Linköping Studies in Science and Technology

Licentiate Thesis No. 1722

Efficient Temporal Reasoning with Uncertainty

by

Mikael Nilsson

Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

Linköping 2015

This is a Swedish Licentiate's Thesis

Swedish postgraduate education leads to a doctor's degree and/or a licentiate's degree.
A doctor's degree comprises 240 ECTS credits (4 year of full-time studies).
A licentiate's degree comprises 120 ECTS credits.

ISBN 978-91-7685-991-9
ISSN 0280–7971
Printed by LiU Tryck 2015

URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-119409

Abstract

Automated Planning is an active area within Artificial Intelligence. With the help of computers we can quickly find good plans in complicated problem domains, such as planning for search and rescue after a natural disaster. When planning in realistic domains the exact duration of an action generally cannot be predicted in advance. Temporal planning therefore tends to use upper bounds on durations, with the explicit or implicit assumption that if an action happens to be executed more quickly, the plan will still succeed. However, this assumption is often false. If we finish cooking *too early*, the dinner will be cold before everyone is at home and can eat. Simple Temporal Networks with Uncertainty (STNUs) allow us to model such situations. An STNU-based planner must verify that the temporal problems it generates are executable, which is captured by the property of *dynamic controllability (DC)*. If a plan is not dynamically controllable, adding actions cannot restore controllability. Therefore a planner should verify after each action addition whether the plan remains DC, and if not, backtrack. Verifying dynamic controllability of a full STNU is computationally intensive. Therefore, *incremental* DC verification algorithms are needed.

We start by discussing two existing algorithms relevant to the thesis. These are the very first DC verification algorithm called MMV (by Morris, Muscettola and Vidal) and the incremental DC verification algorithm called FastIDC, which is based on MMV.

We then show that FastIDC is not sound, sometimes labeling networks as dynamically controllable when they are not. We analyze the algorithm to pinpoint the cause and show how the algorithm can be modified to correctly and efficiently detect uncontrollable networks.

In the next part we use insights from this work to re-analyze the MMV algorithm. This algorithm is pseudo-polynomial and was later subsumed by first an $O(n^5)$ algorithm and then an $O(n^4)$ algorithm. We show that the basic techniques used by MMV can in fact be used to create an $O(n^4)$ algorithm for verifying dynamic controllability, with a new termination criterion based on a deeper analysis of MMV. This means that there is now a comparatively easy way of implementing a highly efficient dynamic controllability verification algorithm. From a theoretical viewpoint, understanding MMV is important since it acts as a building block for all subsequent algorithms that verify dynamic controllability. In our analysis we also discuss a change in MMV which reduces the amount of regression needed in the network substantially.

In the final part of the thesis we show that the FastIDC method can result in traversing part of a temporal network multiple times, with constraints slowly tightening towards their final values. As a result of our analysis we then present a new algorithm with an improved traversal strategy that avoids this behavior. The new algorithm, EfficientIDC, has a time complexity which is lower than that of FastIDC. We prove that it is sound and complete.

This research work was funded in part by CUGS (the National Graduate School in Computer Science, Sweden), the Swedish Research Council (VR) Linnaeus Center CADICS, the ELLIIT network organization for Information and Communication Technology, the Swedish Foundation for Strategic Research (CUAS Project), the EU FP7 project SHERPA (grant agreement 600958), and Vinnova NFFP6 Project 2013-01206

Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

Acknowledgements

The writing of this thesis and the research behind have required hard work. It would not have been possible without the support of many people, to whom I am grateful.

I would like to thank both my supervisors: Patrick Doherty and Jonas Kvarnström. Patrick for suggesting the research topic and giving me the opportunity to pursue it. Jonas for making sure that all co-authored publications were of the highest quality. That we never had a rejection testifies to this.

For reviewing and suggesting improvements of the thesis I would like to thank: Patrick Doherty, Jonas Kvarnström, Håkan Warnqvist, Daniel de Leng and Olov Andersson.

I am also grateful for discussions with past and present members at the AIICS division: Olov Andersson, Cyrille Berger, Gianpaolo Conte, Fredrik Heintz, Karol Korwel, David Landén, Daniel de Leng, Martin Magnusson, Tommy Persson, Piotr Rudol, Andrzej Szalas, Mattias Tiger, Håkan Warnqvist, Mariusz Wzorek.

During my years at IDA I have also had assistance from administrative personnel and would especially like to thank Karin Hendry and Anne Moe.

Finally I would like to thank Linda and our parents for unconditional and untiring support over the years.

Contents

List of Figures

Chapter 1

Introduction

Automated Planning is the area of Artificial Intelligence which studies the problem of finding plans, which could be sequences of actions, for achieving objectives. There are many different types of automated planners. They have in common that the end result is a plan detailing which available actions to perform in order to be able to reach the goal. It is the goal of this thesis to provide an efficient algorithm allowing automated planners to reason about time in realistic environments. The environments are realistic in the sense that they may contain events which cannot be controlled, only observed. In the rest of this introductory chapter we will introduce these ideas as well as use cases for planners in more detail.

Simple Temporal Problems. Time and concurrency are increasingly considered essential in automated planning, with temporal representations varying widely in expressivity. Many temporally expressive planners make use of well-known temporal formalisms that have been extensively studied in their own right. One such well-known temporal formalism is that of Simple Temporal Problems (STPs, Dechter et al. [9]), which will be formally defined in the next chapter. In this formalism there are *events* and *requirement constraints* between pairs of these. The formalism can be used to reason about the possibility of assigning timepoints to the events in a way that respects all constraints. This concept, known as *consistency*, is central to reasoning with STPs.

From a planning perspective it is clear that the start and end of every action in a plan can be represented by STP events. Constraints between start and end events can then be used to model action durations. The planner can also use STP constraints as ordering constraints between pairs of ac-

tions that must be executed in a specific order. We now present an example scenario which can be modeled by an STP.

Example 1. *I am at my vacation house in a remote area somewhere where the weather is nice. Suddenly I feel the need to read a good book. I may then enlist the help of an online bookstore which has the book in question and which can also deliver it to my vacation house. I visit the bookstore web page and order the book to be delivered by Unmanned Aerial Vehicle (UAV) to my location. However due to safety concerns delivery will take place at a safe designated area which is close by my vacation house. We then agree on a time of delivery and the bookstore promises that the delivery will be within 10 minutes of the agreed upon time. They also require that I am not more than 5 minutes late when signing off the delivery, so that the UAV may return for its next delivery run.*

Figure 1.1 shows an STP built from the described scenario. The STP is shown in graph form, where nodes represent events and labeled edges represent requirement constraints. When STPs are shown in graph form they are referred to as Simple Temporal Networks or STNs (to be detailed later). Here we have also modeled that the UAV has deliveries prior to mine, and we have included requirement constraints that specify time bounds for all actions: *Deliveries*, *Fly*, *Wait* and *Walk*. They connect the events which mark the start and end of the actions. For example, the interval $[33, 40]$ on the *Fly* action is used to model the fact that flying between the UAV start location and the designated delivery location can take between 33 and 40 minutes.

Given this STP, it is possible to reason about the existence of a consistent assignment of timepoints to all events that satisfies all constraints. However, a limitation within the STP formalism is that reasoning about consistency is limited to the case where the planner chooses all times for when events happen. In a realistic setting this is not possible, since if the planner's actions are used to model real world tasks, the durations of these may be affected by outside influences. An example of this is the time it takes to for the UAV to fly in the scenario. According to our model, we believe it takes between 33 and 40 minutes to fly the distance. However, it is not up to us to *choose* what the exact duration will be. We can only *find out* the *outcome* of the duration after we have actually seen the UAV execute the fly action. Consistency is necessary, but not sufficient if we want to model realistic domains.

Simple Temporal Problems with Uncertainty. A richer temporal formalism which deals with the problem just described is provided by STPs with Uncertainty (STPUs, Vidal and Ghallab [44]). This formalism contains both

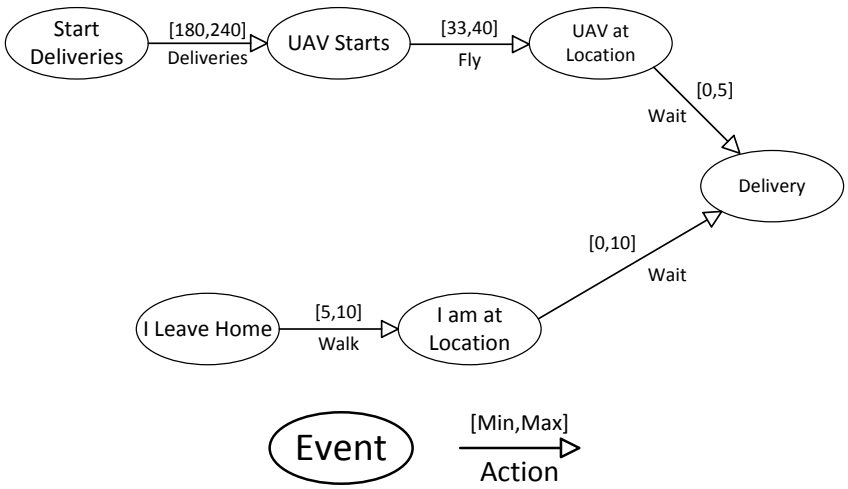

Figure 1.1: Example of a consistent STP in graph form.

controlled events, which are the only events available in the STP formalism, and *uncontrollable* events, corresponding to events which we cannot fully control. An event becomes uncontrollable if it is the target of a *contingent constraint*. Contingent constraints are used to model uncontrollable action durations with lower and upper bounds on durations. STPUs can be displayed in graph form in the same way as STPs can be displayed as STNs. The graph form of the STPU is called STNU and will be defined in Chapter 2.

For STPUs, the interesting issue is not only whether the problem is consistent but how controllable events should be assigned timepoints, depending on our knowledge about those uncontrollable outcomes that have already occurred. This leads to the concept of *controllability*. Depending on which assumption is made regarding information about uncontrollable events, three different controllability types are identified [42, 44]. We will discuss all three in detail in Section 2.4.1. Most importantly, it is reasonable to assume that a realistic system that is executing a plan corresponding to an STPU can make decisions about plan execution dynamically when receiving information about uncontrollable events that have occurred. If an STPU allows us to devise a strategy for *incrementally* scheduling controlled events (starting actions) given that we immediately receive information when a contingent event occurs (an action ends), it is *dynamically controllable* (DC) and can be efficiently executed by what is called a dispatching

algorithm.

When this work began, four algorithms had been published that operated directly on a complete STNU to determine whether it was dynamically controllable.

1. **MMV** (from the authors' surnames: Morris, Muscettola and Vidal) [21]. This is the first non-exponential algorithm for verifying dynamic controllability. Because of this we will refer to it also as **the classical algorithm**. All later algorithms make use of the ideas and the core proof given in the MMV paper.

   The algorithm works by applying a set of tightening rules to triangles, which are triples of nodes in the STNU that are connected by edges in specific ways. A tightening rule is applied if there is an implicit constraint between the nodes in the triangle. The implicit constraint is then turned explicit by being added to the STNU as a new constraint (edge), or by tightening an already existing constraint (decreasing the value of an edge label). MMV continually applies tightening rules until quiescence, when no more tightenings can be derived, or the evaluation of a simple criterion verifies that the STNU is not dynamically controllable. We will see the algorithm presented in detail in Chapter 3. A beneficial property of this algorithm is that it is easily implemented.

   The algorithm is pseudo-polynomial[1], but the ideas from MMV can be used in conjunction with ideas from the next algorithm, FastIDC, to create a modified MMV algorithm which is $O(n^4)$. The details are presented in Chapter 5.

2. **FastDC/FastIDC** [34] builds on MMV but applies the derivations in only one temporal direction, towards the start of the temporal network. Of the two algorithms FastDC is a full DC verification algorithm whereas FastIDC is an incremental version. We will present FastIDC in Chapter 3. As will be seen, a difference between this approach and MMV is that MMV focuses on triangles whereas the other algorithms take an edge-focused approach.

   The run-time of FastDC was conjectured to be $O(n^3)$, but we show that the incremental FastIDC is $\Omega(n^4)$ (see Chapter 6) which makes this conjecture less likely to be true.

[1] An algorithm is *pseudo-polynomial* if its running time is polynomial in the numeric value of the input, but is exponential in the length of the input (the number of bits required to represent it).

3. **MM** (again from the authors' surnames: Morris and Muscettola) [20] builds on the theory of MMV but uses new, less intuitive derivation rules. Its run-time complexity is $O(n^5)$ which is an improvement over the pseudo-polynomial complexity of MMV.

4. **The Morris algorithm** [17] builds in turn on MM. Its theory and especially analysis depends on several new complex concepts taking it further from the simple intuition behind MMV. This was the fastest algorithm when our work started. It runs in $O(n^4)$.

Above, we also presented FastIDC, which is a FastDC derivative. This was the only known algorithm for incremental DC verification prior to our work. The algorithm verifies whether an STNU that was originally DC remains DC after changes are made to it.

Note that the complexities presented are worst case run-times. There is no official test suite for comparing actual run-times on typical problems.

1.1 Temporal Planning

The use of temporal reasoning in an automated planner can be further motivated by the following example scenario.

Example 2. *Imagine a large scale disaster, such as a tsunami or earthquake, strikes an area making roads impassable and travel in the affected region dangerous. In the area there may be several people at various locations and they are now in need of assistance. If one has access to unmanned aerial vehicles (UAVs), these could be used to help. In this scenario it is assumed that there exist one or more depots where UAVs and supplies are located. The supplies could be different kinds of medicine, food and for instance communication devices with longer range than mobile phones.*

A first task for a fleet of UAVs would be to use different sensors to scan the area and detect people. The sensors could include heat cameras, normal cameras and sound sensors. Step two would then be to get the needed supplies out to these persons. In order to do so, larger UAVs may use carriers to move a large amount of supplies at once. Forward supply depots may be set up so that smaller UAVs flying single deliveries do not need to fly all the way back to the depot when reloading.

The area also needs continuous monitoring to detect people moving into it. Since UAVs have a limited fly time, the UAVs that do the monitoring need to replace each other over time.

It is clear that an automated planner can help humans to find a plan in such a complex scenario. Temporal aspects of the plan include ordering between UAVs entering a depot. Deadlines will have to be met for delivering emergency medicine and there are limited life spans for some medicines leading to different deadlines. When leaving supplies at remote depots, the depots should not be unguarded for too long to discourage looters. The same is true for watching pathways entering the area.

