Fast Execution of Temporal Plans with Mixed Discrete-Continuous State Constraints

by

Jingkai Chen

B.E., Zhejiang University (2016)

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

Author ..... Signature redacted
Department of Aeronautics and Astronautics
June 1, 2019

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

Accepted by ..... Signature redacted
Sertac Karaman
Associate Professor of Aeronautics and Astronautics
Chairman, Graduate Program Committee

Fast Execution of Temporal Plans with Mixed Discrete-Continuous State Constraints

by

Jingkai Chen

Submitted to the Department of Aeronautics and Astronautics
on June 1, 2019, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

There has been a dramatic rise in networked embedded systems that play a central role in complex tasks. To achieve high performance and robustness, these systems need to configure and reconfigure on the fly, in light of the task requirements and system states. Communications networks, for example, plan routes and allocate bandwidth resources over time for different communication activities, while respecting throughput, delay, loss, and deadline constraints. However, existing approaches use simple discrete models to achieve goal sequences and thus cannot provide a high-fidelity plan for complex plan specifications in terms of time and state.

In this thesis, we deliver Amundsen, an efficient configuration manager that supports complex concurrent tasks over time and state by reasoning over high-fidelity models. These models can encode different actuation modes with discrete and continuous specifications and temporal influences. Amundsen provides plans meeting mission requirements, which specify the timing of events, outline the mode changes throughout the mission, and allocate resources. The primary challenge of the configuration management problem is the computation required to handle the state and time constraints that are highly coupled.

We address this challenge through the critical insight that the configuration management problem may be efficiently solved by dividing the problem into smaller subproblems such as scheduling and resource allocation, which is achieved by total ordering the events that represent time points in the goal specification. Each sub-problem may then be solved efficiently with existing highly optimized algorithms. We make two main technical contributions in this thesis. First, we identify the relevant subproblems in the existing configuration management problems and provide tractable encodings. Second, we provide an algorithm to efficiently order the stages of the problem by learning and communicating the requirements for successful solutions to the sub-problems.

We provide empirical evidence of the efficiency of Amundsen by benchmarking against UnifyHistory, a state-of-the-art solver to configure systems by unifying multiple timelines, on a communication network simulator. We show that our approach

can dynamically manage hundreds of network requests over a network with hundreds of communication links in simulated missions given strict time limits. Our approach is also able to find plans in 10 times as many scenarios as the baseline solver. The work described in this thesis thus significantly advances the configuration management problem, both theoretically and practically.

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

Acknowledgments

This thesis would not have been possible without the support of my mentors and friends during my time at MIT.

First and foremost, I would like to thank my advisor, Professor Brian Williams, for welcoming me into MERS and providing brilliant insights, support, and advice for my research. His guidance and detailed comments on drafts helped me go through every step of writing this thesis. I would also like to thank Dr. Howard Shrobe and Dr. Christian Muise for mentoring me over the past years.

I am thankful to all the members of the MERS group for sharing their genius with me. In particular, I would like to thank Simon Fang and Christian Muise, who worked with me on the EdgeCT project, and Yuening Zhang and Marlyse Reeves, who have been working with me on the Creative Problem Solver Project. It is also a great experience to work with Eric Timmons and other labmates to develop the constraint optimization solver OpSat. Thanks everybody who have provided detailed comments on this thesis: Andrew Wang, Simon Fang, Cyrus Huang, Yuening Zhang, Marlyse Reeves, and Nick Pascucci.

I am grateful to my family and friends at MIT and around the world for encouraging me, inspiring me, and sharing good times and hard times with me.

Finally, I would like to thank and acknowledge DARPA for sponsoring this work for two and a half years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A wide range of real-world problems involves configuring systems in light of tightly coupled time and state constraints. For example, an automated system commanding multiple underwater vehicles to explore the seafloor [33] must schedule multiple data-gathering tasks and configure the control signals during these tasks for vehicles given the presence of currents along with timing and spatial requirements on the tasks. As a second example, Livingstone, the configuration manager of the spacecraft Cassini, configures the components of its engines such as fuel tanks and valves over time, such that the desired thrust can be provided by the engines to navigate the spacecraft [39]. The third example is a mission-aware communication manager that must plan routes and allocate bandwidth resources for network flows with requirements on bandwidth, loss, delay, and deadlines. In both instances, the planner must decide on not only the time at which each activity or flow is executed but also the system settings during an activity. These decisions must also conform to constraints on timing, system dynamics, and goal specifications. Thus, there is a strong demand for automated systems to schedule tasks and configure system settings under a combination of discrete and continuous state constraints together with timing requirements.

In this thesis, we show that networked embedded systems can achieve high levels of performance through Amundsen, a timed configuration manager that reasons over hybrid models with discrete and continuous constraints.

This chapter motivates the problem of timed configuration management in hy-

brid discrete and continuous domains and presents Amundsen, a timed configuration manager as the solution approach. Amundsen is able to configure both discrete and continuous parameters of an embedded networked system to achieve desired behaviors over time. We begin by introducing scenarios that motivate us to develop a high-performance timed configuration manager. Then, we present this timed configuration manager that supports complex concurrent tasks and highlight several modules of this manager that are critical to the efficiency. We end this section by summarizing our contributions and the organization of the rest of this thesis.

1.1 Motivating Scenario

One example of a timed configuration management problem is network configuration. Judicially configuring a communication network and being mission-aware can enable it to transfer more data, especially when the link capacities fluctuate. To configure a communication network, a communication manager plans routes and allocates bandwidth resources for multiple network flows subject to throughput, delay, loss, and deadline constraints, given a temporal plan of flow modes. A network flow is a communication request, such as a data transfer request for a teleconference, and a flow mode is an actuation state such as transferring or off. More specifically, the manager generates routes satisfying loss and delay requirements and allocates usable bandwidth for each flow over time. For example, a File Transfer Protocol (FTP) flow, which demands a high bandwidth channel, and a Voice over Internet Protocol (VoIP) flow, which demands low delay, will be provided with different routes. In addition, configuring network flows requires the manager to reason over temporal constraints in order to fulfill deadlines and precedence relations. For example, two flows that cannot transfer concurrently need to be scheduled at different times, such that both of them can complete.

Our example scenario is modeled after a communication mission between four stations. The network topology is shown in Figure 1-1, and the loss, delay, bandwidth capacity, and available duration of each link are given in Table 1.1. Table 1.2 shows

the requirements of each network flow that is described in terms of the duration of communication, minimum allowable bandwidth, and maximum allowable loss rate and delay. In addition to the communication durations, which are specified as temporal constraints, FTP-A must complete at least 10 seconds and at most 15 seconds before FTP-B.

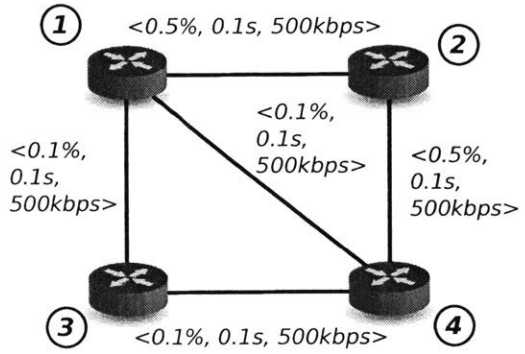

Figure 1-1: Network topology and link characteristics.

Table 1.1: Link characteristics and available durations.

ID	Link	Loss	Delay	Bandwidth	Available Duration
1	1-2	0.5%	0.1s	500kbps	[0, 8]mins
2	2-1	0.5%	0.1s	500kbps	[0, 8]mins
3	1-3	0.1%	0.1s	500kbps	[4, 8]mins
4	3-1	0.1%	0.1s	500kbps	[4, 8]mins
5	1-4	0.1%	0.1s	500kbps	[0, 4]mins
6	4-1	0.1%	0.1s	500kbps	[0, 4]mins
7	2-4	0.5%	0.1s	500kbps	[0, 8]mins
8	4-2	0.5%	0.1s	500kbps	[0, 8]mins
9	3-4	0.1%	0.1s	500kbps	[4, 8]mins
10	4-3	0.1%	0.1s	500kbps	[4, 8]mins

Given the temporal constraints, the requirements of network flows, the network topology, and the link characteristics, a communication manager should be able to provide a temporal plan that specifies the configurations of the system over time, where a configuration is a route and bandwidth allocation along this route for each

Table 1.2: Requirements of network flow requests.

ID	Flow	Source	Sink	Loss	Delay	Throughput	Data	Duration
1	*VOIP*	1	4	0.5%	0.3s	200kbps	72000kb	[360, 400]s
2	*FTP-A*	3	4	3%	1s	360kbps	64800kb	180s
3	*FTP-B*	2	3	3%	1s	360kbps	14400kb	40s

flow. An example of this temporal plan is shown in Figure 1-2. The reasoning process to obtain this plan is as follows: because VoIP's duration is more than 6 minutes, we cannot fit it into either four-minute interval of the entire eight-minute horizon. In the first four minutes, path 1-3-4 is broken, and path 1-2-4 has too high loss rate. Hence the only available path for this VoIP is directly from 1 to 4. In the second four minutes, since path 1-4 is broken, VoIP should be routed to path 1-3-4. We can only start the flow in the second four minutes because path 3-4 and path 3-1 are broken. Hence no flow can go out of 3. In the second four minutes, VoIP only leaves 300kbps bandwidth on link 1-4, which is not enough for any FTP, therefore FTP-A can be only routed to path 3-1-2-4. Given the temporal constraints of FTP-B, it is the last one to transfer, and two routes are available.

The above reasoning process is challenging given the mixed discrete and continuous nature of the constraints. More specifically, choosing the route for each flow is a discrete problem, and reasoning over bandwidth capacity requires dealing with continuous variables. Another significant part of this problem is the presence of time such as temporal constraints, and the state constraints and temporal constraints are highly coupled. For example, flows can be either scheduled to different times or routed to different paths to save resources for other flows.

These features are also common in other problems beyond network configuration. For example, to configure a fleet of underwater vehicles, we need to decide on the task assignment over vehicles, which are discrete, and the motion trajectories and timing constraints, which are continuous.

To capture these problem features, we define a new class of problems called timed configuration management problems (TCMP), where systems with both discrete and

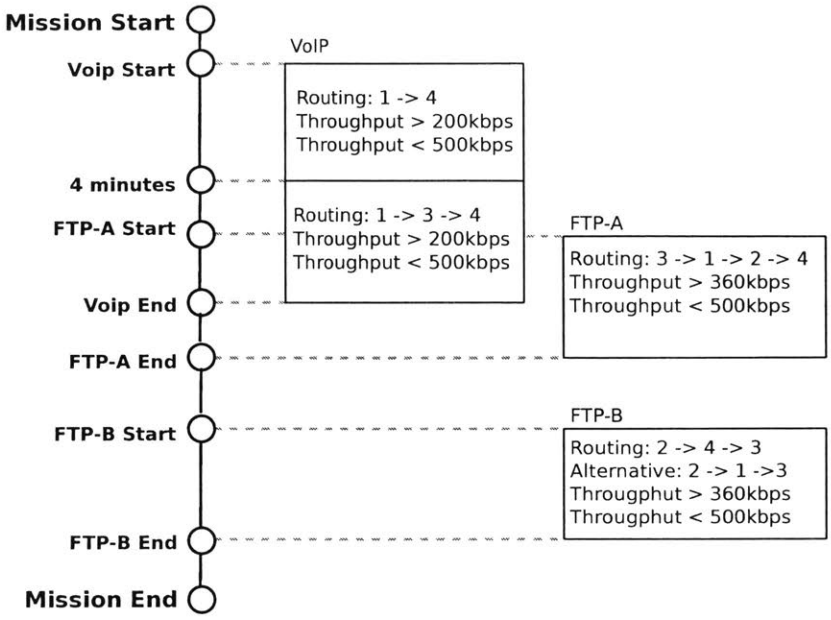

Figure 1-2: A temporal plan of configurations that specifies routes and bandwidth allocations of network flows along these routes.

continuous settings are configured over time to achieve temporally concurrent goals. We then develop Amundsen, a high-performance timed configuration manager that supports complex tasks in terms of state and time by reasoning over high fidelity models with mixed discrete and continuous constraints, and temporal influences.

1.2 Approach in a Nutshell

In this section, we begin by reviewing the related work this thesis is built on. Then, we introduce the features of TCMPs and briefly introduce its solution, the timed configuration manager Amundsen.

Related Work

This thesis solves a configuration problem that was originally proposed by Livingstone [39], a configuration manager for space systems. Livingstone is able to identify a sequences of optimal configurations that achieve a set of goals. It also introduces a form of Conflict-directed Search that allows Livingstone to scaled to large sets of

devices. This core solution method and problem was then generalized in the form of Optimal Satisfaction Problems and the OpSat solver [40].

As this thesis is directly built upon the concept of a configuration problem and its generalization as an optimal satisfaction solver, we are able to handle continuous temporal or state constraints, which are two of the key additional features of our configuration problems.

The tBurton planner generalizes Livingstone to handle both temporal constraints, and planning of concurrent sequences of temporal actions [38, 37]. Addressing the interplay between temporal constraints and planning for concurrent actions is notoriously difficult. Key to tBurton's effective management of this interaction is an algorithm called UnifyHistory, which accumulates all goals related to a timeline and consistently orders these goals along the timeline, before planning for these goals all at once. tBurton does not handle systems with mixed discrete and continuous constraints since further planning sequences of actions in this hybrid domain is difficult.

Amundsen takes a middle ground between Livingstone and tBurton. While Amundsen handles temporal constraints and temporal goals along a timeline like tBurton, but like Livingstone, it does not extend planning into planning action sequences. We create this temporal variant of Livingstone by combining OpSat from Livingstone and UnifyHistory from tBurton, to produce a timed configuration manager and a general purpose Episodic CSP solver. Finally, we extend the configuration manager into hybrid domains by leveraging both discrete and continuous state constraints, supported by this extended Opsat.

Finally, a key component of OpSat is its conflict-directed search algorithm. To achieve efficiency for solving Episodic CSPs, we incorporate a novel form of conflict-directed search to the core algorithm of UnifyHistory called Incremental Total Ordering. The enhanced new algorithm is called Conflict-directed Incremental Total Ordering.

TCMP: Timed Configuration Management Problem

TCMPs model complex configuration management problems over time by generalizing simple goal sequences to a set of temporally synchronized concurrent goals described by temporal plans on goal states. To be high-performance, the problems use hybrid automaton that encode different actuation modes with discrete and continuous constraints and temporal influences.

The problem consists of a plant model, a goal plan, and the initial state of the system being configured. The solution of this problem is a control plan that is a temporal plan of configurations, such that the system can achieve the goal plan by executing these configurations.

A plant model is a hybrid concurrent automaton (HCA), which is a set of interacting hybrid automata. Each automaton specifies several locations to represent system modes under which different sets of constraints are specified. In our motivating example, we have two kinds of automata for links and network flows, respectively. For example, a link automaton has the locations On and Off, and network flows can be only transferred when this link is at the location On.

We give the initial state and goal plan of the motivating scenario in Figure 1-3, where LS_i and FS_j are a link automaton and a flow automaton, respectively. The initial state gives the locations of these automata when time $t = 0$; the goal plan specifies a set of desired behaviors of these automata and a set of temporal constraints between these behaviors.

To solve this problem, a timed configuration manager should reason over the plant model and generate a temporal plan of configurations for the system being configured, such that all the automata can achieve the behaviors specified in the goal plan. We call this temporal plan of configurations a control plan of this problem, which is given in Figure 1-4 as a formal representation of Figure 1-2. Again, a configuration in the motivating example is the route and bandwidth allocations that are represented by the control variables $Path$ and BW, respectively. To efficiently generate such a control plan is hard since constraints on both discrete variables (i.e., $Path$) and continuous

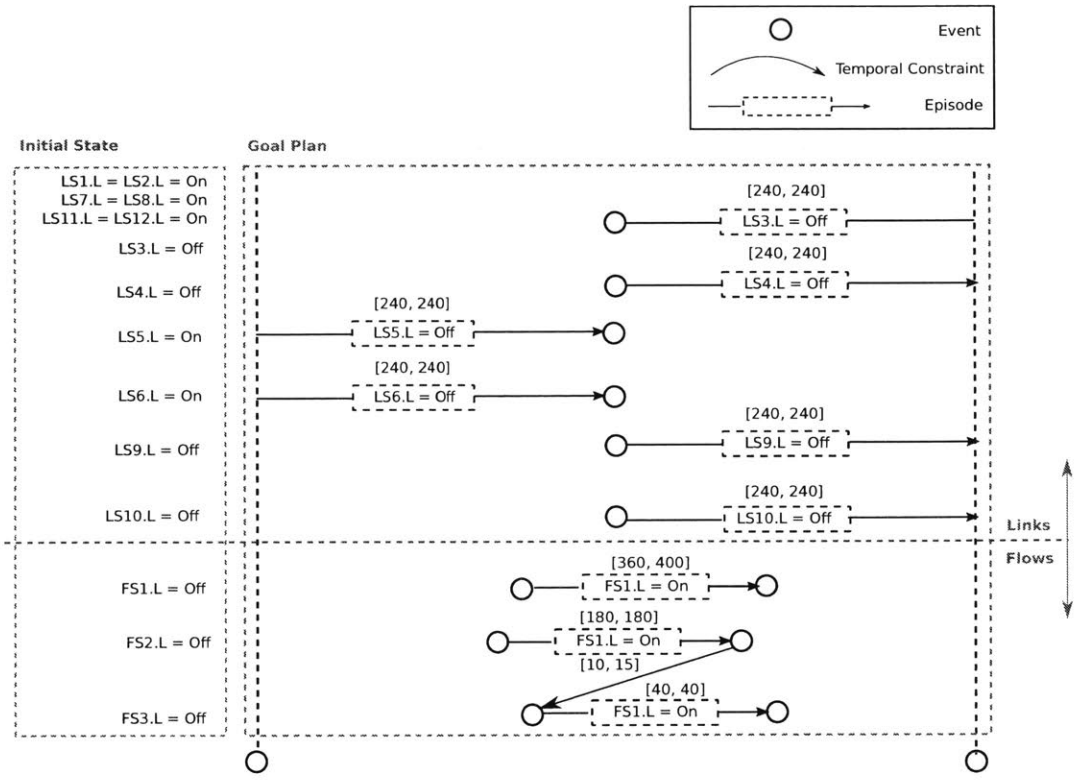

Figure 1-3: Initial state and goal plan of the motivating example. LS_i is a link automaton with the locations On and Off, and FS_j is a flow automaton with the locations On and Off.

variables (i.e., BW) are involved, and timing requirements are also significant in this example.

Amundsen: Timed Configuration Manager

Now we briefly introduce how the timed configuration manager Amundsen solves a TCMP. The architecture of Amundsen is given in Figure 1-5. Amundsen is comprised of a plan compiler that translates the problem into an episodic constraint satisfaction problem (Episodic CSP), an Episodic CSP Solver that solves the translated problems, and a solution complier that translates a solution of the Episodic CSP to a control plan. An Episodic CSP consists of events that are non-negative real variables to represent points in time, temporal constraints between event pairs, state variables, and episodes that are timed constraints between event pairs over a set of state variables.

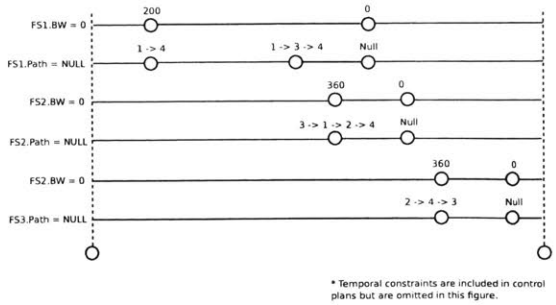

Figure 1-4: Control plan of the motivating example that gives the bandwidth and paths at the start of the plan, and their configuration changes during the mission. The temporal constraints between configuration changes are omitted in this figure.

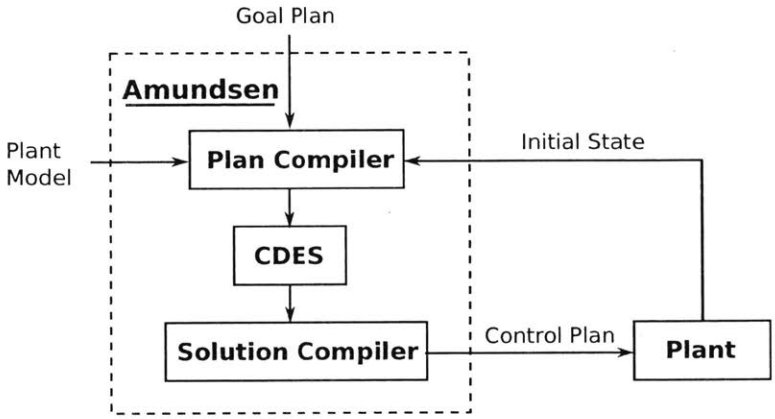

Figure 1-5: Amundsen architecture.

