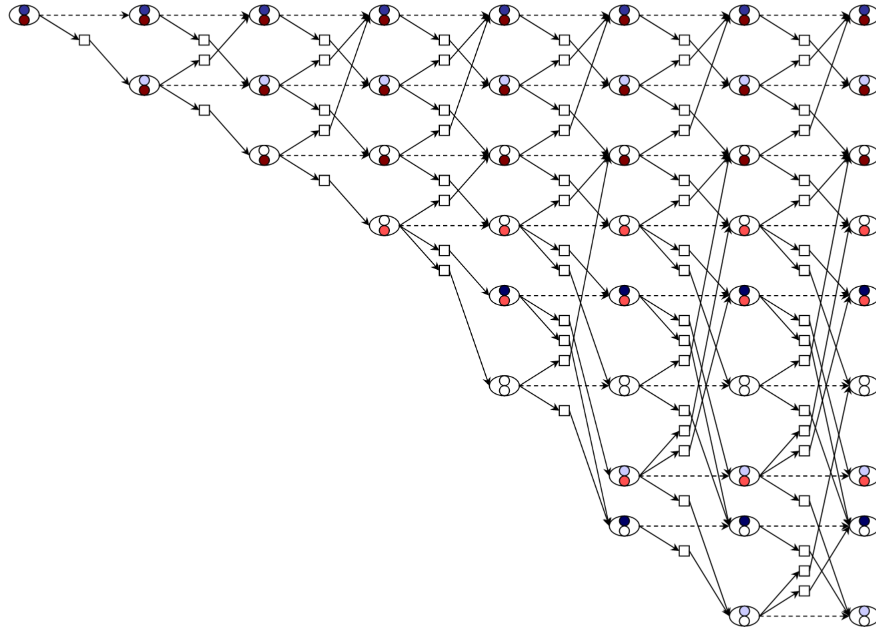


Figure 2-2: Standard reachability graph.

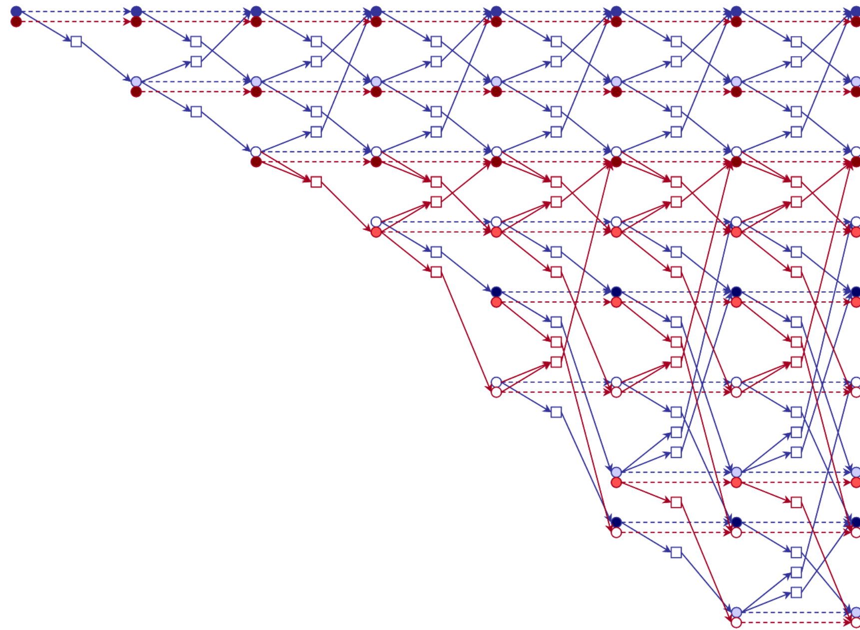


Figure 2-3: Reachability graph with individual component states decoupled.

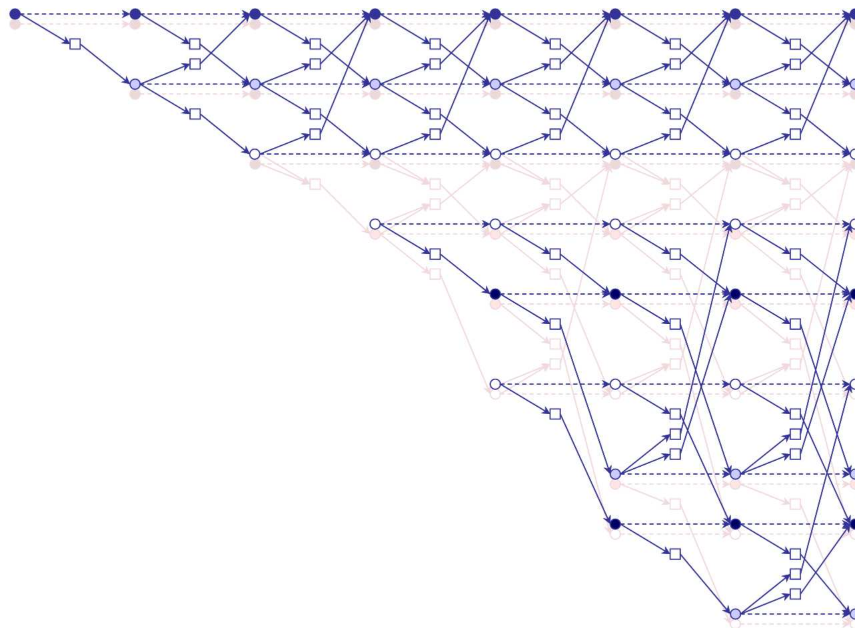


Figure 2-4: Bus controller reachability graph with redundant states.

the device state, and blue, the bus controller state. For example, the transition from ($bus = on, dev = on$), row 6, to ($buson$), row 6, to ($bus = on, dev = off$), row 3, depends on the state of and the device, that is, red and blue variables. All blue, that is, bus controller, transitions, however, are independent of red variables assignments, that is, does not precondition on the state of the device.

Bus Controller State Reachability Graph

Since all bus controller state transitions are independent of the device state, bus controller state reachability can be analyzed independently from the device state reachability. Removing the device information from Figure 2-3 result in a bus controller reachability graph shown in Figure 2-4.

Note that the same state variable assignments, bus controller states, are repeated multiple times on the graph, e.g. state assignment rows 3, 4, and 6. We can collapse the same assignments and still represent the same reachability graph. Figure 2-5 is the result of collapsing the same assignments.

As a result, it is easier to see that for a given transition column, there are identical

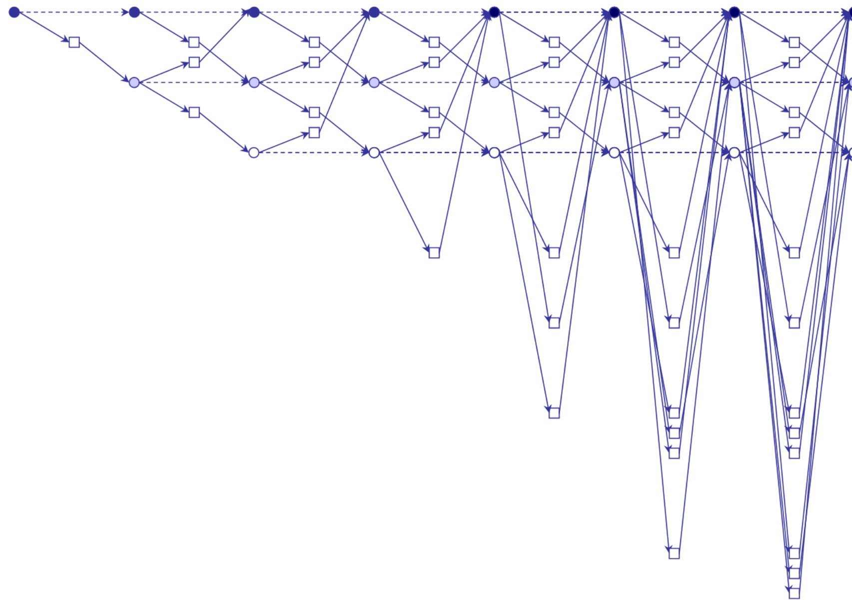


Figure 2-5: Bus controller reachability graph without redundant states.

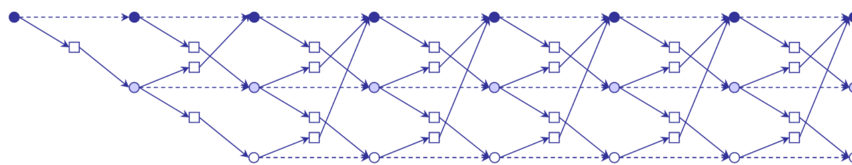


Figure 2-6: Bus controller reachability graph without redundant transitions.

transitions that can also be collapsed. Note that two transitions are identical if the precondition (previous state and command) and effects (next state) are identical. For example, in transition column 4, transition rows 4 and 5 are identical. With both redundant state assignments and transitions removed, the reachability graph for the bus controller becomes much more compact and simplified as shown in Figure 2-6. While the reader will notice the redundancy in the columns of state-transition-state triple, that is, the reachability graph levels out after three steps, this redundancy will be removed after analyzing the reachability graph of the device.

Device State Reachability Graph

The same reduction method can be applied to the portion of the reachability graph the pertains to the device (red). Unlike the blue transitions, however, the red transitions depends on the state of the blue variable (see Figure 2-7, e.g. transition column 3 row

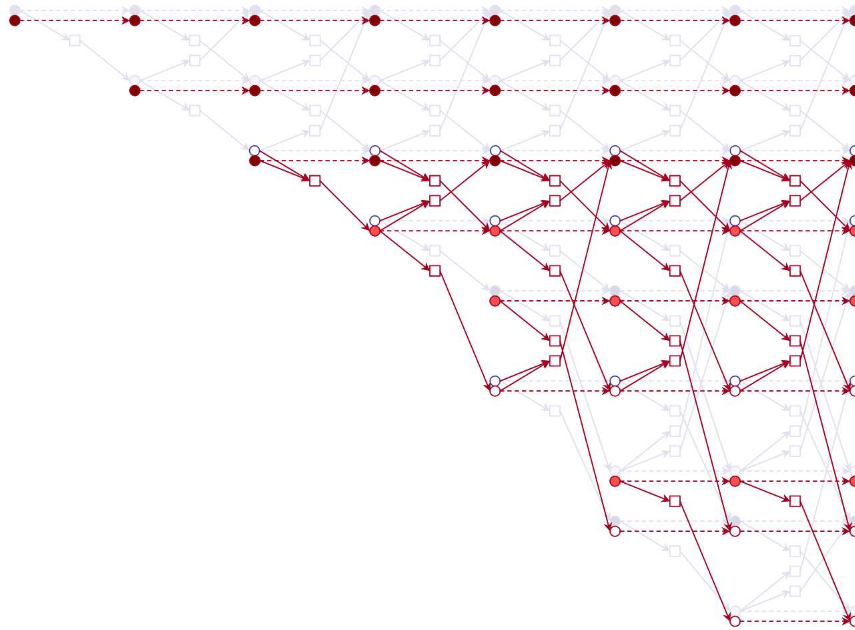


Figure 2-7: Device reachability graph with redundant states.

4.

Using the same procedure as before, all identical state assignments are collapsed into one as show in Figure 2-8.

Now, all identical red transitions can be collapsed as shown in Figure 2-9. For example, in transition column 6, transition rows 4 and 6 can be collapsed into one.

For clarity, Figure 2-9 can be rearranged as shown in Figure 2-10.

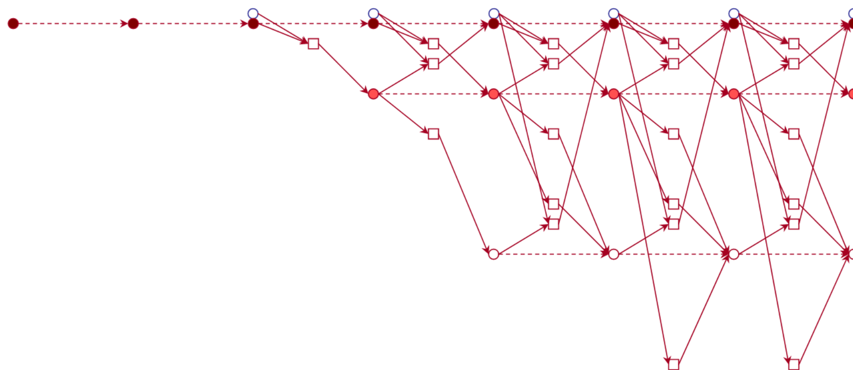


Figure 2-8: Device reachability graph without redundant states.

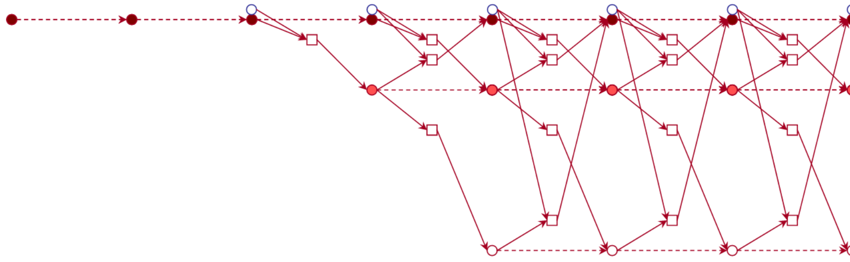


Figure 2-9: Device reachability graph without redundant transitions.

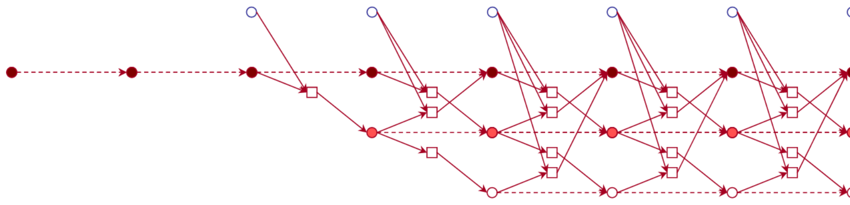


Figure 2-10: Device reachability graph without redundant states or transitions.

Decomposed Reachability Graph

The two reduced reachability graph of the bus controller and the device can be recombined into a single graph as shown in Figure 2-6. Combining the two into a single graph results in coupled concurrent reachability graph, or a decomposed reachability graph.

The decomposed reachability can be further reduced by recognizing the fact the graph levels off after the fifth level. Figure 2-12 is the result of removing the remaining redundant levels.

Figure 2-13 compares the original reachability graph (Figure 2-13(a)) to the decomposed reachability reachability graph (Figure 2-13(b)). Note the size reduction of the graph. The decomposition possess two important attributes. First, the decomposition

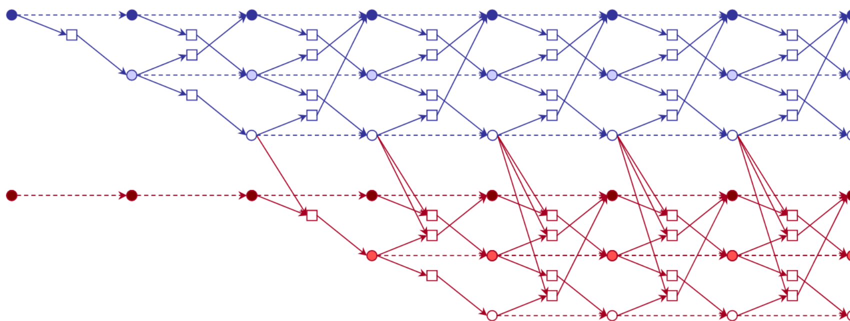


Figure 2-11: Bus controller-Device reachability graph without redundancies.

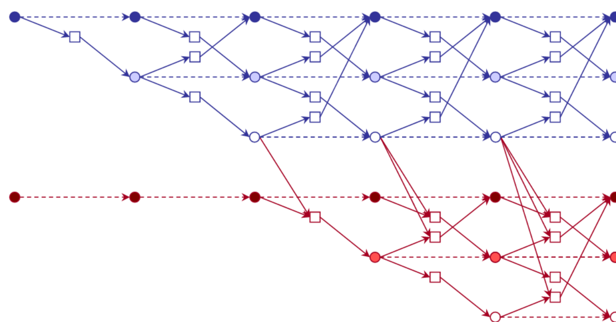
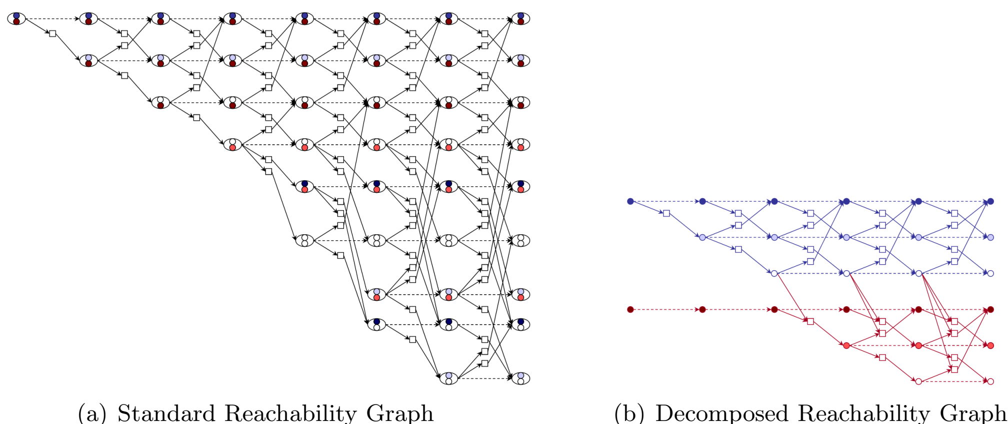


Figure 2-12: Decomposed bus controller-Device reachability graph.



(a) Standard Reachability Graph (b) Decomposed Reachability Graph

Figure 2-13: Comparison of standard and decomposed reachability graphs.

inhibits state space explosion by decoupling a global state into a set of individual or partial state variable assignments. Thus, not all possible combinations of global states need to be enumerated. Two, the decomposition allows the reachability of each individual or partial state variable assignments to be analyzed concurrently. Thus, not all possible sequences of global state transitions need to be enumerated.