We see that temporal reasoning of some kind is needed to adequately model the scenario. It is not sufficient to model the temporal relations between actions in a plan using an ordinary STP, since there are many uncontrollable aspects that affect the durations of actions. These include, but are not limited to, weather, flight paths of other UAVs and manned aircraft, as well as other built in uncertainties in the actions (e.g. how long it takes to connect a winch to a supply crate). STPUs on the other hand offer a temporal model that allows us to model these uncontrollable aspects.

Automated planning is often based on search through a specific search space. When a successor of a given search node is created, most search strategies used in planning never loosen or remove existing constraints. For example, sequential forward search will add a new action at the end of a plan, which requires adding two new events and several new constraints to an STPU. Partial-order planners may add an action at any point in the existing plan, but still only add events and constraints to the STPU, and similarly for many other search strategies.

For these types of automated planners, if the STPU corresponding to a search node is not dynamically controllable, then no descendant of the search node can have a dynamically controllable STPU. Searching further along the same branch can only lead to more constraints in a network where there already is a violation, which cannot restore dynamic controllability. Therefore a planner can verify after each search step whether the plan remains DC, and if not, backtrack in order to prune the search tree as early as possible.

Testing dynamic controllability takes non-trivial time. Therefore, one can benefit greatly from using an *incremental* DC verification algorithm rather than redoing the analysis for each new action or constraint. This precludes the use of the four algorithms discussed above. The work presented in this thesis starts with the FastIDC algorithm [32, 33], which was the only available incremental DC verification algorithm prior to our contributions.

FastIDC supports incremental tightening/addition of constraints, which can be used for efficient planning, as well as incremental loosening or re-

moval. As argued above, most search strategies used in planning never loosen or remove constraints when a successor of a given search node is created. Therefore, this thesis will only focus on tightening/addition of constraints.

To summarize the introduction we have seen that automated planners can make use of temporal formalisms to reason about time. In order to efficiently be able to reason about uncertainty through the use of STPUs, an efficient incremental DC verification algorithm is needed. As mentioned briefly in the beginning of this chapter: it is the goal of this thesis to provide such an algorithm for integration with a planner capable of handling large problems with the complexity levels seen in this section.

1.2 Contributions

The contributions of this thesis appear in their respective context. Therefore, they are located in various places of the thesis. To make them clear we summarize them in this section. The details pertaining to each contribution can be found in the specified sections.

The following are the larger contributions of the thesis:

1. The FastIDC algorithm presented by Stedl and Williams [33] and Shah et al. [32] incrementally verifies dynamic controllability, which is essential when generating plans with the full expressivity of Simple Temporal Networks with Uncertainty. We have shown that the algorithm in certain cases fails to detect that networks are not dynamically controllable, which could potentially lead to planners accepting invalid plans. The problem was localized to the incremental dispatchability and consistency checking part of the BackPropagate-Tighten algorithm. We then analyzed the properties of the problem, resulting in a modification ensuring that inconsistencies will be detected while retaining the incremental properties of the algorithm. The modification does not increase the worst case run-time of the algorithm.

   This contribution appeared in [24].
   The details of this contribution can be found in Chapter 4.

2. The classical MMV algorithm is pseudo-polynomial. We prove that the ideas from MMV can be used in conjunction with ideas from FastIDC to create a modified MMV, which has a run-time of $O(n^4)$. The modified algorithm is an excellent and viable option for determining

whether an STNU is dynamically controllable. The modified algorithm preserves the results relating to MMV, with the exception of its improved complexity, and therefore offers a simpler and more intuitive theory than later algorithms. We also show that a large part of work done by MMV can be avoided, as it does not provide information that affects the result of the algorithm.

This contribution appeared in [25].
The details of this contribution can be found in Chapter 5.

3. We show that a corrected version of FastIDC in the worst case takes $\Omega(n^4)$ time to handle one incremental change. We then proceed to present a new algorithm that uses additional analysis together with a different traversal strategy to avoid this behavior. The new algorithm, EfficientIDC, has an amortized $O(n^3)$ time complexity. We prove that it is sound and complete.

   This contribution appeared in [26].
   The details of this contribution can be found in Chapter 6.

Additional contributions, which have not yet been published elsewhere, include the following:

1. Proof that MMV must use all edges from the All-Pairs Shortest Paths (APSP) graph in order to be correct (see Chapter 3).

2. Proof that Direct Dispatchability Verification (see Section 2.2.3) can be done correctly with the help of a filtering algorithm.

1.3 Outline

In Chapter 2 the STN and STNU formalisms are presented along with related concepts and properties. Chapter 3 introduces the MMV and FastIDC algorithms that the thesis work is based upon. This chapter also contains observations and experiences gained while implementing the algorithms. Chapter 4 points out a flaw in FastIDC and gives an efficient solution for it. In Chapter 5 insights from FastIDC are used to propose a revised MMV algorithm with a worst case run-time of $O(n^4)$. This is followed by the presentation of a new algorithm in Chapter 6: EfficientIDC, which subsumes the corrected version of FastIDC. The new algorithm has a better worst case complexity of amortized $O(n^3)$. Related work is reviewed in Chapter 7 and final conclusions are presented in Chapter 8.

Chapter 2

Temporal Formalisms

The study of temporal reasoning has a long history in computer science. For example, "Mechanization of Temporal Knowledge" by Kahn [15] dates back to 1975. There exist many different temporal formalisms. Many of these can be used to define events and constraints between these. The simplest and most popular of these event-based formalisms is the **Simple Temporal Problem (STP)**, which was defined by Dechter et al. [9]. We have already seen an example of this in Chapter 1. In more detail a Simple Temporal Problem consists of a set of real variables $x_1, \ldots, x_n$ and a set of constraints $T_{ij} = [a_{ij}, b_{ij}], i \neq j$ limiting the temporal distance $a_{ij} \leq x_j - x_i \leq b_{ij}$ between the variables. These constraints can also be expressed equivalently as $x_j - x_i \in [a_{ij}, b_{ij}]$.

There are many extensions to the STP formalism. One branch of extensions is based on increasing the expressivity of the constraints. For example, the *Temporal Constraint Satisfaction Problem (TCSP)* allows each constraint to express a time difference which must be in one of several disjoint intervals [9]. Constraints such as $t_2 - t_1 \in [a, b] \cup [c, d] \cup [r, f]$ can be represented in this formalism. A further extension is made in the *Disjunctive Temporal Problem (DTP)* formalism [35]. Constraints in DTPs can contain disjunctions between ordinary STP constraints. This allows constraints of the type $t_2 - t_1 \in [a, b] \vee t_4 - t_3 \in [c, d]$ to be represented.

An orthogonal branch of extensions introduces *uncertainty*, a concept that was introduced in the previous chapter. In these formalisms uncertain durations can be represented by the use of *contingent constraints*. Uncertain durations represented by contingent constraints lead to *uncontrollable* events which are assigned timepoints by external processes. These formalisms require that the bounds of the uncontrollable durations are known.

Since the extension branches of formalisms mentioned so far are orthogonal it is possible to combine them. This gives the *Simple Temporal Problem with Uncertainty (STPU)* [44], *Temporal Constraint Satisfaction Problem with Uncertainty (TCSPU)* [40], and *Disjunctive Temporal Problem with Uncertainty (DTPU)* [41].

The *Probabilistic Simple Temporal Problem (PSTP)* can be seen as an extension of STPUs in which durations are still uncertain, but their outcomes are modeled by probabilistic density functions [37]. In this formalism it therefore becomes interesting to reason about the likelihood of execution scenarios which do not violate constraints.

Conditional Temporal Problems (CTP) extend STPs by allowing choices to be made within the problem depending on facts that become known at execution time [38]. In this formalism events can be labeled by choice labels and different choices enable different constraints which must be satisfied. Another formalism which allows representation of choice is that of *Temporal Plan Networks (TPN)* [16], which allows more programming-like structures such as loops.

There are also temporal formalisms that are not STP-based. Allen's interval algebra [1] and the point algebra [46] are two such examples. Allen's interval algebra can be used to compare time intervals to each other in a qualitative way. Thirteen different relations can be defined when comparing two intervals. These are: before, meets, overlaps, starts, during, finishes and equal. Point algebra is similar but compares only the qualitative relations between timepoints.

We will now discuss two further orthogonal aspects of temporal formalisms. The first aspect is how time is modeled, the second is which type of relations that exist in the formalism.

Discrete vs. Continuous: Time can be *discrete* or *continuous*, an aspect which is often modeled by using a set of integers or reals to represent events in time. When the underlying granularity of time is decided upon, events can be fixed in time. Two events can then be used to define the start and end of an interval.

Qualitative vs. Quantitative: Relations between events and intervals can be either *qualitative* or *quantitative*. The qualitative relations use symbolic comparisons. Results can be *before* or *after* as in Allen' interval algebra. Good sources for knowledge about qualitative temporal constraints are given by [11], [45], [31] and [8]. Quantitative relations instead use metric expressions to relate events and intervals. Simple Temporal Problems and their derivatives fall into this category.

A planner can be designed to use a temporal formalism to be able to reason about and use temporal constraints. Depending on which formalism is chosen, the planner will have different capabilities. As temporal formalisms add a higher complexity to the already high planning complexity, many planners reason about time through built-in mechanisms. For instance, the forward chaining planner TALplanner [10] represents time explicitly by time-stamping states as it incrementally builds a plan. Therefore, while it does not explicitly create constraints it can still reason about them.

To be able to plan search and rescue missions, support for uncertainty is a requirement. It is unrealistic to presume that no external influences exist. Among the mentioned formalisms both DTP and TCSP are NP-hard [29]. Therefore it is not realistic to base a planner on DTPU or TCSPU although these formalisms have high expressivity. The most realistic choice falls on STPU which is a formalism in which reasoning is polynomial. In this thesis we will therefore focus on STPUs and to some extent STPs since many concepts transfer from STPs to STPUs. Although it is common to think of advancing time in quanta (often integer times), there are no limitations to STPs or STPUs when it comes to deciding if they should be applied to reason about discrete or continuous time.

2.1 Simple Temporal Problems

In this section we give the definition for the basis of temporal reasoning in this thesis, Simple Temporal Problems. We also present interesting properties that are closely tied to the definition.

Definition 1 (Simple Temporal Problem (STP) [9]).
*A **simple temporal problem** (**STP**) consists of a set of real variables $x_1, \ldots, x_n$ and constraints $T_{ij} = [a_{ij}, b_{ij}], i \neq j$ limiting the temporal distance $a_{ij} \leq x_j - x_i \leq b_{ij}$ between the variables.*

The definition does not specify the values that are allowed for the a_{ij} and b_{ij} bounds. They are by default real-valued since they relate real-valued variables. However, the bounds are often allowed to take on the values $\pm\infty$ which have special meaning. An upper bound $b_{ij} = \infty$ means that the constraint does not constrain the maximum time difference between the involved variables. Similarly a lower bound $a_{ij} = -\infty$ means that the constraint does not constrain the minimum time difference between the involved variables.

Note that the definition mentions *temporal* distance. This is only a formality associated with the perspective taken, where we want to relate temporal events. As we will see later (Section 2.1.1), STP constraints can also be encoded as distances in graphs. Reasoning with STPs can then be done by more general methods for finding shortest paths in graphs.

Reasoning with STPs. We now provide several definitions related to reasoning with STPs. The definitions in the rest of this section were all specified by Dechter et al. [9].

- An STP is *consistent* if there is an assignment of real values to the variables, $x_1, \ldots, x_n$, so that all constraints T_{ij} are satisfied. Such an assignment is also referred to as a *solution* to the STP.
- A constraint interval, T_{ij}, which can be shrunk without losing any solution is called *redundant*.
- When no constraint interval is redundant the STP is *minimal*.

Many of the algorithms used for solving STPs will both determine if the STP is consistent and find its minimal constraint intervals [9, 29].

Since STP constraints only constrain the relative difference between variable values, it is not possible to relate the variables to absolute values. For example, if $\langle t_1 = 0, t_2 = 2\rangle$ is a solution to a small STP, then $\langle t_1 = x, t_2 = x + 2\rangle$ must be a solution for every $x \in \mathbb{R}$. To be able to relate variables to absolute values, a special variable, called a *Temporal Reference (TR)* [27, 28], is often introduced. Intuitively the TR denotes the start of time in the STP and it is defined to occur at time 0. To ensure that no other event occurs before TR, constraints $T_{ij} = [0, \infty]$, with i being the index of the TR, can be introduced for all j.

For the next definition we need to discuss partial assignments of values to the STP variables. We say that a partial assignment to an STP is consistent if the assignment do not violate any of the constraints that constrains the assigned variables. With the help of a partial assignment we now describe the concept of a *decomposable* STP [9]. An STP is decomposable if any consistent partial assignment can be extended to a full assignment which is consistent. A decomposable STP can be used to generate solutions by assigning one variable at a time, respecting the choices made so far. If the assignments are done with increasing values, this is a direct way of carrying out execution as we will see in Section 2.2.

As a side note we want to point out that the constraint represented by the interval $a_{ij} \leq x_j - x_i \leq b_{ij}$ can be viewed as a pair of inequalities $x_j - x_i \leq b_{ij}$ and $x_i - x_j \leq -a_{ij}$. By representing all constraints in this way we get a system of inequalities. These systems have been studied extensively in the area of operations research. However many of the solution methods available there, e.g. the simplex method [7], are aimed at linear programming problems where optimization is an integral part. STPs are simpler than general linear programming problems, which for instance allow constraints between more than two variables, and therefore we will see simpler algorithms that are faster when solving these problems.

Simple Temporal Networks

The temporal constraints of STPs are binary. Therefore, they always capture a relation between exactly two events. Because of this any STP can be put into graph form where nodes correspond to events and edges to relations between events.

Definition 2 (Simple Temporal Network (STN) [9]).
*Given an STP with variables $x_1, \ldots, x_p$ and constraints $T_{ij} = [a_{ij}, b_{ij}]$ a **Simple Temporal Network (STN)** for this STP is a graph $\langle N, E \rangle$ consisting of nodes $N = \{n_1, \ldots, n_p\}$ and edges $E = \{e_{ij} = (n_i, n_j) | T_{ij} \in STP\}$. Each edge e_{ij} in the STN is labeled by the interval $[a_{ij}, b_{ij}]$.*

Although STPs and STNs are equivalent representations of the same temporal problem, it is more common to see references to STNs in the context of planning and shortest path algorithms [39] and STPs in constraint solving contexts [9].