An example of an Episodic CSP is given in Figure 1-6. It is natural to think of this problem as an extension of a constraint satisfaction problem (CSP) over time since solving a TCMP finds a control plan that satisfies the specified behaviors in the goal plan, given the initial state and the plant model. For example, network management requires a plan of routes and bandwidth allocations over time for the network flows, such that all the flows can be transferred during their corresponding time windows.

The plan compiler collects the partial states to represent the system behaviors specified in the goal plan, and the constraints required to hold under each location. As a result, we obtain an Episodic CSP with all the constraints specified at a time point or across intervals. While the solution of a CSP is an assignment of all the variables, which is consistent with all the constraints, the solution of an Episodic CSP is a group of control trajectories that satisfy the episodes. The obtained control

trajectories are represented as a control plan.

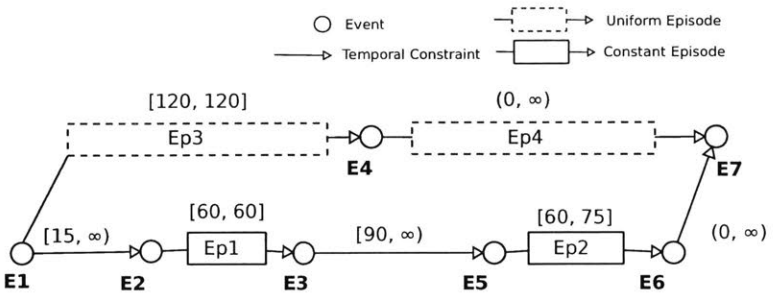

Figure 1-6: An Episodic CSP example with 7 events, 3 temporal constraints, 2 uniform episodes and 2 constant episodes.

CDES: Conflict-directed Episodic Satisfaction

The performance of Amundsen highly depends on the efficiency of the underlying Episodic CSP solver. Thus, to efficiently solve Episodic CSPs, we develope a solver called Conflict-directed Episodic Satisfaction (CDES), which is built on the same hybrid framework as UnifyHistory in tBurton [37] but improves its ordering module and the state consistency checking algorithm. To solve an Episodic CSP, CDES decomposes the problem into sub-problems and solve them independently. An example of decomposition is given in Figure 1-7. Then, to handle the temporal constraints (e.g., deadlines) and the state constraints (e.g., routing constraints and resource constraints), the solver focuses on searching for a total order of events under which a feasible solution exists. As in the reasoning process outlined above for the motivating example, such a total order is important since it is futile to decide routes before a good total order is found. With this total order, the entire horizon can be divided into a sequence of stages, where a stage is an interval between two adjacent events. Then, the solver can further determine the stage lengths and the assignments of control variable during each stage. Three total ordered instances are given in Figure 1-8, where each instance is dived into 4 stages by the event ordering. The architecture of CDES is given in Figure 1-9.

We implement CDES by extending OpSat [40] to an Episode CSP solver and incorporating an enhanced ordering module from UnifyHistory [37], which is also in-

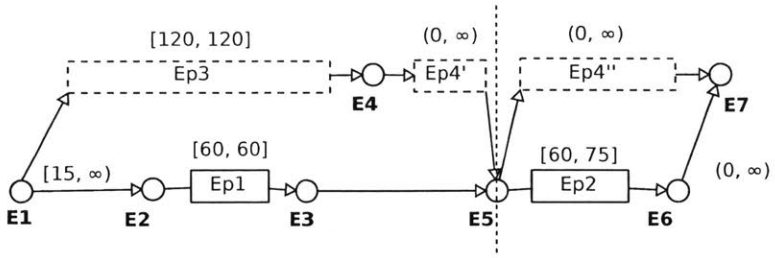

Figure 1-7: Decomposition example with $E5$ as the breakpoint.

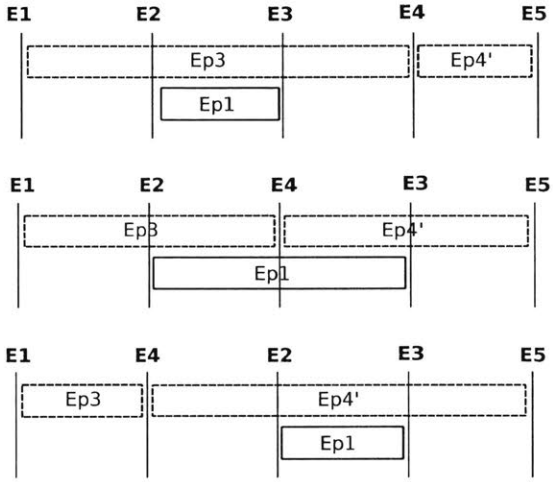

Figure 1-8: Ordered Episodic CSPs.

dicated by eOpSat. As shown in Figure 1-9, CDES is comprised of three parts: (1) the decomposition module divides an Episodic CSP into several pieces according to the independences between episodes over time and state, and then these pieces can be solved independently; (2) Conflict-directed Incremental Total Ordering (CDITO) starts with a total order of all the events in an Episodic CSP and generates a new total order by altering part of the total order when the current total order is infeasible; (3) Incremental Temporal Consistency (ITC) and Incremental State Consistency (ISC) are used to check the temporal consistency and state consistency of the ordered Episodic CSP, respectively. If the ordered Episodic CSP is consistent, a solution is returned. Otherwise, the inconsistency is summarized as clauses of partial orders that are leveraged by CDITO for the next order generation. Note that our approach is incremental: CDITO only changes a small portion of partial orders within the current total order at each generation, and our algorithms only check the consistency of part

of the plan that is different from the previous plan in most cases, which is important to the efficiency of our algorithm.

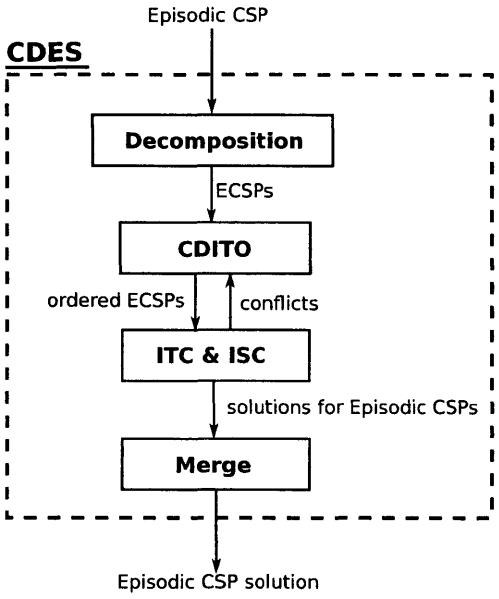

Figure 1-9: Conflict-directed Episodic Satisfaction (CDES) architecture.

1.3 Summary of Contributions

In this thesis, we solve timed configuration management problems for networked devices over long horizons. We summarize our contributions as follows:

1. **Solving Timed Configuration Management**

   We introduce a new class of configuration management problems called TCMP, in which we configure the discrete and continuous parameters to achieve the desired behavior of systems over time. We also developed the timed configuration manager Amundsen to solve TCMPs, and the efficiency of Amundsen was demonstrated by benchmarking on a communication network management simulator. This approach combines key insights from Livingstone for configuring networked devices, tBurton and UnifyHistory for planning concurrent processes, and constraint programming for managing hybrid constraints.

2. **Solving Decision-Making Problems over Time**

   We propose a new class of constraint satisfaction problems with timed constraints called Episodic CSPs. This representation can model many real-world scenarios featuring decision making over time as well as capture the TCMPs proposed in this thesis. We developed an efficient Episodic CSP Solver called CDES to power Amundsen.

1.4 Organization of Thesis

The rest of this thesis is organized as follows: we begin by reviewing the background and literature related to timed configuration management in Chapter 2. Then, we introduce the formal definitions of TCMPs in Chapter 3. As we solve TCMPs by solving their corresponding Episodic CSPs, we first introduce the definitions of Episodic CSPs in Chapter 4. In Chapter 5, we present the architecture of Amundsen, which is used to generate control plans for TCMPs. Then, we describe CDES, the solution approach of Episodic CSPs, in Chapter 6. In Chapter 7, CDITO, which is the core module of CDES to generate total orders, is presented. We end this thesis by demonstrating the efficiency of our solver in Chapter 8 and discussing the conclusions and future works in Chapter 9.

Chapter 2

Related Work

In this chapter, we review previous works related to our timed configuration management problems and timed configuration manager. We first discuss the predecessor of our work, the configuration manager Livingstone [39]. Then, we introduce other temporal planners that solve similar problems. We will also discuss the classical problems that are special cases of the problem we are solving.

2.1 Configuration Management

As this thesis is directly built upon the concept of a configuration problem and its generalization as an optimal satisfaction solver. We draw a lot of insights from Livingstone [39], a model-based reactive self-configuring autonomous system. Livingston along with HSTS planning system [27] and RAPS executive [13] managed the spacecraft Deep Space One. Livingstone models system components through concurrent transition systems that communicate through shared variables. It reasons over the control variables to achieve the desired trajectory specified by the planner HSTS, where the trajectory is a sequence of qualitative states of system components. Our timed configuration manager and Livingstone are similar: they both configure the low-level control variables to instantiate the specified behaviors as the goal. Our planner extends the problem to be timed and hybrid in two aspects. First, our planner uses concurrent hybrid transition systems and reasons over both discrete and continuous

control variables, while the control commands in Livingstone are of discrete domains. Second, to specify the desired behaviors of system components, we use a goal plan that not only includes the qualitative state of components but also requires metric temporal features, such as how long these components stay within a qualitative state. This encoding provides more accurate modeling power for time-critical and complex missions.

2.2 Timeline-based Planning and Scheduling

In complex system management problems, planning and scheduling are usually represented with the same paradigms and interleaved with the same mechanisms. Planning and scheduling are traditionally two distinct but complementary processes of managing complex systems over a receding temporal horizon. Planning decides on the course of actions to achieve specific goals, and scheduling instantiates a set of plans by assigning execution times to these actions concerning resource and temporal constraints.

Classical planning systems such as STRIPS [12] and PDDL [26] represent actions as instantaneous transitions between states. Then, PDDL2.1 [14] extends PDDL to describe temporal planning problems by introducing durative actions. Timeline-based planning exploits stronger structuring assumptions used in classical scheduling [1] and was first introduced in HSTS [27]. Timelines are continuously-varying state variables that represent the evolution of system features up to a given time, and constraints are expressed on the trajectories of these timelines. Other timeline-based planning paradigms are also introduced in [4, 15, 6]. Advanced timeline-based planners, including EUROPA [2], ASPEN [7], and tBurton [38] have been developed in the past decades. Recent advances in the PLATIUm system integrate planning and scheduling to manage discrete and reservoir resources and take into account execution flexibility and temporal uncertainty [25, 35, 34].

Overall, our advance over existing representations is to include complex constraints over trajectories such as constraints over discrete and continuous variables, which are

useful in describing complex systems and goal specifications.

More specifically, Amundsen takes a middle ground between Livingstone and tBurton. While Amundsen handles temporal constraints and temporal goals along a timeline like tBurton, but like Livingstone, it does not extend planning into planning action sequences. We create this temporal variant of Livingstone by combining OpSat from Livingstone and UnifyHistory from tBurton, to produce a timed configuration manager and a general purpose Episodic CSP solver. In addition, we extend the configuration manager into hybrid domains by leveraging both discrete and continuous state constraints, supported by this extended Opsat.

2.3 Classical Problems Featuring Timed Configuration

Making decisions over time is a significant topic in scheduling and planning communities, and there are many classical problems featuring time.

The classical problem Vehicle Routing Problem with Time Windows (VRPTW) is a well-known integer programming problem which belongs to the category of NP-hard problems. It originates from the Vehicle Routing Problem (VRP), extending it with additional time constraints. In the VRPs, we need to configure a fleet of vehicles to pick up and deliver the packages of customers, and VRPTW specifies certain time windows during which customers are available in specific locations. In VRPTW, there are resource constraints such as the maximum capacity of each vehicle to carry packages, and time windows. VRPTW can be a special case of timed configuration management problems and has been well studied. Thus our solution methods also draw insights from methods used to solve VRPTW, such as tackling ordering problems.

Other problems such as Job Shop Scheduling Problems also feature resource allocation and time reasoning in which we need to allocate limited resources to multiple tasks over time carefully. In general, our framework and many classical problems are

different in two aspects. First, these problems such as Job Shop Scheduling and their solution methods always deal with discrete time, which restricts solution efficiency and execution flexibility. In our framework, we can freely choose the time representation as a continuous or discrete timeline. In addition, our output control plan does not fix the schedule, enabling robust and flexible execution with the presence of execution uncertainty over time. Second, instead of only considering the resource constraints that are linear inequalities, we allow much more complex constraints over time such as All-Different, Circuit, and Element. These constraints are required for the expressiveness to plan over more complex systems and requirements.

2.4 Summary

In this chapter, we summarized the previous works of configuration management from which our framework is extended. We also discuss the timeline-based planners that are related to our work, and the classical problems that can be regarded as special cases of the problem this thesis aims at.

In the following chapters, we present a formal definition of timed configuration management problems that provide high-fidelity models to represent systems being configured, and a goal specification to describe the required system behaviors with the presence of concurrency and metric timing requirements. We also give a solution approach that is efficient to reason over these high-fidelity models and complex goal specifications.

Chapter 3

Problem Statement

In this chapter, we define the timed configuration management problem (TCMP) that Amundsen solves. Amundsen is designed to configure embedded networked systems with constraints over both discrete and continuous variables to achieve temporally concurrent goals.

One example of a TCMP is the network management problem introduced in Section 1.1, where multiple flow requests must be configured, respecting requirements on bandwidth, delay, loss, deadlines over a set of communication links. Given the link characteristics and these network flow requirements, Amundsen must generate routes and bandwidth allocations along the route over time for all the flow requests, such that all the requirements are satisfied while respecting the capacity of links.

To model and solve such a problem, TCMPs extend the configuration management problem Livingstone solves [39] to be hybrid and timed: (1) the model is high-fidelity and has both discrete and continuous variables under different modes of systems, while the variables of the previous configuration management problems are discrete; (2) to specify the desired behaviors of system components, TCMPs uses a goal plan that extends the step-wise transitions to include metric temporal features, such as how long they must stay within a mode, which provides more accurate modeling power for time-critical and complex missions. In the following two paragraphs, we use the example in Section 1.1 to illustrate the necessity of these two features.

Network management requires modeling different system behaviors under vari-

ous system modes. For example, the capacity of a link to transfer flow requests is different when a link is functional or unavailable. In addition, the constraints of these behaviors might involve both continuous and discrete variables — such as continuous bandwidth capacity and discrete routes — which requires our model to be hybrid. This motivates us to use hybrid automata that can capture both discrete and hybrid constraints under different modes over time. Since an embedded networked system always consists of multiple sub-systems, we compose several hybrid automata along with the constraints between these automata to generate a hybrid concurrent automaton (HCA) to describe a system.

Another significant feature of TCMPs is temporally concurrent goals. As we saw in Section 1.1, each network flow must be transferred within a time window where the start and end are not fixed time points, and multiple flow requests are possibly transferred concurrently. Similar to the Chekov, tBurton and Kongming planners, these temporal configuration goals are represented using qualitative state plans [17, 37, 24]. Thus, to describe these temporally concurrent goals, we use a temporal plan whose activities specify the behaviors of sub-systems, such as flow requests being transferring or turned off, while metric temporal constraints are used to describe the timing relations among these activities.

A TCMP is comprised of a plant model of a system, a goal plan, and the initial state of the system. The solution of a TCMP is a set of trajectories of control variables over time as a control plan, such that the system can evolve from the initial state and follow the requirements of the goal plan by executing this control plan.

In the rest of this chapter, we first present the definition of TCMP. Then, we introduce hybrid automata and HCA, which specifies feasible transitions between different modes and corresponding constraints under each mode for sub-systems and the entire systems respectively. Then, we defined the goal plan that specifies the desired system behaviors. We also define a control plan that is the solution of a TCMP. We end this chapter by modeling our motivating example introduced in Section 1.1 as a TCMP.

3.1 Timed Configuration Management Problem

A TCMP consists of a plant model expressed as a set of concurrent and interacting hybrid automata, plus a temporal goal specification that imposes temporal constraints on the desired bebaviors of those hybrid automata. We also specify the initial state of the system and assume all the variables are known when $t = 0$. Formally, TCMP is defined as follows:

Definition 1 (TCMP). *A TCMP is a tuple* $\langle M, GP, \Theta \rangle$*, comprised of a hybrid concurrent automaton* M*, a goal specification* GP*, and an initial state* Θ*.*

While the HCA specifies constraints over both discrete and continuous variables under different modes for a set of sub-systems, the goal plan is a temporal plan that gives a the desired behaviors of the system. An HCA consists of a set of sub-systems, and e ach sub-system is represented by a hybrid automaton that describes the behavior of a single component or process, such as a communication link or a network flow. We begin by defining each sub-system as a hybrid automaton as follows:

Definition 2 (Hybrid Automaton). *A hybrid automaton is a 5-tuple* $\langle P, L, X, U, F^C, T \rangle$*:*

- P *is a set of discrete or continuous parameters that are constants over time.*
- L *is the unique location variable. dom(L) consists of the locations over which this system transitions.*
- X *is a set of state variables, and each state variable* $x \in X : \mathbb{R}_{\geq 0} \rightarrow dom(x)$ *is a function of time.* X *consists of a set of discrete state variables* X^d*, and a set of continuous state variables* X^c*.*

  A valuation val for X *is a function that assigns a value to each state variable of* X*.*

  A state $s = (l, v)$ *consisting of a location* $l \in dom(L)$ *and a valuation* v *of* X*. A partial state is an assignment to a subset of* $\{L\} \cup X$*.*

- $U = U_d \cup U_c \cup U_X$ *is a set of interface variables.* U_d *is a set of discrete control variable;* U_c *is a set of continuous control variable;* U_X *is a set of state variables of other automata;*

- F^C*:* $dom(L) \rightarrow 2^C$ *is a function that maps a location to a set of constraints* C *that scope on* P*,* X*, and the derivations of the continuous state variables* $\dot{X}^c$*, which specifies a feasible space* $\Sigma_{X \cup \dot{X}^c \cup U}$ *of* X *and* $\dot{X}^c$ *under the parameter setting* P*.*

- T *is a set of transitions of the form* $\tau = \langle l, l', g \rangle$*, each of which associates with a source location* $l \in dom(L)$*, a target location* $l' \in dom(L)$*, and a guard* g*. The transition is enabled when* g *is satisfied. We assume* g *is in the form of assertion of a discrete control variable* $u_d \in U_d$*.*

We extend the timed automata in the tBurton planning problem to be hybrid [37]. The main difference is that hybrid automata supports a larger range of constraints under each location. We restrict the supported constraints to all the constraints in [3]. We also make the assumption that the guard of transitions is the form of assertion, and there exists a transition between every pair of locations. This is the same assumption used in the Livingstone configuration problem [39].

To model several components within a system, we compose a set of hybrid automata to produce an HCA. An HCA specifies a set of interacting hybrid automata and the global constraints of their state variables under different location combinations. We define the HCA as follows:

Definition 3 (HCA). *An HCA is a tuple* $M = \langle CS, U, F^{GC} \rangle$*:*

- $CS = \{S_1, S_2, .., S_n\}$ *is a set of hybrid automata.*

- $U = U_d \cup U_c$ *is a set of control variables.* U_d *is a set of discrete control variables, and* U_c *is a set of continuous control variables.*

The goal plan specifies a set of temporally concurrent goals. This is useful when we plan for networked devices over a long horizon with multiple goals as well as timing constraints between these goals. The goal plan is defined as follows:

Definition 4 (Goal Plan). *Goal plan GP is a tuple $\langle E, EP, TC \rangle$:*

- *E is a set of events with real domains.*

- *EP is a set of episodes. Each episode is a a tuple $\langle e^{\vdash}, e^{\dashv}, D, x \rangle$, where $e^{\vdash}$ and $e^{\dashv}$ are the start and end events, D is a duration constraint between the start and end events, and x is a partial state that must hold during this episodes.*

- *TC is a set of temporal constraints scoping on E.*

Note that the goal plan GP gives all the desired states. Thus, the transitions of the locations are known to the configuration manager. However, the time when the transitions happen is partially known. Although every transition τ is associated with a symbolic event that is allowed to happen during an interval, the exact timing of the transition is unknown.