2.3 Planning using the Decomposed Reachability Graph

While the search space of the decomposed reachability graph can be much smaller in comparison to the original reachability graph, in contrast, searching for a feasible plan, a feasible trajectory, within the decomposed reachability graph can be much more complex. Assuming that the goal state is indeed reachable, planning based on the original reachability graph simply requires searching a path from the node that represents the

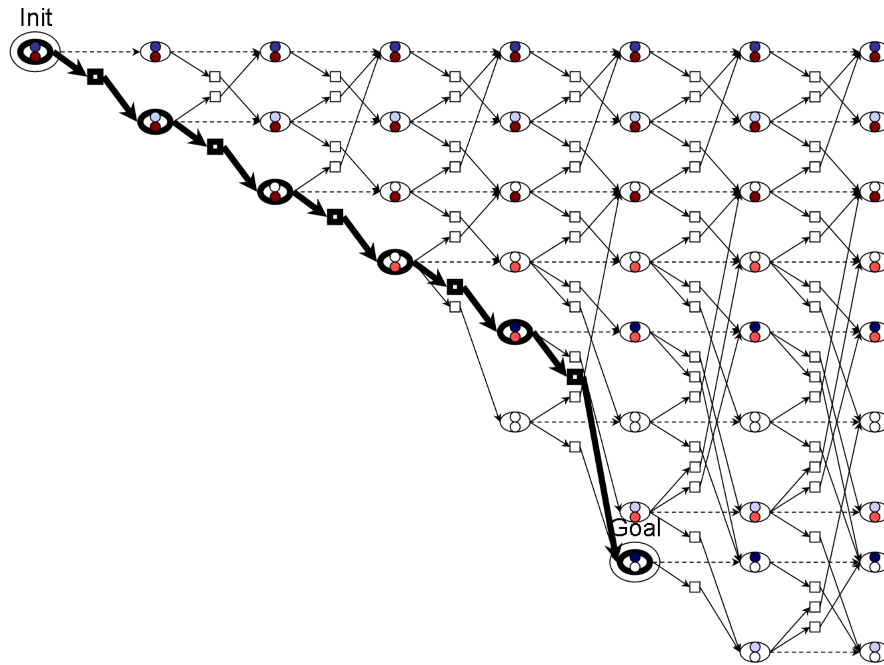


Figure 2-14: Planning problem for standard reachability graph.

initial state to the node that represents the goal state. For Problem 2.1, a feasible plan from the initial state of $(bus = \mathit{off}, dev = \mathit{off})$ to the goal state of $(bus = \mathit{off}, dev = on)$ is shown within the original reachability graph in Figure 2-14.

Planning using a decomposed reachability graph, however, requires searching for a consistent set of trajectories. Because a state is decomposed into a set of individual or partial state assignments, the initial and goal states must also be decomposed into a set of individual or partial state assignments. Then, a feasible plan is a consistent set of trajectories from the set of initials state assignments to the set of goal state assignment. For example, for Problem 2.1, the initial state is decomposed into $bus = \mathit{off}$ and $dev = \mathit{off}$. Similarly, the goal state is decomposed into $bus = \mathit{off}$ and $dev = on$. Then, a feasible plan is a consistent set of trajectories of $bus = \mathit{off}$ to $bus = \mathit{off}$ and $dev = \mathit{off}$ to $dev = on$. Such feasible plan is shown in Figure 2-15.

While the search space may have been reduced from a single large graph to inter-connected set of smaller graphs, searching for a feasible plan has become more complex. Instead of searching for a path within a single graph, now a path must be found within

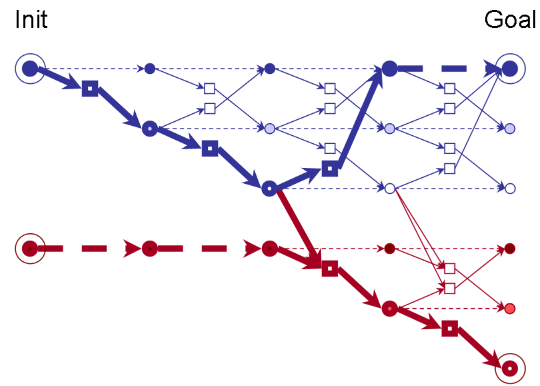


Figure 2-15: Planning problem for decomposed reachability graph.

each of the smaller graphs while assuring that the paths chosen are consistent with one another.

Chapter 3

Concurrent Constraint Automata

Contents

3.1 A Simple System . . . 38
3.1.1 Formal Definition of a Constraint Automaton . . . 38
3.1.2 Execution of a Constraint Automaton . . . 42
3.1.3 Feasible Trajectory of a Constraint Automaton . . . 42
3.2 Concurrent Constraint Automata . . . 43
3.2.1 Formal Definition of CCA . . . 43
3.2.2 Execution of a CCA . . . 45
3.2.3 Feasible Trajectory of a CCA . . . 47
3.3 Related Work . . . 47

A central idea in the model-based programming paradigm is the notion of an *executable specification* [23]. In an executable specification, the system behavioral description is used directly for planning. Thus, the conceptual description of the system behavior must be written in, or automatically mapped to, some form of model on which deductive algorithms can operate. Furthermore, the computational model must be capable of representing complex behaviors of a system, while facilitating computationally tractable planning.

Within the model-based execution framework, the behavior of the system being controlled is modeled as concurrent partially observable Markov decision processes (POMDPs) that is compactly encoded as a probabilistic *concurrent constraint automaton* (CCA) [24]. Concurrency is used to model the behavior of a set of components that operate synchronously. Constraints are used to represent co-temporal interactions and intercommunication between components. Probabilistic transitions are used to model the stochastic behavior of components, such as failure.

In this chapter, a formal description of the CCA computational model is introduced. CCA will be described in two parts. First, a constraint automaton for a single component is first defined formally. Then, a CCA is defined as a set of such constraint automata. These definitions are similar to the definition of a CCA in [24, 25, 23].

3.1 A Simple System

A CCA represents a set of concurrently operating constraint automata.

3.1.1 Formal Definition of a Constraint Automaton

A constraint automaton is a finite state transition system with constraints that specify the behavior of the system. The notation $C(X)$ is used to denote a set of all possible constraints over variables X. Each automaton has an associated *state variable* x^q with domain $\mathcal{D}(x^q) = \{v_1, \ldots, v_m\}$. Each state x^q is associated with state constraints involving its attributes, that is inputs $\Sigma(X^u)$ and outputs $\Sigma(X^y)$ used to define its requirements and output behaviors. The notation $\Sigma(X)$ is used to denote the set of all possible full assignments to variables X. If the automaton in state $q_i \equiv (x^q = v_i)$, its *state constraint* $\mathcal{Q}(q_i)$ must be satisfied. Given the current state q_i, an automaton transitions its state in the next time step, according to its *transition function* δ. A transition function is conditioned on a *gaurad constraint* that must be satisfied for the transition to occur.

Before formal defining constraint automaton, a finite domain constraint is first defined. A finite domain constraint is a constraint on finite domain variables. Formally:

Definition 3.1 (Finite Domain Constraint) Given a set of finite domain variables X, a *finite domain constraint* $\varphi \in C(X)$ over variables X is a sentence in a propositional state logic, that is:

$$\varphi ::= \texttt{true} \mid \texttt{false} \mid (x = v) \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2, \tag{3.1}$$

where $x \in X$ and $v \in \mathcal{D}(x)$. Variables X and sentence φ respectively represent the

scope and the relation of the constraint.

Definition 3.2 (Full Instantiation) A set of *full instantiations* $\Sigma(X) = \prod_{x \in X} \mathcal{D}(x)$ is used to denote a complete set of all possible assignments to variables X.

A constraint automaton is then defined as follows:

Definition 3.3 (Constraint Automaton) A *constraint automaton*[1] $\mathcal{A}$ is a 4-tuple $\langle X, D, \mathcal{Q}, \delta \rangle$, where

- $X = \{x_1, \ldots, x_n\}$ is a finite set of *variables* of the automaton, partitioned into a set of *state variables* X^q, a set of *input variables* X^u and a set of *output variables* X^y
- $D = \{\mathcal{D}(x_1), \ldots, \mathcal{D}(x_n)\}$ is a set of finite *domains* of X,
- $\mathcal{Q} : Q \to C(X^u \cup X^y)$ is a *state constraint* function,
- $\delta : Q \times \Sigma(X^u) \times C(X^u) \to 2^Q$ is a *transition* function.

Variables

$X = X^q \cup X^u \cup X^y$ is a finite set of variables of an automaton, partitioned into a set of *state variables* $X^q \subseteq X$, a set of *input variables* $X^u \subseteq X$ and a set of *output variables* $X^y \subseteq X$, such that none of the sets overlap with one another, that is, $X^q \cap X^u = \emptyset$, $X^q \cap X^y = \emptyset$ and $X^u \cap X^y = \emptyset$. The set of state variables $X^q = \{x\}$ is a singleton set containing the *state variable* of the automaton, denoted x^q. The behavior of the automaton depends on the values of the input variables X^u. Alternatively, the value of the output variables X^y depend on the state of the automaton.

[1]As described in [23], a constraint automaton also includes a transition probability function, an observation probability function and a reward function. These are omitted here for clarity of the discussion, as the scope of the discussion is limited to deterministic planning. For decision theoretic planning, transition probability, observation probability, and reward functions should be included in the description of a constraint automaton.

Domains

D is a finite set of domains of variables X. The domain of the state variable x^q, $\mathcal{D}(x^q) = \{v_1, \ldots, v_m\}$, directly corresponds to the finite set of discrete states $Q = \{q_1, \ldots, q_m\}$ of the automaton, where state $q_i \in Q$ is equivalent to the assignment of the value $v_i \in \mathcal{D}(x^q)$ to the state variable x^q, that is, $q_i \equiv (x^q = v_i)$. A finite set $\Sigma(X^u) = \prod_{x_j \in X^u} \mathcal{D}(x_j)$ is the set of all possible full instantiation of the input variables X^u. A full instantiation $u \in \Sigma(X^u)$ is a set of assignments to the input variables, $u = \{(x_1 = v), \ldots, (x_l = v)\}$, where variable $x_i \in X^u$ is assigned a value $v \in \mathcal{D}(x_i)$ and $l = |X^u|$. A finite set $\Sigma(X^y) = \prod_{x_j \in X^y} \mathcal{D}(x_j)$ is a set of all possible full instantiations of the output variables X^y. A full instantiation $y \in \Sigma(X^y)$ is a set of assignments to the output variables, $y = ((x_1 = v), \ldots, (x_l = v))$, where variable $x_i \in X^y$ is assigned a value $v \in \mathcal{D}(x_i)$ and $l = |X^y|$.

State Constraint

The state constraint function $\mathcal{Q}$ associates each state $q \in Q$ with a finite domain constraint $\mathcal{Q}(q) \in C(X^u \cup X^y)$. The constraint set $C(X^u \cup X^y)$ denotes the set of all finite-domain constraints over input variables X^u and output variables X^y. The state constraint function specifies that if the automaton is in state $q \in Q$, its state constraint $\mathcal{Q}(q)$ must be satisfied by input $u \in \Sigma(X^u)$ and outputs $y \in \Sigma(X^y)$ of the automaton, that is, $q \cup u \cup y$ must satisfy $q \Rightarrow \mathcal{Q}(q)$[2]. In effect, a state constraint specifies a set of feasible outputs given a state and an input. Given state $q \in Q$ and input $u \in \Sigma(X^u)$, if there exists exactly one output $y \in \Sigma(X^y)$ that satisfies state constraint $\mathcal{Q}(q)$, the output is deterministic. Accordingly, if more than one output can satisfy the state constraint, then the output is nondeterministic. Nondeterministic output is typically associated with a faulty state, but not always. If the output is deterministic for every state of the automaton, the state constraint function of the automaton is said to be

[2] In general, an input may be a partial instantiation of the input variables, $u \in \tilde{\Sigma}(X^u)$. Without loss of generality, the definition specifies an input $u \in \Sigma(X^u)$ as a full instantiation of the input variables. A partial instantiation can be represented as a full instantiation by requiring a value of *unknown* to be included in the domain of each input variable, that is, *unknown* $\in \mathcal{D}(x)$ for $x \in X^u$.

deterministic.

Definition 3.4 (Deterministic State Constraint Function) Given constraint automaton $\mathcal{A} = \langle X, D, \mathcal{Q}, \delta \rangle$, the state constraint function $\mathcal{Q}$ is *deterministic*, if and only if, for each state $q \in Q$ and input $u \in \Sigma(X^u)$, there exists exactly one output $y \in \Sigma(X^y)$ that satisfies the state constraint $\mathcal{Q}(q)$.

Transition Function

In the transition function $\delta : Q \times \Sigma(X^u) \times C(X^u) \to 2^Q$, $C(X^u)$ is the set of all finite domain constraints over input variables X^u. Within the context of the transition function δ, a constraint $\varphi \in C(X^u)$ is called a *guard constraint*, also known as a transition guard. Given state $q \in Q$, an input $u \in \Sigma(X^u)$ and a guard constraint $\varphi \in C(X^u)$, where φ is satisfied by input u, the transition $\delta(q, u, \varphi)$ specifies a set of possible states that the automaton can transition to at the next time step. Under this condition, transition $\delta(q, u, \varphi)$ is said to be *enabled*. The transition function captures both nominal and faulty behavior, represented by $\delta^n \subseteq \delta$ and $\delta^f \subseteq \delta$, respectively, where $\delta^n \cap \delta^f = \emptyset$. In the absence of a faulty behavior, the nominal transition function is always deterministic, that is, $\delta^n : Q \times \Sigma(X^u) \times C(X^u) \to Q$. The fault transitions introduce nondeterminism into the system. Therefore, if and only if $\delta^f = \emptyset$ and state constraint $\mathcal{Q}(q)$ is restricted to be deterministic, then the constraint automaton is deterministic.

Deterministic Constraint Automaton

In this thesis, we will focus on *deterministic constraint automaton*.

Definition 3.5 (Deterministic Constraint Automaton) Constraint automaton $\mathcal{A} = \langle X, D, \mathcal{Q}, \delta \rangle$ is deterministic, if and only if state constraint function is deterministic (see Def. 3.4) and transition function δ is deterministic, that is $\delta : Q \times \Sigma(X^u) \times C(X^u) \to Q$.

3.1.2 Execution of a Constraint Automaton

Definition 3.6 (Execution of a Constraint Automaton) Given constraint automaton $\mathcal{A} = \langle X, D, \mathcal{Q}, \delta \rangle$ in state $q \in Q$, the *execution* of input $u \in \Sigma(X^u)$ is $\text{exec}(\mathcal{A}, q, u) \in \delta(q, u, \varphi)$, where u satisfies transition constraint $\varphi \in C(X^u)$ and a feasible next state is nondeterministically chosen from the transition function δ. The notion of execution of an input is extended to execution of input sequence $u(0:n) = (u(0), \ldots, u(n))$, where

$$\text{exec}(\mathcal{A}, q, u(0:n)) = \begin{cases} \text{exec}(\mathcal{A}, q, u(0)), & \text{if } n = 0 \\ \text{exec}(\mathcal{A}, \text{exec}(\mathcal{A}, q, (u(0), \ldots, u(n-1))), u(n)), & \text{if } n > 0 \\ \text{undefined} & \text{otherwise} \end{cases} \tag{3.2}$$

In general, an execution of a constraint automaton is nondeterministic. However, if a constraint automaton is deterministic (Def. 3.5), then the execution is also deterministic.

3.1.3 Feasible Trajectory of a Constraint Automaton

Let $u(0:n) = (u(0), u(1), \ldots, u(n))$ be a finite input sequence and $q(0)$ be the initial state of constraint automaton $\mathcal{A}$. Then, $q(0:n+1) = (q(0), q(1), \ldots, q(n+1))$ is a *feasible trajectory* if $q(0:n+1)$ is one of possible trajectories that results from executing $u(0:n)$ on $\mathcal{A}$. If constraint automaton $\mathcal{A}$ is deterministic, then there exists exactly one feasible trajectory $q(0:n+1)$. Formally:

Definition 3.7 (Feasible Trajectory of a CA) Given constraint automaton $\mathcal{A} = \langle X, D, \mathcal{Q}, \delta \rangle$ and input sequence $u(0:n) = (u(0), \ldots, u(n))$, a finite sequence $q(0:n+1) = (q(0), \ldots, q(n+1))$ is a *feasible trajectory*, if and only if,

- $q(0) \in Q$ is the initial state of $\mathcal{A}$,
- $q(t+1) = \text{exec}(\mathcal{A}, q(t), u(t))$, for $0 < t \leq n$.

3.2 Concurrent Constraint Automata

Concurrent constraint automata (CCA) is a set of concurrently operating automata. Within this formalism, all automata are assumed to operate synchronously, that is, at each time step every component performs a single state transition. In this section, CCA and its *legal execution* are formally defined.

3.2.1 Formal Definition of CCA

A system, is modeled as a composition of concurrently operating constraint automata $\mathcal{A}$ that represent the system's individual components or processes. This composition, including interconnections between component automata and interconnections with the environment, is captured by a CCA. Formally, a CCA is defined as follows:

Definition 3.8 (Concurrent Constraint Automaton) A concurrent constraint automaton $\mathcal{M}$ is a 3-tuple $\langle \mathcal{A}, X, \mathcal{I} \rangle$, where:

- $\mathcal{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ is a finite set of *constraint automata*,
- $X = \{X_1, \ldots, X_m\}$ is a finite set of *system variables*,
- $\mathcal{I} \in C(X')$ is a finite domain constraint, called an *interconnection constraint*, where $X' = \bigcup_{i=1\ldots n} X_i^u \cup X_i^y$, in which X_i^u and X_i^y are the input and output variables of constraint automaton $\mathcal{A}_i \in \mathcal{A}$, respectively.

Variables

A finite set of variables $X = \bigcup_{i=1\ldots n} X_i$ of CCA $\mathcal{M}$ is composed of all variables X_i of each constraint automaton $\mathcal{A}_i \in \mathcal{A}$. Similarly, the finite set of domains $D = \bigcup_{i=1\ldots n} D_i$ of $\mathcal{M}$ is composed of the set of domains D_i of each constraint automaton $\mathcal{A}_i \in \mathcal{A}$ of $\mathcal{M}$.

$X = X^s \cup X^c \cup X^o \cup X^d$ is partitioned into a set of *state* variables $X^s \subseteq X$, a set of *control* variables $X^c \subseteq X$, a set of *observable* variables $X^o \subseteq X$ and a set of *dependent* variables $X^d \subseteq X$, such that none of the sets overlap with one another. State variables

represent the state of each component. Actuator commands are relayed to the system by assigning the desired values to control variables. Observable variables capture the information provided by the system's sensors. Finally, dependent variables represent interconnections between components. They are used to transmit the effects of control actions and observations throughout the system model.