Figure 2.1 shows an example of an STP in its STN graph form. In the example each edge is labeled with the interval that constrains the events corresponding to the nodes connected by the edge. Therefore the label $[10, 20]$ between t_1 and t_2 means that $10 \leq t_2 - t_1 \leq 20$. If the assignment $t_1 = 0, t_2 = 20, t_3 = 10, t_4 = 25$ is made, all constraints are satisfied and hence the STP has a solution. It is therefore consistent.

2.1.1 Distance Graph Representation

There exists another graph representation which is equivalent to that of the STN but is easier to work with since it allows working with only one of the interval bounds at a time. The graph representation is called a *distance graph* and we now describe how it is constructed from an STN [9].

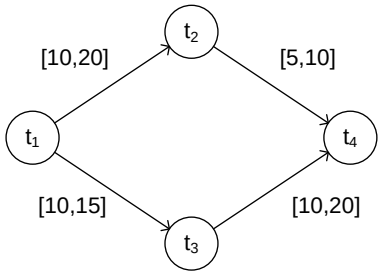

Figure 2.1: Example of a consistent STP in graph form.

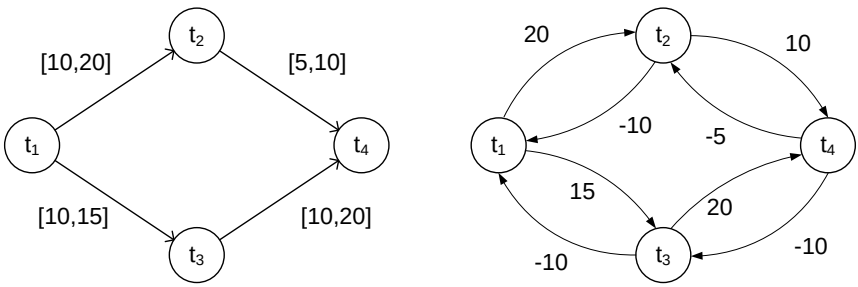

Figure 2.2: Example of a consistent STN and its distance graph.

For each time variable $x_i (i = 1, \ldots, n)$, create a node labeled t_i. Then split the STN constraints into two inequalities each, and let the values of the inequality bounds be the weights of directed edges between the two nodes:

- The constraint $x_j - x_i \leq b_{ij}$ becomes an edge $i \rightarrow j$ with weight b_{ij}. This can be read "j can never be later than b_{ij} after i".
- The constraint $x_i - x_j \leq -a_{ij}$ becomes an edge $j \rightarrow i$ with weight $-a_{ij}$. This can be read "i can never be later than $-a_{ij}$ after j", which can also be expressed as 'i is at least a_{ij} before j'.

Figure 2.2 shows an example of an STN and its distance graph. When constructing the distance graph, edges corresponding to upper bounds of ∞ and lower bounds of $-\infty$ are left out. This means for example that only one edge is used to model the constraint $[-\infty, 25]$ and similarly for $[13, \infty]$.

In a consistent STN, $a_{ij} \leq b_{ij}$ due to the definition of the constraints. Furthermore, with the help of the distance graph it is easy to see that $T_{ij} = [a, b]$ is equivalent to $T_{ji} = [-b, -a]$.

We want to point out that the value for the lower bound becomes negated in the distance graph. A constraint $[-a, b]$ where $a \geq 0$ and $b > 0$ is translated into two positive edges in the distance graph. This means that this constraint does not constrain the nodes to happen in a specific order. If two nodes are related by such a constraint in a minimal STN, they are *unordered*. If instead there is a constraint $[a, b]$ where $a, b > 0$ between the nodes they are *ordered*, i.e. i must always happen before j.

As mentioned before each edge encodes an inequality. It is possible to calculate the sum of the inequalities along a path of edges. For instance we can add the inequalities corresponding to the edges $t_1 \rightarrow t_3 \rightarrow t_4 \rightarrow t_2$ from Figure 2.2. This gives

$$\begin{aligned} t_3 - t_1 &\leq 15 \\ t_4 - t_3 &\leq 20 \\ t_2 - t_4 &\leq -5 \\ \hline t_2 - t_1 &\leq 30 \end{aligned}$$

In general a path from X to Y gives a sum that can be expressed as a bound on Y, $Y \leq X + sum\_of\_weights$. If the path is a cycle we get an expression of the type $X \leq X + sum\_of\_cycle\_weights$. We see that if the sum is negative the event X must be assigned a timepoint before itself which is impossible. It can be proven that an STN is consistent if and only if its distance graph contains no negative cycle [9]. Assuming there is no negative cycle in the distance graph (thus a consistent STN), it is possible to calculate the shortest path distances between all nodes. A complete graph where every edge $i \rightarrow j$ has a weight that equals the (possibly infinite) shortest path distance between i and j is called a *d-graph* [9]. Because the term d-graph can easily be confused with the term distance graph, we will instead use the more intuitive term *All-Pairs Shortest Paths* graph or *APSP* graph in this thesis. For a consistent STN, it is shown that the constraints corresponding to the edges in the APSP graph represent a decomposable STN with minimal constraints [9].

2.1.2 Consistency Checking

We mentioned previously that an STP can be regarded either as a constraint satisfaction problem or as a shortest path problem. Taking a constraint approach allows tests of consistency by applying techniques from the con-

straint reasoning domain. These include full, directional and partial path consistency techniques [8]. Constraint-based techniques can also leverage concepts such as the induced width [27, 28] of the constraint graph. The fastest algorithms use triangulation sub-algorithms [27, 47] to process constraints in an efficient order. Published algorithms includes DPC [9], PPC [28], ΔSTP [47] and P^3C [27]. Regardless of approach, the worst case complexity for verifying consistency is $O(n^3)$ where n is the number of events.

We saw earlier that the consistency of an STP can be established by detecting the absence of negative cycles in the corresponding distance graph [9]. This can be done by use of the Floyd-Warshall algorithm [6], again in $O(n^3)$. For sparse graphs Johnson's algorithm [6] gives a slightly better complexity at $O(n^2 \log n + n|E|)$, where $|E|$ is the number of edges in the graph.

Although the constraint methods have more detailed complexity results (based for instance on induced width of graphs and other graph theoretic properties) than the distance methods [29], only the latter have been transferred to STNUs.

2.2 Execution of STNs

Simple Temporal Networks can be used to schedule events. For a temporal planner, the start and end of each action are events needing scheduling. Suppose a planner has generated a plan for which the corresponding STN is consistent. This means that there is at least one solution to the STN. Therefore there is at least one assignment of timepoints to the variables which satisfies all constraints. Each such assignment corresponds to one execution schedule for the plan. There may be several such schedules and to execute the plan one of these have to be chosen.

There are two alternative ways of choosing a solution:

1. Choose a complete solution before execution. This can be done for instance incrementally if the STN is decomposable. The chosen solution may optimize some condition, for instance *makespan*, which is defined to be the difference in time between the earliest and the latest timepoints in the schedule. If all events in the STN are completely controllable by the execution system, this way of deciding which schedule to execute works well. As soon as the solution is decided upon, execution starts the actions corresponding to the events in the order and at the timepoints chosen in the solution.

2. Gradually, dynamically choose during execution when to start and finish actions. This is a harder alternative for several reasons. First, a partial solution must be gradually extended and the choices made must be such that it is possible to extend the partial solution to a full solution. Second, even if the STN is decomposable, so that this is always possible, timepoints for events must be chosen in the order they are executed. The fact that any partial solution can be extended to a complete solution is not sufficient if all such extensions require an action to be executed "10 minutes ago".

   There are advantages of dynamic execution. It is sometimes possible to introduce flexibility to handle actions whose durations are not entirely controllable by the executor. This is a way of reaching a bit towards the flexibility of STNUs, but without any guarantees. The issue was discussed by Muscettola et al. [23] and further developed by Tsamardinos [36] and Tsamardinos et al. [39]. A short discussion of flexibility is included in Section 2.2.1. An algorithm which can assign times dynamically as described herein is called a *dispatcher* [36].

We continue in the next section by presenting a dispatcher: An execution algorithm which for a certain class of STNs can assign timepoints to events dynamically during execution. We then discuss a way of verifying if an STN belongs to the class of networks the dispatcher can execute. Fast verification and execution is dependent on edge filtering which is covered in Section 2.2.2. In Section 2.3 we present the DBP algorithm which can convert an STN to an equivalent STN that has the same solutions as the original, and belongs to the class of STNs a dispatcher can execute correctly.

The DBP algorithm is an integral part of the FastIDC algorithm (Chapter 3) for STNUs. We will also see that the execution of STNs is related to the execution of STNUs.

2.2.1 Dispatching STNs

In the context of STNs, assigning values to the variables is called *execution* [39]. In this context it is assumed that variables are assigned values in order of increasing value. This captures the situation where actions are started and ended as time is progressing. Different assignments to the variables give different *execution scenarios*. An STN is *executable* if there is an execution scenario which satisfies all constraints. Note that this is equivalent to the STN being consistent. The context determines the terminology used.

An execution algorithm called a *dispatcher* was proposed by Muscettola et al. [23]. The algorithm uses the distance graph of an STN to facilitate the execution. Execution by the algorithm is referred to as *dispatch* of an STN.

Definition 3 (Directly Dispatchable STN [This Thesis]). *An STN is **directly dispatchable** if it can be dispatched correctly by the dispatcher without modification.*

Definition 4 (Dispatchable STN [This Thesis]). *An STN is **dispatchable** if it is directly dispatchable or can be made directly dispatchable through the addition of a set of constraints that preserve the original set of solutions to the STN.*

In essence, the process of making an STN dispatchable will compute a set of constraints that are already implicit in the STN and add these constraints explicitly to the graph. For example, given edges $A \xrightarrow{[10,20]} B$ and $B \xrightarrow{[10,20]} C$, we may infer and add $A \xrightarrow{[20,40]} C$. This does not change the set of solutions, but makes the constraint between A and C more explicit.

If a consistent STN is not directly dispatchable, it can always be made so by addition of constraints, as will be seen later in this chapter. This means that the concepts of dispatchability, consistency and executability are equivalent, but different from direct dispatchability.

Notation. Two concepts used when selecting the next node to execute are defined by Muscettola et al. [23]. A node n is a *predecessor* of another node p, if there is a negative edge $n \leftarrow p$. As an example, in Figure 2.2 node t_3 is a predecessor of node t_4 since there is a negative edge from t_4 to t_3. A negative edge, $t_3 \xleftarrow{-10} t_4$, encodes a positive lower bound in the opposite direction so that t_4 is required by the constraint to execute at a timepoint at least 10 time units after t_3. A node becomes *enabled* when all its predecessors in the distance graph have been executed. In the example of Figure 2.2, t_4 becomes enabled when both t_2 and t_3 have been executed.

The notation T_i is used by [23] to refer to the chosen execution time of node i. Furthermore, a *start_time_point* which is executed at time 0 is defined. This is equivalent to the temporal reference (TR) that we use throughout this thesis.

Execution Windows. During execution the dispatcher associates each node i with a time interval $[lb_i, ub_i]$ representing the dispatcher's current knowledge about the time interval in which the node must be executed to avoid violating constraints. This interval is referred to as the *execution window* of

Algorithm 1: Dispatcher [23]

function DISPATCH(*G - STN*)
enabled ← {Temporal-Reference}
executed ← {}
currentTime ← 0
while not *all nodes are executed* **do**
  Remove *e* from *enabled*, for which
    currentTime ∈ [LowerBound(*e*), UpperBound(*e*)]
  set execution time of *e* to *currentTime*
  add *e* to *executed*
  update execution windows of neighbors to *e*
  add all nodes which become enabled to *enabled*
  wait until
  currentTime ∈ [*min*(LowerBound(*enabled*)), *min*(UpperBound(*enabled*))]
end

the node [32]. The execution window is initialized to $[0, 0]$ for the TR and $[0, \infty]$ for all other nodes. During execution the execution windows will shrink as constraints from neighboring nodes affect them. Note that nodes that have been executed can be forgotten about since they were safely executed and their execution times cannot be altered [23].

Propagation of Bounds. The dispatcher uses the edges of the distance graph to propagate execution window bounds. For simplicity we say that execution window bounds are propagated via nodes at execution time while in fact it is the algorithm which propagates the bounds as nodes are executed.

Whenever a node i is executed, so that its execution time T_i becomes known, every positive edge $i \xrightarrow{b} j$ with $b > 0$ will result in the propagation of an upper bound of $T_i + b$ to node j. If the existing execution window for j was $[lb_j, ub_j]$, the new window becomes $[lb_j, min(T_i + b, ub_j)]$ because the tightest limits must be chosen to accommodate all constraints.

Similarly, every negative edge $i \xleftarrow{-a} j$ (note the reversed order), with $a \geq 0$, will propagate a lower bound of value $T_i + a$ to j. The resulting execution window for j becomes $[max(T_i + a, lb_j), ub_j]$ in this case. Note that 0 is treated as negative. Therefore, lower bounds will be propagated along 0-weight edges.

Dispatcher Algorithm. Algorithm 1 lists the pseudo-code for the dispatcher [23]. The algorithm keeps two sets, **enabled** containing enabled nodes and **executed** containing executed nodes. It loops until all nodes are executed. In each iteration it executes one node, *e* by setting its execution time to the current time. The node is then moved from **enabled** to **executed** and all the execution windows for neighboring nodes are updated as described before. Any node which becomes enabled due to the execution of *e* is put in **enabled**. Time is then advanced until the next node is chosen, which happens when *currentTime* is somewhere in the interval $[\min(\mathit{LowerBound}(\mathit{enabled})), \min(\mathit{UpperBound}(\mathit{enabled}))]$. Here the minimum is taken over all nodes in *enabled*. To start a new iteration, it is sufficient that *currentTime* exceeds the lower bound of one node. The algorithm must choose a time within the time interval limited by the lowest upper bound of any node in *enabled*. Also, the algorithm cannot postpone execution of any node until after its upper bound is passed, or execution will fail.