The configuration manager returns a valid control plan CP that provides a set of piecewise control trajectories and entail the desired transitions. Meanwhile, it maintains execution flexibility by associating each control value change with a symbolic event, which is similar to the goal plan GP. A control plan is a solution of $P = \langle M, GP, \Theta \rangle$, and we formally define the control plan as follows:

Definition 5 (Control Plan). *A control plan CP is a tuple $\langle E, CT, TC \rangle$:*

- *$E \subseteq GP.E$ is a set of events with real domains.*

- *T is a set of timed sequences of configurations. For each control variable $u \in U$, there is a sequence $t \in T$, and each element is a timed configurations $\langle e, \mu \rangle$. The control variable is changed to value μ at the event $e \in E$.*

- *TC is a set of temporal constraints scoping on E.*

3.2 Example: Temporal Network Configuration Problem

In this section, we formulate the motivating example in Section 1.1 as a Temporal Network Configuration Problem (TNCP), which is an example of TCMP focusing on managing flow requests in terms of routing, throughput, loss, delay, and deadlines. We begin by introducing two components in the form of hybrid automata: the link model LM and the flow model FM, and then we compose multiple links and flow requests to produce a network model that is an HCA in which several flow requests and links are interacting with each other. We also give examples of its goal plan and control plan. Note that we only use abstract functions to represent the constraints over routing, delay, loss, and bandwidth, and more encoding details can be found in Appendix A.

Link Model

Links are fundamental components of a communication network. Each link starts from a source node s and ends at a sink node t, and it can transfer a group of flow requests from s to t if the total throughput BW does not exceed its bandwidth capacity bw. In addition, links are also characterized by loss rate l and delay d. Formally, the link model $LM = \langle P, L, X, F^C, T \rangle$ is defined as follows:

- $P = \{s, t, bw, l, d\}$
- $L \in \{\text{On}, \text{Off}\}$.
- $X = \{BW\}$.
- $U = \{\text{cmdL}\} \cup \{Path_i, BW_i \mid \text{for every Flow i}\}$.
- F^C is:
  - $F^C(\text{On})$: $BW \leq bw$; $ComputeBW(Path_i, BW_i)$
  - $F^C(\text{Off})$: $BW = 0$; $ComputeBW(Path_i, BW_i)$

- T is:
  - $\tau_1 = \langle \text{On}, \text{cmdL} = \text{Off}, \text{Off} \rangle$
  - $\tau_2 = \langle \text{Off}, \text{cmdL} = \text{On}, \text{On} \rangle$

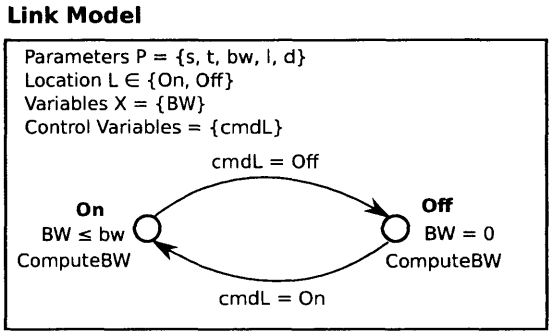

Figure 3-1: Link model.

As we can see in Figure 3-1, a link can be in two locations: On or Off, which means this link is functional or unavailable because of some unknown reasons. τ_1 and τ_2 are two transitions between On and Off. The details about the constraint ComputeBW are given in Appendix A.

Flow Model

Another fundamental component is the network flow. Each flow is a data transfer request that starts from a source node s and ends at a sink node t for transferring an amount of data. When a flow is transferring, the loss and delay must not exceed the maximum allowable loss l and the maximum allowable delay d that are accumulated along the chosen route. Formally, the flow model $FM = \langle P, L, X, F^C, T \rangle$ is defined as follows:

- $P = \{s, t, bw, l, d\}$
- $L \in \{\text{Off}, \text{On}\}$.
- $X = \{L, D\}$.
- $U = \{\text{cmdF}, BW, \text{Path}\} \cup \{s_j, t_j, l_j \mid \text{for every Link j}\}$

- F^C is:
  - F^C(Off): $BW = 0$
  - F^C(On):
    - $BW > bw$;
    - $LossConstraint(l, Path, \bigcup_j \{\langle s_j, t_j, l_j \rangle\})$;
    - $DelayConstarint(d, Path, \bigcup_j \{\langle s_j, t_j, l_j \rangle\})$.
- T is:
  - $\tau_1 = \langle \text{Off}, \text{cmdF} = \text{On}, \text{On} \rangle$
  - $\tau_2 = \langle \text{On}, \text{cmdF} = \text{Off}, \text{Off} \rangle$

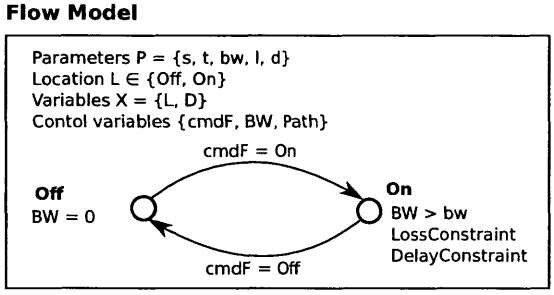

Figure 3-2: Flow model.

As we can see, a flow can be in two locations On and Off. While the location On and Off can transition to each other by setting the command *cmdR* to On and Off, respectively. When a flow is Off, no constraint is applied. However, when this flow is On, the loss and delay must not exceed the specified values, and the data is accumulated at the destination with the rate *bw*. The details about the constraints Compute-Delay and Compute-Loss are given in Appendix A.

We also define another flow model CFM that forces the bandwidth to be constant during the flow being transferring. This is the case when some flow requests must have constant throughput for the sake of service quality, such as a VoIP flow for an online meeting. Therefore, when CFS.L = On, we add another constraint $\frac{\partial BW}{\partial t} = 0$.

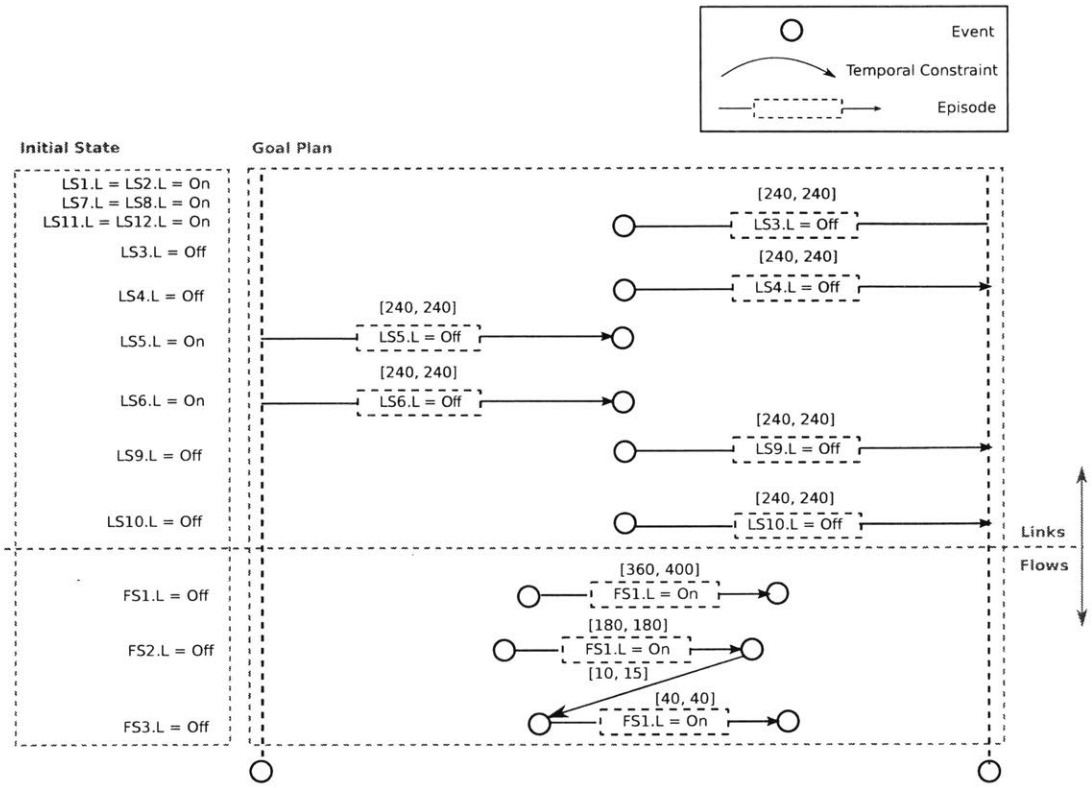

Figure 3-3: Initial state and goal plan of the motivating example.

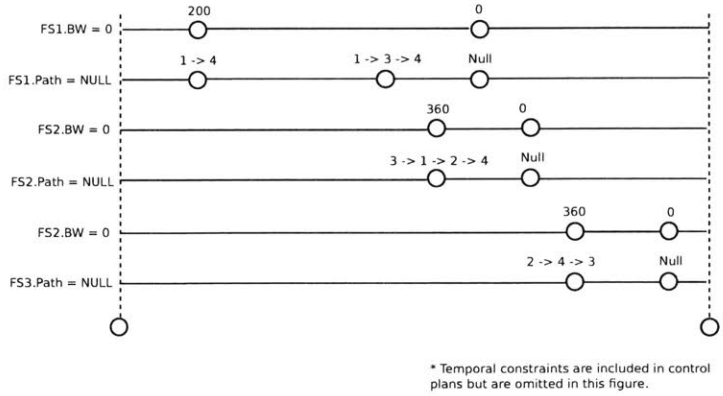

Figure 3-4: Control plan of the motivating example. We omit all the temporal controls and cmd variables.

Goal Plan and Control Plan

We show an example of a goal plan in Figure 3-3, where all three flow requests are required to be On for a certain amount of time. No episodes are specified for links $LS1$, $LS2$, $LS7$, $LS8$, $LS11$, and $LS12$. Other links should be turned off either in the first four minutes or the second four minutes, where temporal constraints $[240, 240]$ are specified for the corresponding episodes. We also show the initial locations of all the systems on the left of this goal plan.

The corresponding control plan is given in Figure 3-4, where the values of control variables, such as the bandwidth allocation and route of each flow, change at some events.

3.3 Summary

In this chapter, we presented the formal definition of the timed configuration management problem that consists of a plant model, a goal plan, and an initial state. The plant model is a hybrid concurrent automaton that composes several interacting hybrid automata. The goal plan specifies the partial states of these automata over time with metric temporal constraints. We also show an instance of this problem for the configuration of communication networks.

As reasoning over such a high-fidelity model and complex goals is hard in terms of efficiency, we present a high-performance, efficient solution approach called Amundsen in Chapter 5. As Amundsen translates the TCMPs to a a class of problems called Episodic CSPs and then leverages an efficient solver to solve them, we first introduce the definitions of Episodic CSPs in the next chapter.

Chapter 4

Episodic Constraint Satisfaction Problem

In this chapter, we introduce the formal definitions of Episodic CSPs, which are decision-making problems with state constraints over time. We also review the similar representations that are used to model decision-making problems over time.

An Episodic CSP is comprised of a set of events to represent time points, a set of temporal constraints between these events, a set of state variables, and a set of episodes to constrain these state variables during some intervals between event pairs. An example of an Episodic CSP is given in Figure 4-1.

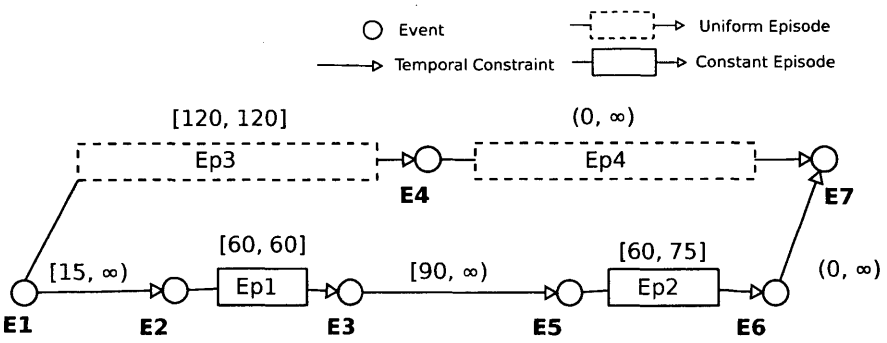

Figure 4-1: An Episodic CSP example with 7 events, 3 temporal constraints, 2 uniform episodes and 2 constant episodes. Note that state variables are omitted here.

The motivation to introduce Episodic CSP is that we solve timed configuration management problems by solving their correspoding Episodic CSPs. While the plant

model provides the constraints of a system under certain locations, the goal plan specifies the desired behaviors of this system over time. Solving this problem is equivalent to finding a set of trajectories of the control variables, such that all the constraints to achieve the desired behaviors are satisfied. This problem is a constraint satisfaction problem scoping on these control variables. Thus, we use Episodic CSPs to capture the constraints extracted from the initial state, the plant model, and the goal plan.

4.1 Related Work

As Episodic CSPs extend the classical CSPs with the notion of time, it can also be regarded as a composition of simple temporal constraints, which are used in simple temporal networks (STNs) [10], and state constraints to describe state trajectories over bounded time. The episode is similar to the time token used in the time map management system [9].

Other similar representations include temporal planning networks (TPNs) [19, 23, 31, 8] and qualitative state plans (QSPs) [22, 17, 38, 24, 11], which are widely used in many planners and executives. For example, tBurton [38] uses QSPs to describe a set of temporally concurrent goals the system needs to achieve over time.

The state-of-the-art constraint-based scheduling solver is CP Optimizer that deals with continuous time and introduces the notion of state functions to constrain the trajectories of state variables over time [20, 21]. Powered by advanced stochastic search methods such as Large Neighborhood Search (LNS), CP Optimizer can efficiently and iteratively improve the plan quality of complex scheduling problems. However, CP Optimizer only models and solves problems with resource constraints over state variables. As is the case for most scheduling methods, they lack richer constraints to model complex systems.

In the field of decision-making, there are many successful solvers, such as Gecode [16] for constraint programs, and Gurobi [30] for mixed integer and linear programs. Episodic CSPs are as expressive as these programs. Since we naturally support the

notion of time, our framework provides more concise models and explicit dependency structures to enable efficient solution.

4.2 Definitions

We give the formal definitions of Episodic CSPs as follows:

Definition 6. *(Epiosodic CSP) An Episodic CSP is a tuple* $\langle EV, SV, TC, EP \rangle$.

- EV *is the set of events.*
- SV *is the set of state variables.*
- TC *is the set of temporal constraints.*
- EP *is the set of episodes.*

In the rest of this thesis, we also abbreviate Episodic CSPs as episode satisfaction. We use events to describe to time point, which is defined as follows:

Definition 7. *(Event) An event is a real variable or an integer variable.*

The temporal constraints specify the relations between events, which are defined as follows:

Definition 8. *(Temporal Constraint) A temporal constraint is a a tuple* $\langle e^{\vdash}, e^{\dashv}, D \rangle$, *where:*

- $e^{\vdash}$ *and* $e^{\dashv}$ *are two events events.*
- D *is the duration constraint, and is given by a tuple* $\langle D_l, D_u \rangle$ *such that* $D_l \leq e^{\dashv} - e^{\vdash} \leq D_u$.

We define the state variables and state trajectories as follows:

Definition 9. *(State Variable) A state variable maps time* $t \in \mathbb{R}_{\geq 0}$ *to the values from an associated domain.*

Definition 10. *(State Trajectory) A state trajectory of S over I maps $t \in I$ to an assignment of S, where S is a vector of state variables and I is a continuous interval. S and I are said to be the scope and the interval of this state trajectory.*

Episodes are timed constraints over state variables that are defined as follows:

Definition 11. *(Episode) An episode is a a tuple $\langle e^{\vdash}, e^{\dashv}, D, S, SC \rangle$, where:*

- *$e^{\vdash}$ and $e^{\dashv}$ are the start and end events of the episode.*
- *D is the duration constraint, and is given by a tuple $\langle D_l, D_u \rangle$ such that $D_l \leq e^{\dashv} - e^{\vdash} \leq D_u$.*
- *S is the scope, and is given by a set of state variables.*
- *SC is the state constraint that must hold during this episode.*

To restrict the problems we are given, we restrict the considered state constraint SC within two categories that can be expressed as a tuple $\langle \text{uniform}, C \rangle$ or $\langle \text{constant}, C \rangle$, where C is a constraint as defined in CSP. We call the corresponding episodes uniform episodes and constraint episode, which are defined as follows:

Definition 12. *(Uniform Episode) A uniform episode applies the same constraint to the scoped state variables at every time point between the start event and the end event.*

Definition 13. *(Constant Episode) A constant episode is a uniform episode where the assignments to the scoped state variables are constant between the start event and the end event.*

We use two examples to illustrate uniform episodes and constant episodes. That the throughput x of a network flow should be larger than 100kbps during transfer can be an episode with the state constraint $\langle \text{uniform}, x \geq 100 \rangle$. If the throughput should be constant for the sake of service stability, the state constraint will be $\langle \text{constant}, x \geq 100 \rangle$.

We also add another special class of episodes, point episodes that describe a constraint over time points, which are defined as follows:

Definition 14. *(Point Episode) A point episode is an episode whose start event and end event are same. Let $e = \zeta$ be this event, and ϵ be an enough small quantity, there are three types of point constraints:*

1. *Left point constraint constrains the state trajectory during the interval $[\zeta - \epsilon, \zeta]$.*
2. *Right point constraint constrains the state trajectory during the interval $[\zeta, \zeta + \epsilon]$.*
3. *Point constraint constrains the state trajectory during the interval $[\zeta - \epsilon, \zeta + \epsilon]$.*

A candidate solution of an Episodic CSP assigns all the events and state variables over time, while a solution is a candidate that satisfies all the episodes, which is defined as follows:

Definition 15. *(Episodic CSP Solution) An Episodic CSP solution sol is a tuple $\langle s, st \rangle$:*

- *s is a schedule that assigns every event with a value.*
- *st is a state trajectory of all the state variables from the earliest time to the latest time of s.*
- *sol is said to be consistent if and only if all the episodes are satisfied. An episodes is said to be satisfied if and only if the schedule of its start event and end event satisfy its duration constraints, and the state trajectory satisfies its state constraints.*

4.3 Summary

In this chapter, we presented Episodic CSPs to model the decision-making problems with state constraints over time. In the next chapter, we will introduce the architecture of Amundsen, which translates the timed configuration management problems into Episodic CSPS and then leverages an efficient solver to solve them.

Chapter 5

Amundsen: Timed Configuration Manager

In this chapter, we introduce the timed configuration manager, Amundsen, which takes as input a timed configuration management problem (TCMP) and generates a control plan. First, we present the architecture of Amundsen consisting of a translator from TCMPs to Episodic Constraint Satisfaction Problems (Episodic CSPs), and a corresponding Episodic CSP Solver called Conflict-directed Episodic Satisfaction (CDES). Then, we introduce the translator that compiles TCMPs to Episodic CSPs. In this chapter, we only briefly describe CDES, and more details about CDES are introduced in Chapter 6.

5.1 Amundsen Architecture

The architecture of Amundsen is given in Figure 5-1. Amundsen takes as input a TCMP and generates a control plan. This TCMP includes a plant model of the system being configured, the goal plan that specifies the desired behavior of the system over time, and the initial state of the system. The output control plan is a temporal plan whose activities are configurations, where a configuration is an assignment of the control variables, such as the routes and bandwidth allocation. The examples of plant models, state plans, and control plans can be found in Chapter 3 (Figure 3-1,

Figure 3-2, Figure 3-3, and Figure 3-4).

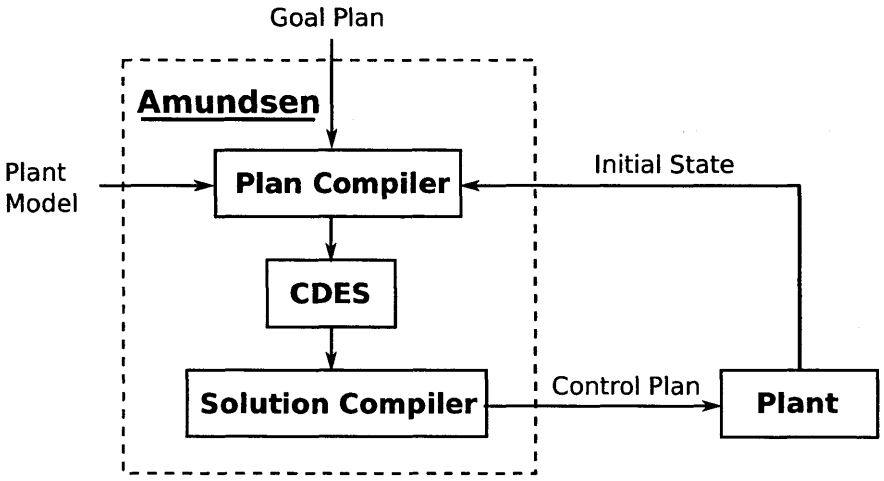

Figure 5-1: Amundsen architecture.

To generate a control plan for a TCMP, Amundsen first compiles the TCMP into an Episodic CSP, which is solved by an efficient Episodic CSP Solver called CDES. While the plant model provides the constraints of a system under certain locations, the goal plan specifies the locations of this system over time. Solving this problem is equivalent to finding a set of trajectories of the control variables, such that all the constraints under the desired partial states are satisfied. This problem is a constraint satisfaction problem scoping on these control variables. Thus, we use Episodic CSPs to capture the constraints extracted from the initial state, the plant model, and the goal plan. CDES can efficiently solve Episodic CSPs with both discrete and continuous variables.