State Variables The set of state variables $X^s = \bigcup_{i=1\ldots n} X_i^q$ of CCA $\mathcal{M}$ is a set composed of the state variables of each constraint automaton $\mathcal{A}_i \in \mathcal{A}$. The state space of a CCA, $S = \prod_{i=1\ldots n} Q_i$, is the Cartesian product of the state spaces Q_i of the individual constraint automaton $\mathcal{A}_i \in \mathcal{A}$, for all constraint automata $\mathcal{A}$. Note that the state space Q represents the set of all possible full assignments to the state variables X^s. A full assignment $s \in S$ represents a state of the system the CCA represents.

Control Variables The set of control variables $X^c \subseteq \bigcup_{i=1\ldots n} X_i^u$ of a $\mathcal{M}$ is a subset of all input variables of each constraint automaton $\mathcal{A}_i \in \mathcal{A}$. A finite set $\Sigma(X^c) = \prod_{x \in X^c} \mathcal{D}(x)$ is a set of all possible full assignments over the control variables. A full assignment $\mu \in \Sigma(X^c)$ represents an instance of a control input.

Observable Variables Similarly, the set of observable variables $X^o \subseteq \bigcup_{i=1\ldots n} X_i^y$ of a $\mathcal{M}$ is a subset of all output variables of each constraint automaton $\mathcal{A}_i \in \mathcal{A}$. A finite set $\Sigma(X^o) = \prod_{x \in X^o} \mathcal{D}(x)$ is a set of all possible full assignments over the observable variables. A full assignment $o \in \Sigma(X^o)$ represents an instance of an observation.

Dependent Variables The set of dependent variables $X^d = X - X^s - X^c - X^o$ are all remaining variables. A finite set $\Sigma(X^d) = \prod_{x \in X^d} \mathcal{D}(x)$ is a set of all possible full assignments over the dependent variables. A full assignment $d \in \Sigma(X^d)$ represents an instance of a valuation to the dependent variables.

Interconnection Constraint

A finite domain constraint $\mathcal{I} \in C(X')$, where $X' = \bigcup_{i=1\ldots n} X_i^u \cup X_i^y$, provides a means to describe the interconnections between the outputs, $\prod_{i=1\ldots n} \Sigma(X_i^y)$, and the inputs,

$\prod_{i=1\dots n}\Sigma(X_i^u)$, of the constraint automata. For a CCA to be *proper*, all inputs of the automata must be defined. That is, an input must either be a control input or be connected to the outputs of the constraint automata. Thus to assure that CCA is proper, all inputs that are not control inputs, i.e. $X^u - X^c$, are required to be connected to the outputs X^y through the interconnection constraint $\mathcal{I}$. Similar to a state constraint function of a constraint automaton, an interconnection constraint is deterministic if there exists exactly one input $u \in \prod_{i=1\dots n}\Sigma(X_i^u)$ that satisfies interconnection constraint $\mathcal{I}$ for given output $y \in \prod_{i=1\dots n}\Sigma(X_i^y)$ and control input $\mu \in \Sigma(X^c)$.

Definition 3.9 (Deterministic Interconnection Constraint) Given concurrent constraint automaton $\mathcal{M} = \langle \mathcal{A}, X, \mathcal{I} \rangle$, interconnection constraint $\mathcal{I}$ is *deterministic*, if and only if, for each output $y \in \prod_{i=1\dots n}\Sigma(X_i^y)$ and control $\mu \in \Sigma(X^c)$, there exists exactly one input $u \in \prod_{i=1\dots n}\Sigma(X_i^u)$ that satisfies interconnection constraint $\mathcal{I}$.

A deterministic CCA is then defined as follows:

Definition 3.10 (Deterministic CCA) A concurrent constraint automaton $\mathcal{M} = \langle \mathcal{A}, X, \mathcal{I} \rangle$ is deterministic, if and only if each constraint automaton $\mathcal{A}_i \in \mathcal{A}$ is deterministic (see Def. 3.5) and interconnection constraint $\mathcal{I}$ is deterministic (see Def. 3.9).

3.2.2 Execution of a CCA

Before formally defining an execution of a CCA, a projection operator $\mathrm{proj}_{X'}(w)$ is defined. The projection operator will be used to define an *execution* of a CCA, as well as a *feasible trajectory* of a CCA described in next section,

Definition 3.11 (Projection of an Instantiation) Let $w \in \Sigma(X)$ be a full assignment to variables in X and $X' \subseteq X$ be a subset of the variables in X. Then $\mathrm{proj}_{X'}(w)$ is the projection of the assignments w to variables X', that is $\mathrm{proj}_{X'}(w) \equiv \{(x = v) \mid$

$(x = v) \in w, x \in X'\}$. An assignment may also be projected to a single variable x_j, where $\text{proj}_{x_j}(w) = \{(x_j = v) \mid (x_j = v) \in w\}$ is a singleton set of an assignment.

For example, let $X = \{x_1, x_2, x_3, x_4, x_5\}$, $w = (x_1 = v_1, x_2 = v_2, x_3 = v_3, x_4 = v_4, x_5 = v_5)$ and $X' = \{x_2, x_3, x_5\}$. Then, the projection of w to X' is $\text{proj}_{X'}(w) = (x_2 = v_2, x_3 = v_3, x_5 = v_5)$, and the projection to variable x_4 is $\text{proj}_{x_4}(w) = (x_4 = v_4)$.

With the use of the projection operator and the definition of an execution of a constraint automaton (Def. 3.6, an execution of a CCA can be formally defined as follows:

Definition 3.12 (Execution of a CCA) Given CCA $\mathcal{M} = \langle \mathcal{A}, X, \mathcal{I} \rangle$ in state $s \in S$, the *execution of control input* $\mu \in \Sigma(X^c)$ is $\text{exec}(\mathcal{M}, s, \mu) = \bigcup_i \text{exec}(\mathcal{A}_i, q_i, u_i)$, where

1. $\mathcal{A}_i \in \mathcal{A}$ is a constraint automaton of CCA $\mathcal{M}$
2. $q_i \in \text{proj}_{X_i^q}(s)$ is the current state of constraint automaton $\mathcal{A}_i$,
3. $u_i = \text{proj}_{X_i^u}(u)$, where $u \in \prod_i \Sigma(X_i^u)$ is an input, such that $u \wedge \mu \wedge \mathcal{I} \wedge \bigwedge_{q_i \in s} \mathcal{Q}_i(q_i)$ is consistent,

The notion of execution of a control input is extended to *execution of control sequence* $\mu(0:n) = (\mu(0), \ldots, \mu(n))$, where

$$\text{exec}(\mathcal{M}, s, \mu(0:n)) = \begin{cases} \text{exec}(\mathcal{M}, s, \mu(0)), & \text{if } n = 0 \\ \text{exec}(\mathcal{M}, \text{exec}(\mathcal{M}, s, (\mu(0), \ldots, \mu(n-1))), \mu(n)), & \text{if } n > 0 \\ \text{undefined} & \text{otherwise} \end{cases} \tag{3.3}$$

The third condition of $\text{exec}(\mathcal{M}, s, \mu)$ specifies that the input to the constraint automata must be consistent with control input μ, output of the automata defined by $\bigwedge_{q_i \in s} \mathcal{Q}_i(q_i)$ and interconnection constraint $\mathcal{I}$. In essence, this condition assures that the flow of

information from control μ and output y to input u is consistent with the behavior specified by the CCA.

Again, as was with executions of a constraint automaton, an execution of a CCA is in general nondeterministic. However, if a CCA is deterministic (Def. 3.10), then the execution is also deterministic.

3.2.3 Feasible Trajectory of a CCA

Consider a state trajectory $s(0:n{+}1) = (s(0), s(1), \ldots, s(n{+}1))$ and a control sequence $\mu(0:n) = (\mu(0), \mu(1), \ldots, \mu(n))$, where $s(t)$ and $\mu(t)$ represent the state, and control input of a CCA at time t, respectively. Then, $s(0:n+1)$ is a *feasible trajectory* for a CCA if $s(0:n+1)$ is one of the state trajectories that results from executing control sequence $\mu(0:n)$. Formally,

Definition 3.13 (Feasible Trajectory of a CCA) Let $\mathcal{M} = \langle \mathcal{A}, X, \mathcal{I} \rangle$ be a CCA, where an automaton $\mathcal{A}_i \in \mathcal{A}$ is described by a 4-tuple $\langle X_i, D_i, \mathcal{Q}_i, \delta_i \rangle$. Then, a finite sequence $s(0:n{+}1) = (s(0), \ldots, s(n{+}1))$ is a *feasible trajectory* generated by executing control sequence $\mu(0:n) = (\mu(0), \ldots, \mu(n))$, if and only if

- $s(0) \in S$ is the initial state of $\mathcal{M}$,
- $s(t+1) = \mathrm{exec}(\mathcal{M}, s(t), \mu(t))$, for $0 < t \leq n$.

3.3 Related Work

Reactive planning methods [25, 3] used a the same CCA formalism to model a system to be controlled. Due to the planning complexity introduced by the maintenance and observation constraints of a CCA, those constraints were assumed to be "compiled" away. This compilation method requires an additional assumption that reduces the expressivity of the modeling formalism and the planning problem description. Within the compilation, the constraints on the dependent variables are eliminated by substituting them with entailed constraints on state and control variables. The essential elements of

the CCA model are extracted using knowledge compilation methods [25, 7] and encoded as *concurrent automata* ($\mathcal{CA}$) for reactive planning. In essence, $\mathcal{CA}$ represent a nondeterministic transition system with finite state and concurrently operating components. Compiling a CCA into a $\mathcal{CA}$ eliminates the need for constraint-based reasoning. Also, the elimination of the dependent variables reduces the size of the state space. For example, the Deep Space One (DS1) CCA model developed for the Remote Agent included approximately 3000 propositional variables; with the dependent variables eliminated, only about 100 variables remained.

Chapter 4

Planning Problem for CCA

Contents

4.1 Planning Problem for CCA **50**
4.2 Qualitative Time QSP **50**
4.2.1 Qualitative Time in QSP 52
4.2.2 Formal Definition of Qualitative Time QSP 52
4.3 Qualitative Time QCP **57**
4.3.1 Partially Ordered Plan 58

In a general planning problem, the objective is to generate a sequence of control actions that achieves the desired goal given the initial state and the description of the system under control. The description of the system specifies a set of legal states and actions, usually described using a set of domain axioms and action operators. The complexity of the planning problem depends on the the expressivity of the description of the goal and the system.

In model-based programming formalism, the system under control is represented as a Concurrent Constraint Automata (CCA) described in Chapter 3. The benefit in describing a system under control as a CCA is in its expressivity. However, any type of reasoning on CCA, including planning, becomes quickly intractable as the number of constraint automata grows in a CCA. Recall that the number of states in a CCA is $|S| = \prod_{i=1...n} |Q_i|$, where $n = |\mathcal{A}|$ is the number of constraint automata in a CCA and $|Q_i| = |\mathcal{D}(x_i^q)|$ is the number of states for each constraint automaton $\mathcal{A}_i$. The number of actions can be as large as the number of possible control inputs $|\Sigma(X^c)| = \prod_{x \in X^c} |\mathcal{D}(x)|$, where X^c is a set of control variables. In order to assure the tractability of planning for CCAs, the problem of state space explosion must be mitigated, if at all possible.

A simple goal description typically specifies the desired final state of the system. In this thesis, we are interested in a more complex goal that specifies the desired behavior of a system over time, typically referred to as a temporally *extended goal* [11]. In model-based programming, a goal specification formalism called *Qualitative State Plan* (QSP) is used to describe a temporally extended goal.

In this chapter, we describe and formally define a planning problem for CCA. First, we briefly describe the planning problem. Then, in the subsequent sections, we describe and define the components of the planning problem. As Chapter 3 defines CCA and its initial state, we start with the description of a QSP, followed by the definition and formalism of a solution to the planning problem, called *Qualitative Control Plan* (QCP).

4.1 Planning Problem for CCA

In this thesis, we are concerned with the problem of generating a control sequence that achieves a temporally extended goal given the initial state of a finite state system. A temporally extended goal is a set of goals that are temporally constrained. In effect, a temporally extended goal constrains the state trajectory of the system. As previously described, the system is specified as a CCA. The initial state of a CCA is specified simply as a full assignment to the state variables of the system. Formally the problem is as follows:

Definition 4.1 (Planning Problem for a CCA) A planning problem $\mathcal{P}^{CCA}$ is a 3-tuple $\langle \mathcal{M}, s(0), \mathcal{QSP} \rangle$, where $\mathcal{M} = \langle \mathcal{A}, X, \mathcal{I} \rangle$ is CCA that represent the system under control, $s(0) \in S$ is the initial state of $\mathcal{M}$ (Def. 3.8), and $\mathcal{QSP}$ is a qualitative time, qualitative state plan (Def. 4.2) that specifies the desired time-evolved goals.

4.2 Qualitative Time QSP

A mission objective and an operator's intent can be described explicitly as a desired evolution of goal states. The desired evolution of goal states can be formally represented as a *qualitative state plan* (QSP) [17, 14]. Given a QSP and the current state, a planner

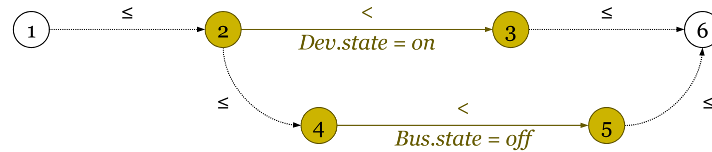


Figure 4-1: An example of a qualitative time, qualitative state plan (QT-QSP) with qualitative temporal constraints represented using point algebra.

must generate an executable plan that will achieve the desired goal states as described by the QSP.

For example, consider Bus1Dev1 system described in Chapter 3. A simple mission objective for the system may be to first turn on the device ($Dev.state = on$). Once $Dev.state = on$ is achieved, we wish to turn off the bus controller ($Bus.state = off$) at any time thereafter. Such mission objective may be described as a QSP illustrated in Fig. 4-1.

In Fig. 4-1, each vertex represents a *time point*. The inequality constraint labeled on each directed edge represents a *temporal constraint* on two time points connected by the edge. For example, $\leq$ constraint labeled on the directed edge connecting time point one, t_1, to time point two, t_2, specifies that t_1 must occur before or at the same time as t_2, that is $t_1 \leq t_2$.

While all directed edges are associated with a temporal constraint, some directed edges are also associated with an *episode*. An episode specifies a restriction on time and state space. For example, the directed edge from t_2 to t_3 specifies an episode, in addition to the temporal constraint specified by $<$ constraint. Let us refer to this episode as ep_1. Episode ep_1 restricts the time-space by requiring that the episode start at time t_2, referred to as the *start event* and denoted $e_s(ep_1)$, and end at time t_3, referred to as the *end event* and denoted $e_e(ep_1)$. The episode also restricts the state-space through the state constraint, $Dev.state = on$. Consequently, episode ep_1 requires that the device state must be on starting at time t_2 until time t_3.

Another episode, referred to as ep_2, occurs between time points t_4 and t_5, which specifies that the bus controller must be off ($Bus.state = off$). Note that the temporal constraint between the start time of ep_1, $e_s(ep_1) = t_2$, and the start time of ep_2,

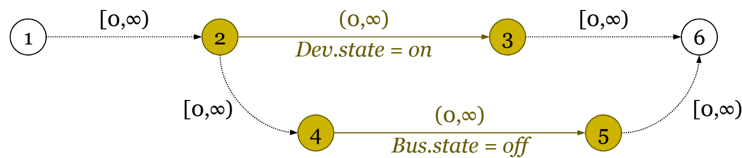


Figure 4-2: An example of a qualitative time, qualitative state plan (QT-QSP) with qualitative temporal constraints represented using simple temporal constraints.

$e_s(ep_2) = t_4$, that is $e_s(ep_1) \leq e_s(ep_2)$, requires the state constraint $Bus.state = off$ be true at the same time or thereafter the event at which $Dev.state = on$ becomes true.

4.2.1 Qualitative Time in QSP

In general a qualitative state plan may specify quantitative (metric) temporal constraints, in which a set of time points of a QSP are constrained by *binary temporal constraints* of the form $l \leq t_j - t_i < u$, where l and u are real numbered constants that represent the lower and upper bounds of the difference of the two time points t_i and t_j. The lower and upper bounds, in effect, specify the time interval for which $t_j - t_i$ is allowed.

In this chapter, however, we are concerned with QSP's with qualitative temporal constraints. We refer to such QSP's as *qualitative time QSP* (QT-QSP). In QT-QSP, point algebra [22] is used to describe the relationship between time points. In point algebra, only equality and inequality operators, $\{<, \leq, =, >, \geq\}$, are used to describe the relationship between two time points, e.g. $t_i < t_j$. Note that a time interval can be use to represent a point algebra by restricting the lower and upper bounds to be either zero or infinity, that is, $l \in \{-\infty, 0\}$ and $u \in \{0, \infty\}$. Figure 4-2 is an example of a QT-QSP for which a qualitative temporal constraint is represented using a time interval.

4.2.2 Formal Definition of Qualitative Time QSP

As described above a QSP describes a desired evolution of states. More specifically, a QSP describes the desired evolution of states by specifying a set of time points bound by temporal constraints and state constraints. In this section a qualitative time QSP is

formally defined.

Definition 4.2 (Qualitative Time QSP) A *qualitative time, qualitative state plan* $\mathcal{QSP}^{qt}$ is a 5-tuple $\langle X, D, T, \mathcal{C}^t, E\rangle$, where

- $X = \{x_1, \ldots, x_n\}$ is a finite set of *variables*,
- $D = \{\mathcal{D}(x_1), \ldots, \mathcal{D}(x_n)\}$ is a set of finite *domains* of X,
- $T = \{t_1, \ldots, t_m\}$ is a finite set of time points (Def. 4.3),
- $\mathcal{C}^t : T \times T \rightarrow \widetilde{\mathfrak{I}}$ is a temporal constraint function (Def. 4.5), and
- $E = \{ep_1, \ldots, ep_l\}$ is a finite set of externally imposed episodes that specify the desired state over time (Def. 4.8).

Variables and Domains

A finite domain variables X and their corresponding domains D are used in describing an episode to specify the desired state of a CCA. As such, X and D are identical to that of the CCA for which the QT-QSP is specified.

In the QT-QSP shown in Fig. 4-2, only the state variables *Bus.state* and *Dev.state* are used. Note that any variables of Bus1Dev1 CCA could have been used to specify the state constraints of the episodes.

Time Point

Definition 4.3 (Time Point of a QSP) A *time point*, $t \in \mathbb{R}$, is a real valued variable that represents a point in time-space. The time for each time point t_i is measured from a reference time point called *starting time point*, denoted t_s.