The algorithm does not specify the exact time within $[\min(\mathit{LowerBound}(\mathit{enabled})), \min(\mathit{UpperBound}(\mathit{enabled}))]$ chosen for *currentTime*. There are two reasons for this. First, by having a least commitment approach, the dispatch algorithm can produce all possible execution scenarios. Secondly, if the end times of some tasks are not controllable by the executor, the dispatcher can be modified to "execute" action end nodes when notified that the action actually ended, as long as this happens within each node's execution window. The dispatch approach then permits a degree of uncontrollability without using the more complex concept of STNUs (see Section 2.4).

A greedy alternative would be to always execute a node as soon as possible, i.e. when *currentTime* reaches its lower bound. This approach clearly gives the lowest makespan, but excludes most scenarios and provides less flexibility. In a multi-agent environment, maximum flexibility gives the agents maximum freedom to choose when to execute their task. This makes it easier for them to combine tasks from different schedules.

Example. We will now go through an example dispatch of an STN. Figure 2.3 shows a consistent STN and its distance graph. Figure 2.4 shows how the dispatcher may update execution windows while dispatching the example graph. The steps of the execution are outlined below:

a) At the start all execution windows are $[0, \infty]$ except for the temporal reference's which is $[0, 0]$.

b) Execution of node 0 at time 0 leads to propagation of bounds along both

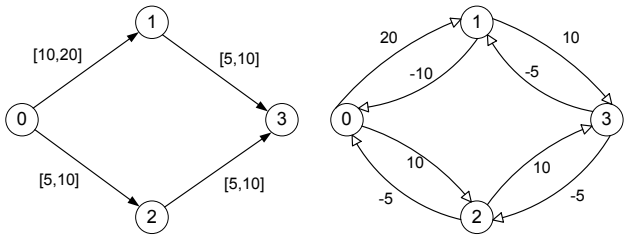

Figure 2.3: Example of a consistent STN and its distance graph dual.

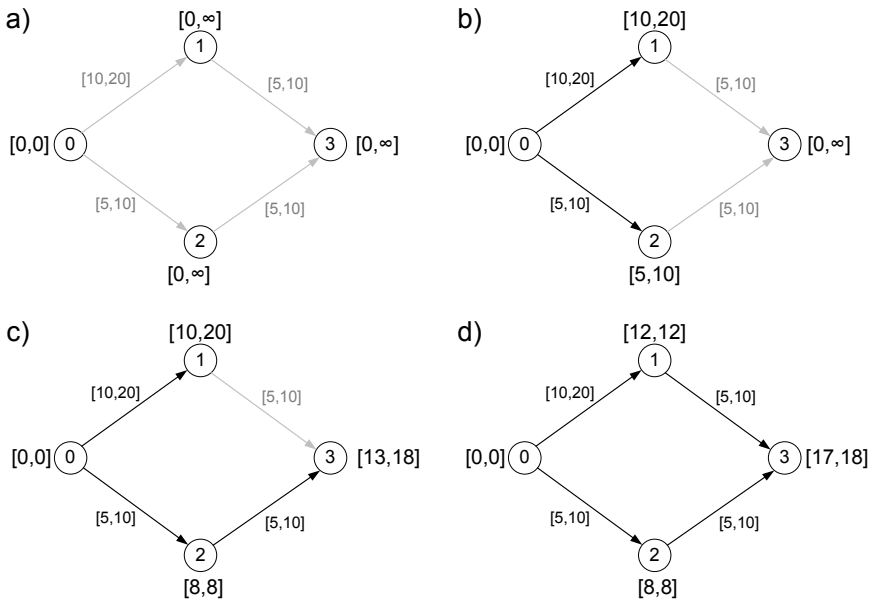

Figure 2.4: Dispatching the STN in Figure 2.3.

the outgoing positive and the incoming negative edges. This enables nodes 1 and 2 since all nodes that are sources of incoming constraints towards node 1 and 2, with positive lower bounds, have been executed (namely node 0).

c) The dispatcher chooses to execute node 2 at time 8 which leads to a propagation of tighter bounds to node 3. The new bounds become $[13, 18]$ but node 3 does not become enabled.

d) Node 1 is executed at time 12 and bounds are again propagated to node 3.

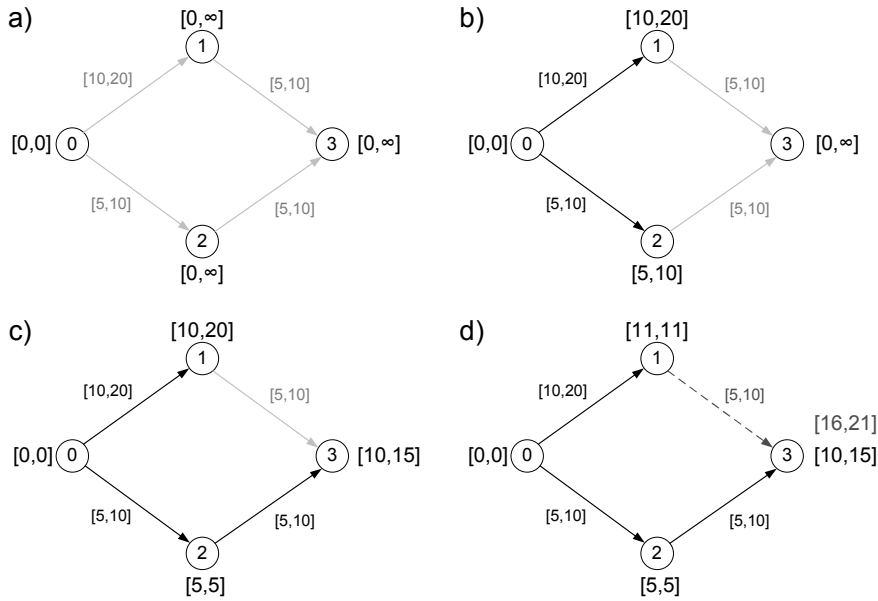

Figure 2.5: Example of a consistent STN where dispatch fails due lack of propagation between nodes.

The upper bound propagated is 22. Since node 3 already has an upper bound of 18, which is tighter than 22, the upper bound is not changed. The final execution window for node 3 becomes $[17, 18]$ and the node is now enabled. The dispatcher may choose to execute node 3 at time 17 or 18 arriving at a valid execution scenario where all constraints are met.

Note that each node is assigned a timepoint that is later than the timepoints previously assigned. This means that execution can be carried out while new timepoints are assigned. There is no need to wait until the full solution is decided upon.

Direct Dispatchability. The dispatcher is only *guaranteed* to work if the existing constraints between nodes in the STN are explicit enough so that the algorithm propagates tight enough bounds. For the STN in Figure 2.3 this is not the case. We saw previously that the dispatcher made choices that led to a successful execution. It could have made other choices. Figure 2.5 shows what happens if the dispatcher decides to execute $T_2 = 5$ (Figure 2.5c). The execution window for node 3 becomes $[10, 15]$. When

the dispatcher follows this decision by choosing a low T_1 value of 11 in the last step (Figure 2.5d), the bounds propagated to node 3 will be too high: $[16, 21]$. This means that the intersection between the interval propagated from node 1 and that propagated from node 2 is empty, so that no timepoint satisfies them both. To prevent this situation the choice of T_2 must affect the possible choices for T_1.

Figure 2.6 shows the APSP graph for the example STN, which has a different set of constraints but exactly the same solutions. According to this graph there is in fact a constraint between node 2 and node 1 that was implicit in the original STN formulation. The constraint requires the difference to be within $[0, 5]$. We can also see that T_1 must be below 15. If an STN is missing edges, as in the $[0, 5]$ case, or edges are too loose, as in the case with the $[10, 20]$ constraint between 0 and 1, dispatch may fail. From this intuition we formulate and prove the following lemma:

Lemma 1. ***(Direct Dispatchability) [This thesis]*** *Let G be a consistent STN and G′ be the APSP STN version of G. We now assume that G and G′ are dispatched simultaneously and that all timepoints that are assigned to events in G are also assigned to the corresponding events in G′. Then G is directly dispatchable if and only if, for all execution scenarios of G, all execution windows for enabled nodes in G are sub-intervals of the execution windows for the corresponding enabled nodes in G′*

Proof. We start with the "if'-part. We know that G′ is dispatchable. We also know that any edge present in G is present in G′ and the edge in G′ can only be tighter. Therefore the dispatch of G′ will allow as many execution scenarios as possible. This is because the APSP version of an STN is a minimal decomposition. The minimality of G′ means that no constraint contains redundant values, if any constraint interval is reduced at least one execution scenario is lost. That G′ is a decomposition means that for any allowed timepoint assignment to an event, with respect to constraints, assignments to the other events can be made so that the resulting execution scenario is valid. If all choices for the dispatch of G are within the choices allowed for G′, G can clearly be directly dispatched.

Now we prove the "only if"-part. Suppose that there exists an execution scenario for G, such that an execution window for the enabled node n during dispatch of G is not a sub-interval of the corresponding window for the enabled node n′ in G′. This means that the dispatcher may chose a time for n which cannot be assigned to n′. But dispatching G′ allows all possible valid execution scenarios. Therefore the scenario executed by the dispatcher for G cannot be valid and dispatch is incorrect. Therefore G is

in this case not directly dispatchable.

□

It is not trivial to use this lemma to prove that an STN is directly dispatchable. We present a novel efficient solution to the direct dispatchability verification problem in Section 2.2.3. Before this can be done the concept of edge filtering must be presented.

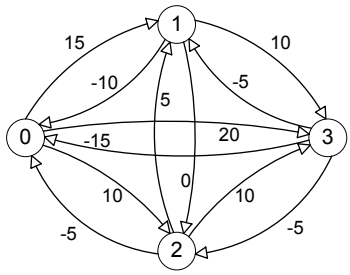

Figure 2.6: The all-pairs shortest paths of the STN example.

2.2.2 Edge Filtering

One way to make an STN directly dispatchable is to add edges between all nodes with weights corresponding to the costs of the shortest paths between nodes, i.e. add all edges from the APSP graph. This makes sure that any decision made by the dispatcher will be propagated by the dispatcher to limit its later choices of timepoints for events. A problem with a complete graph, such as the APSP graph, is that once a timepoint has been decided for an event, the dispatcher must propagate this decision to affect the execution windows of all non-executed nodes. Therefore the dispatcher will have to carry out on average $\Theta(n^2)$ propagations in each iteration for a total of $\Theta(n^3)$ propagations altogether during execution. The idea behind the dispatcher, as mentioned before, is to execute plans in real-time to be able to accommodate a small amount of uncontrollability in action durations. As a reminder, the alternative to dispatching execution is to choose timepoints for the events before the actual execution of plans. This results in a static time schedule which has a lower probability of success in scenarios with uncontrollable durations. A concern for dispatching is if an STN has a high degree of connectedness. Then propagations by the dispatcher during execution may become a bottleneck affecting real-time performance.

If there are several paths along which the dispatcher may propagate the same information, some of these paths can be removed without affecting the direct dispatchability of the STN. This is observed by Muscettola et al. [23] who presents so called domination rules that can be used to remove unnecessary edges from the graph. Before discussing these rules we remind the reader that when the dispatcher executes a plan, upper bounds on execution windows are propagated by positive edges and lower bounds by negative edges (in the reverse direction). There is a difference in the importance of these edges. A positive edge $i \xrightarrow{b} j$ means that the execution of j should not be later than b time units from when i is executed. Notice that this says nothing about which node should executed first. A negative edge however always means precedence. A negative edge $i \xleftarrow{-a} j$ requires that i is executed at least a time units before j. Edges with weight zero are a bit special and can be regarded in both ways. The procedure presented in this section considers zero weight edges as non-negative and groups them with positive edges. The DBP algorithm, presented in Section 2.3, considers zero weight edges as negative. We want to point out this difference in definitions so that the reader is not confused.

The idea proposed by Muscettola et al. [23] is that given a directly dispatchable STN, redundancy in propagation can be reduced by removing certain edges. These edges are *dominated* by others and they can be discovered by applying a triangle rule. The triangle rule has two conditions, one for when a positive edge is dominated and can be removed and one for when a negative edge is dominated and can be removed. An algorithm is also given which filters out any unneeded edges from a directly dispatchable STN. The algorithm runs in $O(|V|^3)$ where V is the set of nodes in the STN.

The intuition behind the domination rules is that some edges will be used by the dispatcher to propagate looser or identical bounds compared to propagation along other edges, and so only the tightest of the propagation paths is required. This intuition leads to the following definitions:

Definition 5 (Domination [23]). *A non-negative edge **AC** is **upper-dominated** by another non-negative edge **BC**, both having **C** as target, if the upper bound propagated by the dispatcher during dispatch along **BC** is always at least as tight as the upper bound propagated by the dispatcher along the **AC** edge.*

*A negative edge, **AC**, is similarly **lower-dominated** by another negative edge **AB**, both having **A** as source, if the dispatch propagation of a lower bound along **AB** is always at least as tight as the corresponding propagation along **AC**.*

Theorem 1 states the requirements for when domination applies. In the fol-

lowing theorem and proof, $|\boldsymbol{AB}|$ represents the weight of the edge between nodes $\boldsymbol{A}$ and $\boldsymbol{B}$.

Theorem 1. ***(Triangle Rule) [36]***
In an STN where the associated distance graph satisfies the triangle inequality:

1. *A non-negative edge* $\boldsymbol{AC}$ *is upper-dominated by another non-negative edge* $\boldsymbol{BC}$ *if and only if* $|\boldsymbol{AB}| + |\boldsymbol{BC}| = |\boldsymbol{AC}|$.
2. *A negative edge* $\boldsymbol{AC}$ *is lower-dominated by another negative edge* $\boldsymbol{AB}$ *if and only if* $|\boldsymbol{AB}| + |\boldsymbol{BC}| = |\boldsymbol{AC}|$.