CDES takes as input an Episodic CSP that is translated from a TCMP and outputs an Episodic CSP in which events are totally ordered and state variables are assigned over time. We implement CDES by extending OpSat [40] to an Episode CSP solver and incorporating an enhanced ordering module from UnifyHistory [37], which is also indicated by eOpSat. The output of CDES is a set of state trajectories for all the state variables. The solution compiler only chooses the state trajectories of the control variables and compile them into a control plan. Since this compilation is trivial, we do not give the details in this thesis.

5.2 From TCMP to Episodic CSP

In this section, we present the plan compiler that takes as input a TCMP *tcmp* and outputs an Episodic CSP *ecsp* that can be solved by an Episodic CSP solver.

As introduced in Chapter 3, a TCMP is $tcmp = \langle M, GP, \Theta \rangle$, where $M = \langle CS, F^{GC} \rangle$ is a plant model. We create the corresponding Episodic CSP $ecsp = \langle EV, SV, TC, EP \rangle$ by following these steps:

1. **Events**: The events EV are all the events in the goal plan GP, and we have $EV = EV(GP)$. We also denote the reference event $e_0 \in EV$ and the global end event $e_\infty \in EV$, and they happen earliest or latest respectively.

2. **State Variables**: The state variables SV are the composition of the location variable L_i, state variables X_i, and exogenous variables U_i of every hybrid automaton $S_i \in CS$, and we have $SV = \bigcup_{S_i \in CS} SV_i = \bigcup_{S_i \in CS} (\{L_i\} \cup X_i \cup U_i)$.

3. **Temporal Constraints**: The temporal constraints come from the temporal constraint in the goal plan.

Then, we instantiate the episodes:

4.1 Episodes from Initial State: We add a point constraint over e_0 to capture the information in the initial state Θ. This point constraint is $ep_0 = \langle e_0, C \rangle$, and C is a conjunction of a set of *equality* constraints over location variables and state variables X:

$$(\bigwedge_i (L_i = l_i^\Theta)) \wedge (\bigwedge_j (X_j = x_j^\Theta))$$

where l_i^Θ and X_j^Θ are the state specified by Θ at $t = 0$.

4.2 Episodes from Constraints Implied by Locations We also add a uniform episode $ep_{0 \to \infty} = \langle e_0, e_\infty, (0, \infty), \langle \text{uniform}, C \rangle \rangle$, with e_0 preceding every other event and e_∞ succeeding every other event, to include the constraints implied by location assignments. As the location constraint function F_i^C of each automaton

$S_i \in SC$ maps a location $L_i = l_{ij}$ to a constraint $F_i^C(l_{ij})$ over its variables S_i, we have $C_i = \underset{j}{\wedge} C_{ij}$ and each C_{ij} is represented as follows:

$$(L_i = l_{ij}) \implies F_i^C(l_{ij})$$

Then, we complie out the implication:

$$(L_i \neq l_{ij}) \vee F_i^C(l_{ij})$$

As for each automaton, we have the constraint C_i. The constraint C included in the episode is $C = \underset{i}{\wedge} C_i$.

4.3 Episodes from Goal Plan: Finally, we add all the episodes from the goal plan to the Episodic CSP.

5.3 Summary

In this chapter, we presented a timed configuration manager for solving timed configuration management Problems, which translates this problem to an Episodic CSP and then solves this equivalent problem by using an Episodic CSP solver called CDES.

As the core of Amundsen, CDES leverages temporal dependency to decompose Episodic CSPs, and then orders the events to further decompose the problem into several parts that are handled by highly-optimized sub-solvers independently. The algorithmic details of CDES are introduced in the next chapter.

Chapter 6

CDES: Conflict-directed Episodic Satisfaction

In this chapter, we present a fast conflict-directed algorithm for episode satisfaction, Conflict-directed Episodic Satisfaction (CDES) (Figure 6-1). CDES is comprised of a decomposition method followed by three interacting modules: Conflict-directed Incremental Total Ordering (CDITO), Incremental Temporal Consistency (ITC), and Incremental State Consistency (ISC). We first introduce the decomposition module that takes as input an Episodic CSP and outputs a set of Episodic CSPs that can be solved independently (Section 6.3). Then, each Episodic CSP will be totally ordered by CDITO in a generate-and-test fashion that incrementally changes a small portion of the previous generated total order in a new generation until a total order under which a solution exists is found. We briefly introduce the properties of CDITO in Section 6.4, and the algorithmic details are given in Chapter 7. Given a total order provided by CDITO, the temporal consistency and state consistency of the ordered plan will be incrementally checked by ITC (Section 6.5) and ISC (Section 6.6), respectively.

6.1 Related Work

Our approach is built on UnifyHistory that is used in tBurton [37] to unify multiple timelines. UnifyHistory is a hybrid algorithm with a master and a pair of sub-solvers. The master is Incremental Total Order (ITO) that generates candidate orderings of episode events. The first sub-solver is ITC that is used to check the temporal consistency of the candidate order, while the second sub-solver searches for a candidate assignment to state variables that is consistent with the candidate ordering.

CDES uses the exact same framework as UnifyHistory. We add a decomposition module that is able to decompose an Episodic CSP into several independent sub-problems. We improve ITO by considering the conflicts discovered by the sub-solvers such that the ordering generation can be guided and more efficient.

We also use ITC to check the temporal consistency of a total order. As UnifyHistory searches a candidate assignment of finite-domain state variables, we uses the module ISC that extends OpSat [40] to handle mixed continuous-discrete state variables to find the candidate assignment. To improve the efficiency, ISC decomposes the ordered problem to a set of CSPs and only checks a subset of these CSPs that are most important to the consistency. As many total orders are generated by the master in sequence, ISC is also incremental, which only checks the CSPs that have been changed with respect to the previous orders.

6.2 CDES Algorithm

As described in Algorithm 1, the approach starts with an initial total order $\mathcal{L}$ of all the events EV that is consistent with respect to partial orders Φ (Line 6). Then, CDES invokes CDITO to generate consistent total orders (Line 10) whose corresponding temporal consistency and state consistency are incrementally checked by $\mathcal{ITC}$ and $\mathcal{TSC}$, respectively.

Algorithm 1 is called with an Episodic CSP. Following decomposition, CDES infers a partial ordering on events of an Episodic CSP, which is similar to UnifyHistory. It

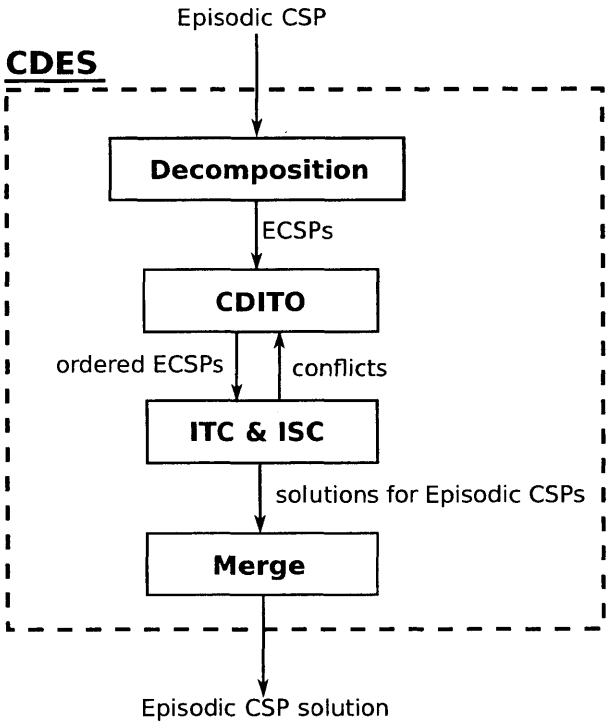

Figure 6-1: Conflict-directed Episodic Satisfaction (CDES) architecture.

accomplishes this by inferring all implicit temporal constraints from the STN *stn* of the Episodic CSP and extracting partial orders from the implied STN (Line 4). The implied STN is inferred by computing All-Pairs Shortest Paths (APSP) on the distance graph of the STN [10] . Each implied simple temporal constraint implies a partial order when its lower bound is positive.

Then, the generator of candidate orderings is initialized before moving to the main candidate generation loop. The initialization involves the total order $\mathcal{L}$, the temporal consistency algorithm $\mathcal{ITC}$, the state consistency algorithm $\mathcal{ISC}$, and the search state $\mathcal{P}$. A total order $\mathcal{L}$ of all the events is initialized that is consistent with respect to Φ. For simplicity and without loss of generality, we refer to events by indices from 1 to n, thus the first $\mathcal{L}_r$ is $(1, 2, ..., n)$, where n is the number of events.[1] With *stn* and $\mathcal{L}$, we initialize ITC $\mathcal{ITC}$ by adding a temporal constraint $(0, +\infty)$ to *stn* for every pair of partial orders implied by $\mathcal{L}$ (Line 7). Line 8 initializes ISC $\mathcal{ISC}$ under $\mathcal{L}_r$. We

[1] In other total order $\mathcal{L} = (p_1, p_2, ..., p_n)$, $p_x = y$ means y is the event index or event, and x is the position of y.

Algorithm 1: CDES

Input: an Episodic CSP instance $ecsp = \langle EV, SV, EP \rangle$
Output: an ordered and constrained Episodic CSP or {}

1 $P = \{p_1, p_2, ..\} \leftarrow$ Decompose($ecsp$) ;
2 **for** $p_i \in P$ **do**
3 $\quad n \leftarrow$ the number of events in EV ;
4 $\quad stn \leftarrow EV$ and duration constraints in p_i ;
5 $\quad \Phi \leftarrow$ all partial orders in APSP(stn) ;
$\quad$ // $\mathcal{L} = (p_1, p_2, ..., p_n)$
6 $\quad \mathcal{L}_r \leftarrow (1, 2, ..., n)$;
7 $\quad \mathcal{ITC} \leftarrow$ initialize $\mathcal{ITC}$ with stn and $\mathcal{L}_r$;
8 $\quad \mathcal{ISC} \leftarrow$ initialize $\mathcal{ISC}$ with p_i and $\mathcal{L}_r$;
9 $\quad \mathcal{P} \leftarrow [(n, 1, 1)]$;
10 $\quad \mathcal{L}_i \leftarrow$ CDITO($\mathcal{L}_r, \mathcal{P}, \Phi, (\mathcal{ITC} \circ \mathcal{ISC})$);
11 $\quad sol_i \leftarrow$ Ground($p_i, \mathcal{L}_i, (\mathcal{ITC} \circ \mathcal{ISC})$)
12 $sol \leftarrow$ merge($\{sol_1, sol_2, ..\}$) ;
13 **return** sol

compute the solution under a consistent total order (Line 11) for each decomposed problem then merge all the solutions together for the original problem (Line 12).

Algorithm 1 is identical to UnifyHistory except that we add a decomposition module and improve the ordering generation and the state consistency check algorithm.

6.3 Decomposition

To accelerate the solution procedure, we provide a decomposition method that exploits the temporal structure and outputs decomposed problems. Rather than reasoning over the whole problem, decomposing an Episodic CSP into several small problems that can be solved independently is much more efficient.

The decomposition is to divide an Episodic CSP into two Episodic CSPs by an event called the breakpoint. The breakpoint must precede some events and succeed the other events. A breakpoint in Figure 6-2 is the event $E5$, and its partial orders with other events are given in Figure 6-3. We also require that no temporal constraints or state constraints are cross the divided problems, such that these problems can be solved independently in terms of the schedule of events and the state vari-

able assignment. An example of the corresponding decomposition by $E5$ is given in Figure 6-4.

The first step of the decomposition is to translate the STN of an Episodic CSP into a distance graph and apply All-Pairs Shortest Path (APSP) to this distance graph to expose the implicit temporal constraint between every pair of events [10]. This procedure is the same as initializing the partial orders of the problem. Then, a Directed Acyclic Graph (DAG) is constructed by colleting all these events and partial orders .

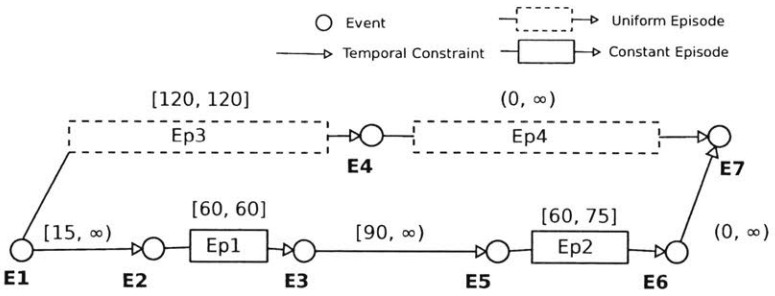

Figure 6-2: Am Episodic CSP example with 7 events, 3 temporal constraints, 2 uniform episodes and 2 constant episodes.

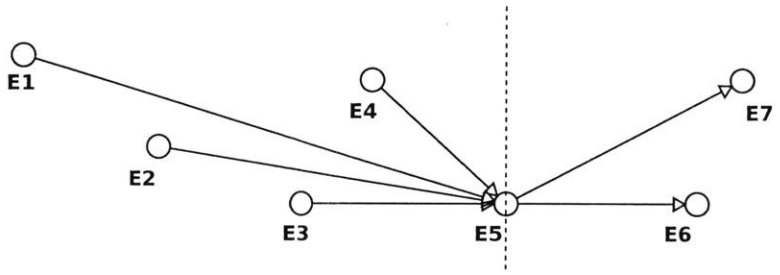

Figure 6-3: The DAG of the example with only the edges starting or ending at the candidate breakpoint $E5$.

Based on this graph, an event that precedes a set of events and succeeds the rest of the events will be found as the candidate breakpoint for the decomposition, which is the necessary condition of decomposition. Event $E5$ is the candidate breakpoint as shown in Figure 6-3 for an example given in Figure 6-2. We keep searching these candidate events until all candidates are exhausted or we find a candidate event whose preceded events and succeeded events have no connection. Specifically, no connection

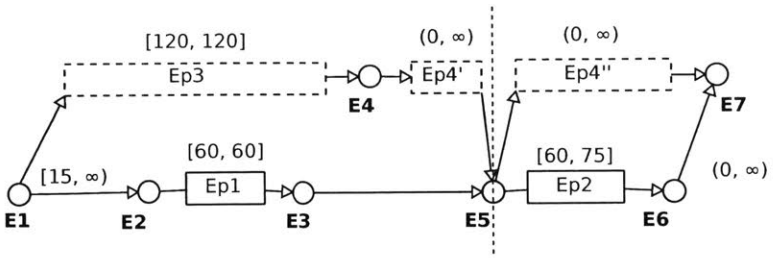

Figure 6-4: Decomposition example with $E5$ as the breakpoint.

equals the conjunction of two conditions:

1 No constant episode is across these two event sets. This is necessary since constant episodes will force state variables to be assigned with the same value across the decomposed problems.

2 No temporal constraint is across these two event sets, in the Minimal Dispatchable Network form of the STN [28]. Minimal Dispatchable Network is equivalent to the original STN but removing all the redundant constraints.

Finally, the STN is divided into two STNs concerning the breakpoint, and the episodes in the original Episodic CSP are projected to each STN. As constant episodes are guaranteed to not cross two STNs, we decompose a uniform episodes that is cross two networks by cutting two episodes into two episodes connected to the breakpoint, as shown in Figure 6-4. The example (Figure 6-2) can be divided into two Episodic CSPs given in Figure 6-4.

6.4 CDITO: Conflict-directed Incremental Total Ordering

The core of CDES is CDITO, which systematically and incrementally explores all the total orders of the events in an Episodic CSP such that temporal consistency and state consistency can be checked independently. We briefly introduce the properties of CDITO in this section and more details can be found in Chapter 7.

CDITO starts with a total of all the events and keeps generating new total orders until reaching a total order under which a consistent plan can be found. For each generation, the new total order is associated with an order change that is a sequence of order moves from the previous total order to the current total order. Consider generating total orders of four events 1, 2, 3, 4. While the previous total order is 1234 and the current total order is 1324, the order change is $[(2 \rightarrow 3), (3 \rightarrow 4)]$, which means 1324 is created from 1234 by moving the 3^{th} event after the 4^{th} event and then 2^{tg} after the 3^{th} event. This ability to output order changes enables ITC and ISC to incrementally check the consistency and avoid unnecessary consistency checking, which is important to the efficiency in solving time.

Another feature of CDITO is that it can interact with ITC and ISC through ordering conflicts that are conflicting partial orders extracted by ITC and ISC from discovered inconsistency along the solving procedure. By leveraging these ordering conflicts, CDITO achieves great efficiency by avoiding generating total orders with similar inconsistency. The method to extract ordering conflicts is introduced Section ??.

6.5 ITC: Incremental Temporal Consistency

Each candidate order needs to be checked for consistency against the STN of the Episodice CSP. This is performed by using the ITC algorithm, which is the same as UnifyHistory to check the temporal consistency of an ordered STN [37]. ITC is capable of updating the previous STN with incremental or decremental changes and continuing the consistency checking, and the algorithmic details of ITC is given in [32].

Extracting Ordering Conflicts from Temporal Inconsistency

When ITC detects some temporal constraints are conflicting, it is able to summarize the inconsistency as ordering conflicts and return them to the CDITO algorithm. In sake of time efficiency, we add this module upon ITC to to avoid similar inconsistency

in the future generations.

As we formally model the temporal requirements as a STN, a total order on the events in the network is equivalent to imposing temporal constraints $(0, \infty)$ on every pair of events whose precedence relation is specified by this total order. As these imposed constraints tighten the network, temporal inconsistency may be introduced. In the distance graph form of this network, a temporal consistency checking algorithm is able to detect negative cycles that are composed of inconsistent temporal constraints. Given a negative cycle, we use a partial order $R_i^t = x_i^- \prec x_i^+$ to represent every temporal constraint that is added because of total ordering and involved in the cycle. The ordering conflict c^t is used to represent this negative cycle as follows:

$$c^t = \underset{i}{\wedge} R_i^t = \underset{i}{\wedge}(x_i^- \prec x_i^+). \tag{6.1}$$

6.6 ISC: Incremental State Consistency

If an ordered Episodic CSP is temporally consistent, the final step is to check its state consistency. As mentioned in the previous sections, we already decompose the temporal constraints and the state constraints of the Episodic CSP by ordering the events. When it comes to checking the state consistency, the approach of considering the ordered Episodic CSP as a single CSP is complete but inefficient. Plus, it is unsound to check the state constraints of every stage between adjacent event pairs separately because of the presence of the constant episodes. Because these constant episodes force their scoped state variables to keep constant during their stages and results in the interaction between stages.

Instead of considering the ordered Episodic CSP as a single CSP, we decompose it into several CSPs. In contrast to the constant episodes, the uniform episodes does not pose constraining effects across stages. Thus, when two stages are not constrained by the same constant episodes, their consistency can be checked, respectively. For example, in Figure 6-5(a) and Figure 6-5(c), each stage can be considered independently. However, in Figure 6-5(b), the stage between $E2$ and $E3$ need to be checked together because they are covered by the same constant episode $Ep1$.

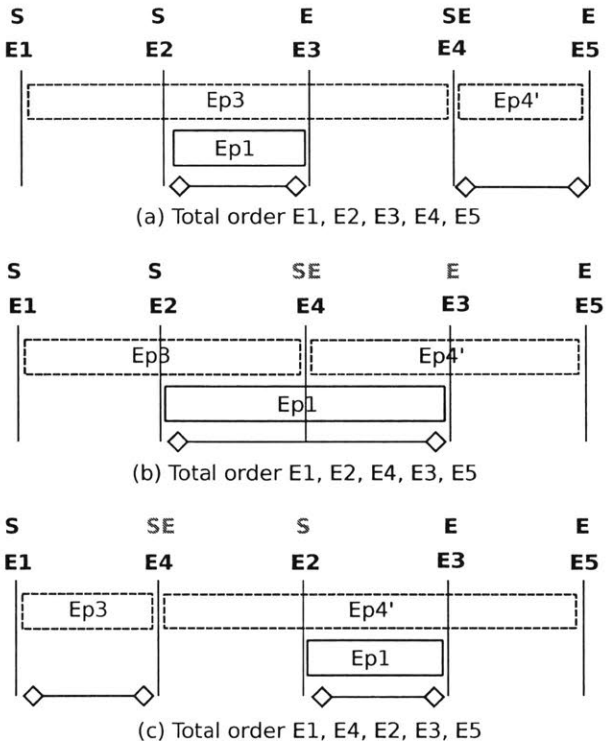

Figure 6-5: Main Components (MCs) of ordered Episodic CSPs.