In the QT-QSP shown in Fig. 4-2, there are total of six time points, that is $T = \bigcup_{i=1,\ldots,6}\{t_i\}$. Of the six time points, t_1 is the starting time point, that is $t_s \equiv t_1$.

Temporal Constraint

Adopting the temporal constraint formalism of [5], a temporal constraint $\mathcal{C}^t_{ij} \equiv \{I\}$ specifies a set of feasible *qualitative time intervals* for time points t_i and t_j. For example, a time interval $I = [l, u)$ of $\mathcal{C}^t_{ij}$ represents a temporal constraint $l \leq t_j - t_i < u$. Since we are interested in representing qualitative temporal constraints, we restrict an interval to also be qualitative. A qualitative time interval is formally defined as follows:

Definition 4.4 (Qualitative Time Interval) A *qualitative time interval* $I \in \widetilde{\mathfrak{I}}$ of a temporal constraint $\mathcal{C}^t_{ij}$ is an interval of a binary domain that represents a feasible duration between time points t_i and t_j, that is $t_j - t_i$. Intervals $\widetilde{\mathfrak{I}} \equiv \{\emptyset, [l, u], (l, u), [l, u), (l, u] \mid l \in \{-\infty, 0\}, u \in \{0, \infty\}\}$ may be empty, closed, semi-open, or open, where l and u respectively denote the lower and upper bounds of the interval.[1]

The lower bound of an interval may be unconstrained simply by setting $l = -\infty$. Similarly the upper bound may be unconstrained simply by setting $u = \infty$. Note that [17, 14] only uses non-strict inequalities, $\{\leq, \geq\}$, but we generalize the formalism to more expressive form in which strict inequalities, $\{<, >\}$, are also allowed.

As described before, a temporal constraint $\mathcal{C}^t_{ij} \equiv \{I_1, I_2, \ldots, I_n\}$ is, in general, a set of intervals. A set of intervals represents a disjunction of the intervals, $I_1 \vee I_2 \vee \ldots \vee I_n$. In this thesis, however, we restrict a temporal constraint to be a singleton set, thus disallowing disjunctive intervals.

Definition 4.5 (Qualitative Temporal Constraint) A *temporal constraint function* $\mathcal{C}^t : T \times T \rightarrow \widetilde{\mathfrak{I}}$ associates any two distinct time points, t_i and t_j for $i \neq j$, to a singleton set of a qualitative time interval specified by the temporal constraint $\mathcal{C}^t_{ij}$.

A temporal constraint $\mathcal{C}^t_{ij}$ is said to be consistent if time points t_i and t_j can be assigned to some value such that $t_j - t_i$ is in interval $I \in \mathcal{C}^t_{ij}$. For conciseness, $\mathcal{C}^t_{ij} =$

[1] The notation $\widetilde{\mathfrak{I}}$ is used to distinguish qualitative time intervals from quantitative time intervals, denoted $\mathfrak{I}$.

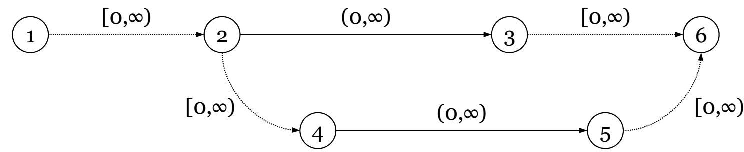


Figure 4-3: The simple temporal constraint of QSP Fig. 4-2.

$\{(-\infty, \infty)\}$ by default, if a temporal constraint $\mathcal{C}^t_{ij}$ is undefined, that is, the duration between t_i and t_j is unconstrained.

A set of temporal constraints $\mathcal{C}^t$ is said to be consistent if all time points in T can be assigned to some value such that all temporal constraints are consistent. In general, a temporal constraint satisfaction problem (TCSP) is NP-hard [5] due to the disjunctions of intervals. By disallowing disjunctions, that is, allowing only a single interval for a temporal constraint, the temporal satisfaction problem can be solved in polynomial time [5]. Such problems, in which a temporal constraint is restricted to a singleton, is called *simple temporal problem* (STP) [5]. Accordingly, the temporal constraint of a STP is called *simple temporal constraint.* Note that in this thesis we are only concerned with QSP's with simple temporal constraints.

Definition 4.6 (Simple Temporal Constraint) A *simple temporal constraint* is a temporal constraint that is restricted to a singleton set of a time interval.

Figure 4-3 illustrates the temporal constraint of the QT-QSP shown in Fig. 4-2. Each directed edge is labeled with a time interval. Each time interval represents the simple temporal constraint of the corresponding two time points connected by the directed edge. For example, the interval $[0, \infty)$ on edge (t_2, t_4) corresponds to simple temporal constraint $\mathcal{C}^t_{2,3} = \{[0, \infty)\}$. Each temporal constraint $\mathcal{C}^t_{ij}$ for which the directed edge (t_i, t_j) does not exist is assumed unconstrained, that is $\mathcal{C}^t_{i,j} = \{(-\infty, \infty)\}$. Note the use of dotted and solid lines for the edges. A dotted line edge is used to denote an interval $[0, \infty)$ and a solid line edge is used to denote an interval $(0, \infty)$. We use this convention of notation throughout the thesis.

Episode

In general, a state constraint describes a time-varying state region, e.g. *flow-tube* [14], by constraining both the state and the time. In this thesis, however, the state-space constraint is restricted to discrete, time-invariant constraints, that is, time-invariant relation over variables with finite domains. Furthermore, we restrict the state-space constraint to a finite domain constraint, $C(X)$ as defined in Def. 4.7, but in which conjunction, $\varphi_i \wedge \varphi_j$, and disjunction, $\varphi_i \vee \varphi_j$, are disallowed:

Definition 4.7 (State Constraint of an Episode) Given a set of finite domain variables X, a *state constraint* of an episode $\varphi \in C(X)$ over finite domain variables X is a sentence in a propositional state logic, that is:

$$\varphi ::= \texttt{true} \mid \texttt{false} \mid (x = v) \mid \neg\varphi_1, \tag{4.1}$$

where $x \in X$ and $v \in \mathcal{D}(x)$. Variables X and sentence φ respectively represent the scope and the relation of the constraint.

An episode is defined formally as follows:

Definition 4.8 (Episode) An *episode* ep is a 3-tuple $\langle e_s, e_e, c\rangle$, where

- $e_s \in T$ is an event representing the starting time point of the episode.
- $e_e \in T$ is an event representing the end time point of the episode, such that $e_s < e_e$.
- $c \in C(X)$ is a state constraint that restricts the state-space from the time of e_s to e_e.

In the QT-QSP shown in Fig. 4-2, there are two episodes $E = \{ep_1, ep_2\}$, where $ep_1 \equiv \langle t_2, t_3, Dev.state = on\rangle$ and $ep_2 \equiv \langle t_4, t_5, Bus.state = off\rangle$.

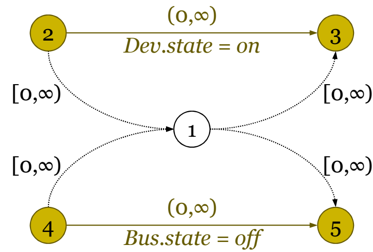


Figure 4-4: Concurrent episodes whose state-space constraints *Dev.state* = *on* and *Bus.state* = *off* must be true at the time point t_1.

Note that while $\varphi_i \wedge \varphi_j$ is disallowed in a state constraint of an episode, a conjunction can still be represented in a QSP by use of temporal constraints. Let ep_i be an episode with constraint φ_i and ep_j be an episode with constraint φ_j. Then, $\varphi_i \wedge \varphi_j$ can be represented by requiring both episodes ep_i and ep_j to occur over the same time point. For example, if we desire to specify that $Dev.state = on$ and at the same time $Bus.state = off$, or equivalently, $(Dev.state = on) \wedge (Bus.state = off)$, we can represent it by requiring the two episodes corresponding to $Dev.state = on$ and $Bus.state = off$ occur over the same time point. This is illustrate in Fig. 4-4.

4.3 Qualitative Time QCP

The solution to a CCA planning problem can be represented as a set of feasible control sequences that achieve the goal specified in the form of a QSP. That is, a partially ordered controls in effect represent a set of totally ordered control sequences. Qualitative Control Plan (QCP) is a formalism that specifies such partially ordered control sequence.

The criteria for goal achievement depends on the goal type. In our problem, a goal formalism called Qualitative State Plan (QSP) is used to specify the desired goal. The following sections will describe QSP and the definition of goal achievement for QSP.

Definition 4.9 (Solution to CCA Planning Problem) A control sequence $\mu(0 : n) = (\mu(0), \ldots, \mu(n))$ is a solution to the planning problem $\mathcal{P}^{CCA} = \langle \mathcal{M}, s(0), \mathcal{QSP} \rangle$ if and only if the following two conditions are met:

1. Feasible Trajectory: $s(0 : n) = (s(0), \ldots, s(n))$ is a *feasible state trajectory* that

results from executing control sequence $\mu(0:n-1)$ from initial state s(0).

2. Goal Satisfaction: $s(n) \wedge \mu(n) \wedge g(n) \wedge \mathcal{I}(n) \wedge \bigwedge_{q_i(n) \in s(n)} \mathcal{Q}_i(q_i(n))$ is consistent.

The first condition requires that the trajectory $s(0:n)$ generated by executing $\mu(0:n-1)$ is indeed a feasible trajectory as specified in Def. 3.13. Intuitively, $s(0:n)$ is a feasible trajectory if the execution of the the control sequence $\mu(0:n-1)$ on CCA $\mathcal{M}$ in initial state $s(0)$ results in the state trajectory $s(0:n+1)$. The second condition requires that the feasible state trajectory guarantees that the goal is achieved. The goal is achieved if the state and control at time n and the goal is consistent with the state and the interconnection constraints.

As specified in Def. 4.9, a solution to the planning problem is a control sequence $\mu(0:n) = (\mu(0), \ldots, \mu(n))$, which, upon execution from initial state $s(0)$, generates a feasible state trajectory $s(0:n) = (s(0), \ldots, s(n))$ that achieves the goal in its final state. Instead of searching for the control sequence, we can find a sequence of inputs, $u(0:n) = (u(0), \ldots, u(n))$ where $u \in \prod_i \Sigma(X_i^u)$, which, upon execution from the initial state, achieves the goal. From input sequence $u(0:n)$, we can extract the solution trough projection, as defined in Def. 4.10, where $\mu(0:n) = \text{proj}_{X^c}(u(0:n))$.

Definition 4.10 (Projection of a Sequence) Let $w \in \Sigma(X)$ be a full assignment to variables in X and $X' \subseteq X$ be a subset of the variables in X. Let $w(0:n) = (w(0), \ldots, w(n))$ be a sequence of instantiations to variables X. Then $\text{proj}_{X'}(w(0:n))$ is the projection of the sequence $w(0:n)$ to variables X', that is $\text{proj}_{X'}(w(0:n)) \equiv (\text{proj}_{X'}(w(0)), \ldots, \text{proj}_{X'}(w(n)))$.

4.3.1 Partially Ordered Plan

Before we define partially ordered plan, we first introduce partial instantiation.

Definition 4.11 (Partial Instantiations) A set of *partial instantiations* $\tilde{\Sigma}(X) =$

$\bigcup_{X' \in 2^X} \Sigma(X')$ is used to denote a complete set of assignments to all possible subset of variables $X' \in 2^X$.

Definition 4.12 (Qualitative Control Plan) A *qualitative control plan* (QCP) is a 5-tuple $\langle X^c, \mathcal{D}, T, \mathcal{C}^t, E \rangle$, where

- X^c is a set of discrete domain, control variables,
- $\mathcal{D}$ is the corresponding set of control variable domains,
- T is a set of time points (Def. 4.3),
- $\mathcal{C}^t$ is a set of temporal constraints on time points (Def. 4.5), and
- E is a set of episodes that specify the required control actions over time (Def. 4.8).

Note that the syntax of QCP is nearly identical to that of QSP. The only difference is that the variables are restricted to only the control variables. Correspondingly, the state-space constraint of an episode is also restricted to control variables. Furthermore, while the episodes of a QSP specify the desired goal state, the episodes of a QCP specify the required control actions that must executed in a timely manner as specified by the temporal constraint $\mathcal{C}^t$.

An example of a QCP for Bus1Dev1 problem is shown in Fig. 4-5.

A partially ordered plan represents a set of control sequences, where $\mu(0:n) = (\mu(0), \ldots, \mu(n))$ is a control sequence described by a partial plan $\Pi = \langle T, \Sigma(X^c), \mathcal{C}^< \rangle$ if the corresponding sequence of time points $t_{0:n} = (t_s, \ldots, t_n)$ satisfies the ordering constraint $\mathcal{C}^<$. Furthermore, a partial plan $\Pi = \langle T, \Sigma(X^c), \mathcal{C}^< \rangle$ is a *valid* plan for the problem $\mathcal{P}^{CCA} = \langle \mathcal{M}, s(0), g \rangle$ if and only if each control sequence $\mu(0:n) \in \Pi$ is a *solution* to the problem $\mathcal{P}^{CCA}$.

Definition 4.13 (Valid Partially Ordered Plan) Given a planning problem $\mathcal{P}^{CCA} = \langle \mathcal{M}, s(0), g \rangle$, a partially ordered plan $\Pi = \langle T, \Sigma(X^c), \mathcal{C}^< \rangle$ is *valid* if and only if

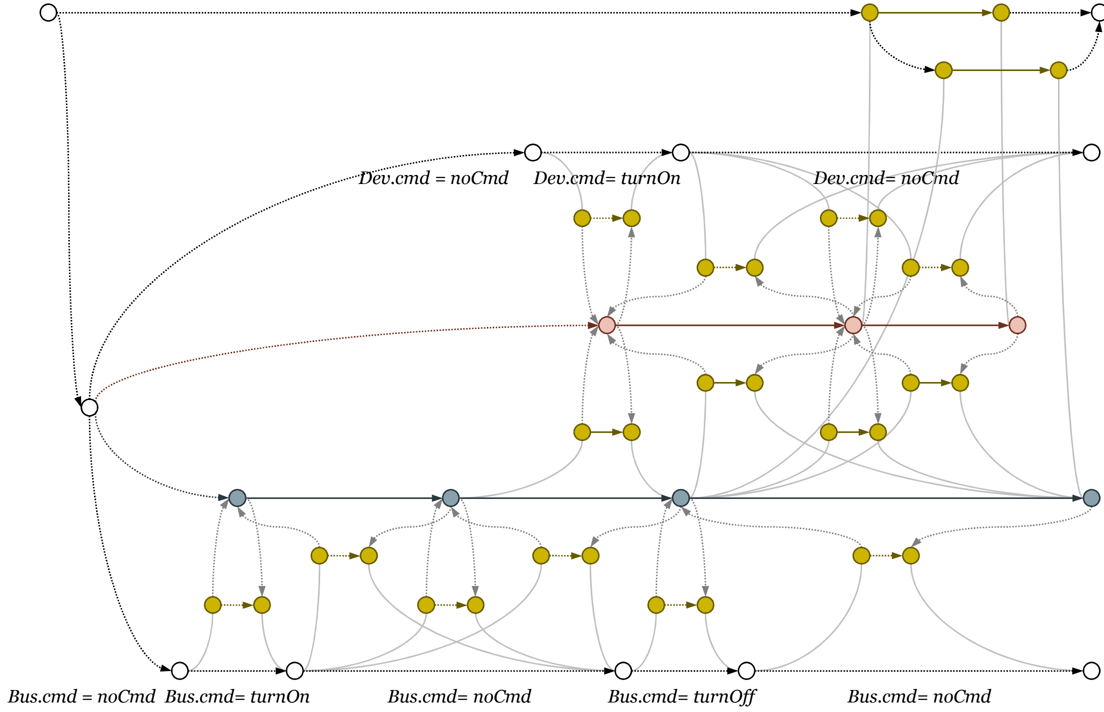


Figure 4-5: Qualitative control plan (QCP) for the goal QSP shown in Fig. 4-2.

1. $\left\{ t_{0:n} \,\middle|\, \left(\bigwedge_{i=1,\ldots,n} (t_{i-1} < t_i) \right) \text{ satisfies } \mathcal{C}^< \right\} \neq \emptyset$.

2. For each sequence of time points $t_{0:n} \in \left\{ t_{0:n} \,\middle|\, \left(\bigwedge_{i=1,\ldots,n} (t_{i-1} < t_i) \right) \text{ satisfies } \mathcal{C}^< \right\}$, the corresponding control sequence $\mu(0:n)$ is a *solution* to the planning problem $\mathcal{P}^{CCA}$ as defined in Defintion 4.9.

The first condition specifies that at least one sequence of time points $t_{0:n}$ must exist that is, consistent with the ordering constraint $\mathcal{C}^<$. The second condition specifies that for a given sequence of time points $t_{0:n}$ that is, consistent with the ordering constraint $\mathcal{C}^<$, the corresponding control sequence $\mu(0:n)$ is a solution to the planning problem.

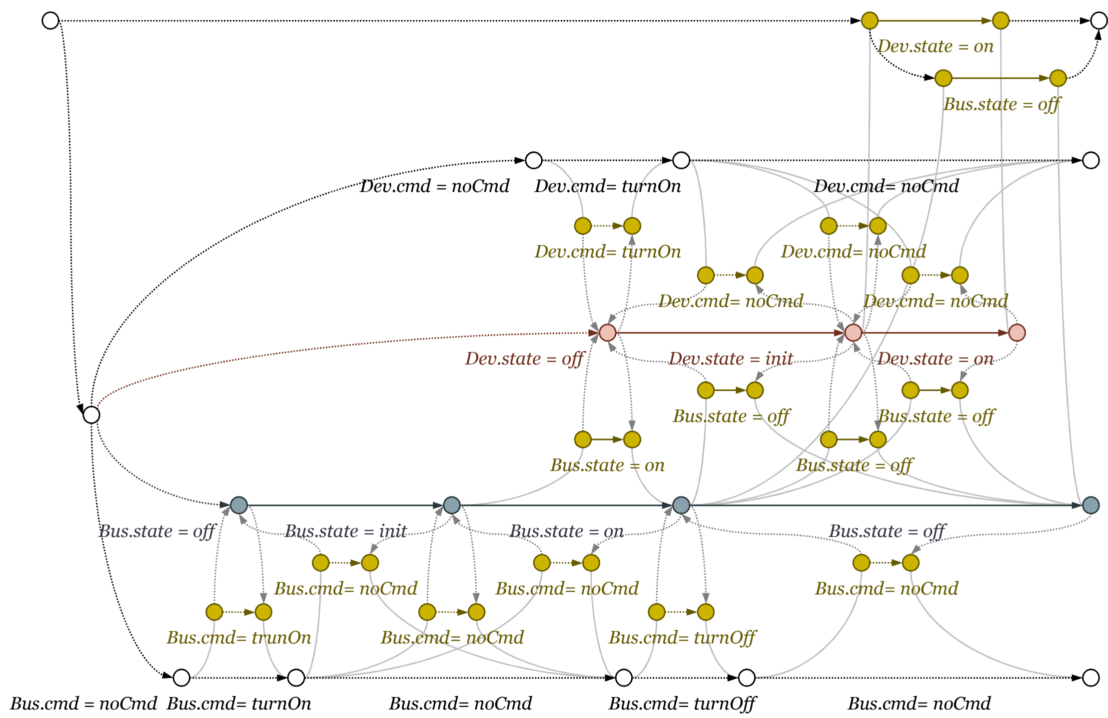


Figure 4-6: Elaborated qualitative control plan (eQCP) for the goal QSP shown in Fig. 4-2.