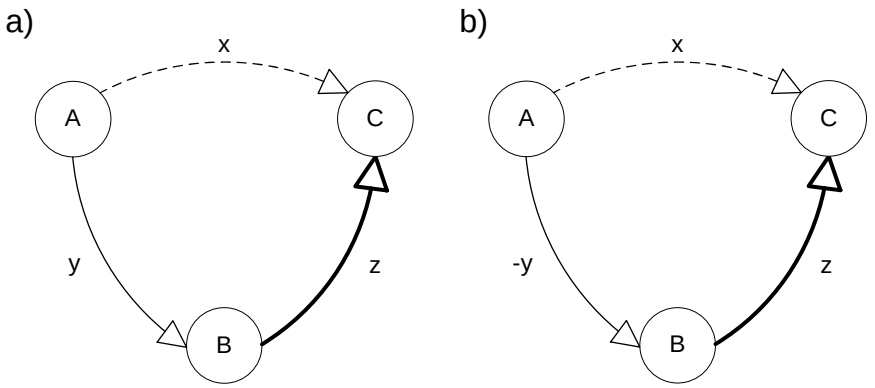

Figure 2.7: The two cases of upper domination.

Proof. This was originally given a brief proof in Muscettola et al. [23]. Here we provide a more detailed proof that should be easier to follow.

The proof is divided into two parts, each of which covers one domination type. Throughout the proof we will use a triangle with nodes A, B and C as shown in Figure 2.7. The absolute weight of the edges will be denoted by the variables x, y and z. These variables will therefore always be positive, and expressions such as $-x$ always represent negative weights.

Recall that T_A is used to denote the execution time of node A.

Upper Domination (1). For upper domination, Figure 2.7a shows the case when AB has a non-negative weight and Figure 2.7b when its weight is negative. These are the only two cases of interest, since both AC and BC are required to be non-negative. The proof will be split in these two cases due to the fact that different argumentation is possible depending on the sign of the weights.

Upper Domination (1) / "if". Assume that $|\mathbf{AB}| + |\mathbf{BC}| = |\mathbf{AC}|$. We must then show that the AC edge is dominated by the BC edge.

Upper Domination (1) / "if" / non-negative. We start with the case in Figure 2.7a. Since all edges have non-negative weights, there is no forced ordering between when A, B and C are executed. We will therefore examine each ordering in turn and show that the upper bound of $T_A + x$, on the node C, can be achieved without AC. In other words, we want to show that $T_C \leq T_A + x$ without the AC edge. From the assumption we know that $y + z = x$ and can therefore use the following two facts: $x \geq y$ and $x \geq z$.

If $T_C \leq T_A$ this is trivially shown since $x \geq 0$ in cases where upper domination is applicable.

If $T_A < T_C \leq T_B$, i.e. A is executed before B, the upper bound of y affects the choices for when the dispatcher executes B so that $T_B \leq T_A + y$ and hence $T_C \leq T_A + y \leq T_A + x$ since $y \leq x$. Therefore AC is upper dominated if this execution order is chosen by the dispatcher.

The next case we need to verify is $T_A \leq T_B < T_C$. From the ordering we know that B is executed before $T_A + y$. At that time an upper bound is propagated to C: $T_B + z \leq T_A + z + y = T_A + x$ so the propagated bound is never larger than $T_A + x$, and it is in place before C is executed making sure that $T_C \leq T_A + x$ as required.

The last case is if $T_B \leq T_A$ which trivially makes $T_B \leq T_A + y$ and the same argument applies again. In conclusion, by removing AC the upper bound on T_C is not loosened, hence AC is upper dominated by BC.

Upper Domination (1) / "if" / negative. In Figure 2.7b, $|AB|$ is negative. This means that A and B are ordered so that $T_B < T_A$, since a negative edge propagates a positive lower bound. In this instance we have $T_A \geq T_B + y$. Therefore, the upper bound of $T_B + z$ is propagated to C before A executes. Again, there are two possible execution orders: $T_C \leq T_A$ and $T_C > T_A$.

If $T_C \leq T_A$ then trivially $T_C \leq T_A + x$ since $x \geq 0$.

If $T_C > T_A$, then the upper bound propagated via A, $T_A + x \geq T_B + y + x = T_B + z$ so the bound propagated via AC cannot be tighter than the bound propagated vi the BC edge. Therefore in both possible orderings AC is upper dominated by BC.

Upper Domination (1) / "only if". To conclude the proof of part 1 we show that if $|\mathbf{AB}| + |\mathbf{BC}| \neq |\mathbf{AC}|$, AC cannot be upper dominated by BC. We first see that $|\mathbf{AB}| + |\mathbf{BC}| < |\mathbf{AC}|$ is not possible in the distance graph of the STN since it was assumed to satisfy the triangle inequality which states that $|\mathbf{AB}| + |\mathbf{BC}| \geq |\mathbf{AC}|$.

If $|\mathbf{AB}| + |\mathbf{BC}| > |\mathbf{AC}|$ and the edge weights match the scenario in Figure 2.7a where all edges have positive weights, there exists a dispatch scenario in which the upper bound propagated via BC is $T_A + y + z > T_A + x$. This means that the upper bound propagated via BC in this dispatch scenario is not at as tight as the bound propagated via the AC edge and hence the requirement of upper domination: "...is always at least as tight...", is not satisfied.

If $|\mathbf{AB}| + |\mathbf{BC}| > |\mathbf{AC}|$ and the edge weights match the scenario in Figure 2.7b there is a dispatch scenario in which A is executed as soon as possible, i.e. y time units after B, giving C the upper bound $T_B + y + x = T_B + x - (-y) = T_B + |AC| - |AB| < T_B + |BC|$ and so we see that the upper bound propagated via BC is not as tight as the upper bound propagated via AC. Therefore BC cannot upper dominate AC.

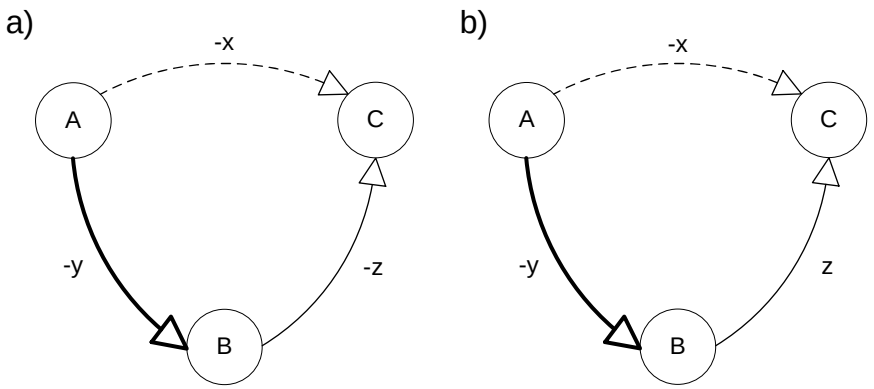

Figure 2.8: The two cases of lower domination.

Lower Domination (2). We now continue with part 2 of the proof. Figure 2.8 shows the corresponding two cases of lower domination. The difference between the cases is the sign of the BC edge. Recall that zero is non-negative.

Lower Domination (2) / "if". To show lower domination the requirement becomes that $T_A \geq T_C + x$ in all cases.

Lower Domination (2) / "if" / all negative. In Figure 2.8a all three edges are negative. Therefore $T_C < T_B < T_A$, so B executes before A and the lower bound of AB is applied to A. We have $T_B \geq T_C + z$, and therefore the lower bound propagated via B becomes $T_B + y \geq T_C + y + z = T_C + x$, which is required for lower domination.

Lower Domination (2) / "if" / one positive. In Figure 2.8b, C is unordered relative to B. However, we still have $T_A > T_B$ and $T_A > T_C$. From the fact that $-x = -y + z$ and $z \geq 0$ it follows that $-y \leq -x$ (i.e. $y \geq x$). Combined with the upper bound definition $T_C \leq T_B + z$, we get $T_B + y = T_B + z + (y - z) \geq T_C + (y - z) = T_C + x$ since $-x = -y + z$ and therefore $x = y - z$. To summarize, $T_A \geq T_B + y \geq T_C + x$, which is required for lower domination.

Lower Domination (2) / "only if". To conclude the proof of part 2 we show that if $|\mathbf{AB}| + |\mathbf{BC}| \neq |\mathbf{AC}|$, AC cannot be lower dominated by BA. Again $|\mathbf{AB}| + |\mathbf{BC}| < |\mathbf{AC}|$ is not possible since it violates the triangle inequality assumption.

If $|\mathbf{AB}| + |\mathbf{BC}| > |\mathbf{AC}|$ and the edge weights match the scenario in Figure 2.8a there is a dispatch scenario where the lower bound propagated via AB is $T_C + z + y < T_C + x$ since $|\mathbf{AB}| + |\mathbf{BC}| > |\mathbf{AC}| \Leftrightarrow -y - z > -x \Leftrightarrow y + z < x$. This means that there is a scenario in which the lower bound propagated via AC is tighter than the lower bound propagated via AB and therefore AC cannot be lower dominated by AB.

If $|\mathbf{AB}| + |\mathbf{BC}| > |\mathbf{AC}|$ and the edge weights match the scenario in Figure 2.8b there is a dispatch scenario where the lower bound propagated via AC is $T_B + z + x$. This scenario happens if C is executed as late as possible. In this scenario the lower bound propagated via AB is $T_B + y < T_B + z + x$ since $|\mathbf{AB}| + |\mathbf{BC}| > |\mathbf{AC}| \Leftrightarrow -y + z > -x \Leftrightarrow y < z + x$. Therefore the bound propagated by AC is tighter and AB cannot lower dominate AC. □

The filtering algorithm for creating minimal directly dispatchable distance graphs makes use of the dominance relation. Since there is no mixing between signs in domination, the filtering can take care of positive and negative edges separately.

Algorithm 2 shows the pseudo-code for the filtering algorithm. The algorithm shown is the first filtering algorithm by Muscettola et al. [23] with complexity $O(|V|^3)$ (where $|V|$ is the number of nodes in the graph). A faster, more advanced version exists [36]. It has a run-time complexity of $O(|V||E| + |V|^2 \log |V|)$ (where $|E|$ is the number of edges in the initial graph). It is shown that that the dominance relation is an equivalence relation [23]. All edges in an equivalence class dominate each other. The equivalence classes form a partial order since domination is shown to be a transitive relation. It is further shown that if only one edge from each top-level equivalence class is kept the resulting filtered network becomes minimal and dispatchable.

Algorithm 2: Edge Filtering [23]

```
function MINIMUM_DISPATCHABLE(G - STN)
Calculate the APSP-graph using Johnson's or Floyd-Warshall's algorithm
for each triangle in the APSP-graph do
    check dominance of the edges
    if one edge dominates another then
        mark the dominated edge
    end
    if both edges dominate each other then
        make sure one of them is marked
    end
end
return a graph consisting of all unmarked edges
```

The algorithm first calculates the dispatchable APSP graph for the STN. It then goes through all triangles and marks the dominated edges. In case two edges dominate each other, one is kept unmarked. Then all marked edges are removed and the resulting graph is returned. It is shown that the strategy for marking edges ensures that only one edge from each top-level equivalence class is kept in the final graph.

We now have the machinery needed to present the direct dispatchability verification algorithm.

2.2.3 Direct Dispatchability Verification

STNs are not normally dispatched in their original form. As shown in the previous section they need to be processed and additional constraints must be added before dispatchability is attained. The added constraints have the effect of relaying all the needed information for dispatch to succeed. However, in some cases the original STN cannot be altered. This prevents the standard procedure presented in the previous section where the algorithm first calculates the APSP graph to ensure dispatchability. The reason for not altering the STN may be that execution is distributed or that nodes are owned by separate entities so that new constraints between nodes cannot be imposed on the global STN. It could also be caused by the fact that communication can only follow certain paths, along the edges in the STN. Regardless of which reason that is preventing alteration of the STN, it is important to verify whether the given STN can be dispatched. There does not exist any algorithm for verifying this property. Therefore, we present a

new solution based on the following lemma:

Lemma 2. ***[This thesis]*** *Let the dominance equivalence classes of a consistent STN G be defined as in the previous section [36]. Then G is directly dispatchable if and only if for each dominance equivalence class of G, G either contains at least one edge from that class or contains at least one edge from a class ordered before it in the preorder induced by the dominance relation.*

Proof. **"if":** An STN which contains one edge from each dominance equivalence class, or a class ordered before it, is directly dispatchable. This follows from the theory of filtering where it is proven that such an STN is minimal and directly dispatchable [36]. Since such an STN is minimal it is not affected by any additional edges in G, they cannot increase or decrease the execution intervals created during dispatch. An increase is impossible since adding more constraints cannot prevent the existing ones from limiting the execution intervals. A decrease is impossible since the STN was already minimal.

"only if": We now show that a consistent directly executable STN must contain an edge from each dominance equivalence class, or a class ordered before it in the preorder induced by the dominance relation.

Suppose for contradiction that no edge from a certain dominance class, or class ordered before it, were present in the STN. Let T be the target node if it is an upper domination equivalence class and the source node if it is a lower domination equivalence class. All the edges from the involved equivalence classes propagate the same upper/lower bound to T relative to the temporal reference. This can be seen by inspection of the triangle theorem (Theorem 1) where the bound propagated is relative to A. It is clear that other dominations of the same edge must relate to nodes which relate to A and since this is related to the temporal reference we can relate the bounds directly to this for any equivalence class. Since the dispatcher propagates the tightest possible execution windows along the edges in the dominance classes by virtue of them representing a minimal STN, the fact that the contradicting STN does not contain any of these edges means that any bounds propagated to T will create an interval which is not a subset of the execution window propagated in the APSP version of the STN. By Lemma 1 this STN cannot be directly dispatchable.

□

Algorithm 3 is based on Lemma 2 and verifies direct dispatchability. The algorithm first builds the equivalence classes used by the filtering algorithm (Algorithm 2) presented in section 2.2.2. Since the filtering algorithm

Algorithm 3: Direct Dispatchability Verification

```
function VERIFY_DIRECT_DISPATCHABILITY(G - STN)
Build the dominance equivalence classes
    using a modified MINIMUM_DISPATCHABLE
Construct the partial order of the classes
for each equivalence class with order o do
    if G contains no edge in a class ≤ o then
        return false
    end
end
return true
```

does not output these directly, a modified version is needed. The filtering algorithm produces a minimal dispatchable result. The verification algorithm may need to verify an STN which is dispatchable but not minimal. Therefore it does not suffice to compare the STN to the result of the filtering algorithm. Instead the criterion of Lemma 2 is used directly. Algorithm 3 checks if any of the classes are not covered, either directly by an edge from the class being in the STN, or indirectly if an edge in a class ordered before is included. If all classes are covered the algorithm returns true, otherwise false.