To efficiently check state consistency, we only check the state consistency of part of the stages that are most critical to the consistency of the problem. Note that state constraints in one stage can be a superset of state constraints in other stages. In this case, the consistency of the superset stage implies the consistency of its subset stages, and a problem is compacted by only including superset stages but still being sound. For example, in Figure 6-5(a), the stage between $E2$ and $E3$ contains all the state constraints of its neighbors, and thus its two neighbors are pruned, and only the marked stages are considered. We call a set of connected superset stages as a Main Component (MC).

To introduce how to extract an MC (i.e., a set of connected superset stages), we first label the events according to whether they start or end episodes. There are three labels: S means the event starts some episode (e.g., $E2$); E means the event ends some episode (e.g., $E3$); an SE event start some episode and end other episodes (e.g., $E4$); We do not consider events neither starting or ending any event since they are

Table 6.1: Order change effects to Main Components (MCs).

MC Relaxed	**MC Tightened**
$S \leftrightarrow E$	$E \leftrightarrow S$
$S \leftrightarrow SE$	$SE \leftrightarrow S$
$SE \leftrightarrow E$	$E \leftrightarrow SE$
No Change	**MC Changed**
$S \leftrightarrow S$	$SE \leftrightarrow SE$
$E \leftrightarrow E$	

not involved in checking state consistency. The algorithm to extract MCs starts from the earliest event and ends when reaching the latest event.

As CDITO searches total orders by moving an event after another event, we can improve the state consistency checking efficiency by analyzing these moves. As shown in Table 6.1, the moves between different events can relax, tighten, change (i.e., the effects are unknown) or make no change to the state constraints of an MC. For a new total order, ISC only needs to check two kinds of MCs:

1 MCs that were inconsistent in the previous total order but have been relaxed and changed by the current total order.

2 MCs that were consistent in the previous total order but have been tightened and changed by the current total order.

If there is one inconsistent MC, the total order is inconsistent. By using main components and incrementally check the changed MCs, ISC is able to quickly determine the state consistency of an ordered Episodic CSP.

Extracting Ordering Conflicts from State Inconsistency

Now we introduce how to extract ordering conflicts from state inconsistency with are inconsistent overlap. Consider an example where two episodes $EP1$ and $EP2$ cannot overlap. As $EP1$ starts at 1 and ends at 5, and $EP2$ starts at 2 and ends at 4, this ordering conflict can be represented as $c_5 = (1 \prec 4) \wedge (2 \prec 5)\text{m}$ where $(a \prec b)$ means a should precede b. This conflict compactly captures all the combinations of this

overlap: 1254, 1245, 2154, and 2145, which means the start events of these episodes happen before the end events.

We represent the ordering conflict of the overlap between two episodes as a conjunction of two partial orders: $R^s_{ij} = (x_i^{\vdash} \prec x_j^{\dashv}) \wedge (x_j^{\vdash} \prec x_i^{\dashv})$, where i and j are the indices of these two episodes; $x_i^{\vdash}$ and $x_j^{\vdash}$ are the start events; $x_i^{\dashv}$ and $x_j^{\dashv}$ are the end events. When multiple episodes overlap, the ordering conflict c^s is as follows:

$$c^s = \bigwedge_{i,j} R^s_{ij} = \bigwedge_{i,j} (x_i^{\vdash} \prec x_j^{\dashv}). \tag{6.2}$$

where each R^s_{ij} represents the overlap of two episodes.

Assume that m episodes overlap with each other, Equation 6.2 can capture this overlap with a conjunction of $m(m-1)$ partial orders. In every total order featuring this overlap, if we halve the involved events into two groups with respect to this total order, all the start events are in the first group, and all the end events are in the second group. Thus, this overlap can be described by specifying all the precedence relations between start events and end events while it can happen in at least $(m!)^2$ total orders.

6.7 Summary

In this chapter, we present CDES, an Episodic CSP solver that consists of a decomposition module and an ordering method called CDITO followed by two sub-solvers ITC and ISC that check the state and temporal consistency of the ordered problem, respectively. In addition to efficiently decomposing the problems, CDES is able to leverage ordering conflicts from the sub-solvers to manipulate the total orders of the events and then incrementally checks the consistency of the changed parts.

We have introduced the details of all these four modules except CDITO. We discuss the algorithmic details of CDITO in this next chapter.

Chapter 7

CDITO: Conflict-directed Incremental Total Ordering

In this chapter, we address the ordering problem that decides the execution order of events in Episodic CSPs. By total ordering the events, we can decompose the temporal and state constraints of Episodic CSPs and solve them independently. For example, when two network flows represented as episodes are competing over the bandwidth resource on a communication link, and their starts and ends are not ordered, they can be routed to different links or scheduled to different times to avoid conflicts. This is a joint decision over states such as routing and time such as schedules. If the flows are ordered, then we know they are overlapped or disjoint over the timeline. Based on the ordering, we can decide their routes between adjeacent events although the exact schedule is unknown. Meanwhile, the schedule only needs to be consistent with the temporal constraints and the imposed orderings.

As a total order is decided first for decomposition, finding a consistent total order under which a solution exists is central to solving Episodic CSPs. However, the ordering problem with complex underlying constraints is NP-hard, and the solution space is factorial in the event number. To efficiently generate such a consistent total order, we use an ordering algorithm called Conflict-directed Incremental Total Ordering (CDITO). This algorithm incrementally and systematically generates total orders by applying conflict-directed search in a special tree structure of total orders.

CDITO is built upon Incremental Total Ordering (ITO) that is used in the tBurton planner [37]. CDITO extends ITO with conflict-directed search [40] to leverage the ordering conflicts, Since CDITO is enhanced to handle disjunctive partial orders, it can leverage the ordering conflicts from both the ordering relations and sub-solvers. Both CDITO and ITO use the same tree structure for arranging total orders, which was first introduced by [29].

We begin by introducing a motivating example that will be used to illustrate the algorithmic details of CDITO. In Section 7.1, we give the formal definition of our ordering problem. We also formulate the motivating example as an ordering problem in this section. Then, we review the related work that inspires CDITO in Section 7.2. As a background, we introduce the total order tree and a search strategy within this tree in Section 7.3 and Section 7.4, respectively. We also present the definitions of ordering conflicts and the method to extract ordering conflicts in Section 7.5 . The method to resolve ordering conflicts in introduced Section **??**. Then, we introduce the method to obtain a resolving move that resolves ordering conflicts and considers the search state in Section 7.7. In Section 7.8, the algorithmic details of CDITO are introduced, and we focus on how to apply a resolving move to the total order search. We end the chapter by introducing a method to relax strictly-ordered total orders generated by CDITO to non-strict orderings in Section 7.9.

First, we present a motivating example in Figure 7-1. This problem is an Episodic CSP with five events, three episodes, and five temporal constraints, three bounding the duration of the three episodes, and two relating different episodes. Each episode specifies a network request. Because determining whether multiple network flows can overlap in our motivating example is equivalent to routing them with the constraints of bandwidth, loss and delay, which is NP-hard, we assume it is computationally expensive to check the consistency of the overlaps between episodes. We also assume that all these episodes can overlap with each other except $EP1$ and $EP3$. In addition, there is an ordering constraint that forces $E3$ or $E4$ to precede $E1$, and we denote it as two temporal constraints $(0, \infty)$ with a circular arc between them.

There is only one consistent total order $\{E2, E4, E1, E3, E5\}$ for the events under

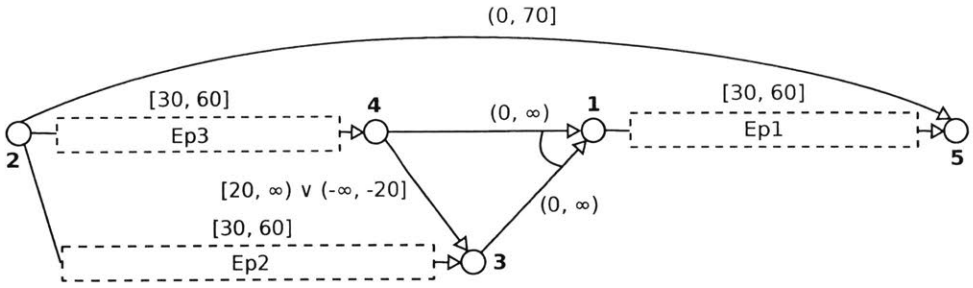

Figure 7-1: Ordering problem example as an Episodic CSP.

which a consistent solution exists. Given the temporal constraints, we know $EP1$ has to overlap with either $EP2$ or $EP3$. Since $EP1$ and $EP3$ cannot overlap, $E1$ should precede $E3$ and thus succeed $E4$. Therefore, there exists one and only one consistent total order that is given in Figure 7-1.

As episodes represent network flow requests, this problem involves routing and resource allocation concerning multiple characteristics as shown in Section 1.1, which is already NP-hard and computationally expensive to check the consistency. Solving this problem is even more complex since we need to make these decisions over time. We will show that our method can efficiently find this consistent total order by leveraging the ordering information extracted from the state and temporal requirements on demand, and the conflicts discovered during the search.

7.1 Problem Formulation

In this section, we introduce the definitions of the ordering problem and give an ordering problem instance of the motivating example in Figure 7-1. We also introduce the definitions of the consistency checking function and the conflict extraction function, which CDITO takes as input along with an ordering problem.

Ordering Problem

The ordering problem is defined as follows:

Definition 16 (Ordering Problem). *An ordering problem is a triple* $\langle E, \Phi, \Pi \rangle$*:*

- *E is a set of n events represented by the natural numbers $\{1, 2, .., n\}$.*
- *Φ is an ordering relation represented by a set of clauses whose disjuncts are partial orders. Each partial order $(a \prec b)$ constrains $a \in E$ to precede $b \in E$.*
- *Π is a set of constraints over total orders $\mathcal{L}$ of E, where a total order of E is an order sequence of the elements of E.*

A candidate solution of this ordering problem is a total order of E. $\mathcal{L}$ is a solution if and only if $\mathcal{L}$ satisfies both Φ and Π. Note that total orders of events E are subject to not only disjunctive partial orders Φ but also additional constraints Π. Constraints Π should hold but are difficult to formulate as disjunctive partial orders, such as metric temporal constraints and consistency of overlaps in the motivating example.

The motivating example can be formulated as follows:

- $E = \{1, 2, 3, 4, 5\} \equiv \{E1, E2, E3, E4, E5\}$ as shown in Figure 7-1.
- $\Phi = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}$, where $\varphi_1 = 1 \prec 5$, $\varphi_2 = 2 \prec 3$, and $\varphi_3 = 2 \prec 4$ constrain each episode's start to precede its end, and $\varphi_4 = (3 \prec 1) \vee (4 \prec 1)$ captures the disjunctive ordering constraint.
- $\Pi = \{\pi_1, \pi_2\}$, where constraint π_1 represent the state constraints, and constraint π_2 represent the metric temporal constraints.

Consistency Checking Functions

CDITO also takes as input a set of consistency checking functions F. For each constraint π that scopes on total orders $\mathcal{L}$ and the constraint π, we associate π with a consistency checking function f. If $\mathcal{L}$ is consistency with π, $f(\mathcal{L}, \pi) = \top$; otherwise, $f(\mathcal{L}) = \bot$.

For the ordering problem of the motivating example, we have two consistency functions $F = \{f_1, f_2\}$: f_1 is the CP solver in [5] to check the state consistency of overlaps. In this example, the CP solver only returns $\bot$ when $EP1$ and $EP3$ overlap;

we use Incremental Temporal Consistency [32] as f_1 to check the temporal consistency of a total order with respect to the temporal constraints. In the motivating example, if both of the events 3 and 4 are before 1, the problem is temporally inconsistent since the duration between 2 and 5 would be at least 80, which exceeds its upper bound 70. This inconsistency is summarized as an ordering con

Conflict Extraction Function

In addition to the consistency checking functions, We also associate each constraint $\pi \in \Pi$ with a conflict extraction function h that maps a total order and the constraint π to a set of ordering conflicts if $f(\mathcal{L}, \pi) = \bot$, where f is the consistency checking function of π. The definition of ordering conflicts is given in Section 7.5

For the the motivating example, we have two conflict extraction functions $H = \{h_1, h_2\}$ for state inconsistency and temporal inconsistency, which are introduce in Section 6.6 and Section 6.5, respectively.

Along the search, two ordering conflicts c_5 and c_6 will be extracted from temporal and state inconsistency, respectively, where $c_5 = \{(1 \prec 4), (2 \prec 5)\}$ and $c_6 = \{(1 \prec 3), (1 \prec 4)\}$. The methods to extract ordering conflicts from temporal and state inconsistency are introduced in Section 6.5 and Section 6.6, respectively.

7.2 Related Work

Our solution approach CDITO is built upon the idea of searching the total order tree introduced in [29]. In this tree, total orders are arranged in a specific structure such that some subtrees can be pruned concerning violated partial orders. As [29] generates all total orders in parallel, ITO modifies it to generate one total order at a time on demand [38, 37]. ITO is used in tBurton to unify multiple timelines [38, 37] and can solve our ordering problem. CDITO extends ITO with a conflict extraction and pruning algorithm in the spirit of the conflict-directed search for satisfiability but tailored to the unique properties of the total order tree.

For total orders found to be inconsistent, we may reason over which partial orders

led to this inconsistency. As we are working over a total order tree, we may leverage the insights of [40] to focus our search by pruning the search space that has similar inconsistency. In addition to common conflict-directed search techniques, CDITO leverages the special structure of the total order tree to jump to promising candidate orderings quickly. However, applying conflict-directed search to this total order tree is not intuitive since the operations of resolving conflicts might be restricted within this tree. We use the same idea of [40] to compute the resolving move of multiple conflicts but apply the resolving move by considering the properties of this tree.

7.3 Total Order Tree

We begin by reviewing the ordering tree and its construction, first introduced in [29]. While the ideas of this tree structure and the notion of the level was first introduced by [29], we add the notion of order moves and pseudo moves for the convenience of describing CDITO.

We first intorude the total order tree. In the total order tree of $E = \{1, 2, .., n\}$, each node is a total order of E, and each edge is an operation of altering partial orders of total orders. Rooted at $\mathcal{L}_r$ = (1,2,..,n), the tree is constructed by iteratively expanding all the children of each total order. An example of the tree of four events are givin in Figure 7-2. The tree expansion uses the notion of total order level that is defined by Ono:

Definition. (Level) Given a root total order $\mathcal{L}_r$, the level of a total order $\mathcal{L} = (p_1, p_2, .., p_n) \neq \mathcal{L}_r$ is the minimum index l such that $p_l \neq l$. The level of $\mathcal{L}_r$ is n.

One example of a level is that the second element in 1324 is not 2, which is the first index from the left whose element is not in the right place, such that the level of 1324 is 2. Another example is that the level of 4321 is 1.

To generate a child of a total order $\mathcal{L} = (p_1, p_2, .., p_n)$, Ono deletes $p_i = i$ from $\mathcal{L}$ and then inserts it in somewhere $(p_{i+1}, p_{i+2}, .., p_n)$. In this paper, we formally define this operation of moving an event within a total order as an order move:

Definition. (Order Move) An order move $(i \rightarrow j)$ deletes p_i from a total order $\mathcal{L} = (p_1, p_2, .., p_n)$ and inserts it right after p_j to obtain a total order $\mathcal{L}'$, where $i \leq n$ and $j \leq n$. This operation is denoted as $\mathcal{L}' = \mathcal{L} \oplus (i \rightarrow j)$.

One example of an order move is that for order move $(1 \rightarrow 3)$, $1234 \oplus (2 \rightarrow 3) = 1324$. Another example is that $4321 \oplus (2 \rightarrow 4) = 4213$.

As each edge in the tree is an order move from a parent to its child, only a subset of order moves is included in the tree. Thus, we have the notion of a feasible move, which is defined as follows:

Definition. (Feasible Move) Let $\mathcal{L}$ be a total order with level l, an order move $(i \rightarrow j)$ from $\mathcal{L}$ is a feasible move if and only if $i < l$ and $i < j$.

Basically, a feasible move $(i \rightarrow j)$ from $\mathcal{L}$ with level l should satisfy two conditions: (1) $i < l$: a feasible move should only move an event that is less than l. For example, $(2 \rightarrow 3)$ is a feasible move for 1234 whose level is 4, but an infeasible move for 4321 whose level is 1; (2) $i < j$: this move should only right shift an event. For example, $(2 \rightarrow 3)$ is feasible move, but $(3 \rightarrow 2)$ is infeasible. The total order tree of $E = \{1, 2, 3, 4\}$ is given as an example in Figure 7-2, which gives are the total orders of E and all the feasible moves.

In addition to feasible moves, we also define the pseudo move that moves from a total order to a sibling with the same level. Although these pseudo moves are not feasible edges in the total order tree, they can be resolving moves that are resolving order moves in CDITO, which will be seen in Section 7.8. The pseudo moves are defined as follows:

Definition. (Pseudo Move) Let $\mathcal{L} = (p_1, p_2, .., p_n)$ be a total order with level l, an order move $(i \rightarrow j)$ from $\mathcal{L}$ is a pseudo move if and only if $p_i = l$ and $i < j$.

As we can see, the difference between feasible moves and pseudo moves are that while the former moves the event with indices less than level l, the latter only moves the event l. An example of pseudo move is that $(2 \rightarrow 4)$ is an pseudo move from 2134 and moves to the sibling 2341.

Following Definition 7.3, we have Proposition 1, which says a pseudo move always moves an event with an index larger than its level.

Proposition 1. If an order move $(i \rightarrow j)$ is a pseudo move from a total order $\mathcal{L}$, we have $i > l$.

Proposition 1 holds because if a total order has level l, the event l has been right shifted, and thus its index will be larger than l. Therefore, every pseudo move $(i \rightarrow j)$ has $(i > l)$. As we will use pseudo move to update the search state in CDITO, Proposition 1 is important to distinguish pseudo moves from other moves.

As the total order tree constrains feasible moves with respect to the total order's level and moving directions, an important property of this tree can be obtained as Lemma 1.

Lemma 1. For a total order with level l, the subsequence of events $\{l, l+1, .., n\}$ remain in its descendants.

Proof. Since child generation does not allow moving events that are larger than the parent's level, $\{l, l+1, .., n\}$ are not moved, and the subsequence of $\{l, l+1, .., n\}$ remains in this total order's children. Given that all descendants' levels are less than l, the subsequence of the events $\{l, l+1, .., n\}$ remain in the descendants of this total order as well. □

Lemma 1 gives the partial orders that remain in a subtree of a total order, such that CDITO can prune a subtree when these partial orders are inconsistent. By using Figure 7-2, one example of Lemma 1 is that 1243 has level 3, and thus all its descendants have the partial order 43 since both 3 and 4 are no less than 3.

7.4 Total Order Search

Next, we introduce the search strategy within this total order tree. Our search strategy is depth-first search, which is different from the breadth-first search order used in [29]. We explore the tree in the same order as ITO [37]. However, while ITO

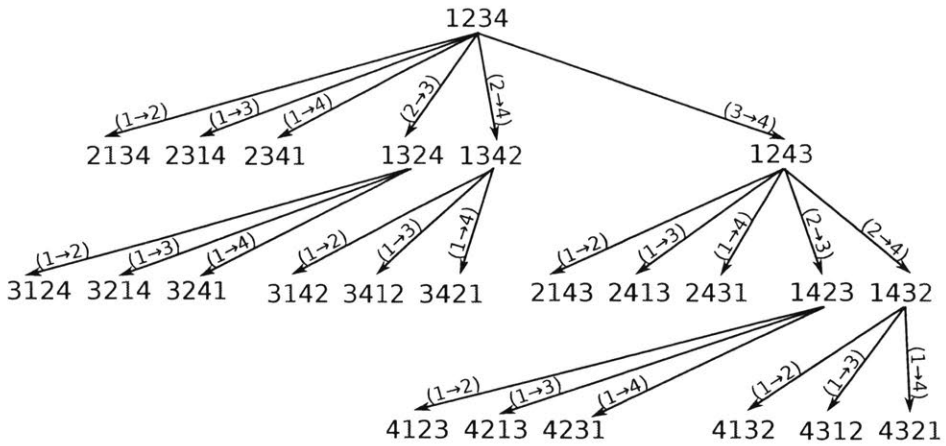

Figure 7-2: Total order tree of $E = \{1, 2, 3, 4\}$. The nodes are total orders; the edges are order moves; the levels of total orders are blue.

uses a queue to store the next several order moves for both children expansion and backtrack, we use a stack of search states to record which children can be explored from the current total order. Thus, we know which children can be visited to resolves conflicts, which makes our search strategy easier to be extended to a conflict-directed algorithm.