Chapter 5

Planning for CCA & Qualitative Time QSP

Contents

5.1 Compiled Bus Controller and Device Example 63
5.2 Solving the Planning Problem 64
5.2.1 Plan Space: A Trellis Diagram 65
5.2.2 Solution within a Trellis Diagram 67
5.3 Decomposed Planning 68
5.3.1 Decomposing the Trellis Diagram 69
5.3.2 Searching through the Decomposed Trellis Diagram 75
5.3.3 Extending to Temporal Planning 81
5.4 Related Work 81
5.4.1 State Constraint 82

In this chapter, we describe a new decomposed planning method for the CCA planning problem. Finally, the chapter ends with the description of related work.

5.1 Compiled Bus Controller and Device Example

To keep the description of the planning problem and the planning process simple to understand, we consider a *compiled* CCA of a bus controller (Bus) and a generic device (Dev) whose state constraints have been compiled away [7]. We refer to this compiled model as *CompiledBus1Dev1*. *CompiledBus1Dev1* is graphically depicted in Fig. 5-1.

The bus controller, depicted in Fig. 5-2(a), is responsible for relaying data, including commands, to the devices that are connected to it. It has three states: An off state (*off*), an initializing state (*init*), and an on state (*on*). The device can be commanded to turn on or turn off. When turned off, the bus controller immediately transition into

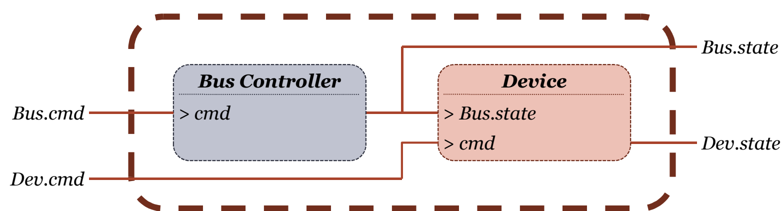
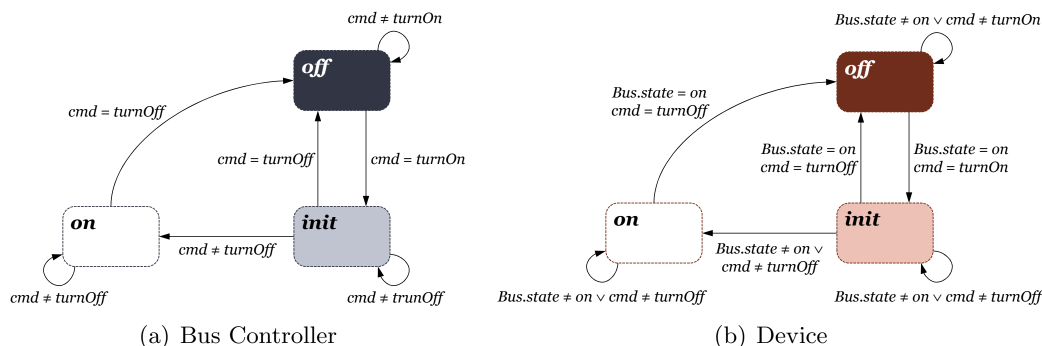


Figure 5-1: Compiled model of a device attached to a bus controller.



(a) Bus Controller (b) Device

Figure 5-2: 5-2(a) is a compiled model of a bus controller and 5-2(b) is a compiled model of a device attached to the bus controller.

the off state from any other state. When turned on from the off state, however, the bus controller first transitions into the initializing state. Unless commanded to turn off, the bus controller autonomously transitions into the on state once initialization is complete. The bus controller must be in the on state for it to be able to relay any data to its devices. The specification of this behavior has been *compiled* away and is reflected in the device model.

Similar to the bus controller model, the device depicted in Fig. 5-2(b) can also be in any one of the three states $\{off, init, on\}$. The behavior of the state transition is identical to that of the bus controller except that any commanded transitions require that the bus controller be on. This is due to the fact that the bus controller must be on for it to relay any command to the device.

5.2 Solving the Planning Problem

We can view the CCA planning problem $\mathcal{P}^{CCA} = \langle \mathcal{M}, s_0, \mathcal{QSP} \rangle$ as a graph search problem. The objective is to find a path that is rooted at a vertex corresponding to

initial state s_0 and achieves qualitative state plan (QSP) $\mathcal{QSP}$.

5.2.1 Plan Space: A Trellis Diagram

The plan space of $\mathcal{P}^{CCA}$ is defined by a set of feasible execution traces specified by CCA $\mathcal{M}$. We can depict this search space graphically using a trellis diagram. For example, Fig. 5-3 is a trellis diagram that depicts the search space of a planning problem for *CompiledBus1Dev1* CCA. Each column t_i represents a set of feasible states at ith time step. In each row, a circle that circumscribes two smaller circles represents a state of *CompiledBus1Dev1* CCA. Of the two smaller circles, the top circle represents a state of the Bus Controller and the bottom circle represents a state of the device. Note that the shades of colors used directly correspond to the shades used to distinguish the states of the Bus Controller and the device in Fig. 5-2. An edge that connects a state of *CompiledBus1Dev1* CCA to a successive state represents a feasible transition between the two successive states. For example, state $\{Bus.state = off, Dev.state = off\}$ (first row of the trellis diagram) may transition to state $\{Bus.state = init, Dev.state = off\}$ (second row).

Formally, the set of states $S(t_i) = \prod_i Q_i(t_i)$ of CCA $\mathcal{M} = \langle \mathcal{A}, X, \mathcal{I} \rangle$ for time steps $i = 0, 1, \ldots, n$ directly corresponds to the set of vertices V of a trellis diagram $G = \langle V, E \rangle$, where constraint automaton $\mathcal{A}_i \in \mathcal{A}$ is described by a 4-tuple $\langle X_i, D_i, \mathcal{Q}_i, \delta_i \rangle$ and $Q_i = \Sigma(X_i^q)$ is a set of states of $\mathcal{A}_i$. For example, *CompiledBus1Dev1* CCA has two constraint automata, that is $\mathcal{A} = \{\mathcal{A}_{Bus}, \mathcal{A}_{Dev}\}$. The state variables of *Bus* and *Dev* are $X_{Bus}^q = \{Bus.state\}$ and $X_{Dev}^q = \{Dev.state\}$, respectively. For any time point t_i, the states of *CompiledBus1Dev1* is defined by all combinations of the states of two constraint automata, that is $S(t_i) = Q_{Bus}(t_i) \times Q_{Dev}(t_i)$. Since each constraint automaton has three states, there are total of nine states for the CCA at each time step.

In a trellis diagram, a set of directed edges is formally defined as $E = \{(s_k, s_l) \mid \exists_{\Sigma(X^c)}(\mu).(s_l = \text{exec}(\mathcal{M}, s_k, \mu))\}$. Intuitively, CCA $\mathcal{M}$ in state s_k may transition to state s_l if and only if there exists a control input $\mu \in \Sigma(X^c)$, such that execution of μ on CCA $\mathcal{M}$ may transition the CCA to state s_l in the next time step, that is

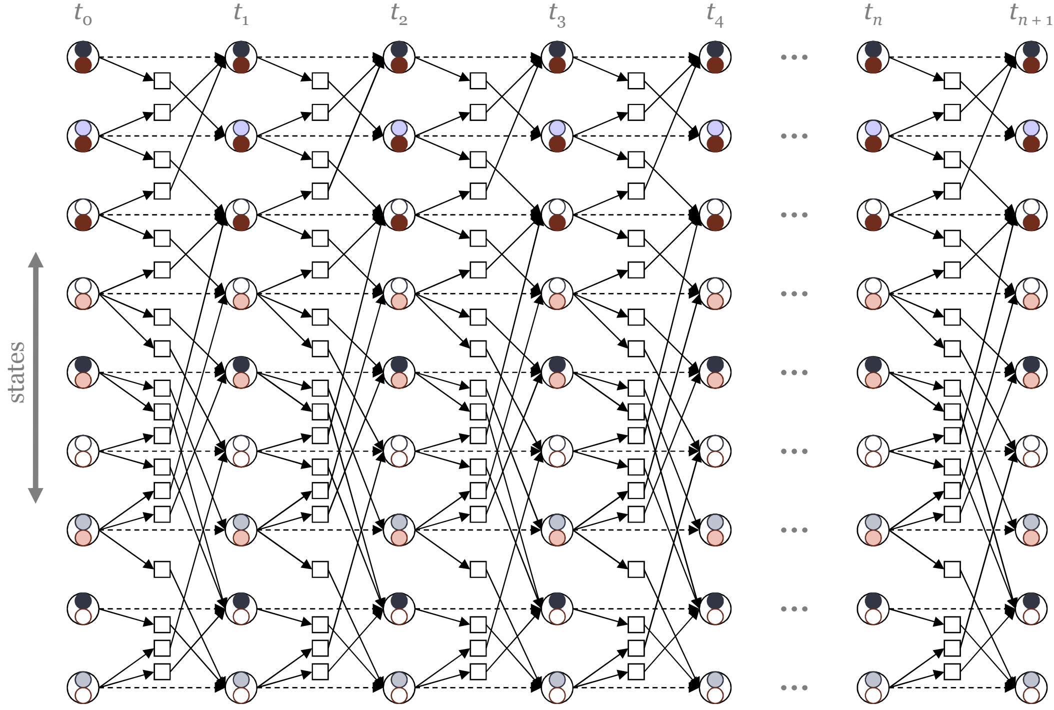


Figure 5-3: Trellis diagram of *CompiledBus1Dev1* model.

$s_l = \text{exec}(\mathcal{M}, s_k, \mu)$. Furthermore, $s_l = \text{exec}(\mathcal{M}, s_k, \mu)$ if and only if there exists input $u(t) \in \prod_i \Sigma(X_i^u(t))$, such that $\mu(t) = \text{proj}_{X^c}(u(t))$, that is control $\mu(t)$ is a projection of input $u(t)$ to control variables X^c, and the following two constraints are satisfiable:

State Consistency State $s_k(t)$ and input $u(t)$ are consistent with the specified state behavior of $\mathcal{M}$:

$$\mathcal{I}(t) \wedge \bigwedge_i \bigwedge_{q_i(t) \in Q_i} (q_i(t) \Rightarrow \mathcal{Q}_i(q_i(t))) \tag{5.1}$$

Transition Consistency There exists a constraint $\varphi_i(t) \in C(X_i^u(t))$, such that $s_k(t) \cup u(t) \cup s_l(t+1)$ is consistent with the specified transition behavior of $\mathcal{M}$:

$$\bigwedge_i \bigwedge_\delta (q_i(t) \wedge \varphi_i(t)) \Rightarrow q_i(t+1) \tag{5.2}$$

.

Note that state consistency and transition consistency are the necessary and sufficient conditions of a feasible trajectory (Def. 3.13) rewritten as a set of constraints. The state consistency constraint corresponds to the third requirement of CCA execution in

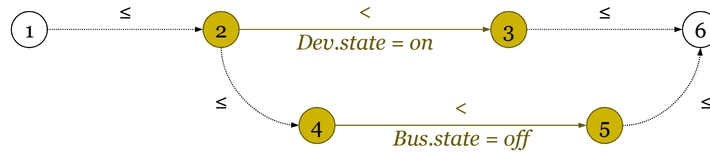


Figure 5-4: An example of a QSP for CCA *CompiledBus1Dev1*.

Def. 3.12. Intuitively, this condition ensures that the flow of information from output y to input u is consistent with the state constraint specified by the CCA. The transition consistency constraint corresponds to the execution requirement of constraint automata in Def. 3.6. Intuitively, this condition ensures that the transition from s_k to s_l is consistent with the transition relation of each constraint automaton.

5.2.2 Solution within a Trellis Diagram

A path within the trellis diagram is a solution to the CCA planning problem $\mathcal{P}^{CCA} = \langle \mathcal{M}, s_0, \mathcal{QSP} \rangle$ if and only if the path is rooted at a vertex corresponding to initial state s_0 and achieves qualitative state plan $\mathcal{QSP}$. For example, consider a planning problem with CCA $\mathcal{M} = \mathit{CompiledBus1Dev1}$, initial state $s_0 = \{Bus.state\ = \mathit{off}, Dev.state\ = \mathit{off}\}$, and QSP $\mathcal{QSP}$ depicted in Fig. 5-4. In Fig. 5-5, the path traced in bold is an example of a solution to the planning problem. Note that the path is rooted (time point t_0) at a vertex that corresponds to the initial state. Also, states $\{Bus.state\ = on, Dev.state\ = on\}$ and $\{Bus.state\ = \mathit{off}, Dev.state\ = on\}$ achieve goal $Dev.state\ = on$ at time points t_4 and t_5, and state $\{Bus.state\ = \mathit{off}, Dev.state\ = on\}$ achieves goal $Bus.state\ = \mathit{off}$ at time point t_5. Also as required by the QSP, the solution achieves goal $Bus.state\ = \mathit{off}$ after goal $Dev.state\ = on$ has been achieved.

Recall that a qualitative state plan (QSP) is a 5-tuple $\langle X, D, T, \mathcal{C}^t, E \rangle$, where set of episodes E specifies a set of desired goals and temporal constraint function $\mathcal{C}^t$ specifies a partial ordering by which the goals should be achieved. Thus, a path achieves QSP $\mathcal{QSP}$, if the path visits a set of vertices that satisfy the goals of E and visits them in an order that satisfies the partial order specified by $\mathcal{C}^t$. Formally, goal $g \in E$ of the QSP is achieved if:

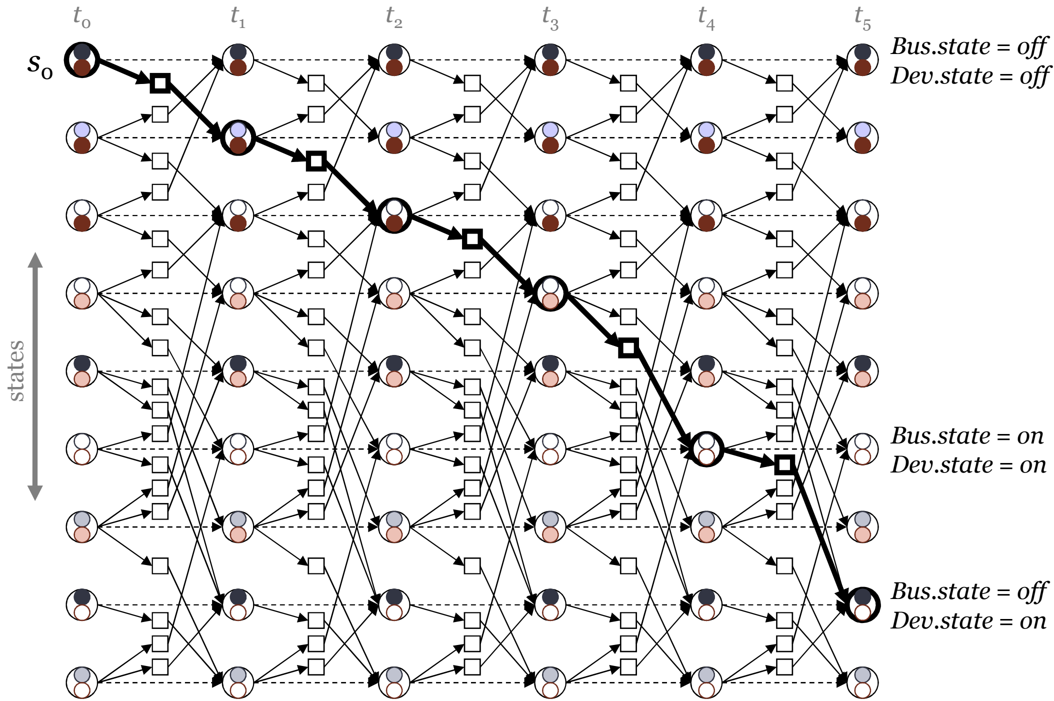


Figure 5-5: A solution within the trellis diagram for a planning problem involving CCA *CompiledBus1Dev1*.

Goal Achievement State $s_k(t)$ and input $u(t)$ satisfies the goal:

$$g(t) \wedge \mathcal{I}(t) \wedge \bigwedge_{q_i(t) \in s(t)} \mathcal{Q}_i(q_i(t)) \tag{5.3}$$

The goal achievement constraint simply checks to see if their exists an output such that the current state, input and output satisfy the goal.

5.3 Decomposed Planning

Constructing and searching trellis diagram G, however, poses three issues. First, enumerating all vertices of G, which is equivalent to enumerating all states of a CCA, is intractable, as discussed earlier. The number of vertices grow exponentially with the number of constraint automata in a CCA. Second, we must enumerate all models of state consistency and transition consistency constraints in order to construct all edges of graph G. Finally, to determine the goal states, we must either check the satisfiability of the goal achievement constraint on each branch of the search or enumerate all models

of the goal achievement constraint. Enumerating all edges and goals is also intractable given the exponential nature of the number of states, and the fact that in the worst case, the number of outgoing edges from a vertex can be as large as the number of possible control inputs ($|\Sigma(X^c)| = \prod_{x \in X^c} |\mathcal{D}(x)|$, where X^c is a set of control variables).