2.3 The Dispatch Back-Propagation Approach

In this section we present an excerpt of a larger algorithm. Stedl and Williams [33] introduce the concept of Dispatch Back-Propagation (DBP) as part of their Fast-DC algorithm. The Fast-DC algorithm is the basis for the FastIDC algorithm which is the focus of our research. Similar to the edge filtering in the previous section, the DBP algorithm works in the distance graph of the STN. Addition of a new constraint to an STN is seen as a tightening of an existing constraint (with infinite bounds).

The idea behind DBP is that if a constraint $i \rightarrow j$ is tightened this will not affect how the dispatcher dispatches nodes that are executed after j. Any propagation along edges during dispatch to nodes executed after j are not affected by the actual time assigned to j. Their constraints toward j are all relative to j's assigned time. Therefore, in order to keep a dispatchable STN dispatchable after a constraint is tightened, only nodes that are executed before j need to be checked for possible inconsistency. Since different

execution scenarios may execute nodes in different order, some nodes may be either before or after the node j depending on the scenario. Recall that the dispatcher may assign any execution time to a node as long as it is within the execution window of the node.

The effect of tightening the $i \rightarrow j$ constraint is that the execution window for j becomes narrower. Then the STN distance graph may become non-dispatchable and the STN even inconsistent (remember that consistent STNs cannot always be directly dispatched through their distance graph). In order to detect inconsistencies, and preserve dispatchability, the DBP algorithm imposes new constraints on any direct path along which an inconsistency could be propagated. The DBP algorithm will detect if the tighter constraint leads to the STN not being dispatchable anymore, with exceptions. We will see in Chapter 4 that there are cases in which it fails to detect non-dispatchability correctly. The solution to this problem is an important result of this thesis.

In more detail, DBP visits all possible predecessors of j and imposes new constraints on these to preserve dispatchability. This is done recursively, affecting the predecessors of j's predecessors, and so on towards the start of the STN until either no more predecessors exist or DBP detects that dispatchability cannot be preserved. Figure 2.9 shows the two cases of back-propagation which are needed to preserve dispatchability or detect non-dispatchability. The bold edges in the figure are the result of DBP propagation. As mentioned before, 0 is treated as a negative edge by DBP, Fast-DC and FastIDC.

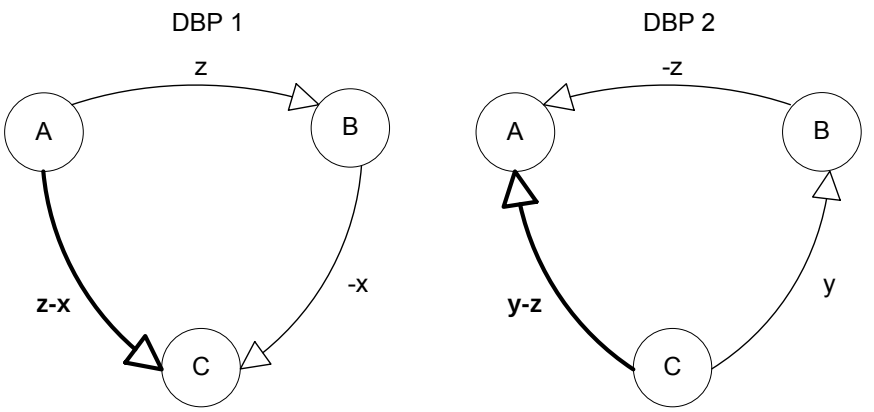

Figure 2.9: The two cases of back-propagation used by the DBP algorithm.

The DBP-1 propagation is triggered if a positive edge AB is tightened and remains positive. The triggering condition for adding or tightening edge AC is that BC is an outgoing negative edge. The rationale is that an upper bound which is propagated from A to B may be threatened by an incoming lower bound from C. If the lower bound requires B to execute after the upper bound's requirement then the STN is inconsistent. By adding the AC edge, DBP forces C to be executed enough time before B so that the time propagated by the lower bound cannot be above the upper bound.

The DBP-2 propagation is triggered if a negative edge AB is tightened or a previously positive edge AB is tightened to a negative value. It will add or tighten edge CA if there is a positive incoming edge CB. The rationale is now the opposite. A lower bound propagated via A will lead to propagation of a bound to B that may be threatened by an upper bound propagated via C. By adding the CA edge, the DBP algorithm makes sure that A will not be executed too late, preventing an empty execution window for B.

DBP propagations are focused on the tightened edge. If the current focus is disregarded, the triangles become isomorphic. An incoming positive edge is followed by an outgoing negative edge producing a tightening from the source of the positive edge to the target of the negative edge. The weight of the tightened edge is the sum of the weights of the other two edges.

Algorithm 4 lists the pseudo-code of the DBP algorithm. This algorithm is not given directly in the paper on Fast-DC [34] where the concept is first presented. It is only included as an integrated part of Fast-DC.

The algorithm starts with the tightened edge and then recursively applies the back-propagation rules where applicable. First the algorithm checks if it is tightening a loop. A positive loop results when DBP is called with node A equal to node C in Figure 2.9. This is a case of testing local consistency between two nodes. A positive loop cannot be further recursed and presents no threat to dispatchability. A negative loop is the result of discovering an inconsistency. In case such an edge is found the algorithm returns false.

If the tightened edge is not a self-loop, DBP checks if it can apply DBP-1 or DBP-2. After each application, a recursive call is made to verify that the STN is still dispatchable. If any of the recursions fail, false is returned. Otherwise no inconsistencies were found and true is returned. The STN, G, contains the tightened edges derived by DBP. This should be kept for use in future invocations of the algorithm.

If the DBP algorithm returns false, the STN which was just processed

Algorithm 4: Dispatch Back-Propagation

```
function DISPATCH_BACK-PROPAGATION(G - STN, e - tightened edge)
if Source(e) = Target(e) then
    if Weight(e) ≥ 0 then
        return true      // positive loop
    else
        return false     // negative loop
if e is positive then
    for each negative edge f with Source(f) = Target(e) do
        if !DISPATCH_BACK-PROPAGATION(G, tighten according to DBP-1) then
            return false
    end
else
    for each positive edge f with Target(f) = Source(e) do
        if !DISPATCH_BACK-PROPAGATION(G, tighten according to DBP-2) then
            return false
    end
end
return true
```

cannot be dispatched and hence is inconsistent. The self-loops encountered by the algorithm are never added to the distance graph. Therefore the STP definition is not violated by the use of these edges. It is possible to interleave filtering of edges [36] as presented in Section 2.2.2 since this does not affect dispatchability.

In Chapter 4 we analyze DBP further as a cause of FastIDC failures. We end by remarking that it is possible to come up with an STN with n nodes where DBP is as expensive as an all-pairs shortest-path calculation, i.e. $\Theta(n^3)$.

Comparison between APSP and DBP

The two triangles in Figure 2.9 both show derivations of shortest paths. If tightening the AB edge produces a shorter path between A and C, the AC edge is updated to reflect this. Therefore, DBP applies a subset of the tightenings that would be done if an APSP algorithm were used. There are four types of triangle interaction between two edges dependent on the edge signs: plus-plus, plus-minus, minus-plus and minus-minus. All these interactions are applied in APSP calculations, but only the plus-minus com-

bination is applied by the DBP algorithm. This is all that is needed to ensure dispatchability, assuming there are no negative cycles (which is a case which we will discuss in Chapter 4). Plus-minus interaction derives shortest paths along negative edges. This means that the path is derived towards nodes that are executed earlier. If a consistent STN is built using calls to DBP, all nodes will have shortest path distances towards the temporal reference.

An interesting property is that, since dispatchability is maintained, any consistent STN can be dispatched after it was processed through DBP. This means that even though not all edges in the STN distance graph have weights that are the shortest distances between nodes, these shorter distances are in fact recovered during execution / dispatch. If this did not happen, execution windows would allow values outside those of the minimal STN which would cause dispatch to fail. Therefore, for a consistent STN, DBP does a "lazy"-APSP calculation which shifts the work done to the STN so that less work is done during processing and more during execution.

2.4 Simple Temporal Networks with Uncertainty

We have now seen the basic concepts relating to STNs: Distance graphs, consistency verification, execution, and dispatchability. These will now be discussed in the more general model of temporal problems where uncertainty is allowed. We start with the core definition:

Definition 6 (Simple Temporal Problem with Uncertainty (STPU) [44]). *A **simple temporal problem with uncertainty** (**STPU**) consists of a number of real variables $x_1, \ldots, x_n$, divided into two disjoint sets of* controlled events *R and* contingent events *C. An STPU also contains a number of* requirement constraints $R_{ij} = [a_{ij}, b_{ij}]$ *limiting the distance $a_{ij} \leq x_j - x_i \leq b_{ij}$, and a number of* contingent constraints $C_{ij} = [c_{ij}, d_{ij}]$ *limiting the distance $c_{ij} \leq x_j - x_i \leq d_{ij}$. For the constraints C_{ij} we require that $x_j \in C$ and $0 < c_{ij} < d_{ij} < \infty$.*

Controlled events can be assigned timepoints by choice. The timepoints at which contingent events occur can only be observed and they are therefore sometimes also referred to as *observable events* or *uncontrollable events*. Requirement constraints are constraints that are required to hold in the STPU. A contingent constraint is used to describe the uncertainty of a process that is started at one event, which may or may not be controllable, and ends in a contingent event.

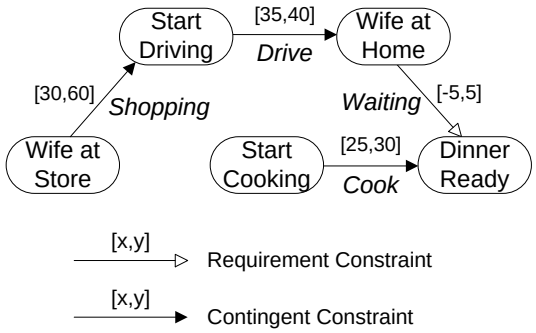

Figure 2.10: Example STNU cooking scenario.

Requirement constraints can be placed between any types of events. In contrast, it makes no sense to allow two contingent constraints to end in the same contingent event. This would correspond to stating that timepoint t is the timepoint when *both* action A and action B end, but it is impossible to guarantee that such a timepoint will exist except in the trivial case where both actions actually have a single known predefined duration.

STPUs in graph form are called STNs with Uncertainty (STNUs). We now give an example of a scenario which can be modeled by an STNU.

Example 3. *Suppose that a man wants to surprise his wife with some nice cooked food after she returns from shopping. For the surprise to be pleasant he does not want her to have to wait too long for the meal after returning home. He also does not want to finish cooking the meal too early so it has to lay waiting. We can model this scenario with an STNU as shown in Figure 2.10. Here the durations of shopping, driving and cooking are uncontrollable (but bounded). This is modeled by using contingent constraints between the start and end events of each action. The fact that the meal should be done within a certain time of the wife's arrival is modeled by a requirement constraint which must be satisfied for the scenario to be correctly executed. The question arising from the scenario is: can we guarantee that the requirement constraint is satisfied regardless of the outcomes of the uncontrollable durations, assuming that these are observable.*

We will return to the question posed in this example later in this chapter.

Wait Constraints. In addition to the types of constraints already existing in an STNU, some algorithms can also generate *wait constraints* that make certain implicit requirements explicit for use in further computations and execution of the STNU.

Definition 7 (Wait Constraint [21]).
*A **wait constraint**, or **wait**, between the nodes A and C is a constraint AC with annotation $\langle B, t \rangle$. This constraint is respected if execution of node C is not allowed to take place until **either** B has occurred **or** t units of time have elapsed since A occurred.*

Here B is called the *conditioning* node of the constraint and it must be an uncontrollable event. We will see how wait constraints can be derived in certain situations by the MMV algorithm (Chapter 3).

2.4.1 Different Controllabilities

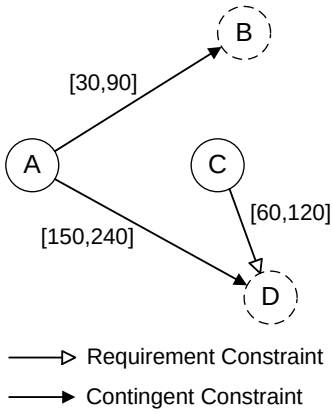

Figure 2.11: Example of a weakly controllable STPU.

For STNUs, consistency is insufficient. Instead, the interesting question is whether an STNU is *controllable* in a certain way.

The definitions of controllability often use the notion of a projection. A *projection* of an STNU is an STN where a value from each contingent constraint interval is chosen to represent the contingent constraint in the STN. The contingent constraint is converted to a requirement constraint in the projection process. Intuitively a projection is what is got after assigning a fixed duration to each uncontrollable duration.

Two types of controllability are defined by Vidal and Ghallab [44]. An STNU is *strongly controllable* if it is possible to assign the controlled events a single set of timepoints so that every projection of the resulting STNU is a consistent STN. This means that there is a universal solution that can be

determined in advance and used regardless of which timepoints the contingent events take from their domains. An STNU is *weakly controllable* if every projection is consistent. This means that for every outcome of the uncontrollable durations it is possible to assign timepoints to the controllable events so that the corresponding projection is consistent. On the other hand, actually doing this may require either luck or knowledge about the future.

The example in Figure 2.11 is not strongly controllable. There is no single timepoint for C that is consistent both when $AD = 150$ and $AD = 240$. However, given a projection of timepoints for the contingent events B and D, we can set $C = D - 80$ and get a consistent network, which shows that the example network is weakly controllable. However, we cannot set $C = D - 80$ until we actually observe the time where D occurred, at which point it is too late to execute C.

It is clear that strong controllability implies weak controllability. Networks that are strongly controllable can be executed, but there exist networks that can be executed that are not strongly controllable. Weak controllability on the other hand gives not much help in deciding whether an STNU can be executed since it assumes a projection and at execution time not all timepoints for contingent events are known. When executing a controllable event, the timepoints available to assign to it may depend on a contingent event that has not yet been observed. This scenario cannot be captured by weak controllability. Hence a better model is needed to handle execution scenarios.

Vidal and Fargier [42] defines this more useful type of controllability. Dynamic controllability for an STNU is defined as the possibility to execute the network consistently, given at each timepoint knowledge of all contingent events that have happened up to this point in time. Dynamic controllability captures the real situation of executing in a dynamic environment. An execution procedure trying to schedule a controllable event that is dependent on a contingent event which has not yet been observed knows this and can compensate for it when deciding the execution timepoint for the controllable event. The concept of dynamic controllability is essential to this thesis. It is formally defined via the concept of dynamic execution strategy.