The search follows these three rules: (1) children are visited before siblings, which means our strategy is a depth-first search. For example, in Figure 7-2, after 1324 is visited, the search visits its child 3124 instead of its sibling 1342; (2) the algorithm backtracks when the children of the current total order are exhausted, or if the current node is a leaf node. For example, after the search visits 3421, it backtracks to 1342; (3) from a total order, the group of children with the lowest level i is generated first, and within a group, children are generated by right shifting i until the right end. For example, the children of 1243 are visited in the order of 2$\underline{1}$43, 24$\underline{1}$3, 243$\underline{1}$, 14$\underline{2}$3, and 143$\underline{2}$, and the corresponding order moves are $(1 \to 2)$, $(1 \to 3)$, $(1 \to 4)$, $(2 \to 3)$, and $(2 \to 4)$. Rule (3) is also given in Lemma 2

Lemma 2. Let $(i \to j)$ and $(i' \to j')$ be different feasible moves from a total order, $(i \to j)$ is applied first if $(ni + j) < (ni' + j')$; otherwise, $(i' \to j')$ is applied first. If $(i \to j)$ is applied first, we say $(i \to j)$ is before $(i' \to j')$, or $(i' \to j')$ is after $(i \to j)$.

Total Order Search (Algorithm 2) takes as input the number of events $n = |E|$

Algorithm 2: Total Order Search

```
   Input: n
   Output: 𝒪
 1 ℒ ← (1, 2, ..., n) ;
 2 𝒪 ← [ℒ] ;
 3 𝒫 ← [(1, 1, n)] ;
 4 while 𝒫 ≠ {} do
 5 |  (i, j, l) ← 𝒫[1] ;
 6 |  if j < n then
 7 |  |  (i†, j†) ← (i, j + 1)
 8 |  else if j = n then
 9 |  |  (i†, j†) ← (i + 1, i + 2)
10 |  if i† < l then
11 |  |  𝒫[1] ← (i†, j†, l);
12 |  |  ℒ ← ℒ ⊕ (i† → j†) ;
13 |  |  push ℒ to 𝒪 ;
14 |  |  push (1, 1, i†) to 𝒫 ;
15 |  else if i† = l then
16 |  |  pop 𝒫;
17 |  |  if 𝒫 ≠ {} then
18 |  |  |  (i′, j′, l′) ← 𝒫[1] ;
19 |  |  |  ℒ ← ℒ ⊕ (j′ → (i′ − 1));
20 return 𝒪
```

and outputs all the total orders $\mathcal{O}$ by following the aforementioned three rules. In Total Order Search, we use the search state (i, j, l) of a total order $\mathcal{L}$ to compute the next move, where $(i \to j)$ records the last applied order move from $\mathcal{L}$, and l is the level of $\mathcal{L}$. Note that we can know the feasible children from level l and unexplored children from the last order move $(i \to j)$. Thus, the children to which $\mathcal{L}$ can move are determine by its search state. We use a stack $\mathcal{P}$ to store all the search states from the root total order to the current total order. We initialize P with $[(1, 1, n)]$ for the root total order, which means no order move has been performed for the root total order and its level is n (Line 3). An example of search state in Figure 7-2 is that, after the search visits 2431 and backtracks to 1243, the search state of 1243 and 1234 are $(1, 4, 3)$ and $(3, 4, 4)$ respectively. Thus, we have $P = [(1, 4, 3), (3, 4, 4)]$. Note that $\mathcal{P}[1]$ is the search state of the current total order. To generate a child, our algorithm reads (i, j, l) from $\mathcal{P}$ (Line 5) and computes the next order move $(i^\dagger \to j^\dagger)$ (Lines 6-

9). If $i^{\dagger} < l$, we apply this order move to obtain a child (Lines 11-14), otherwise the search backtracks (Lines 16-19). Lines 6-9 enforce the feasible moves from a total order $\mathcal{L}$ to be sorted in the ascending order of $(ni + j)$, which follows Lemma 2.

7.5 Conflict Extraction

In this section, we introduce the definition of ordering conflicts, which follows the naming convention of conflicts in [40].

As a total order is equivalent to a set of partial orders or a conjunction of partial orders, an ordering conflict is a subset of these partial orders that are inconsistent with respect to ordering relation Φ or constraint Π. Formally we define the ordering conflict as follows:

Definition 17 (Ordering Conflict). *An ordering conflict c of an ordering relation ϕ is a conjunction of partial orders that is inconsistent with ϕ.*

For example, $c_4 = (1 \prec 3) \wedge (1 \prec 4)$ is inconsistent with clause $\varphi_4 = (3 \prec 1) \vee (4 \prec 1)$, and thus c_4 is an ordering conflict of φ_4. Another example is that $c_5 = (1 \prec 4) \wedge (2 \prec 5)$ is inconsistent with respect to the state constraint π_1 so that c_5 is an ordering conflict of π_1. Note that we also treat constraints as ordering relations when discussing conflicts. While a total order is determined toe be inconsistent by a corresponding consistency checking function, the ordering conflicts are extracted by the corresponding conflict extraction function.

Definition 18 (Minimal Ordering Conflict). *A minimal ordering conflict of an ordering relation ϕ is an ordering conflict of ϕ, no proper subset of which is an ordering conflict.*

An illustrated example of a minimal ordering conflict is that c_4 is the minimal ordering conflict of φ_4. However, $(1 \prec 3) \wedge (1 \prec 4) \wedge (1 \prec 5)$ is not the minimal conflict of φ_4 since its subset c_4 is an ordering conflict of φ_4. We also assume the ordering conflicts returned by a conflict extraction function is minimal.

Definition 19 (State Ordering Conflict). *Let α be a set of partial orders, α manifests c if $c \subseteq \alpha$; otherwise, α resolves c. If α manifests c, then c is called a state conflict of α. For convenience, if an order move $(i \rightarrow j)$ moves a total order $\mathcal{L}$ to another total order that manifests c, we say $(i \rightarrow j)$ manifests c at $\mathcal{L}$; otherwise, $(i \rightarrow j)$ resolves c at $\mathcal{L}$.*

While total order 12345 manifests c_4, 23145 resolves c_4 since $(1 \prec 3)$ is not a partial order included in 23145. While 12345 manifests c_5, and c_5 is a state conflict of 12345, 23415 resolves c_5 since it is not a superset of c_5.

Now we show how to extract minimal ordering conflicts. Recall that a total order $\mathcal{L}$ of E is equivalent to a set of partial orders for each pair events in E, thus it is a full assignment as in the state space search. Given the ordering relation Φ for our ordering problem, each ordering relation $\varphi \in \Phi$ is a disjunction of partial orders, and $\mathcal{L}$ is inconsistent with φ if and only if $\mathcal{L}$ is inconsistent with all disjuncts of φ. Thus, if φ is violated, we collect all the negations of its disjuncts as an ordering conflict. This ordering conflict is minimal since by removing any of its conjunct, φ is satisfied. The methods to extract ordering conflicts from constraints such as temporal constraints and state constraints are introduced in Section 6.5 and Section 6.6, respectively.

7.6 Conflict Resolution

In this section, we present the method to resolve ordering conflicts. We begin by introducing the definitions of resolving moves. Then, we introduce the method to compute first resolving moves of both a single conflict and multiple conflicts. A resolving move jumps to a total order or a subtree such that inconsistent total orders are skipped. Among all the resolving moves, we mainly discuss about the first resolving moves. By first, we refer to the order of applying moves in Lemma 2. All the examples used in this section can be found in Figure 7-3. The definition of the resolving move is as follows:

Definition 20 (Resolving Move). *Let $\mathcal{L}$ be total order with level l, and C is a set of ordering conflicts, a resolving move $(i \rightarrow j)$ is a feasible move or a pseudo move such*

that it moves to a total order that resolves c or a subtree in which a total order that resolves c exists. We say $(i \rightarrow j)$ *is the first resolving move if any other resolving move is after* $(i \rightarrow j)$.

Then, we discuss the resolving moves for both single conflicts and multiple conflicts.

7.6.1 Single-Conflict Resolving Move

We first show an example of a resolving move of a single conflict. Consider the root total order 12345 with level 5 in our motivating example. This total order violates the clause $\varphi_4 = (3 \prec 1) \vee (4 \prec 1)$, and the corresponding ordering conflict is $c_4 = (1 \prec 3) \wedge (1 \prec 4)$. In order to resolve c_4, the next generated total order should include partial order $(3 \prec 1)$ or $(4 \prec 1)$. Therefore, order moves $(1 \rightarrow 3)$, $(1 \rightarrow 4)$, and $(1 \rightarrow 5)$ are the resolving moves of c_4 from 12345. Recall that an order move $(i \rightarrow j)$ with smaller $(ni + j)$ is applied first by Lemma 2, and the first resolving move is $(1 \rightarrow 3)$ that moves to 23145. As we can see, the first resolving move of an ordering conflict c can directly move to a total order that resolves c. Because as ordering conflicts is a conjunction of partial orders, an order move that negates any of its partial order is an resolving move, which does not require a sequence of moves to resolve.

For the sake of completeness, the search only applies the first resolving move even if the the subsequent moves $(1 \rightarrow 4)$ and $(1 \rightarrow 5)$ can also resolve c_4. Consider an example of sacrificing completeness by taking order moves coming after $(1 \rightarrow 3)$. If we choose an order move that comes after $(1 \rightarrow 3)$ such as $(1 \rightarrow 5)$, we will skip the candidate solution 23145. By taking $(1 \rightarrow 3)$, we can still reach 23145 by taking $(1 \rightarrow 4)$ as the next order move.

Note that the computation of resolving moves depends on the current total order. Recall that an order move $(i \rightarrow j)$ means moving the i^{th} event after the j^{th} event in a total order instead of moving event i after event j. The first resolving move varies when the current total order is different. In our motivating example, both 12345 and

12435 violate $\varphi_5 = (4 \prec 1) \vee (5 \prec 2)$ and share the conflict $c_5 = (1 \prec 4) \wedge (2 \prec 5)$. However, their first resolving moves are $(1 \rightarrow 4)$ and $(1 \rightarrow 3)$, respectively, since 4 is the fourth event in 12345 but the third in 12435.

Recall that a resolving move is either a feasible move, which moves to a child, or a pseudo move, which moves to a sibling. Thus, resolving moves also consider the total order's level, which determines the set of these two moves from a total order. The priority between them follows Lemma 2 so that feasible moves are considered before pseudo moves. The above example of applying $(1 \rightarrow 3)$ to resolves c_4 from 12345 is a feasible move. An illustrated example of a pseudo move as the first resolving move is that given 23145 manifesting the conflict $c_5 = (1 \prec 4) \wedge (2 \prec 5)$, $(3 \rightarrow 4)$ is the first resolving move that moves to its sibling 23415. However, when resolving a conflict requires moving an event larger than l, no valid resolving move exists. For example, resolving the conflict $c_2 = (3 \prec 2)$ from 13245 requires moving 3. However, 3 cannot be moved in the subtree rooted at 13245 because the level of 13245 is $2 < 3$.

Formally, we show the method to compute the first resolving move. Consider a total order $\mathcal{L} = (p_1, p_2, ..., p_n)$ with level l and an ordering conflict $c_r = \underset{s}{\wedge} q_{rs}$. We iterate over every $q_{rs} = (p_{i'} \prec p_{j'})$ and then determine the first resolving move $(i_r^\dagger \rightarrow j_r^\dagger)$, which is computed as follows:

$$(i_r^\dagger \rightarrow j_r^\dagger) = \underset{(i' \rightarrow j') \in \Delta_r}{\operatorname{argmin}} (ni' + j'), \tag{7.1}$$

where $\Delta_r = \{(i' \rightarrow j') \mid ((p_{i'} \prec p_{j'}) \in c_r) \wedge (p_{i'} \leq l)\} \cup \{(n \rightarrow n+1)\}$, and $(i' \rightarrow j')$ is the first order move to satisfy partial order $(p_{j'} \prec p_{i'})$ of c_r.

By Lemma 2, the first resolving move of c_r is the order move $(i' \rightarrow j') \in \Delta_r$ with the smallest $(ni' + j')$, which is the first to resolve c_r by negating some partial order in an ordering conflict. Note that Δ_r constrains all the moves that are feasible or pseudo by forcing $(p_{i'} \leq l)$. If no such a move exists, $\Delta_r = \{(n \rightarrow n+1)\}$ and $(i_r^\dagger \rightarrow j_r^\dagger) = (n \rightarrow n+1)$. In another word, if any feasible move or pseudo move exists as resolving moves, $(n \rightarrow n+1)$ will not be selected since it is after any order move by Lemma 2. Therefore, Equation 7.1 outputs $(n \rightarrow n+1)$ or the first resolving move.

7.6.2 Multiple-Conflict Resolving Move

Now we introduce how to combine multiple resolving moves to compute resolving moves of all the ordering conflicts.

Recall that any order move before a first resolving move of a single conflict is inconsistent, and thus the first multiple-conflict resolving move is the furthest among all the first single-conflict resolving moves. For example, the total order 12345 violates $\varphi_4 = (3 \prec 1) \vee (4 \prec 1)$ and $\varphi_5 = (4 \prec 1) \vee (5 \prec 2)$, resulting in the ordering conflicts $c_4 = (1 \prec 3) \wedge (1 \prec 4)$ and $c_5 = (1 \prec 4) \wedge (2 \prec 5)$. The first resolving moves of c_4 and c_5 are $(1 \rightarrow 3)$ and $(1 \rightarrow 4)$, respectively. The first multiple-conflict resolving move of c_4 and c_5 should be $(1 \rightarrow 4)$, which is the first move to resolve both conflicts. If we apply a move before $(1 \rightarrow 4)$ such as $(1 \rightarrow 3)$ and then obtain 23145, we notice that 23145 still manifests c_5.

Formally, we show the method to compute the first multiple-conflict resolving move. Given a total order $\mathcal{L}$ and a set of ordering conflicts $\mathcal{C}$, we first compute the first resolving move $(i_r^\dagger \rightarrow j_r^\dagger)$ of every ordering conflict $c_r \in \mathcal{C}$. Then, the first resolving move $(i^\dagger \rightarrow j^\dagger)$ of $\mathcal{C}$ is obtained as follows:

$$(i^\dagger \rightarrow j^\dagger) = \underset{(i_r^\dagger \rightarrow j_r^\dagger) \in \Delta}{\operatorname{argmax}} (ni_r^\dagger + j_r^\dagger), \tag{7.2}$$

where $\Delta = \{(i_r^\dagger \rightarrow j_r^\dagger) \mid (i_r^\dagger \rightarrow j_r^\dagger)$ is the first resolving move of $c_r \in \mathcal{C}\}$. As any order move before a first resolving move manifests some ordering conflicts, we choose the furthest order move in Δ. Thus, the order move obtained by Equation ?? is the first resolving move before which all the order moves manifest some ordering conflicts.

7.7 NextMove Algorithm

In this section, we introduce NEXTMOVE (Algorithm 3) that considers both ordering conflicts and search states to generate a resolving move. While the resolving moves or the first resolving move discussed in the previous section are for conflicts, we also consider the search state in the implementation Algorithm 3. Figure 7-3 gives the examples of computing resolving moves for eight iterations for the motivating

Algorithm 3: NEXTMOVE

Input: $\langle \mathcal{L}, C, i, j, l \rangle$// $\mathcal{L} = (p_1, p_2, .., p_n)$
Output: $(i^\dagger, j^\dagger)$

```
1  if ni + j ≥ nl then
2  |  (i†, j†) ← (i', i' + 1) for (p_i' = l) in L
3  if j < n then
4  |  (i†, j†) ← (i, j + 1)
5  else
6  |  (i†, j†) ← (i + 1, i + 2)
7  for c_r ∈ C do
8  |  (i_r†, j_r†) ← (n, n + 1);
9  |  for q_rs ∈ c_r do
10 |  |  (p_i' ≺ p_j') ← q_rs in L ;
11 |  |  if p'_i ≤ l and (ni' + j') ≤ (ni† + j†) then
12 |  |  |  (i_r†, j_r†) ← (1, 1) and break
13 |  |  if p'_i ≤ l and (ni' + j') < (ni_r† + j_r†) then
14 |  |  |  (i_r†, j_r†) ← (i', j')
15 |  if p_{i_r†} > l then return (n, n + 1);
16 |  if (ni† + j†) < (ni_r† + j_r†) then (i†, j†) ← (i_r†, j_r†);
17 return (i†, j†)
```

example, and the resolving moves are in blue. Consider the fourth iteration as an example. As 12345 manifest c_4 and c_5, the first resolving move of these two conflicts from 12345 is $(1 \rightarrow 4)$. However, given the search state is $(2, 2, 5)$, the corresponding resolving move is $(2 \rightarrow 3)$.

NEXTMOVE takes as input a total order $\mathcal{L}$, a set of ordering conflicts C, and search state (i, j, l). The algorithm outputs the first resolving move that has not been applied from $\mathcal{L}$.

Lines 1-6 compute the initial next move with respect to the last applied order move: Lines 3-6 are the same as Lines 6-9 in Algorithm 2; Line 1-2 generates an pseudo move to the adjacent sibling when its children are exhausted, or $(n, n+1)$ when both its children and siblings with the same level are exhausted, which is identical to the move of an unresolvable conflict. Recall that for the search state (i, j, l) of $\mathcal{L}$, $(i \rightarrow j)$ is the last applied order move from $\mathcal{L}$ and l is the level of $\mathcal{L}$. By $(i \rightarrow j)$, we know any order move before $(i \rightarrow j)$ has been applied or skipped and thus is

unavailable.

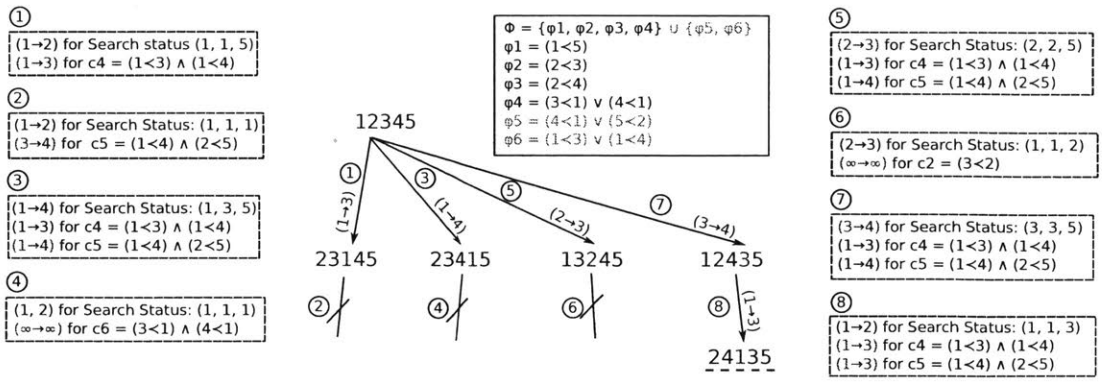

Figure 7-3: Solving the motivating example by using CDITO in eight iterations. The procedures of computing resolving moves for these iterations are given in the dotted boxes; the resolving moves are in blue; the implicit ordering relations φ_5 and φ_6 are in red and discovered by the second and fourth iterations, respectively.

The inner loop (Lines 8-14) computes the first resolving move $(i_r^\dagger \rightarrow j_r^\dagger)$ of each ordering conflict $q_{rs} \in c_r$ by following Equation 7.1, and the outer loop computes the first resolving move $(i^\dagger \rightarrow j^\dagger)$ with respect to all the single-conflict resolving moves by following Equation 7.2. In the inner loop, the algorithm begins with $(n \rightarrow n+1)$ (Line 8) and updates the first resolving move with a nearer order move that negates a partial order (Lines 13-14). To accelerate this procedure, Line 12 breaks the inner loop when $(i_r^\dagger \rightarrow j_r^\dagger)$ would be entering an inconsistent subtree with respect to the incumbent of the resolving move by following Equation 7.2. In the outer loop, Lines 1-6 initialize $(i^\dagger \rightarrow j^\dagger)$ with respect to the last order move (i, j). The algorithm updates $(i^\dagger \rightarrow j^\dagger)$ when a resolving move that jumps further is found (Line 16). To accelerate this procedure, we return $(n, n+1)$ when an ordering conflict is unresolvable (Line 15).

7.8 CDITO Algorithm

In this section, we present the algorithmic details of CDITO (Algorithm 4). CDITO follows the same search strategy of Total Order Search (Algorithm 2) within a total order tree and uses resolving moves to jump over inconsistent total orders (Algorithm 3). The focus of this section is how to use resolving moves to guide search. We

also show the eight iterations of solving the motivation example in Figure 7-3 and Table 7.1.

Algorithm 4 takes as input a total order $\mathcal{L}$, a stack of search states $\mathcal{P}$, an ordering relation Φ, a set of constraints Φ, a set of consistency checking functions H, and a set of conflict extraction functions H. CDITO outputs either an empty set (Line 23) or a total order that satisfies Φ and Π (Lines 2-5).