Computing graph G before searching for a path is an impractical approach to planning for most problems. Instead, we can compute only the relevant portion of the graph incrementally during the search. This is the approach taken by state space and plan graph planners that incrementally compute the reachability graph, which maintains only a portion of the trellis diagram that is reachable from initial state s_0. Nevertheless, the number of reachable states is still potentially exponential. Most state space planners, thus rely heavily on good heuristics to guide the expansion of the reachability graph. Regardless of the heuristic used for CCA planning, we must still perform constraint checking on each outgoing edge to verify state consistency, transition consistency and goal achievement.[1]

To mitigate these issues, we introduce a new method that relies on a combination of two decomposition techniques, constraint decomposition and causal order decomposition. The use of constraint decomposition allows us to maintain a set of reachable states compactly and to verify state consistency, transition consistency and goal achievement in an efficient manner. Constraint decomposition augmented with causal ordering decomposition provides the guidance for searching the reachability graph efficiently.

5.3.1 Decomposing the Trellis Diagram

We classify the three requirements, state consistency, transition consistency and goal achievement, into two main categories. One, state consistency and goal achievement constraints are requirements that must be checked at each time point. We can check these directly from their constraint formulation shown respectively in Eq. (5.1) and Eq. (5.3). Both constraints can be checked separately for each time point t. In com-

[1]Most planners are not concerned with constraint checking as they do not allow state constraints in their planning domain. As shown by [21], inclusion of state constraints increases the complexity of the planning problem.

parison, transition consistency is a requirement that must be checked over every pair of consecutive time points t and $t+1$. This is also reflected in the corresponding constraint in Eq. (5.2), which involves two consecutive time points t and $t+1$.

With this in mind, we decouple the planning problem into two parts. One, we find a path from the current state to a goal state while ensuring transition consistency over the path. Two, we ensure that each vertex of the path is a feasible state, that is, each vertex satisfies state consistency. If a vertex satisfies the goal achievement constraint, then the vertex is a goal. In order to solve the planning problem in an efficient manner, we check the consistency of each vertex as efficiently as possible and minimize the number of edges over which the planner branches. Constraint decomposition allows us to efficiently check the consistency of each vertex and goal serialization based on causal ordering allows us to minimize the branching of the search path.

Checking Vertex Consistency through Constraint Decomposition

Constraint decomposition transforms a constraint satisfaction problem into a binary acyclic network of constraint satisfaction problems, where each vertex of the acyclic network is a CSP and two connected CSPs are constrained by a binary constraint.

Definition 5.1 (Constraint Decomposition) Let $\mathcal{R} = \langle X, D, C \rangle$ be a CSP. A *constraint decomposition* for $\mathcal{R}$ is a triple $\mathcal{R}^D = \langle \mathcal{T}, \mathcal{X}, \mathcal{C} \rangle$, where

- $\mathcal{T} = \langle V, E \rangle$ is a tree.
- $\mathcal{X} : V \rightarrow 2^X$ maps each vertex $v \in V$ to a subset of variables $\mathcal{X}(v) \subseteq X$.
- $\mathcal{C} : V \rightarrow 2^C$ maps each vertex $v \in V$ to a subset of constraints $\mathcal{C}(v) \subseteq C$.

such that

- For each variable $x \in X$, if and only if $v_i, v_j \in \{v \mid x \in \mathcal{X}(v)\}$ and $i \neq j$, then $(v_i, v_j) \in E$.
- For each $c \in C$, there is at least one vertex $v \in V$ such that $c \in \mathcal{C}(v)$ and $\text{scope}(c) \subseteq \mathcal{X}(v)$.

Since the complexity of a binary acyclic CSP is known to be polynomial in the number of vertices of the acyclic network, the complexity of solving a decomposed CSP depends on the complexity of solving the individual CSPs that each vertex represents. The complexity of solving the individual CSPs depends on the number of variables. A measure called *width*, $w = \max_{v \in V} |\mathcal{X}(v)|$, is used to define the maximum number of variables associated with each CSP. Thus, the complexity of a decomposed CSP is exponential in its width. Depending on the domain of a problem, this width may be bounded by a constant. Thus, solving a decomposed CSP depends on the structure of the problem, not the size of the problem.

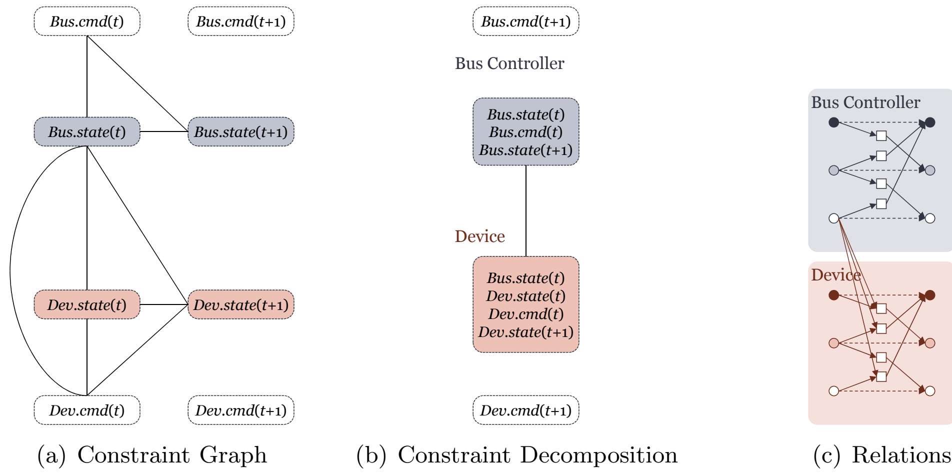
As long as the width of the decomposition is bounded, we can verify state consistency and transition consistency in a tractable manner. There are many decomposition methods, and some are better than others [13]. In this thesis, we are not concerned with developing the best decomposition method, but rather on the use of the technique, under the assumption that the width of the decomposition is bounded. Thus, we only need to define the mapping from consistency of state consistency and transition consistency constraints to a CSP. Then, the decomposition follows directly from the definition of constraint decomposition in Def. 5.1.

Given CCA $\mathcal{M} = \langle \mathcal{A}, X, \mathcal{I} \rangle$, the mapping of state consistency and transition consistency constraints to CSP $\mathcal{R} = \langle X, D, C \rangle$ is straight forward. The variables and domains of CCA $\mathcal{M}$ and CSP $\mathcal{R}$ are identical. We map state consistency and transition consistency constraints into a set of constraints C by first transforming state consistency and transition consistency constraints

$$\varphi = \mathcal{I}(t) \wedge \bigwedge_{i} \bigwedge_{q_i(t) \in Q_i} \left(q_i(t) \Rightarrow \mathcal{Q}_i\left(q_i(t)\right)\right) \wedge \bigwedge_{i} \bigwedge_{\delta} \left(q_i(t) \wedge \varphi_i(t)\right) \Rightarrow q_i(t+1) \tag{5.4}$$

into conjunctive normal form (CNF), denoted CNF(φ). Each clause in CNF(φ) is a constraint $c \in C$ of the CSP.

Figure 5-6(a) illustrates a constraint graph of CCA *CompiledBus1Dev1* mapped to a CSP. Figure 5-6(b) illustrates a constraint decomposition generated using tree decomposition method [6]. Finally, Fig. 5-6(c) graphically depicts the relations of the



(a) Constraint Graph (b) Constraint Decomposition (c) Relations

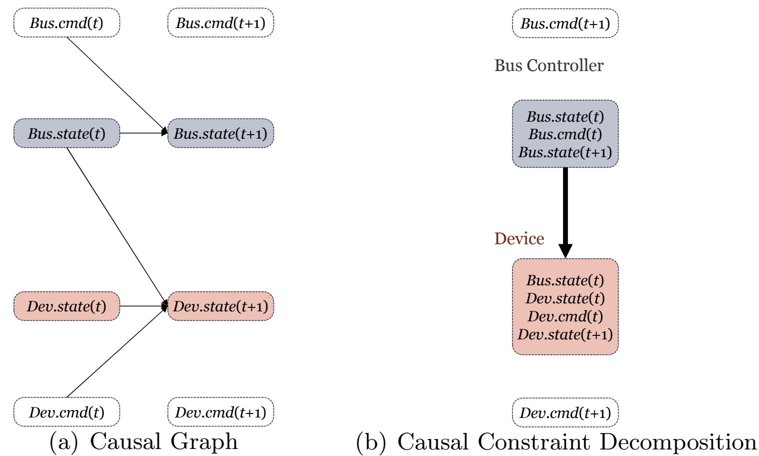
Figure 5-6: 5-6(a) is the constraint graph of the *CompiledBus1Dev1* model, 5-6(b) is a constraint decomposition of the constraint graph and 5-6(c) depicts the relations of the decomposition.

decomposition. The vertices in the first column of Fig. 5-6(c) represent the states at time point t and the vertices in the second column represent the states at time point $t+1$. The edges between two consecutive states represent feasible transitions.

Enable Subgoal Serialization through Causal Decomposition

Decomposition of $\mathcal{CA}$ is based on *subgoal serializability*, where a set of subgoals are serializable if and only if the subgoals can be solved sequentially [16]. [25] has recognized that if the *causal graph* is acyclic, then the subgoals are serializable. Building upon [25], [3] developed a causal decomposition method that allows the subgoals to be serialized even if the causal graph is cyclic. The causal decomposition method simply groups cyclic components of the causal graph such that the resulting meta graph is acyclic. We augment constraint decomposition with causal decomposition to enable subgoal serialization. We refer to the augmented decomposition as *causal constraint decomposition*. An example of causal constraint decomposition is depicted in Fig. 5-7. Figure 5-7(a) illustrates the causal graph of CCA *CompiledBus1Dev1*. With the information from the causal graph, we can construct a causal constraint decomposition shown in Fig. 5-7(b). In this case, there is no cycle, but note the directed edge on the decomposition.

Figure 5-7(a) illustrates a causal graph of CCA *CompiledBus1Dev1*.



(a) Causal Graph (b) Causal Constraint Decomposition

Figure 5-7: 5-7(a) is the causal graph of the *CompiledBus1Dev1* model and 5-7(b) is a causal constraint decomposition of the model.

A causal graph specifies cause and effect relationships between the variables of a CCA. There are thee types of cause and effect relationship: transition dependence, output dependence and interconnection dependence. The transition dependence specifies the temporal cause and effect relationship between the variables. The output dependence specifies the cause and effect relationship between the state and input variables and the output variables. The interconnection dependence specifies the cause and effect relationship between the output variables and the input variables as defined by the interconnection constraints of the CCA. Formally, a causal graph of CCA $\mathcal{M}$ is defined as follows:

Definition 5.2 (Causal Graph) The *causal graph* $G = \langle V, E \rangle$ of concurrent constraint automaton $\mathcal{M} = \langle \mathcal{A}, X, \mathcal{I} \rangle$ is a directed graph, where its vertices V correspond to the variables of CCA $\mathcal{M}$ of two consecutive time steps, $X(t)$ and $X(t+1)$. For each constraint automaton $\mathcal{A}_i$, a directed edge $(u, v) \in E$ is included in G if and only if one of the following two condition is met:

- Transition Dependence: $(u, v) \in \{(u, v) \mid u \in X_i^q(t) \cup X_i^u(t), v = x_i^q(t+1)\}$
- Output Dependence: $(u, v) \in \{(u, v) \mid u \in X_i^q(t) \cup X_i^u(t), v \in X_i^y(t)\} \cup \{(u, v) \mid u \in X_i^q(t+1) \cup X_i^u(t+1), v \in X_i^y(t+1)\}$

Also, for each interconnection constraint $c \in \text{CNF}(\mathcal{I})$, let $X^{y'} = \{x \mid x \in \text{scope}(c), x \in X^y\}$ be a set of output variables in the scope of the constraint and let $X^{u'} = \{x \mid x \in \text{scope}(c), x \in X^u\}$ be a set of input variables in the scope of the constraint. Then, a directed edge $(u, v) \in E$ exists if and only if:

- Interconnection Dependence: $(u, v) \in \{(u, v) \mid u \in X^{y'}(t), v \in X^{u'}(t)\} \cup \{(u, v) \mid u \in X^{y'}(t+1), v \in X^{u'}(t+1)\}$

For each constraint automaton $\mathcal{A}_i$, the transition dependence specifies that the value of the state variable of the next time step, $x_i^q(t+1)$, depends on the values of state variable $x_i^q(t)$ and input variables $X_i^u(t)$ of the current time step. Also for each constraint automaton $\mathcal{A}_i$, the output dependence specifies that the values of output variable X_i^y depend on the values of state variables x_i^q and input variables X_i^u. For each clause of the interconnection constraint in conjunctive normal form, $c \in \text{CNF}(\mathcal{I})$, the interconnection dependence specifies that the values of output variables $X^{y'} \subset \text{scope}(c)$ depend on the values of input variables $X^{u'} \subset \text{scope}(c)$. Note that if only transition dependence is used to construct the causal graph, then the resulting causal graph is identical to the transition dependency graph defined in [25, 3].

A causal constraint decomposition is computed as shown in Alg. 5.1. The algorithm first performs constraint decomposition (lines 1–2). Then, the constraint decomposition is augmented with the causal ordering (lines 3–15). Finally, causal graph decomposition is applied to form a causal constraint decomposition. While constraint decomposition [6] and causal decomposition [3] are well known, the causal ordering of constraint decomposition is novel to this thesis. Clique B is said to depend on clique A if and only if A and B are connected in the constraint decomposition (line 16) and there exists a variable in B, not in A, that is causally dependent on variables of A (lines 7–14). The connectedness (line 16) ensures that the two cliques are in someway related. The dependence relationship identifies the directionality of the causal ordering.

In the case of the constraint decomposition of *CompiledBus1Dev1* CCA shown in Fig. 5-6(b), the clique highlighted *Device* depends on the clique highlighted *Bus Con-*

Algorithm 5.1: ComputeCausalConstraintDecomposition($\mathcal{M}$)

```
   Input: A concurrent constraint automaton M = ⟨A, X, I⟩
   Output: ⟨T'', X', C'⟩
1  R ← CCAtoCSP(M);
2  ⟨T, X, C⟩ ← DecomposeCSP(R);
3  G ← ComputeCausalGraph(M);
4  G* ← ComputeTransitiveClosure(G);
5  E' = ∅;
6  for (u, v) ∈ E(T) do                                   ▷ T = ⟨V, E⟩
7      A ← X(u);
8      B ← X(v);
9      for x_i ∈ A do
10         for x_j ∈ B \ A do
11             if (x_i, x_j) ∈ E(G*) then                 ▷ G* = ⟨V, E⟩
12                 E' ← E' ∪ {(u, v)};
13                 break;
14         if (u, v) ∈ E' then break;
15 T' ← ⟨V(T), E'⟩;
16 ⟨T', X', C'⟩ ← ComputeCausalDecomposition(T', X, C);
17 return ⟨T'', X', C'⟩;
```

troller (see Fig. 5-7(b)) since $Bus.state(t)$ affects $Dev.state(t+1)$ according to the transition dependence. Note that the reverse is not true for the clique dependence.

Searching over the causal constraint decomposition is equivalent to searching decomposed trellis diagram. Extending the relations of the decomposition shown in Fig. 5-6(c) to $n+1$ steps forms a decomposed trellis diagram shown in Fig. 5-8.

5.3.2 Searching through the Decomposed Trellis Diagram

The following is the pseudo code of a decomposed planning algorithm. Intuitively, the algorithm chooses a goal (line 7) and finds a state trajectory to achieve the goal from the current state (line 9), while projecting the state evolution to the future to update the current state (line 15). The chosen state trajectory may have additional requirements, and these requirements are added to the list of goals to achieve (lines 12,14). Once no goal is left to achieve, the planning process successfully returns a plan in the form of a QCP (line 16). At any point, if a goal is not achievable, planning fails (lines 11–12).

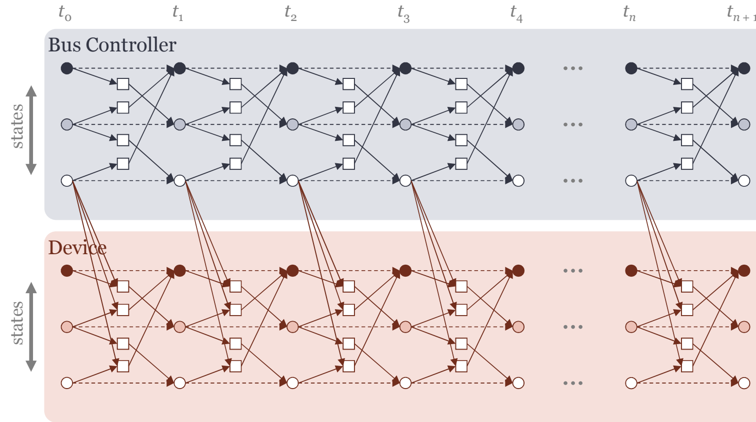


Figure 5-8: Decomposed trellis diagram of *CompiledBus1Dev1* model.