Definition 8 (**Dynamic Execution Strategy [42]**).
*A **dynamic execution strategy** is a strategy for assigning timepoints to controllable events during execution, given that at each point in time, it is known which contingent events have already occurred. The strategy must ensure that all*

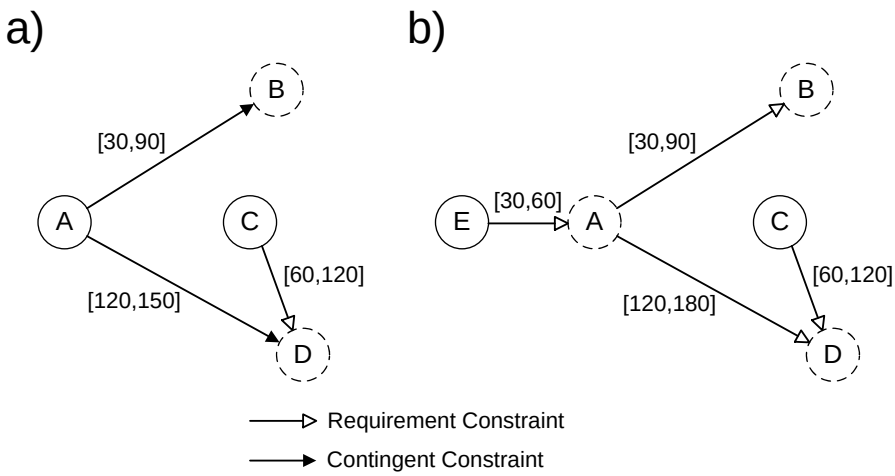

Figure 2.12: Example of a strongly and a dynamically controllable STPUs

requirement constraints will be respected regardless of the outcomes for the contingent events.

Definition 9 (Dynamically Controllable (DC) [42]).
*An STNU is **dynamically controllable (DC)** if there exists a dynamic execution strategy for it.*

We now return to the example in Figure 2.10. A dynamic execution strategy is to start cooking 10 time units after receiving a call that the wife starts driving home. This guarantees that cooking is done within the required time, since she will arrive at home 35 to 40 time units after starting to drive and the dinner will be ready 35 to 40 time units after she started driving.

In contrast the example in Figure 2.11 is not dynamically controllable. Assuming without loss of generality that $A = 0$. We cannot start C after 90 since then if D is observed at 150 we violate the CD constraint by getting a value on the link that is less than 60. We cannot start C before 90 or at 90 since then the CD constraint will be violated if D is not observed until 240 leading to a value for the CD constraint at above 120.

Figure 2.12 shows two variations of the same example. In 2.12a, if A is set to 0 and C to 60, the constraint CD will be satisfied as the possible values will be in the range $[120 - 60, 150 - 60] = [60, 90] \subseteq [60, 120]$. This means that the example is strongly controllable since $C = 60$ works for all the contingent values on D.

The example in Figure 2.12b is a bit different from the earlier examples.

Again assume that the time of the earliest node E is set to 0 for simplicity. Here the value of A will be observed somewhere in the interval 30 to 60 after the execution of E. Then D in turn will be observed 120 to 180 after A. It is not possible to assign a value to C that works for all values of D ($150 \leq D \leq 240$ which is similar to the first example). So the network is not strongly controllable. It is however dynamically controllable. To see this we observe that if we know the time at A we find ourselves at this point in time in a scenario similar to that in Figure 2.12a where we can choose $C = A + 60$ to get an interval for the CD constraint that is $[120 - 60, 180 - 60] = [60, 120]$. Hence the time of D will satisfy the constraint.

In comparison to the other types of controllability, dynamic fits in-between the others. This means that we have $strong \Rightarrow dynamic \Rightarrow weak$.

Figure 2.13 shows the knowledge that is used in the respective ways of controllability. We see that strong controllability cannot make use of any knowledge about the actual timings of the network. In contrast to this, weak controllability makes use of the complete knowledge of all uncertain timings. In-between we find the most realistic type of controllability, which is dynamic controllability. If we take the perspective of an autonomous agent executing a plan, it knows what has happened and plans for the future.

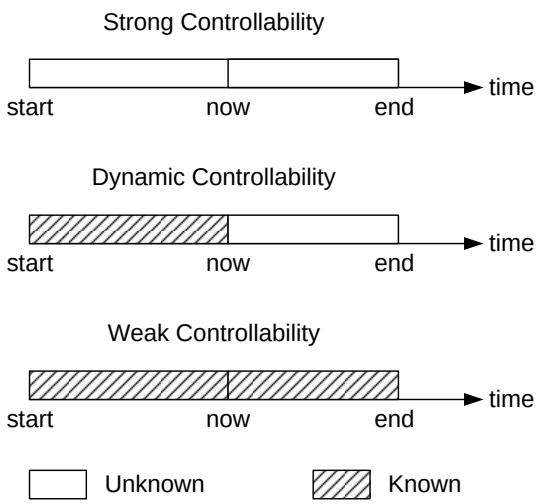

Figure 2.13: Different types of controllability for STPUs

DC STNUs can be executed by a dispatcher taking uncontrollable events into account. The algorithm required depends on whether the STNU has been preprocessed. A dispatcher for STNUs processed by the MMV algorithm will be presented later.

2.4.2 Complexity of Verifying Controllabilities

Vidal and Fargier [42] classify the different controllability problems in terms of complexity. Determining strong controllability is found to be in P since it is possible to take the worst case outcome of a projected STP and check if this is consistent. The worst case is seen when considering the outcome of a contingent event to be the latest timepoint in all constraints were the upper limit is constrained and similarly that it occurs at the earliest possible timepoint in constraints constraining the lower limit. We again use the example in Figure 2.11 to show this. When checking the constraint $D - C \leq 120$ we get the worst case by assuming the upper bound $D = 240$ and when checking the constraint $D - C \geq 60$ we assume the lower bound $D = 150$.

Determining weak controllability is found to be in *co-NP*. This is due to the fact that if we want to check if the network is not weakly controllable it suffices to check the lower and upper bounds of each constraint. Using only the extreme bounds is justified, since if a violation is caused by a value inside an interval, the extremes of the interval would also expose this. A naive algorithm for finding violations can check all combinations. This can be done by a non-deterministic Turing machine in polynomial time.

The complexity of determining dynamical controllability is discussed by Vidal and Fargier [43]. In this article there are conjectures of the complexity classes of both weak and dynamic controllability. The three controllability problems are reformulated using existential and universal quantifiers. Strong controllability is characterized by a list of existential quantifiers outside the scope of an expression which is a conjunction of universal quantifiers for each constraint. This can be solved in polynomial time.

Dynamic controllability is characterized as a game against nature where each side take turns choosing. Nature has a universal operator and the executing entity has an existential operator. The view of dynamic controllability as a game has recently surfaced through the use of Timed Game Automata (TGA). This angle of pursuit will be discussed in Chapter 7. Through the game characterization, verifying dynamic controllability was believed to be *PSPACE*-complete. A later result in Morris and Muscettola [22] conjectured it to be NP-complete. In Chapter 3 we see it is in fact in *P*.

Weak controllability can be characterized by an expression where each

uncontrollable node corresponds to a universal quantifier and each controllable node an existential quantifier. In the expression all universal quantifiers are outside the scope of the existential quantifiers. The expression encodes the concept that for all possible timepoints of all contingent events there exist timepoints for the controlled events so that the resulting network is consistent. The authors believed at the time that this would be simpler to check than dynamic controllability, which at the time was believed to be *PSPACE*-complete. Hence they conjectured that weak controllability was *co-NP*-complete which as later shown to be correct by reduction from the 3-coloring problem [19].

This section has followed the evolving view on the DC verification problem over time. We now continue with a more algorithm-based focus.

2.4.3 Extended Distance Graph Representation

In the same way as we saw previously for STNs, an STNU always has an equivalent *extended distance graph* [34] (in the original paper they were called CDGU for Conditional Distance Graph with Uncertainty, but since there are no CDGs or DGUs we chose to rename it to a more fitting name).

Definition 10 (Extended Distance Graph (EDG) [34]).
*An **extended distance graph** (**EDG**) is a directed multi-graph with weighted edges of 5 kinds: Positive requirement, negative requirement, positive contingent, negative contingent and conditional.*

Requirement edges and contingent edges in an STNU are translated into pairs of edges of the corresponding type in a manner similar to what was previously described for STNs. The *conditional* edges mentioned in the definition, first used by Stedl [34], are used to represent the *wait constraints* that are derived by some algorithms (see Chapter 3). The direction of a conditional edge is intentionally opposite to that of the wait it encodes. This makes the conditional edge more similar to a negative requirement edge in the same direction, the difference being the condition. Conditional edges are never present in the initial EDG, but they can be derived from other constraints through calculations discussed in Chapter 3. We now give a formal definition of a conditional edge:

Definition 11 (Conditional Edge [34]).
*A **conditional edge** CA annotated $\langle B, -w \rangle$ encodes a conditional constraint: C must execute after B* or *at least w time units after A, whichever comes first. The node B is called the* conditioning node *of the constraint/edge.*

Algorithm 5: STNU Dispatcher [21]

```
function DISPATCH(G - STNU)
enabled ← {Temporal-Reference}
executed ← {}
currentTime ← 0
repeat
    minTime ← min_{e∈enabled} lowerBound(e)
    Advance time until uncontrollable event observed or
    currentTime = minTime
    if uncontrollable event e observed then
        execute ← e
        Remove any waits conditioned on e
    end
    else
        execute ← any live event in enabled whose waits are satisfied
    end
    executed ← executed ∪ {execute}
    enabled ← enabled \ {execute}
    Assign currentTime to execute
    Propagate execution bounds along constraints to neighboring events
    enabled ← enabled ∪ {newly enabled events}
until All nodes are executed
```

2.4.4 Execution of STNUs

How an STNU can be executed depends on which algorithm is used to verify its DC status. All algorithms except the Morris algorithm produce, as a side effect of verification, an STNU that can be dispatched by the dispatcher in Algorithm 5, or a slight variation of this for FastDC which uses conditional edges.

This dispatcher which was originally presented in [21] is shown here in a different format. The dispatcher uses two distinct conditions to determine whether an event e can be executed. First, e must be *enabled*, meaning that all events that must be executed before it have actually been executed. These events can be found through the outgoing negative requirement edges, similarly to how it is done when dispatching STNs (Section 2.2). Second, e must be *live*, meaning that it is within its permitted *execution window*. These execution windows are related to the constraints from the original STNU and cannot be determined in advance. Instead they are initialized to $[0, \infty]$ and then dynamically updated as events actually oc-

cur during execution. Observations of uncontrollable events are handled through the same mechanism, causing the execution windows of "dependent" nodes to be updated. When an event becomes enabled, its execution window is guaranteed to be fully updated. For example, suppose that *Start Cooking* in Figure 2.10 is executed at time 50. Then, and only then, can we infer that *Dinner Ready* must occur within the interval $[75, 80]$.

STNUs processed by the Morris algorithm need intermediate processing [12, 13] before they can be executed.

Chapter 3

Algorithms for Verifying Dynamic Controllability

There exist two types of DC verification algorithms, **full** and **incremental**. A full verification algorithm is used to process a whole STNU in one go. The best places to apply these algorithms are to full plans for which there is no prior information of DC status. Incremental algorithms repeatedly verify the DC status of an STNU as small incremental changes are made to it. These algorithms are well suited for use in most types of automated planning. In particular, many planners are based on incrementally extending plan candidates with new actions and constraints. If such a planner determines that a particular plan candidate is not dynamically controllable, none of its descendants can be dynamically controllable, since the descendants can never weaken the current constraints in the STNU. Then the planner can verify after each action addition whether the plan remains DC, and if not, it can safely backtrack without missing solutions. This allows the planner to prune infeasible parts of the search tree as soon as possible.

In this chapter we present the "classical" MMV algorithm for determining dynamic controllability [21]. It is an example of a full algorithm. We then present the FastIDC algorithm [32, 33] which was the only known incremental algorithm at the time our work started. FastIDC does support incremental tightening/addition of constraints, which is needed by planners based on incrementally extending plans. FastIDC also supports incremental loosening/removal of constraints, which is not strictly required by these types of planners.

As a final note it should be said that any full algorithm trivially is an

Algorithm 6: Original MMV Algorithm [21]

```
procedure DynamicallyControllable? (network W)
1. Compute the All-Pairs graph for W.
   If W is not pseudo-controllable then
   return false.
2. Select any triangle such that
   v is non-negative. Introduce any tightenings
   required by the Precede case and any waits
   required by the Unordered case.
3. Do all possible regressions of waits, while
   converting unconditional waits to lower bounds.
   Also introduce lower bounds as provided by the
   general reduction.
4. If steps 2 and 3 do not produce any new
   (or tighter) constraints, then return true,
    otherwise go to 1.
```

incremental algorithm since it can regard the STNU together with the increment as a full STNU.

3.1 The MMV Algorithm

The MMV algorithm has appeared twice in work by Morris et al. [20, 21]. In the first paper it was presented as in Algorithm 6. In the second paper it was presented in a more concise form which is shown in Algorithm 7. These algorithms share the same worst case complexity, but may process the graph differently due to different selection order for triangles. Algorithm 7 is easier to explain and so we will mainly use this in the thesis.

Pseudo-controllability. The algorithm builds on the concept of *pseudo-controllability*, a necessary but not sufficient requirement for dynamic controllability. Figure 3.1 shows how pseudo-controllability can be used when verifying dynamic controllability.