#	$\mathcal{L}$	P	(i,j,l)	Resolving Move
1	12345	$[(1,1,5)]$	$(1,1,5)$	$(1 \rightarrow 3)$
2	23145	$[(1,1,1),(1,3,5))]$	$(1,1,1)$	$(3 \rightarrow 4)$
3	12345	$[(1,3,5)]$	$(1,3,5)$	$(1 \rightarrow 4)$
4	23415	$[(1,1,1),(1,4,5)]$	$(1,1,1)$	$(5 \rightarrow 6)$
5	12345	$[(2,2,5)]$	$(2,2,5)$	$(2 \rightarrow 3)$
6	13245	$[(1,1,2),(2,3,5)]$	$(1,1,2)$	$(5 \rightarrow 6)$
7	12345	$[((3,3,5)]$	$(3,3,5)$	$(3 \rightarrow 4)$
8	12435	$[(1,1,3),(3,4,5)]$	$(1,1,3)$	$(1 \rightarrow 3)$
9	24135	$[(1,1,1),(1,1,3),(3,4,5)]$	$(1,1,1)$	Solution

Table 7.1: Solving the motivating example by using CDIITO in eight iterations.

The consistency checking functions F take as input $\mathcal{L}$ and Π and determine the consistency of a total order with repsect to Π. Some additional ordering relations can be extracted by negating the ordering conflicts $\mathcal{C}_h$ found by the conflict extraction functions H on demand. Then, these relations are added into Φ to avoid generating total orders with similar inconsistency (Line 8). As shown in Figure 7-3, H is invoked two times in the total eight iterations: the second iteration extracts an ordering relation φ_5 from inconsistent concurrency, and the fourth iteration extracts φ_6 from temporal inconsistency.

In order to compute the resolving move $(i^\dagger \rightarrow j^\dagger)$, the algorithm collects all the ordering conflicts $\mathcal{C}$ (Line 10) and inputs $\mathcal{C}$ into NEXTMOVE (Algorithm 3) along with the current total order $\mathcal{L}$ and the search state (i,j,l) (Line 11).

The resolving moves $(i^\dagger \rightarrow j^\dagger)$ output by Algorithm 3 are of four types, and they are handled differently (Lines 12-22): (1) $i^\dagger < l$; (2) $l < i^\dagger < n$; and (3) $i^\dagger = n$. As shown in Figure 7-3, the motivating example is solved by 8 iterations: the second iteration is of type (2), the fourth and the sixth iterations are of type (3), and the

Algorithm 4: CDITO

Input: $\langle \mathcal{L}, \mathcal{P}, \Phi, \Pi, F, H \rangle$// $\mathcal{L} = (p_1, p_2, .., p_n)$
Output: $\mathcal{L}$ or $\{\}$

```
1  while Φ is consistent and P ≠ {} do
2      if L satisfies Φ then
3          consistent? ← F(L, Π);
4          if consistent? = ⊤ then
5              return L
6          else
7              C_h = H(L, Π) ;
8              Φ ← Φ ∪ {φ_r = ¬c_r | c_r ∈ C_h}
9      (i, j, l) ← P[1] ;
10     C ← {c_r = ¬φ_r | L violates φ_r ∈ Φ} ;
11     (i†, j†) ← NextMove(L, C, i, j, l) ;
12     if i† < l then
13         L ← L ⊕ (i† → j†);
14         P[1] ← (i†, j†, l);
15         push (1, 1, i†) to P ;
16     else
17         pop P;
18         if P ≠ {} then
19             (i', j', l') ← P[1] ;
20             L ← L ⊕ (j' → (i' − 1));
21             if l < i† < n then P[1] ← (i', j† − 1, l') ;
22             if i† = n then P[1] ← ((i' + 1), (i' + 1), l') ;
23 return {};
```

other iterations are of type (1).

When a resolving move is of type (1) and $i^\dagger < l$, this resolving move is a feasible move and we directly apply it to generate a child (Lines 13-15). Consider the third iteration as an example. The current total order is 12345, its search state is $(1, 3, 5)$, and the resolving move output by Algorithm 3 is $(1 \rightarrow 4)$. Then, search moves to 23415, a child of 12345. For the other two types, CDITO backtracks (Lines 17-20) but updates the parent's search state in different ways. In the following two paragraphs, we introduce the update rules of type (2) and type (3).

When a resolving move is of type (2) and $l < i^\dagger < n$, this resolving move is a pseudo move, and CDITO updates the parent's search state to $(i', j^\dagger - 1, l')$ (Line 21). Take the second iteration as an example, the total order is 23145 and the resolving move

is $(3 \rightarrow 4)$. If we could apply $(3 \rightarrow 4)$ to 23145, we would obtain 23415, which is the sibling of 23145. However, $(3 \rightarrow 4)$ is a pseudo move from 23145, which cannot be directly applied. Thus we update its parent's search state to $(1, 3, 5)$, and the next order move from its parent is $(1 \rightarrow 4)$ that moves to 23415.

When the resolving move is of type (3) and $i^{\dagger} = n$, this resolving move is neither feasible or pseudo, and CDITO updates the parent's search state to $(i' + 1, i' + 1, l')$ (Line 22). By updateing, CDITO prunes all the descendants of the current total order, all its siblings with level l, and all the descendants of these siblings. Since there exist some ordering conflicts that require moving an event larger than l, these conflicts remain in its siblings with level l. By Lemma 1, these conflicts also remain in the descendants of this total order and these siblings as well. Take the fourth iteration as an example, the current total order is 23415, and its parent search state is $(1, 4, 5)$. As the resolving move is $(5 \rightarrow 6)$, all its siblings with the level 1 are inconsistent, thus we update its parent's search state to $(2, 2, 5)$, and its parent will explore the children with level 2 in the next iteration.

As CDITO can handle ordering conflitcs including multiple partial orders, we are able to guide the total order search by leveraging conflicts from both disjunctive partial orders and temporal and state consistency, which is critical to efficiently solving ordering problems.

7.9 Incorporating Non-strict Orderings

The aforementioned CDITO algorithm is able to generate a consistent total order of the events of Episodic CSPs. As only the strict ordering relation $\prec$ in a total order is considered so far, we introduce a method to relax the total order obtained by CDITO into a more flexible ordering $\mathcal{L}_R$ with the non-strict ordering relation $\preceq$ in this section. While the non-strict ordering method introduced in [37] forces equality constraint between events to disjoint inconsistent overlaps, our method relaxes a strict ordering solution to be non-strict ordering, which avoids an expensive search for disjointing inconsistent overlaps.

For example, we can relax the total order $\mathcal{L} = 23145 \equiv (2 \prec 3 \prec 1 \prec 4 \prec 5)$ into a total order $\mathcal{L} = 23145 \equiv (2 \prec 3 \prec 1 \prec 4 \preceq 5)$, which means $EP1$ and $EP2$ can end at the same time. We begin by proving the relaxed total order still satisfies the temporal and state consistency discussed in this chapter. Then, we provide the relaxation method and solve the motivating example with this method.

We first show that as $\mathcal{L}$ is temporally consistent, its corresponding non-strict total order $\mathcal{L}_R$ is also consistent. As every partial order $a \prec b$ introduces the temporal constraints $(0, \infty)$ into a Simple Temporal Networks, and $a \preceq b$ introduces $[0, \infty)$, changing $(0, \infty)$ to $[0, \infty)$ is equivalent to relaxing the temporal problem. Therefore, the temporal consistency still holds for the relaxed total order. For example, by relaxing the strict total ordering between 4 and 5, the temporal consistency still holds.

The state consistency of $\mathcal{L}$ also holds for its relaxed total order $\mathcal{L}_R$. We prove this by first showing every pair of episodes is still state consistent in two conditions: (1) for every pair of episodes that is disjoint because of the partial order $a \prec b$ in $\mathcal{L}$, the end event of an episode that is a precedes the start event of the other episode that is b. By relaxing $a \prec b$ into $a \preceq b$, these two episodes are still disjoint; (2) for every pair of episodes that overlaps because of some partial orders in $\mathcal{L}$, these two episodes either are disjoint or still overlap when we relax some strict orderings, and the state consistency still holds in either case. As we have proved that the state consistency still holds for every pair of episodes under a relaxed total order $\mathcal{L}_R$, we know $\mathcal{L}_R$ is also state consistent.

As the temporal and state consistency still holds for $\mathcal{L}_R$, the problem is to find such $\mathcal{L}_R$ that satisfies the ordering relation in the ordering problem. Now, we introduce the method to relax $\mathcal{L}$ to a total order $\mathcal{L}_R$ with as less $\prec$ as possible by solving an Optimal CSP. Given a total order $\mathcal{L} = (p_1, p_2, .., p_n) \equiv (p_1 \prec p_2, .., \prec p_n)$ of the events E with size n, and the ordering relation $\Phi = \{\phi\}$, the Optimal CSP for this relaxation problem $\langle X, C, g \rangle$ is as follows:

- X is a set of $(n-1)$ integer variables with the domain $\{0, 1\}$ and determines the relaxed total order by specifying the ordering relation between the adjacent

events in $\mathcal{L}_R$. For $i \in \{1, 2, .., n-1\}$, $x_i \in X$ represents the ordering relation between p_i and p_{i+1} in $\mathcal{L}$, where $(x_i = 0) \implies (p_i \prec p_{i+1})$, and $(x_i = 1) \implies (p_i \preceq p_{i+1})$.

- C is a set of constraints that scope on a subset of X and constrains the relaxed total order to still satisfy the ordering relation Φ. For each clause $\phi_r \in \Phi$, we extract a constraint c_r into C as follows:

  1. We remove all the partial orders of ϕ_r that are inconsistent with $\mathcal{L}$ and obtain a removeed clause $\phi_r^* = \wedge q_{rs}$;
  2. Then, for each $q_{rs} = (p_i \prec p_j) \in \phi_r^*$, we have $c_{rs} = (x_i = 0) \vee (x_{i+1} = 0), .., \vee (x_{j-1} = 0)$.
  3. Finally, we have $C_r = \vee c_{rs}$.

- $g : X \to \{0, 1, 2, .., n-1\}$ is the maximization objective function that represents the number of non-strict orderings in the relaxed total order, and $g = \sum_{x_i \in X} x_i$.

Now we continue with the solution of our motivating example $\mathcal{L} = (2 \prec 4 \prec 1 \prec 3 \prec 5)$ and formulate the corresponding relaxation problem as an optimal CSP $\langle X, C, g \rangle$ as follows:

- $X = \{x_1, x_2, x_3, x_4\}$ with the domain $\{0, 1\}$.
- $C = \{c_1, c_2, c_3, c_4, c_5, c_6\}$ are extracted from $\Phi = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_5, \varphi_6\}$ as shown in Table 7.2.
- $g = x_1 + x_2 + x_3 + x_4$.

We use the conflict-directed search method introduced in [40] to solve this Optimal CSP. For the above example of relaxing $\mathcal{L} = (2 \prec 4 \prec 1 \prec 3 \prec 5)$, the best solution is $\{x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 1\}$ with the maximum objective 1, which leads to the relaxed total order $\mathcal{L}_R = (2 \prec 4 \prec 1 \prec 3 \preceq 5)$.

r	φ_r	φ_r^*	c_r
1	$1 \prec 5$	$1 \prec 5$	$(x_3 = 0) \vee (x_4 = 0)$
2	$2 \prec 3$	$2 \prec 3$	$(x_1 = 0) \vee (x_2 = 0) \vee (x_3 = 0)$
3	$2 \prec 4$	$2 \prec 4$	$x_1 = 0$
4	$3 \prec 1 \vee (4 \prec 1)$	$4 \prec 1$	$x_2 = 0$
5	$(4 \prec 1) \vee (5 \prec 2)$	$(4 \prec 1)$	$x_2 = 0$
6	$(1 \prec 3) \vee (1 \prec 4)$	$(1 \prec 3)$	$x_3 = 0$

Table 7.2: Extracting constraints C from the ordering relation Φ for the motivating example with the solution $\mathcal{L} = (2 \prec 4 \prec 1 \prec 3 \prec 5)$.

7.10 Summary

In this section, we presented CDITO, a systematic and incremental algorithm to efficiently order the events in a partially ordered plan by applying conflict-directed search on a tree of total orders. During the search, our method computes resolving moves by considering ordering conflicts and search states. By leveraging these resolving moves, CDITO is able to skip inconsistent total orders, exploiting the special structure of this tree. We also introduced a method to relax strictly-ordered total orders generated by CDITO to non-strict ordering to achieve higher flexibility.

Chapter 8

Experimental Results

Amundsen has been experimented as part of a network configuration system to manage network flows with mission specifications on loss, delay, and bandwidth over time. Our motivating example is also a temporal network configuration problem. We implemented the following experiments to compare Amundsen against UnifyHistory over these problems with different size and complexity and thus demonstrated Amundsen is able to handle large-scale configuration problems and is also much faster than the state-of-art configuration manager UnifyHistory. UnifyHistory is fast since it leverages the Incremental Total Ordering to fastly generate total orders and incrementally checks the temporal consistency. Amundsen is built upon Unifyhistory and thus uses the same hybrid approach, which is comprised of a master for generating total orders and sub-solvers to check the state and temporal consistency.

8.1 Experiment Description

The problems were provided by a communication network simulator that generates network flows with random duration constraints and characteristic requirements on a meshed network.

The simulator setup is as follows: (1) the mission horizon is 300s; (2) the meshed network has 16 nodes and 240 links; (3) the loss, delay, and bandwidth of each link are uniformly generated from continuous intervals [0.1,0.3]%, [0.1,0.3]s, and

[500,1000]kbps; (4) the loss, delay, throughput, and minimum duration of each transfer mission are uniformly generated from [0.1,0.3]%, [0.1,0.3]s, [600,1000]kbps, and [20,80]s; (5) the generator adds temporal constraints between randomly chosen events with a duration (0,100], and the number of temporal constraints is one fifth of the mission number;

8.2 Results

We test five problems of 10, 20, 30, 40, and 50 missions with Amundsen and a baseline solver that is the core component in tBurton to unify multiple timelines and manage device configurations over time [37]. We ran 100 trials for each scenario, and the timeout was 20 seconds that is the duration between two replan requests in real-world experimental devices.

#flows	Amundsen			UnifyHistory		
	#solved	N_S	N_U	#solved	N'_S	N'_U
10	94	7	221	9	1	233
20	91	6	27	5	1	38
30	86	5	10	6	1	18
40	82	4	16	11	1	21
50	74	5	14	8	1	24

Table 8.1: Experimental results. **#solved**: number of solved trials; N_S, N_U: average number of total order generations in solved and unsolved trails by using Amundsen; N'_S, N'_U: average number of total order generations in solved and unsolved trails by using the baseline solver.

Table 8.1 shows that Amundsen solves nearly 80% of the problems, while the baseline solver solves only 10% of the problems in 20 seconds. It can be seen that Amundsen finds consistent solutions quickly after generating around 5 total orders in all the solved trials, which demonstrates that Amundsen is capable of using conflicts to efficiently guide search and avoid unnecessary order generation or consistency check. However, N'_S equals 1 in all solved trials, which means, in large-scale problems, the baseline solver can find solutions only if a proper initial order is generated. Overall, as #flows increases, checking more grounded consistency is more expensive. Thus,

both solvers generate fewer orders in unsolved trials, and we observe the significant decreases of N_U and N'_S. In every scenario, N_U is slightly less than N'_U, which demonstrates that efforts put on reasoning over conflicts are not expensive compared to other costs.

8.3 Summary

In this chapter, we presented the empirical results of our approach by benchmarking against a baseline solver on a communication network simulator. We show that Amundsen is able to successfully manage hundreds of network requests in simulated missions given strict time limits. Amundsen is also able to find plans in 10 times as many scenarios as the baseline solver.

Chapter 9

Conclusion

9.1 Summary of Contributions

As reasoning over complex temporal behaviors is critical to the successful planning and execution of robotic systems, we develop and present Amundsen, a timed configuration manager that is an executive that takes as input a temporal mission specification and models and outputs an execution plan. This timed configuration manager is able to reason over complex models with both continuous and discrete specification to achieve temporal concurrent goals. We summarize the contributions of this thesis as follows:

1. Solving timed configuration management problem

   Our first contribution is an architecture for solving configuration management problems of networked devices over the continuous timeline to achieve temporally concurrent goals. This problem is hard due to its mixed continuous-discrete constraints that and temporal constraint, which are highly coupled. We handle this problem by modeling it as a timed configuration management problem. Then, Amundsen translates this problem into an Episodic CSPs that are solved by an efficient Episodic CSP solver.

2. Solving problems with complex constraints over time

Our second contribution is to defined a new problem called Episodic CSPs and develope a corresponding solution approach called Incremental and Conflict-directed Episodic Satisfaction (ICES). Episodic CSPs extend classical CSP swith the notion of time and serves as a general definition for many classical problems such as Vehicle Routing Problems with Time Windows and Jobshop Scheduling Problems. We carefully investigated the properties of Episodic CSPs and presented a restricted but expressive class called Time-invariant Episodic CSPs. ICES is able to efficiently solve Time-invariant Episodic CSPs by leveraging an efficient decomposition modules and a conflict-directed ordering algorithm.

9.2 Future Works

There are many interesting avenues for future research related to our timed configuration manager. We describe some of them in this section:

Risk-aware Decision Making

The timed configuration manager currently assumes that the state of the world and the system model are perfectly known. Since this is often not the case in the real world, we should consider the uncertainty of the environment. We could consider searching for plans that guarantee success with a certain degree of confidence. We call these plans as chance-constrained plans. The previous work focuses on the chance-constrained problem over discrete time. However, in our case where the timeline is continuous, we would like to investigate the corresponding problems over continuous time, which has not been well studied.

Another important aspect is the uncertainty over time. Simple Temporal Networks with Uncertainty describe temporal problems with uncontrollable time behaviors [36]. In our problem, we can extend the temporal constraints to uncontrollable temporal constraints to capture the uncertainty over time. Then, to reason over such a problem, we can leverage the advanced temporal consistency checking algorithms in terms of

different types of controllability.

Time-variant Episodic Constraint Satisfaction Problem

In this thesis, we only presented the method to solve Episodic CSPs with constant or uniform episodes. However, the general case where episodes represent changing effects such as motion trajectories are still intractable according to what we know. Studying the state constraints that allow arbitrary functions or more general functions such as linearly changing effects with different rates is important for our configuration manager to handle more complex system dynamics while solving reasonably large-scale problems with various temporal features.

Generative Planning Module

Our timed configuration manager currently assumes an automaton can transition between every pair of transitions and each transition is guarded by a control variable. Thus, we cannot handle more complex guards scoping on state variables. To handle these complex guards requires a generative planning module to determine the system transitions. We would extend our system to be generative. The idea is to build a test and trial search module upon the current system that is responsible for generating all the transitions for all sub-systems. They are expected to interact with each other through conflicts that summarize the infeasible transition combinations.

Appendix A

Network Configuration Problem

A.1 Problem Specification

In this section, we provide a formal definition of the network configuration problem with mission specifications. We begin with a motivating example problem from which we will define the network configuration planning problem. We will then outline the inadequacies of a reactive based approach and highlight the desired features of an automated network configuration planner.

Example 1. (Replanning an FTP transfer with a deadline) We have the network topology as in Figure A-1, with six nodes in a fully connected network. For each link, the bandwidth available is $5Mbps$, with loss 1.5%, and 10 millisecond delay.

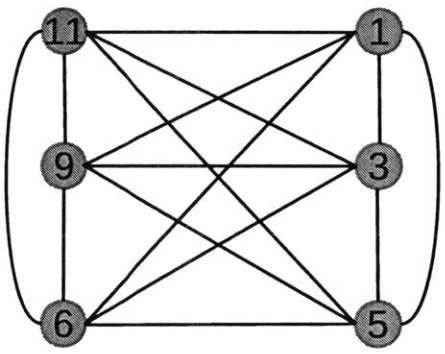

Figure A-1: Network topology for the example problem.

In specifying the mission, we use time index t, such that the start time of the mission occurs at $t = 0$. Initially, there is only an FTP transfer from 9 to 3, of size $560Mb$. The transfer must be completed by time $t = 120s$, and has no delay or loss constraints. At time $t = 30s$, a video flow from 9 to 3 stars, which requires $1Mbps$ bandwidth, less than 1% loss, and up to 15-millisecond delay.

We want to route both flows, such that the FTP transfer finishes by the set time limit, and such that the video flow is placed on a route satisfying the bandwidth, loss and delay constraints. In addition to the choice of routes, we must also choose the configuration of each flow. The configuration choices included dropping the flow, applying FEC with three source packets and one parity packet, or routing without FEC.

The example above serves to illustrate the most essential features of a network configuration planning problem. In such specifications, we must capture features of the network, the mission requirements for the flows, and the allowed configurations for the flows.

Formally, the network configuration problem specifications are defined as follows.