Algorithm 5.2: Plan($\mathcal{M}, s(0), \mathcal{QSP}$)

Input: A concurrent constraint automaton $\mathcal{M} = \langle \mathcal{A}, X, \mathcal{I} \rangle$, Initial state $s(0)$, Goal $\mathcal{QSP}$

Output: Qualitative Control Plan $\mathcal{QCP}$

```
1  eQCP ← InsertInitialState(QSP, s(0));
2  s(0 : t) ← ∅;
3  newGoals ← ∅;
4  R^DD ← ComputeCausalConstraintDecomposition(M)      ▷ R^DD = ⟨T, X, C⟩;
5  s_0 ← s(0);
6  while openGoals(eQCP) ≠ ∅ | newGoals ≠ ∅ do
7      g ← ChooseOpenGoal(eQCP);
8      c ← getComponent(g);
9      s(0 : t) ← ChooseTrajectory(s_0(c), g);
10     if s(0 : t) = ∅ then
11         return failed;
12     newGoals ← ChooseSubgoals(s(0 : t));
13     eQCP ← InsertTrajectory(eQCP, s_0(c), s(0 : t));
14     eQCP ← InsertNewGoals(eQCP, s_0, s(0 : t), newGoals);
15     s_0(c) ← s(t);
16 return QCP = proj_{X^c}(eQCP);
```

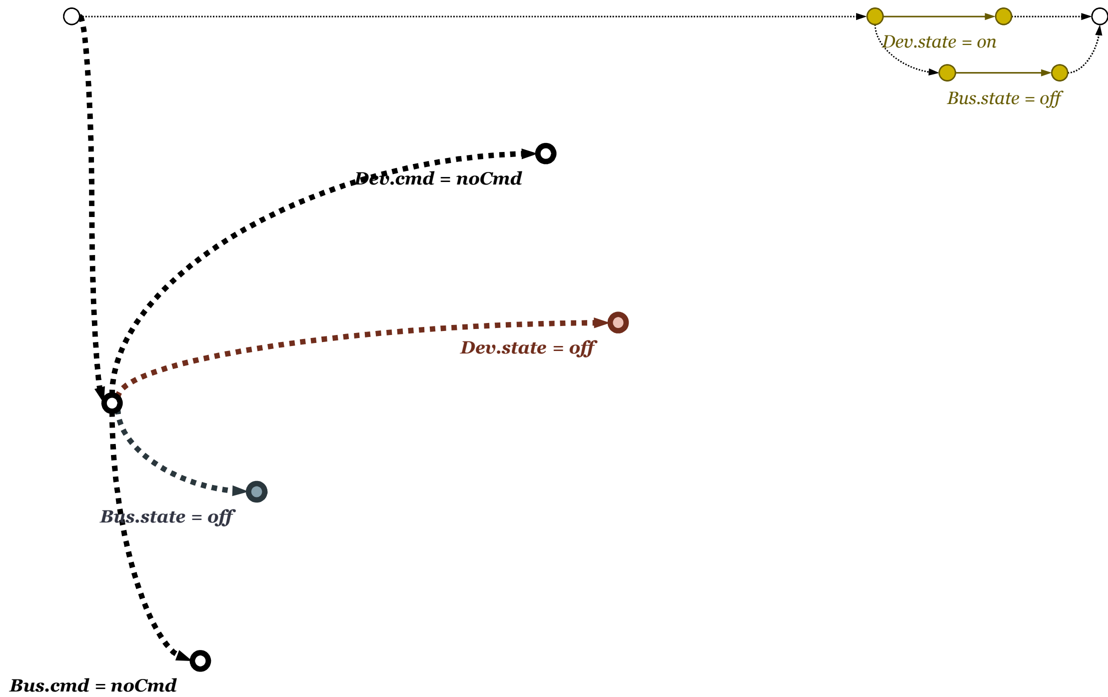


Figure 5-9: eQCP with the initial state inserted.

Insert initial state into elaborated QCP (line 1)

The first step after decomposition, is to insert the initial state into the elaborated QCP (eQCP).

Selecting an Open Goal (line 7)

Given a decomposed causal graph G^c, goals $e\mathcal{QCP}$ in the form of an elaborated qualitative control plan and current state s_0, select an open goal g in $e\mathcal{QCP}$ that is neither *causally* nor *temporally dominated.*

Definition 5.3 (Open Goal of eQCP) Given goal g in $e\mathcal{QCP}$, let c be the component related to goal g, $s_0(c)$ be the current state of component c, ep_i be the episode in $\mathcal{QSP}$ with current state $s_0(c)$ and ep_j be the episode in $e\mathcal{QCP}$ with goal g. A goal g of component c is *open* if and only if current state episode ep_i is guaranteed to end before goal episode ep_j starts, that is $end(ep_i) \leq start(ep_j)$.

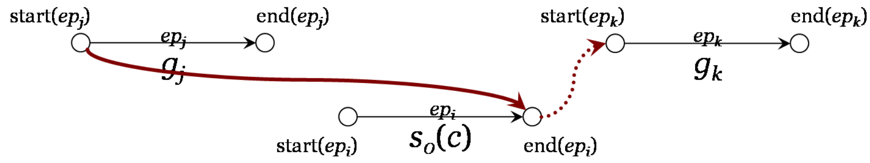


Figure 5-10: Example of a closed goal g_j and an open goal g_k.

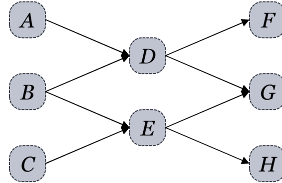
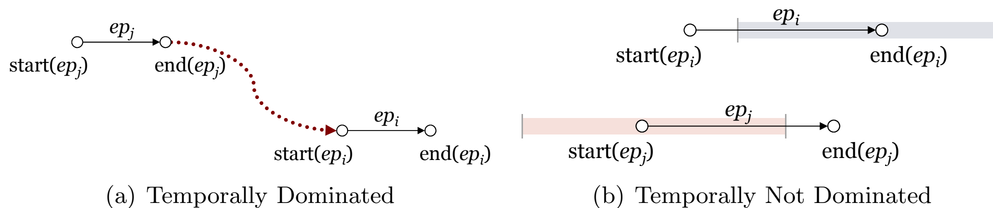


Figure 5-11: An example of a causal graph.

Definition 5.4 (Causally Dominated Goal) Given goals g_i and g_j in $e\mathcal{QCP}$, let c_i be the component for which goal g_i is specified and let c_j be the component for which goal g_j is specified. Goal g_i *causally dominates* goal g_j if and only if there exists a path from c_i to c_j in the decomposed causal graph G^c.

In Figure 5-11 component A dominates component F, but component A *does not dominate* component H.

Definition 5.5 (Temporally Dominated Goal) Given goals g_i and g_j in $e\mathcal{QCP}$, let ep_i be the episode in $e\mathcal{QCP}$ with goal g_i and let ep_j be the episode in $e\mathcal{QCP}$ with goal g_j. Goal g_i *temporally dominates* goal g_j if and only if episode ep_i is guaranteed to end before episode ep_j starts, that is $end(ep_i) \leq start(ep_j)$



(a) Temporally Dominated

(b) Temporally Not Dominated

Figure 5-12: In 5-12(a), ep_j temporally dominates ep_i and in 5-12(b) ep_j does not temporally dominated ep_i.

Figure 5-13: An example of a state trajectory that achieves the goal.

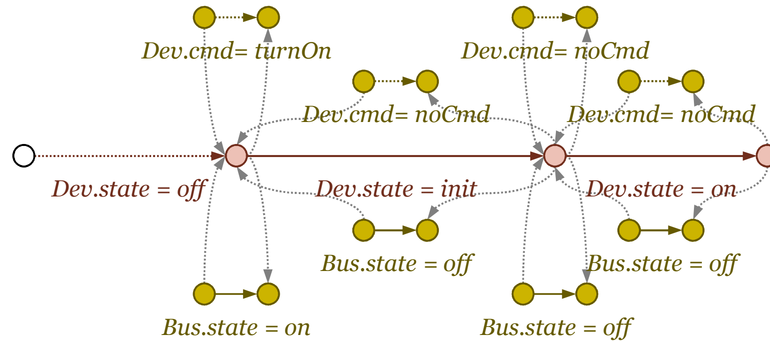


Figure 5-14: An example of extracting the subgoals of the trajectory in Figure 5-13.

Computing a Trajectory (line 9)

Given component c, current state $s_0(c)$ and goal g, compute a feasible trajectory, whose initial state is $s_0(c)$ and final state is g.

Note that this trajectory computation can be done using any planner, e.g. Europa, Burton. As long as each subproblem of the decomposition is small, we are able to precompile a policy for computing the trajectory, similar to the method used by Burton [25, 3].

Extracting New Goals (line 12)

Given component c and trajectory T, compute *time-evolved subgoals*, *newGoals*, that enable trajectory T. If a CCA does not have indirect effects in the form of state-constraints, the subgoals are simply the guard constraints of the transition. We must assure proper qualitative temporal ordering of the subgoals. The guard constraints of a transition into another state must be satisfied during the transition only. The guard constraints of an idle transition (transition into the same state) must be satisfied during the duration of the idling state.

Updating the Plan

If the trajectory generated is a command trajectory, insert it into $e\mathcal{QCP}$.

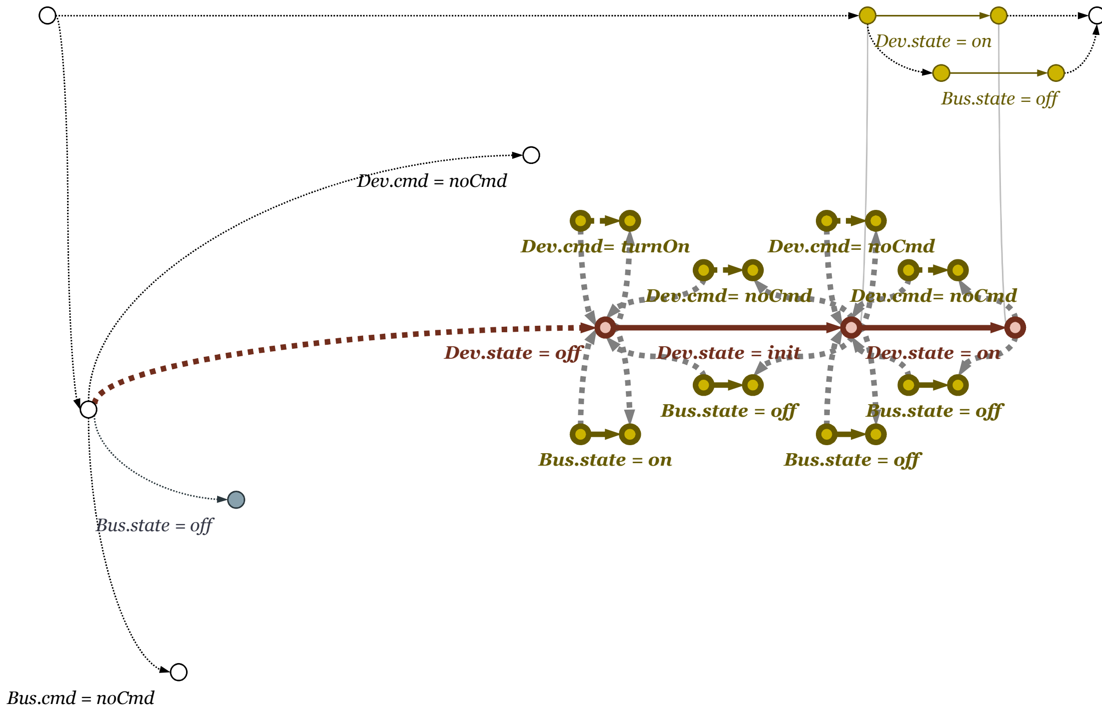


Figure 5-15: eQCP with the subgoals for device trajectory from off to on state.

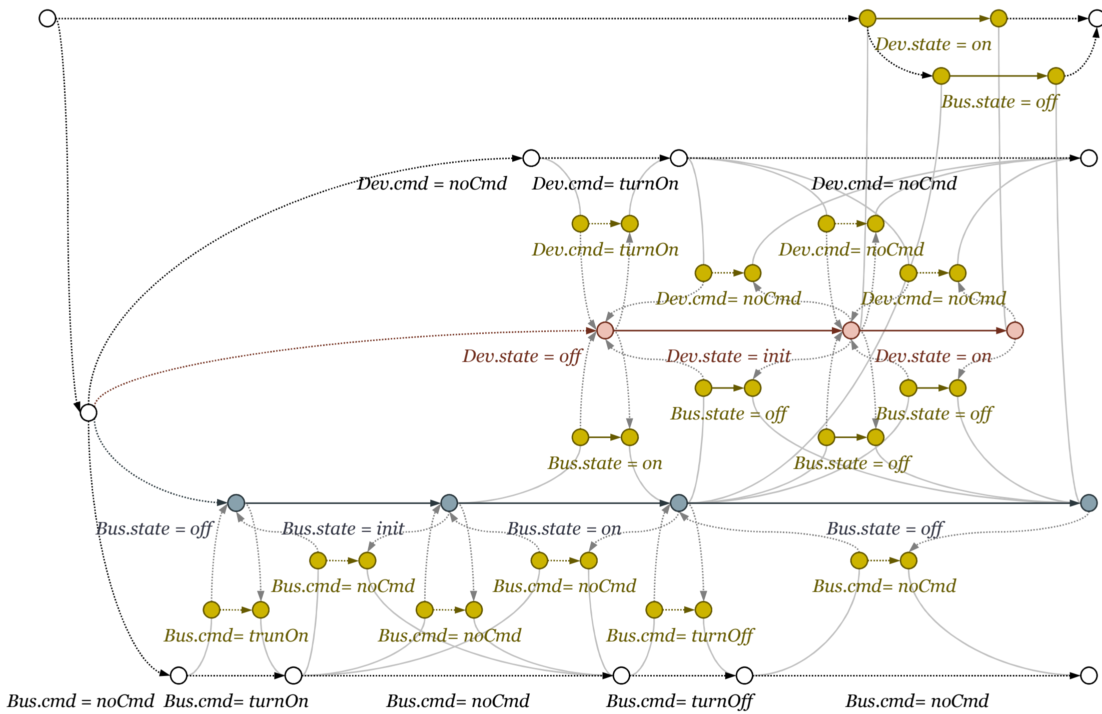


Figure 5-16: Elaborated qualitative control plan (eQCP) for QSP shown in Fig. 4-2.

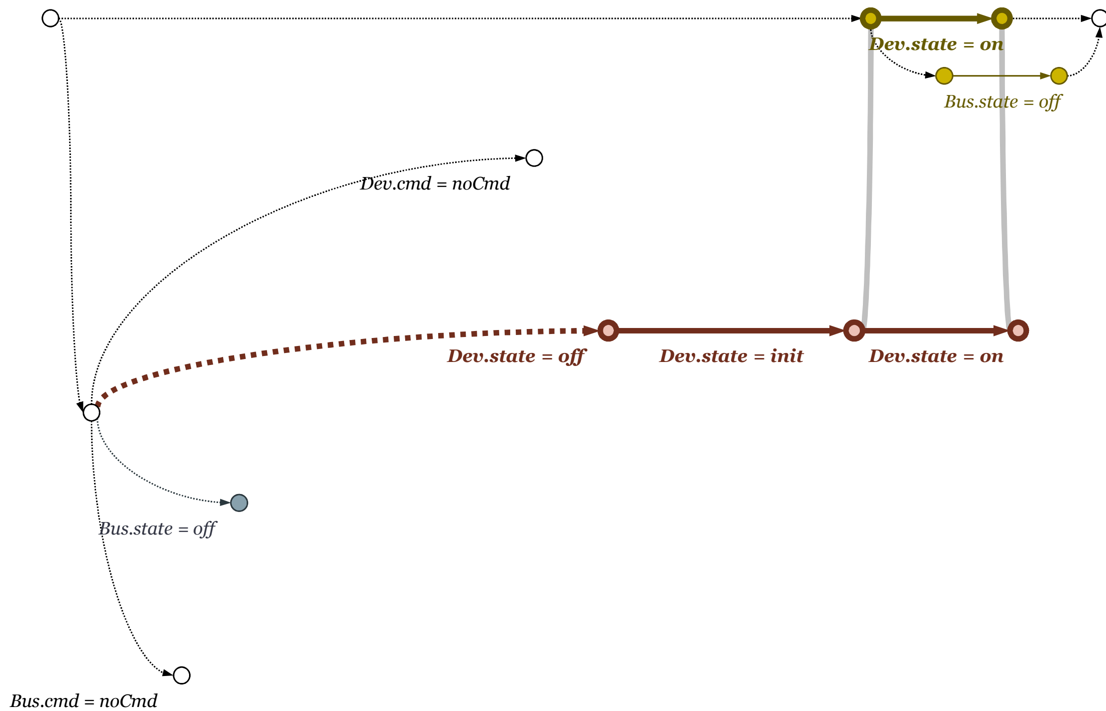


Figure 5-17: An example of updating eQCP with the state trajectory in Figure 5-13.

5.3.3 Extending to Temporal Planning

Extension of the aforementioned algorithm to temporal planning is straight forward. Given a timed CCA (TCCA) and a quantitative time QSP, or simply a QSP, the eQCP becomes augmented with the temporal constraints of TCCA and QSP. The algorithm will require an additional step between lines 14 and 15 that computes temporal consistency check of the updated eQCP. An incremental temporal consistency checking algorithm of [15] can be used for efficiency.

5.4 Related Work

A planning graph essentially maintains the same information as the reachability graph, but more compactly by relaxing the condition for reachability. A planning graph is essentially a linear directed tree, in which each vertex represents a set of all states that are *possibly* reachable from the predecessor vertex. Due to the relaxation, however, not all states of the vertex are guaranteed to be reachable. A goal is reachable if

one of the vertices includes the goal state. However, even if a vertex includes a goal, that does not guarantee that a path exists from the initial state to the goal state. Furthermore, additional search is required to verify that the goal is indeed reachable. Thus, the compactness of the planning graph comes at a cost of insufficient condition for reachability and additional search step required to check the reachability.

5.4.1 State Constraint

As an example, [21] has shown that allowing derived predicates, also known as indirect effect [18, 10, 20] and ramification problem [19, 9], increases the theoretical complexity of a planning problem to EXPTIME-complete.

Chapter 6

Conclusion

Contents

6.1 Implementation **83**
6.2 Results on EO-1 Mission Ops Scenario **83**
6.2.1 CCA of EO-1 Mission Ops System 84
6.2.2 Initial State and QSP 86
6.3 Experimental Results on EO-1 Scenario **86**
6.4 Approach & Innovation **88**
6.4.1 The Importance of Formal Model Specification 90
6.4.2 Capturing Operator's Intent through Goal-directed Plan 90
6.4.3 The Decomposition Approach 91
6.5 Future Work **92**
6.5.1 Decision Theoretic Planning with the Decomposed Planner . . 92
6.5.2 Distributed Planning with the Decomposed Planner 92

6.1 Implementation

The prototype of the decomposed planning algorithm was developed on Matlab R14. As such, the performance of the algorithm is expected to be lower in comparison to what is achievable from more optimized implementation written in C/C++. The problems were solved on a laptop with 1700 MHz Pentium M Processor and 1 GB Ram on 600 MHz bus.

6.2 Results on EO-1 Mission Ops Scenario

The decomposed planner has been verified on an Earth Observing-1 (EO-1) mission operations scenario. The scenario consists of a set of software components that generate a binary plan file and upload to the EO-1 spacecraft. The uploaded binary plan file is

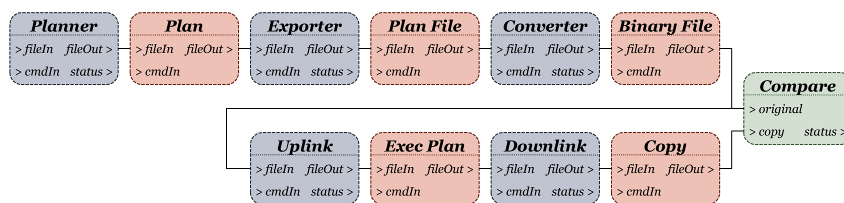


Figure 6-1: Concurrent constraint automaton of EO-1 scenario.

then downlinked and compared against the original binary plan file. The main purpose in benchmarking the performance against the EO-1 scenario is to demonstrate the applicability of decomposed planning on a scenario that represents a real world problem. As is with many complex systems, the EO-1 problem has a large state space with well over ten sextillion states, 1×10^{22}. Representative of most engineered systems, the components of EO-1 mission operations system are loosely coupled. The EO-1 mission operations scenario is as follows:

> Initially the Aspen automated planner, or simply the *planner*, generates a *plan*. The plan is read in by an *exporter* and processed into a *plan file*. Then a *converter* compresses the plan file into a *binary file*. Via the *uplink* communication, the binary file is then uploaded onto EO-1 as an *executable plan*. Once EO-1 receives the upload, the *downlink* communication downloads a *copy* of the binary file. The downlinked copy is then *compared* against the original uplink binary file before executing the uplinked plan.