Definition 21. *(Network configuration problem specification) We consider the Network Configuration Problem to consist of the following network specifications:*

- *A set of nodes:* $N = \{1, ..., n\}$;
- $\forall i \in N \; j \in N$, $BW_e[i, j]$ *is the maximum bandwidth between two nodes;*
- $\forall i \in N \; j \in N$, $L_e[i, j]$ *is the expected loss between two nodes; and*
- $\forall i \in N \; j \in N$, $D_e[i, j]$ *is the expected delay between two nodes.*

The problem also contains the following flow requirement specifications:

- *A set of flows:* $M = \{1, ..., m\}$;
- $\forall k \in M$, $BW_f[k]$ *is the minimum required throughput of flow k;*
- $\forall k \in M$, $L_f[k]$ *is the maximum allowable loss of flow k;*

- $\forall k \in M$, $D_f[k]$ *is the maximum allowable delay of flow* k;
- $\forall k \in M$, $source[k] \in N$ *is the start node of flow* k;
- $\forall k \in M$, $sink[k] \in N$ *is the destination node of flow* k; *and*
- H_f *is the maximum allowable hops of flows.*

The problem also contains the following information on available configurations:

- *For each flow* $k \in M$, *a set of configurations:* $C_k = \{1, ..., c_k\}$;
- *Bandwidth on link function* $b : C_k \times \mathbb{R} \rightarrow \mathbb{R}$, *such that* $b(c_i, \tau)$ *is the actual bandwidth on link required after applying FEC* c_i *to a flow with required throughput* τ;
- *Loss on link function* $l : C_k \times [0, 1] \rightarrow [0, 1]$, *such that* $l(c_i, \delta)$ *gives the reduced loss resulting from adopting configuration* c_i *on a link with loss* δ; *and*
- *Utility* $u : C_k \rightarrow \mathbb{R}$, *such that* $u(c_i)$ *gives the utility of choosing configuration* c_i.

In describing the network features, each link from node i to node j $(i, j \in N)$ has three characteristics: bandwidth $BW_e[i, j]$, loss $L_e[i, j]$, and delay $D_e[i, j]$. As the network is meshed, $BW_e[i, j]$ is set to 0 if nodes i and j are not directly connected.

For flow requirements, each flow $k \in M$ has start node at $source[k]$, and destination node $sink[k]$. In addition, the throughput allocated to flows has a minimum required value $BW_f[k]$. Upper bounds on the allowed cumulative delay and loss along the assigned path are also restricted to be below $L_f[k]$ and $D_f[k]$ respectively. We also model H_f an upper bound on the number of hops allowed for any flow.

Each flow k has a set of possible configurations C_k, which represent Drop, Normal, and various FEC settings. Each choice of configuration leads to different bandwidth on links, loss on links and utilities, calculated according to functions b, l and u respectively.

Current network approaches are reactive controls based and do not consider mission specifications. The behavior observed with current systems given the problem in

Example 1 is as follows. At time $t = 0$, the FTP will be placed on the link $9 \rightarrow 3$, and use up all $5Mbps$ of the available bandwidth. At $t = 30s$, there would be approximately $150Mb$ of the FTP transfer completed. However, the controller will react to the new video transfer, and also place the video flow on the $9 \rightarrow 3$ link. As a result, the remaining $410Mb$ of the FTP transfer will only have $4Mbps$ bandwidth allocated. The FTP transfer will then require an additional 100 seconds to complete, and thus miss the transfer deadline.

By inspection, it can be confirmed that, by placing the FTP on an alternative route from 9 to 3, for example on links $9 \rightarrow 1$ and $1 \rightarrow 3$, we can meet the mission specifications for both flows, provided that we can also apply an FEC to correct for the link loss. We thus require a mission aware network configuration planner that autonomously produce such plans. In the subsequent section, we examine some relevant literature on techniques related to our solution method.

A.2 Constraint Modeling and Encoding

In this section, we describe the model used to encode the network configuration planning problem. We first provide a brief overview of the model for FEC used. We then describe the decision variables used to describe the choice of routes and actuation, as well as the auxiliary variables required for constraint checking. Lastly, we describe the set of constraints which ensure that the routes and actuations are feasible given the network and meet the mission specifications.

FEC modeling

This model was developed as part of the EdgeCT DARPA project, and thus must model well-defined actuators. Packet FEC is one such actuator, used to reduce loss on a link at the expense of consuming more bandwidth.

When FEC is applied, q packets of the source are sent with p packets of parity, such that whenever at least q packets are received out of the $p + q$ sent, then the source can be entirely recovered. In the case when less than q packets are received,

only $q' \leq q$ the number of received source packets can be recovered.

The effect on the bandwidth is thus straightforward: for throughput requirement BW, applying q source and p parity FEC means we have bandwidth over each link of

$$\frac{p+q}{p} BW$$

.

We approximated the effect of FEC on loss as follows. For each FEC block of q source and p parity packets, sent over link with loss δ, we may consider the expectation of the *proportion* of packets successfully sent. We will calculate

$$f_{fec}(q, k, 1-\delta) = \frac{1}{q} E\left[\text{source packets recovered}\right]$$

For convenience, define the probability mass function of a binomial distribution as

$$B(q, q', 1-\delta) = \binom{q}{q'} (1-\delta)^n \delta^{q-q'}$$

for q the number of trials, q' the number of successes, and $1-\delta$ the probability of success for each trial.

For FEC, we consider first the case when at least q packets were delivered:

$$\begin{aligned}
&\frac{1}{q} E\left[\text{source packets recovered when at least q delivered}\right] \\
=&\frac{1}{q} \sum_{i=q}^{p+q} h \times B(p+q, i, 1-\delta) \\
=&\sum_{i=q}^{p+q} B(p+q, i, 1-\delta) \qquad \text{(A.1)}
\end{aligned}$$

Consider now the case when fewer than q packets were delivered:

$$\begin{aligned}
&\frac{1}{q}E\,[\text{source packets recovered when fewer than q delivered}] \\
=&\frac{1}{q}\sum_{i=1}^{q-1}\sum_{j=1}^{i} jP(\text{i delivered})P(\text{j of i source}|\text{i delivered}) \\
=&\frac{1}{q}\sum_{i=1}^{h-1}\sum_{j=1}^{i} j \times B(p+q,i,\delta)B(i,j,\frac{q}{p+q}) \qquad \text{(A.2)}
\end{aligned}$$

Summing the two terms we find

$$\begin{aligned}
&f_{fec}(q,p,1-\delta) \\
&=\sum_{i=q}^{p+q} B(p+q,i,1-\delta) \\
&+\frac{1}{q}\sum_{i=1}^{q-1}\sum_{j=1}^{i} j \times B(p+q,i,1-\delta)B(i,j,\frac{q}{p+q}) \qquad \text{(A.3)}
\end{aligned}$$

Given that FEC with q source and p parity was applied, expected loss on a link with loss δ is thus

$$L_{FEC}(q,p,1-\delta) = 1 - f_{fec}(q,p,1-\delta)$$

Variables

In encoding the network configuration planning problem, we create the following variables:

- $\forall i \in N$, $k \in M$, $s[i,k] \in N$ is the direct successor of flow k on vertex i;
- $\forall i \in N$, $k \in M$, $l[i,k]$ is the cumulative loss of flow k on vertex i;
- $\forall i \in N$, $k \in M$, $d[i,k]$is the cumulative delay of flow k on vertex i;
- $\forall i \in N$, $k \in M$, $h[i,k]$is the cumulative hops of flow k on vertex i;
- $\forall k \in M$, $z[k] \in \{1,...,c_k\}$ is the configuration of flow k;
- $\forall k \in M$, $c \in H_k$, $bw[k,c]$ is the minimum required throughput of flow k;

- $\forall k \in M$, $c \in H_k$, $i \in N$, $j \in N$, $l_e[c, i, j]$ is the expected loss between two vertices when a flow with configuration w passes this edge; and

- $\forall k \in M$, $c \in H_k$, $i \in N$, $j \in N$, $d_e[w, i, j]$ is the expected delay between two vertices when a flow with configuration w passes this edge.

As in encodings common to constraint programming of VRP [18], we define a set of successor variables s. Intuitively, every successor variable denotes the direct successor node of a node A on a flow's routing path with an integer domain whose values represent the nodes connected to vertex A. Note that the successor variables on the same node for different flows are different. Thus, $s[i, k]$ for $i \in N, k \in M$ is the successor variable for node i and flow k. Given a set of assignments to successor variables of a flow, we may recursively extract the path as the set of successors from the source of the flow.

In addition to successor variables, for each node i and flow k, we also have variables l, d, and h, representing cumulative loss, delay and hops when flow k reaches vertex i. These variables are used to maintain upper bounds on the cumulative loss, delay, and hops.

Lastly, we also create variables z which describe the choice of configurations. We create the auxiliary variables bw and l_e which describe the effects on bandwidth required on a link and loss on a link.

Constraints

In this section, we describe the constraints used to represent the effects of configuration choices, constraints for network capacity, and constraints to enforce mission specifications.

We begin with constraints representing the effect of FEC. Assume that we have a mapping F_k which gives us a tuple (q, p) of source and parity packets, for any choice of configuration $z[k]$ for flow k:

- When flows dropped:

$$\begin{aligned}(z[k] = c) \wedge (F_k[c] = (0,0)) \Rightarrow \\ (bw[k, z[k]] = 0) \\ \wedge (l_e[k, z[k], i, j] = 0) \\ \wedge (d_e[k, z[k], i, j] = 0) \\ \forall k \in M, i \in N, j \in N \end{aligned} \tag{A.4}$$

- FEC configuration for each flow not dropped:

$$\begin{aligned}(z[k] = c) \wedge (F_k[c] \neq (0,0)) \Rightarrow \\ \left(bw[k, z[k]] = \frac{p+q}{q} BW_f\right) \\ \wedge (l_e[k, z[k], i, j] = L_{FEC}(q, p, L_e[i, j])) \\ \wedge (d_e[k, z[k], i, j] = D_e[i, j]) \\ \forall k \in M, c \in C_k, i \in N, j \in N \end{aligned} \tag{A.5}$$

When the flow is dropped, then the cumulative loss and delays, as well as the bandwidth required on a link is set to zero. This allows the loss and delay constraints to be trivially satisfied, and the flow does not take up space on the links. However, when the flow is not dropped, the bandwidth on link and loss are calculated given the model for FEC above. Note that the application of FEC does not change the delay on a link.

Routing of flows is encoded using a *ProperCircuit* constraint, as in VRPs.

- ProperCircuit: Each flow circuit visits a subset of N

$$ProperCircuit(s[:, k]) \qquad \forall k \in M \tag{A.6}$$

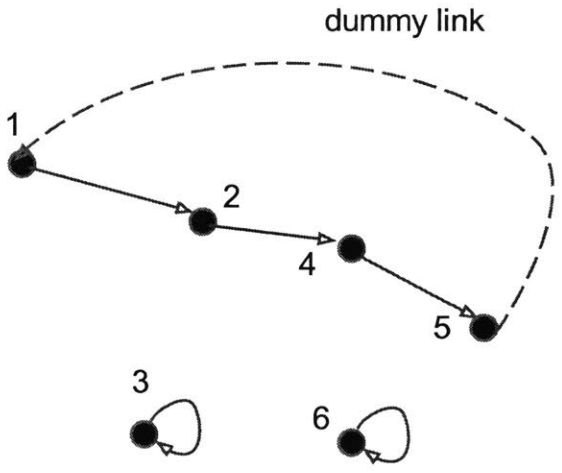

Figure A-2: An example of successor assignments satisfying *ProperCircuit*, for a flow with source 1 and sink 5.

- The successor of each flow's end should be its start:

$$s[sink[k], k] = source[k] \qquad \forall k \in M \tag{A.7}$$

ProperCircuit [?] is a global constraint commonly used in VRP that enforces the requirement for a set of nodes with one circuit visiting once a subset of the nodes. If a node is not connected to any other node, its successor is itself. For example. $\{s[1,1] = 2, s[2,1] = 3, s[3,1] = 1, s[4,1] = 4, s[5,1] = 5, s[6,1] = 6\}$ is a proper circuit, because $\{1,2,3\}$ are in a loop and 4,5,6 point to themselves. We also add a dummy link for each flow, such that the successor of its sink is the source. With this dummy link, the path for every flow is a cycle. An example is given in Figure A-2.

We must also ensure that the flows, given chosen configurations, are routed according to limits on link capacities.

- Edge bandwidth capacity constraint:

$$\sum_{k \in \{k \in M | (s[i,k]=j) \wedge (sink[k] \neq i)\}} bw[k, z[k]] \leq BW_e[i,j]$$
$$\forall i \in N, j \in N \tag{A.8}$$

For each vertex i, the consistency check of bandwidth capacity is performed on all the links. For a link (i, j), if a flow k has passed this link such that $s[i, k] = j$, the throughput requirement of the flow k will considered unless the link (i, j) is a dummy path of flow k such that $sink[k] = i$. Lastly, because the successors of isolated vertices are themselves and these self-loops are also counted, the maximum bandwidth from each vertex to itself should be set as a positive infinite value to satisfy the bandwidth constraint.

Recalling that the flows have upper bounds over allowed accumulated loss, delay, and number of hops, we define the following constraints.

- Loss constraints (conservative approximation with the union bound):

$$\begin{aligned} l[source[k], k] &= 0 \\ l[s[i, k], k] &= l[i, k] + l_e[z[k], i, s[i, k]] \\ l[sink[k], k] &\leq L_f[k] \\ \forall k \in M, i \in \{i \in N | (i \neq s[i, k]) \wedge (i \neq sink[k])\} & \end{aligned} \tag{A.9}$$

- Delay constraints:

$$\begin{aligned} d[source[k], k] &= 0 \\ d[s[i, k], k] &= d[i, k] + d_e[z[k], i, s[i, k]] \\ d[sink[k], k] &\leq delayf[k] \\ \forall k \in M, i \in \{i \in N | (i \neq s[i, k]) \wedge (i \neq sink[k])\} & \end{aligned} \tag{A.10}$$

- Hops Constraints:

$$\begin{aligned} h[source[k], k] &= 0 \\ h[s[i, k], k] &= h[i, k] + 1 \\ h[sink[k], k] &\leq H_f \\ \forall k \in M, i \in \{i \in N | (i \neq s[i, k]) \wedge (i \neq sink[k])\} & \end{aligned} \tag{A.11}$$

For each flow k, except for the isolated nodes, delay is accumulated from the source along the path, stopping when the flow arrives at the sink. We require that the accumulated delay at the sink is less than that allowed in the specifications. To account for the dummy link, we do not accumulate the delay from the sink to the source. Unlike bandwidth constraints, the loss of flow k is not coupled with other flows. As the isolated nodes are not connected to the sink directly or indirectly (because of proper circuit propagation), they do not influence the accumulated delay on every sink.

Similar encodings are used for loss and delay. Note that we chose to enforce the loss bound using by summing loss along the path. This is a conservative approximation using the Union Bound, which is true regardless of whether the losses are independent. The form of the constraint does not change if we assume independence: we can simply impose a constraint over the sum of the log loss.

Objective

In our encoding, the total utility is a linear sum of the utilities for each flow, for utility function for each flow p.

$$\max \sum_{k \in M} p(z[k], h[sink[k], k] \tag{A.12}$$

In our formulation, we would like to use the minimal amount of FEC such that all specifications are met. This allows us to have spare bandwidth for unexpected flows which may arrive during execution. Further, we would like to have short routes, so that we penalize the number of hops required to arrive at the sink. Lastly, we derive zero utility for any dropped flow.

Bibliography

[1] Kenneth R Baker. *Introduction to sequencing and scheduling.* John Wiley & Sons, 1974.

[2] Javier Barreiro, Matthew Boyce, Minh Do, Jeremy Frank, Michael Iatauro, Tatiana Kichkaylo, Paul Morris, James Ong, Emilio Remolina, Tristan Smith, et al. Europa: A platform for ai planning, scheduling, constraint programming, and optimization. *4th International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)*, 2012.

[3] Ralph Becket. Specification of flatzinc, 2014.

[4] Amedeo Cesta and Angelo Oddi. Ddl. 1: A formal description of a constraint representation language for physical domains. *New directions in AI planning*, pages 341–352, 1996.

[5] Jingkai Chen, Cheng Fang, Christian Muise, Howard Shrobe, Brian C Williams, and Peng Yu. Radmax: Risk and deadline aware planning for maximum utility. In *AAAI Workshop on Artificial Intelligence for Cyber Security (AICS'18)*, 2018.

[6] Steve Chien. A generalized timeline representation, services, and interface for automating space mission operations. In *SpaceOps 2012*, page 1275459. 2012.

[7] Steve A Chien, Daniel Tran, Gregg Rabideau, Steve R Schaffer, Dan Mandl, and Stuart Frye. Timeline-based space operations scheduling with external constraints. In *ICAPS*, pages 34–41, 2010.

[8] Patrick R Conrad and Brian Charles Williams. Drake: An efficient executive for temporal plans with choice. *Journal of Artificial Intelligence Research*, 42:607–659, 2011.

[9] Thomas L Dean and Drew V McDermott. Temporal data base management. *Artificial intelligence*, 32(1):1–55, 1987.

[10] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial intelligence*, 49(1-3):61–95, 1991.

[11] Enrique Fernández-González, Erez Karpas, and Brian C Williams. Mixed discrete-continuous heuristic generative planning based on flow tubes. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[12] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[13] R James Firby. The rap language manual. *Animate Agent Project Working Note AAP-6, University of Chicago*, 1995.

[14] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[15] Jeremy Frank and Ari Jónsson. Constraint-based attribute and interval planning. *Constraints*, 8(4):339–364, 2003.

[16] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

[17] Andreas Hofmann and Brian Williams. Exploiting spatial and temporal flexibility for plan execution of hybrid, under-actuated systems. In *AAAI 2006*, 2006.

[18] Philip Kilby and Paul Shaw. Vehicle routing. *Foundations of Artificial Intelligence*, 2:801–836, 2006.

[19] Phil Kim, Brian C Williams, and Mark Abramson. Executing reactive, model-based programs through graph-based temporal planning. In *IJCAI*, pages 487–493, 2001.

[20] Philippe Laborie, Jerome Rogerie, Paul Shaw, and Petr Vilím. Reasoning with conditional time-intervals. part ii: An algebraical model for resources. In *FLAIRS conference*, pages 201–206, 2009.

[21] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. Ibm ilog cp optimizer for scheduling. *Constraints*, 23(2):210–250, 2018.

[22] Thomas Léauté and Brian C Williams. Coordinating agile systems through the model-based execution of temporal plans. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20, page 114. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005.

[23] Steven James Levine and Brian Charles Williams. Concurrent plan recognition and execution for human-robot teams. In *ICAPS*, 2014.

[24] Hui X Li and Brian C Williams. Generative planning for hybrid systems based on flow tubes. In *ICAPS*, pages 206–213, 2008.

[25] Marta Cialdea Mayer, Andrea Orlandini, and Alessandro Umbrico. Planning and execution with flexible timelines: a formal account. *Acta Informatica*, 53(6-8):649–680, 2016.

[26] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language. 1998.

[27] Nicola Muscettola. Hsts: Integrating planning and scheduling. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA ROBOTICS INST, 1993.

[28] Nicola Muscettola, Paul Morris, and Ioannis Tsamardinos. Reformulating temporal plans for efficient execution. In *KR*, pages 444–452, 1998.

[29] Akimitsu Ono and Shin-ichi Nakano. Constant time generation of linear extensions. In *FCT*, pages 445–453. Springer, 2005.

[30] Gurobi Optimization. Gurobi optimizer reference manual, 2016.

[31] Julie Shah, James Wiken, Brian Williams, and Cynthia Breazeal. Improved human-robot team performance using chaski, a human-inspired plan execution system. In *Proceedings of the 6th international conference on Human-robot interaction*, pages 29–36. ACM, 2011.

[32] I-hsiang Shu, Robert T Effinger, Brian Charles Williams, et al. Enabling fast flexible planning through incremental temporal reasoning with conflict extraction. In *ICAPS*, pages 252–261, 2005.

[33] Eric Timmons, Tiago Vaquero, Brian Charles Williams, and Richard Camilli. Preliminary deployment of a risk-aware goal-directed executive on autonomous underwater glider. In *ICAPS*, pages 213–217, 2016.

[34] Alessandro Umbrico, Amedeo Cesta, Marta Mayer, and Andrea Orlandini. Integrating resource management and timeline-based planning. In *ICAPS*, pages 264–272, 2018.

[35] Alessandro Umbrico, Amedeo Cesta, Marta Cialdea Mayer, and Andrea Orlandini. Platinu m: A new framework for planning and acting. In *Conference of the Italian Association for Artificial Intelligence*, pages 498–512. Springer, 2017.

[36] Thierry Vidal. Handling contingency in temporal constraint networks: from consistency to controllabilities. *Journal of Experimental & Theoretical Artificial Intelligence*, 11(1):23–45, 1999.

[37] David Wang. *A Factored Planner for the Temporal Coordination of Autonomous Systems.* PhD thesis, Massachusetts Institute of Technology, May 2015.

[38] David Wang and Brian Williams. tburton: A divide and conquer temporal planner. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[39] Brian C Williams and P Pandurang Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of the national conference on artificial intelligence*, pages 971–978, 1996.

[40] Brian C Williams and Robert J Ragno. Conflict-directed a* and its role in model-based embedded systems. *Discrete Applied Mathematics*, 155(12):1562–1595, 2007.