The schematic of the concurrent constraint automaton for EO-1 scenario is shown in Fig. 6-1. Note that most of the components are serially linked.

6.2.1 CCA of EO-1 Mission Ops System

While each step of EO-1 mission operation has its specific purpose, the components of the process can be categorized into three basic components: File transformer and file compare components. From the above description, plan, plan file, binary file, executable plan and copy are all types of files. Planner, exporter, converter, uplink and downlink are all types of file transformers. Finally, the uplink and downlink files are compared

using a file compare component. Accordingly in Fig. 6-1, file transformers are colored blue, files colored red, and a file compare is colored green.

File Transformer Constraint Automaton

A file transformer is an abstraction on a software that reads data, processes the data and then writes the processed data. An example of a file transformer is an automated planner called Aspen. An automated planner reads in a file with a planning problem, plans and then outputs the generated plan as a file. A file transformer has five states, $\{idle, transforming, writing, done, failed\}$. When a file transformer is in the idle state, the component is not performing any task. Once the file transformer is commanded to *transform*, the file transformer transitions into the transforming state. For the component to perform transformation, the input file *fileIn* must be in a good state. Once the transformation is complete, the component transitions into the write state, in which the component is writing the output to *fileOut*. During the writing state, the output is corrupt. Once writing is complete, the component transitions to the done state. When in done state, the output file is guaranteed to be good. If the input file becomes corrupt or nothing, however, the component transitions into the idle state. At any time, a file transformer may be forced into the idle state by commanding it to *terminate* via the command input *cmdIn*. A file transformer component may also fail at any point, but may be recovered into the idle state by providing a new input or into the transforming state by commanding it to transform. The state of a file transformer is reported through the *status* output. The status is *idle* when the file transformer is in either idle or failed state. Otherwise, the status is reported as *running*.

File Constraint Automaton

A file component models a file on a read/write memory. A file component has three states, $\{good, nothing, corrupt\}$. If good data is written via *fileIn*, a file will be in the good state. If a corrupt file is written, the file will be in the corrupt state. If a file is corrupt, it will remain corrupt until commanded to *delete* or rewritten with a good file. A file in a good state, however, may degrade overtime to the corrupt state. A file

may transition to the nothing state if a file is deleted via the command *cmdIn*. A file component has one output *fileOut* through which other components may read the file. The output directly reflects the state of the file component.

File Compare Constraint Automaton

A file compare component models software that performs comparison on two files. Given two input file *orignalFileIn* and *copyFileIn*, a file compare component will check to see if the two files are in the same state. The component has two states $\{nominal, failed\}$. When the component is in the nominal state, it outputs via *status* whether the two files are *same* or *different*. When the component is in the failed state, it may output any value regardless of the states of the two input files.

6.2.2 Initial State and QSP

Initially all files are assumed to be in the nothing state, all file transformers are in the idle state and the file compare component is in nominal state. Initially, no commands are issued and the planner is provided with a good file.

The goal for the EO-1 scenario is simply to downlink a good file.

6.3 Experimental Results on EO-1 Scenario

One of the difficulties in automated planning is the size of the problem or the number of states. In this scenario, there is total of ten components, five file components, five file transformer components, and one file converter component. A file component has three possible modes, three possible values to each of two inputs and one output. This implies a file component has 54 states. Similarly a file transformer component has a total of 270 states. A file compare component has total of 36 states. Thus, as a system there are over 1×10^{22} states. Nevertheless, the structure of the problem allows us to decompose the state space and search the reduced decomposed state space.

We have tested decomposed planning on several variations of the EO-1 scenario. The size of the problem was varied by incrementally increasing the number of file transformer

Table 6.1: The parameters of the test cases. Each of the test cases are incremental variations on the EO-1 scenario.

Number of Components	2	4	6	8	10
Number of States	1.5×10^4	2.1×10^8	3.1×10^{12}	4.5×10^{16}	6.6×10^{20}
Number of Cliques	13	27	41	55	69
Maximum Clique Size	4	4	4	4	4

Table 6.2: Time performance for various steps of decomposed planning.

Number of Components		2	4	6	8	10
Offline	Decomposition Time (sec)	14	83	243	523	973
	Policy Computation Time (sec)	22	107	299	628	1153
Online	Subgoal Extraction Time (sec)	1.3	14	58	155	344
	Goal Ordering Time (sec)	0.5	5	23	58	128
	Trajectory Computation Time (sec)	0.5	6	22	50	98
	Total Planning Time (sec)	3.8	34	127	315	670

and file chain. The simplest case includes two components, one file transformer and one file linked serially. This link is increased up to 10 components, which includes five file transformers and five files interleaved and linked serially. The 10 component case is equivalent to EO-1 scenario, but only without the compare component. The parameters of each case is shown in Table 6.1. Note that the number of states increases exponential with the number of components, ranging from 150 thousand states to 660 quintillion states. As expected the number of cliques grows linearly with the number of components and the maximum clique size is constant.

Table 6.2 lists the amount of time used in various steps of decomposed planning. The time performance is categorized in to two main parts, offline and online. The memory usage for the two important outputs of the offline step, decomposition and policy, is listed in Table 6.3. Decomposition and policy are precompiled offline to reduce online computation time.

Note that almost half of the time is spent on subgoal extraction. Recall that subgoal

Table 6.3: Memory performance for storing decomposition and policy information.

Number of Components	2	4	6	8	10
Decomposition Memory Size (kB)	108	220	332	444	556
Policy Memory Size (kB)	19	38	58	77	96

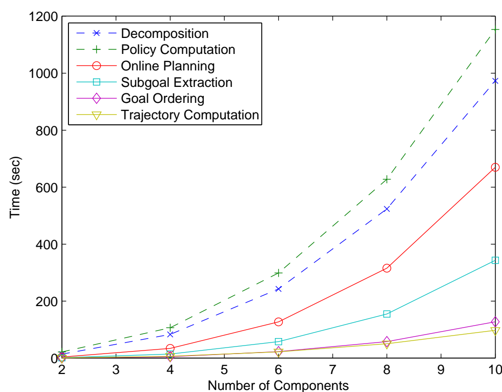


Figure 6-2: Amount of time spent on various steps of the decomposed planning algorithm for varying EO-1 scenario problem size.

extraction is equivalent to implicant generation time. The current implementation computes all possible implicant and chooses one. The time spent on subgoal extraction can be reduced by computing one at a time instead of all at once. The growth in memory usage is linear in number of components as expected.

To visualize the trend on time performance, the data is plotted as shown in Fig. 6-2. As shown in Fig. 6-3, the time performance increases polynomially. This polynomial growth is expected since for each addition of a component, the number of goals increase by a constant factor. That is, a trajectory that achieves a goal will have a length that is bounded by the maximum number of acyclic transitions feasible for the component, and each step of a trajectory will add a constant number of subgoals. In effect, the time performance grows linearly with the expected plan length. A polynomial time bound in the number of decomposed component is a dramatic improvement over the worst case polynomial time bound in the number of states, where in EO-1 scenario there are 1×10^{22} states.

6.4 Approach & Innovation

We improve on the current approach to sequencing the engineering operations of a spacecraft or ground-based asset through the explicit use of verifiable models and state-of-the-art goal-directed planning algorithms. We have developed a model-based decomposition

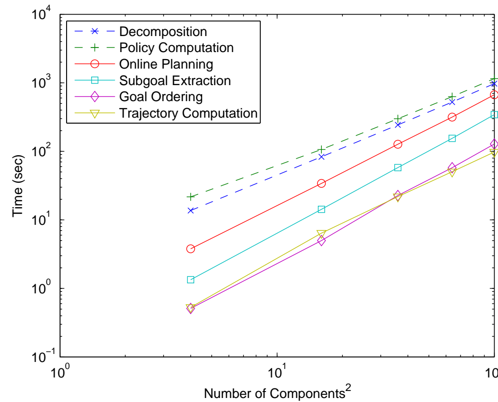


Figure 6-3: Amount of time spent on various steps of the decomposed planning algorithm for varying EO-1 scenario problem size. The data has been linearized by comparing time against the square of the number of components.

planner that generates an executable sequence, based on the behavior specifications of the components of a system. We leverage lessons learned from current operations of systems like MER and the DSN Monitor & Control (M&C) system, in the hope of significantly improving the operations efficiency of future JPL missions, reduce costs and increase the likelihood of mission success.

Based on the specifications of the system behavior, the decomposed planner produces sequences of engineering operations that achieve mission objectives specified by the operator. More specifically:

1. We specify the model of system behavior as concurrent, constraint automata (CCA) that provide the expressiveness necessary to model the behavior of the system components and their interactions.

2. We describe the mission objective as a desired evolution of goal states called qualitative state plan (QSP), explicitly capturing the intent of the operators, rather than implicitly capturing it in a sequence of commands and procedures that achieve the desired goals.

3. We use a "divide-and-conquer" approach, in which an offline reasoning algorithm is used to determine the proper decomposition of the problem and the ordering

by which the decomposed problems should be solved.

4. We use the decomposed problem to solve a planning problem online. The use of precomputed decomposition reduces, and possiblly eliminates, search during the online planning step. Inherent to the decomposed planning, the plan generated, called qualitative control plan (QCP), is partially-ordered. QCP provides additional flexibility and robustness in comparison to totally-ordered plans.

6.4.1 The Importance of Formal Model Specification

The conventional approaches to systems and software engineering inadvertently create a fundamental gap between the requirements on software specified by systems engineers and the implementation of these requirements by software engineers. Software engineers must perform the translation of requirements into software code, hoping to accurately capture the systems engineer's understanding of the system behavior, which is not always explicitly specified. This gap opens up the possibility for misinterpretation of the systems engineer's intent, potentially leading to software errors.

Unlike other planning and sequencing systems, our approach directly exploits engineering models of system component behavior to compose the plan. Specifying the system and software requirements as a formally verifiable specification called concurrent, constraint automaton (CCA) allow the system behavior specification to be directly used to automatically generate sequences. The state constraints of CCA are used to specify the behavior of the states of each component and the interconnection constraints are used to specify the interactions of the components. The transitions and transition guards specify the dynamics of the components.

6.4.2 Capturing Operator's Intent through Goal-directed Plan

The ability to explicitly capture the intent of the operators, rather than implicitly capturing it in a sequence of commands and procedures that achieve the desired goals, is crucial for sequence verification. To enable verifiability of automatically generated sequences, the model-based temporal planner assures that each sequence can be traced

directly to the mission objective and/or operator's intent. Generally, mission objective and operator intent can be described explicitly as an evolution of goal-states. Thus, for model-based planning, the mission objective and operator's intent is described as a desired evolution of goal states called QSP. Then, the decomposed planner uses a goal-directed planning method to elaborate each goal into a sequence, while explicitly associating each sequence to a goal state, that is the intent of the operator and mission objectives.

6.4.3 The Decomposition Approach

The "divide-and-conquer" approach that leverages the structure of the component interactions to simplify the planning problem ensures the tractability of planning, even during time-critical situations. This approach is innovative in that it unifies the existing decomposition techniques used in reactive planning and constraint satisfaction problems. The causal graph decomposition used in reactive planning enables the decomposed planner to determine the proper goal ordering. Using the topological ordering of a causal graph decomposition allows the decomposed planner reduce, and possibly eliminate, the search branching on feasible goal orderings. The causal graph decomposition effectively allows the goals in QSP to be divided and ordered into a set of goals that can be solved sequential without the need to backup to try a new ordering.

An additional search required is in determining the subgoals required for a trajectory chosen to achieve a goal. With no indirect-effects of the state constraints and interconnection constraints, the subgoals are simply the transition guards of the transition. With indirect-effects, however, we must identify how the chosen trajectory could affect components through state and interconnection constraints. Solving for all indirect effects of a trajectory requires a search through the state and interconnection constraints. Nevertheless, we are able to eliminate the search for the require indirect effects through the use of constraint graph decomposition. Constraint graph decomposition decomposes the state and interconnection constraints such that only the portion that is pertinent to the trajectory chosen is solved without the need for a search.

6.5 Future Work

In this thesis, we developed a planning capability to address the real world problem of controlling devices, but limited to the expressivity of time and state constraints. In this section we describe a way to extend the work to address the stochastic nature of the real world. We also describe a way to distribute the processing to enable distributed planning.

6.5.1 Decision Theoretic Planning with the Decomposed Planner

While many highly reliable devices can be viewed as deterministic systems, for unreliable devices, we must model them as stochastic devices. Stochasticity of a system can be modeled as a CCA by introducing nominal and failure state as well as probabilistic transition. With probabilities on transitions, however, the trajectories computed by the decomposed planner is also probabilistic. Thus, we can compute the probability of the QCP generated by the decomposed planner. Furthermore, the decomposed planner can choose a trajectory that is more likely, or, with a utility function, choose a trajectory that has higher expected utility. This implies that we can introduce an optimal search method, such as branch and bound or optimal A* search techniques, so that the decomposed planner generates a QCP with the highest utility.

Furthermore, the decomposed planner can compute the locations of a QCP that are most likely to fail. For the points of high failure probability, we could compute contingent plans that address the possibility of a failure. As such, the decomposed planner could be extended to solve a decision theoretic planning problem as well as contingent planning problem.

6.5.2 Distributed Planning with the Decomposed Planner

There are two reasons for distribution of planning. One of the reasons why one may desire a distributed planning is to modularize the system. One may wish to develop a simple micro processor that is capable of planning for only one device. Such device along with the microprocessor and planning software may be flight qualified separately from

other devices. In such cases, a device of a system may be replaced with another device without the need to flight qualify a new centralized planning software that is capable of supporting the new device. Instead, if the device, micro processor and the planner is self contained, the flight qualification and verification process could be simplified.

As described in the decomposed planner planning algorithm, the trajectory is computed individually for each component, without the need for the information of other components. This allows us to distribute the trajectory computational portion of the algorithm to each of the components. Once trajectory computation is distributed, each component simply needs to pass the required subgoals to the corresponding components of the subgoals. One difficulty in distributing the planning process, however, is in maintaining a consistent knowledge of the temporal constraints of the plan. This must be addressed by communicating the timing bound of the individual component's plans as the plan is generated. This should be similar to the distributed temporal consistency methods used in distributed execution of temporal plans [1].

Bibliography

[1] Stephen A. Block and Brian C. Williams. Robust execution of contingent, temporally flexible plans. In *Proceedings of the AAAI Workshop on Cognitive Robotics*, pages 25–32, Boston, MA, July 2006.

[2] Tom Bylander. Complexity results for planning. In *Proceedings of the Twlefth International Joint Conference on Artificial Intelligence (IJCAI-91)*, volume 1, pages 274–279, Sydney, Australia, August 24–30 1991.

[3] Seung H. Chung. A decomposed symbolic approach to reactive planning. Master's thesis, Massachusetts Institute of Technology, June 2003.

[4] Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 627–634, Edmonton, Alberta, Canada, July 28–August 1 2002.

[5] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1):61–95, September 1991.

[6] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Inteligence*, 38(3):353–366, April 1989.

[7] Paul H. Elliott. An efficient projected minimal conflict generator for projected prime implicate and implicant generation. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, February 2004.

[8] Kutluhan Erol and Dana S. Nau V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1–2):75–88, July 1995.

[9] Debora Field and Allan Ramsay. Planning ramifications: When ramifications are the norm, not the problem. In *Proceedings of the Eleventh International Workshop on Non-Monotonic Reasoning (NMR'06)*, pages 343–351, Lake District, England, May 30–June 1 2006.

[10] Hector Geffner. Causality, constraints and the indirect effects of actions. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, volume 1, pages 555–561, Nagoya, Japan, August 23–29 1997.

[11] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning Theory and Practice*. Morgan Kaufmann Publishers, San Francisco, California, 2004.

[12] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 21–32, Pittsburgh, Pennsylvania, May 31–June 2 1999. ACM Press.

[13] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artificial Inteliigence*, 124(2):243–282, December 2000.

[14] Andreas G. Hofmann. *Robust Execution of Bipedal Walking Tasks From Biomechanical Principles*. PhD thesis, Massachusetts Institute of Technology, January 2006.

[15] I hsiang Shu. Enabling fast flexible planning through incremental temporal reasoning. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 2003.

[16] Richard E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65–68, September 1987.

[17] Thomas Léauté. Coordinating agile systems through the model-based execution of temporal plans. Master's thesis, Massachusetts Institute of Technology, August 2005.

[18] Fangzhen Lin. Embracing causality in specifying the indirect effects of actions. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 2, pages 1985–1991, Montreal, Quebec, Canada, August 20–25 1995.

[19] Sheila A. McIlraith. Integrating actions and state constraints: A closed-form solution to the ramification problem (sometimes). *Artificial Intelligence*, 116:87–121, 2000.

[20] Murray Shanahan. The event calculus explained. *Artificial Intelligence Today: Recent Trends and Developments*, 1600:409–430, 1999.

[21] Sylvie Thiebaux, Jorg Hoffmann, and Bernhard Nebel. In defense of pddl axioms. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 961–966, Acapulco, Mexico, August 9–15 2003.

[22] Marc Vilain and Henry Kautz. Constraint propagation algorithms for temporal reasoning. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, pages 377–382, Philadelphia, Pennsylvania, August 11–15 1986.

[23] Brian C. Williams, Michel D. Ingham, Seung H. Chung, and Paul H. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. In *Proceedings of the IEEE: : Special Issue on Modeling and Design of Embedded Software*, volume 9 of *1*, pages 212–237, January 2003.

[24] Brian C. Williams and P. Pandurang Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 971–978, Portland, Oregon, August 4–8 1996.

[25] Brian C. Williams and P. Pandurang Nayak. A reactive planner for a model-based executive. In M. Pollack, editor, *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1178–1185, Nagoya, Japan, August 23–29 1997. Morgan Kaufmann.