Figure 3.1a shows an example STNU which contains two uncontrollable actions which are modeled by contingent constraint and one requirement constraint that constrains the total time of the two actions. This STNU is not dynamically controllable since the actions may take up to 60 time units whereas the constraint only allows for a maximum of 30 time units. This is

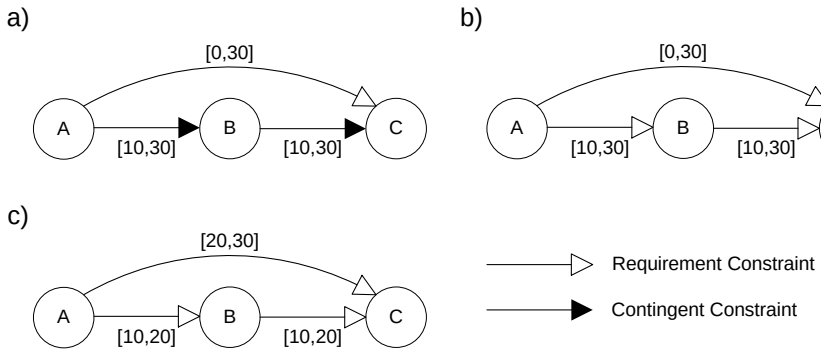

Figure 3.1: Example of pseudo-controllability as a necessity for dynamic controllability.

detected by a pseudo-controllability check.

The first step is to convert the STNU to an STN, as shown in Figure 3.1b. This is done by replacing contingent constraints with requirement constraints having the same bounds. However, to detect if the STNU is not dynamically controllable it is not enough to test its STN version for consistency, since this only determines if there are possible durations for the actions for which the total duration meets the requirement. For example, a consistent solution to the example in Figure 3.1b is A=12, B=12, C=24, which satisfies all requirement constraints.

Suppose we take this one step further to create the corresponding *minimal* STN, as seen in Figure 3.1c. Here only values that will definitely prevent consistency have been removed from the constraint bounds. For example, the original STN cannot be consistent if the temporal distance between A and C is less than 20. Therefore the original constraint interval of $[0, 30]$ is squeezed to $[20, 30]$ in the minimal STN. This is not a problem for the original STNU since squeezing a requirement constraint only limits the possible choices, and as long as there is at least one value in the interval dynamic controllability may still be possible. However, the interval between A and B is squeezed from $[10, 30]$ to $[10, 20]$. If this is translated back to the STNU this means that there are several values of the original *contingent* constraint that are not allowed. If the first action takes 25 time units, which is a possible outcome for the STNU, the requirement constraint cannot be satisfied, and so the STNU cannot be dynamically controllable.

The procedure of testing pseudo-controllability is then as follows:

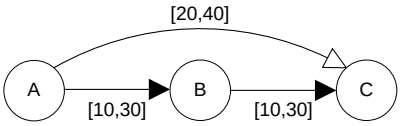

Figure 3.2: Example of a pseudo-controllable STNU which is not dynamically controllable.

1. Convert the STNU into an STN by converting contingent constraints into requirement constraints.
2. Compute the minimal STN (see Chapter 2, page 14).
3. If the STN is found to be inconsistent, the STNU cannot be DC. If there is no possibility for consistency when assigning timepoints to events, this will not become possible by letting external forces assign a subset of the timepoints.
4. Compare all contingent constraints in the STNU with their corresponding constraints in the minimal STN. If any of these constraints are squeezed the STNU cannot be DC.
5. If non-DC was not detected in steps 3 or 4, the STNU is pseudo-controllable.

While pseudo-controllability is a necessary condition for DC, it is not sufficient. Figure 3.2 shows an example STNU which is pseudo-controllable. The corresponding STN is directly minimal and consistent, and does not result in a squeeze. The AB constraint that was squeezed in the previous example can now also take on the full duration of 30, since in this case it is possible that the second duration is 10, and so no values can be pruned from either of these two constraints. But it is still possible that the total duration is 60 which violates the AC constraint and so the STNU is not DC.

In order to detect STNUs that are not DC but pseudo-controllable MMV additionally uses STNU-specific *tightening rules*, also called *derivation rules*, which make constraints that were previously implicit in the STNU explicit (Figure 3.3). Each tightening rule can be applied to a "triangle" of nodes if the constraints and requirements of the rule are matched. The result of applying a tightening is a new or tightened constraint, shown as bold edges in the leftmost part of the triangle. Note that unordered reduction generates wait constraints, which cannot be present in the original STNU.

Algorithm 7: Recent MMV Algorithm [20]

```
Boolean procedure determineDC()
repeat
    if not pseudo-controllable then
        return false
    else
        forall the triangles ABC do
            tighten ABC using the tightenings in Figure 3.3
        end
until no tightenings were found
return true
```

Figure 3.4 shows an example of how MMV could apply a tightening. In Figure 3.4a a contingent constraint and a requirement constraint meet in the node B. If we look at the tightenings in Figure 3.3 we see that the example in 3.4a matches the "Precedes Reduction" rule. This allows MMV to derive the bold edge in the final figure, Figure 3.4c. We use this example to informally reason about why this rule is applicable. In doing so we will first derive the intermediate constraint in Figure 3.4b which contains only the lower bound. Then we will conclude with deriving also the upper bound for a total constraint equal to that of 3.4c. Suppose that node A was executed at time 0. If node C was executed at time 3 there is a possibility that the upper bound, 20, on CB is violated since B may occur at time 25 which is 22 time units after C. If the STNU is dynamically controllable this cannot happen. We can increase the start time of C and apply the same reasoning until we reach time 5. At this time a long duration of AB cannot lead to a violation of the upper CB bound. From this we conclude that C must be executed after time 5, or 5 time units after A for the STNU to be DC. Therefore we can infer the lower bound of 5 which is shown in Figure 3.4b. There is as of yet no upper bound decided so we leave it blank, '_', for now.

To reason about the upper bound we focus on how late we can start C compared to A. If C is executed at time 20, with A still being executed at 0, B could be observed at time 21 which violates the lower bound of CB stating that B should be at least 5 time units after C. By adjusting downwards and reapplying the same reasoning we see that we can infer an upper bound of 15 on the time between A and C. So in order for the STNU to be DC we can in this situation infer a constraint $[5, 15]$ between A and C.

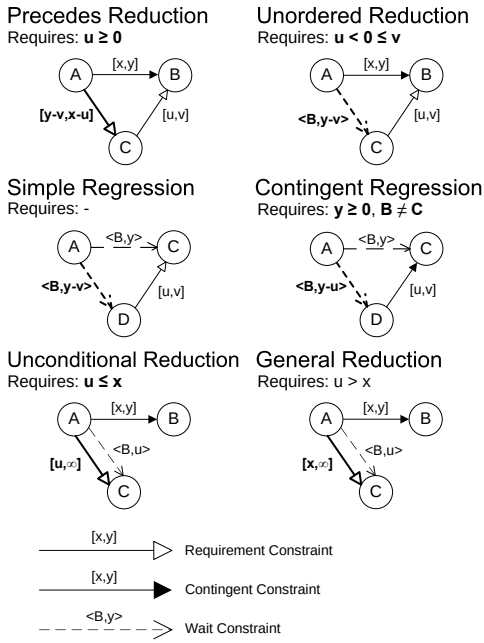

Figure 3.3: Tightenings (derivations) of the MMV algorithm.

Algorithm 7 shows that MMV consists of an outer loop, where it first verifies pseudo-controllability and transfers all tighter constraints found by the associated APSP calculation into the STNU, then applies all possible tightenings through the inner loop. If an STNU is not DC, the tightenings will eventually produce sufficient explicit constraints for the pseudo-controllability test to detect this [21].

The complexity of MMV is said to be $O(Un^3)$ where U is a measure of the size of the domain (the number of constraints times the quotient of the largest and smallest constraint bounds) [21]. This comes from a cost of $O(n^3)$ per iteration and the fact that each iteration must tighten at least one constraint by an amount at least the size of the smallest bound. In the worst case this can occur repeatedly until a negative cycle is formed. Since the complexity bound depends on the size of constraint bounds, it is pseudo-polynomial.

Clarifications. Both descriptions of Algorithm 6 [21] and Algorithm 7 [20] are quite concise and omit certain facts that are essential for using the algorithms. In particular, it is not specified whether the APSP graph computed

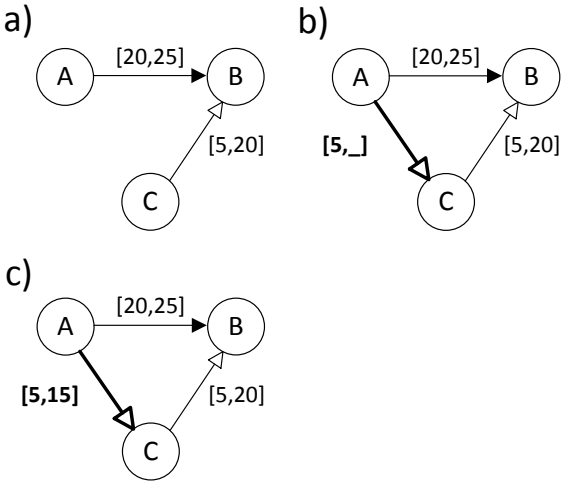

Figure 3.4: Tightening example.

in step 1 is also used in steps 2 and 3, or whether those steps use the original network W. This needs to be clarified.

The authors use Johnson's Algorithm [6] to compute the APSP graph. It is a known fact that this algorithm is more efficient than the much simpler Floyd-Warshall [6] algorithm only for sparse graphs. This contradicts the intuition that the APSP distance calculated in part 1 should be used to determine edge distances for the triangles. If these calculated edges were added to the work graph, it would no longer be sparse and there would be no use for Johnson's Algorithm.

Despite this, we now prove that the algorithm must use the APSP graph when finding triangles to apply tightenings to. We do this with a counterexample. Figure 3.5 shows an example of an EDG for an STNU. The left graph is the starting scenario for the MMV algorithm. The right graph shows the addition of constraints derived through the use of the Precedes Reduction. If the test for pseudo-controllability was run without adding derived shorter distances, the distances between A and D would be $AD : 30$ and $DA : -30$ and there would be no negative cycle detected.

However, if the shortest distance edges $DB : 20$ and $BD : -20$ was added to the graph another application of the Precedes Reduction would give $AD : 30$ and $DA : -80$. Now clearly the APSP calculation would find a negative cycle. This shows that all the APSP edges must be added to the graph in which tightenings are found and applied. It also shows that

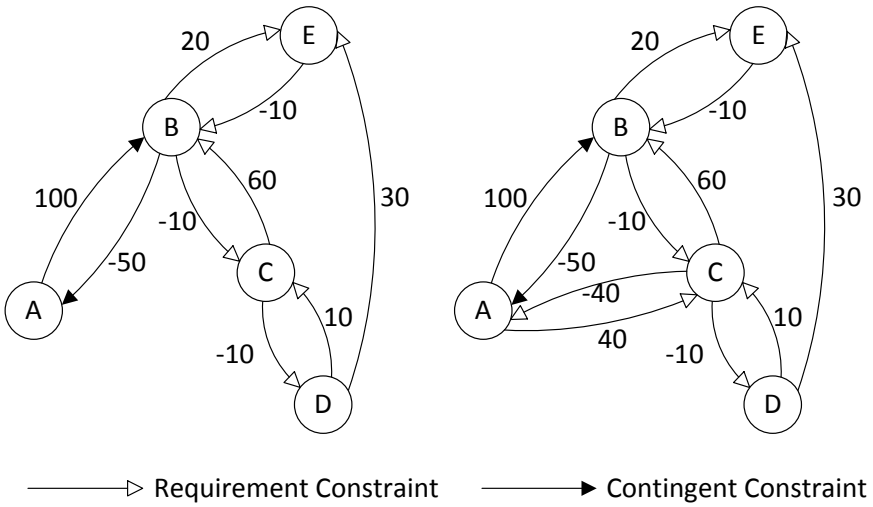

Figure 3.5: Why APSP edges must be added to the working graph.

Floyd-Warshall's algorithm is the algorithm of choice when testing pseudo-controllability.

There are two additional issues which need to be mentioned to successfully implement the algorithm. Tightening rules can be matched by regarding contingent constraints as requirement constraints. This is not so strange since a contingent constraint implies a weaker requirement constraint with the same bounds. Furthermore requirement constraints can be turned around so that $[a, b]$ becomes $[-b, -a]$ in the opposite direction, in order to match tightening rules.

A final note is that all wait constraints conditioned on the same node will have the same source, the source of the contingent constraint where the conditioning node is present. This can be seen on close inspection of the rules in Figure 3.3.

3.2 The FastIDC Algorithm

We now present the part of the FastIDC algorithm which handles additions and tightening of constraints. This part is called the BackPropagate-Tighten algorithm, however as it is the only part of FastIDC we will work with, we will refer to it also as FastIDC.

The Backpropagate-Tighten algorithm (Algorithm 8) presented in [32]

uses a combination of the tightening rules for MMV (see Figure 3.3) and the Dispatch Back-Propagation (DBP) algorithm for STNs (see Section 2.3). The derivation rules of MMV are translated to the Extended Distance Graph format (EDG, see Section 2.4.3) which is the working representation used by Backpropagate-Tighten. Figure 3.6 shows a direct translation of the MMV derivation rules into EDG form. Backpropagate-Tighten uses a subset of these derivation rules together with DBP rules to form its own set of derivation rules. Figure 3.7 shows the final set of rules used by Backpropagate-Tighten. There is another special derivation rule that were not listed among the original derivation rules, but instead mentioned in the text. It is only discussed in the first paper by Stedl [34]. The rule is called unconditional reduction and is a mixture of the unconditional and general reductions of MMV. The result is applied when unconditional reduction is matched, but the resulting edge is that of general reduction. We will come back to discussing this derivation rule in Chapter 4.

The derivation rules of Backpropagate-Tighten are applied recursively similarly to how DBP applies its rules. This makes Backpropagate-Tighten apply derivations in a specific order, instead of arbitrarily which is the case for MMV.

In the presentation of FastIDC there is no mention of how to handle contingent edges. Two correct possibilities exist:

1. Contingent edges are required to be added before adding any other edges having the same target node. This is the approach we have taken and will use throughout this thesis. From a planning perspective it is natural that an action is first decided upon before constraints are put on its end time.

2. After adding a contingent constraint, reprocess all constraints that connect to the target node of the contingent constraint. This is more cumbersome and would affect the readability of most algorithms presented in this thesis.

FastIDC Details. Being incremental, FastIDC assumes that at some point a dynamically controllable STNU was already constructed by FastIDC itself (as a start, the empty STNU is trivially DC). Now one or more requirement edges $e_1, \ldots, e_n$ have been added or tightened, together with zero or more contingent edges and zero or more new nodes, resulting in the graph G. FastIDC should then determine whether G is DC.

It can be seen in Algorithm 8 that FastIDC first adds the newly modified or added requirement edges to a queue, Q. The queue is sorted in order of

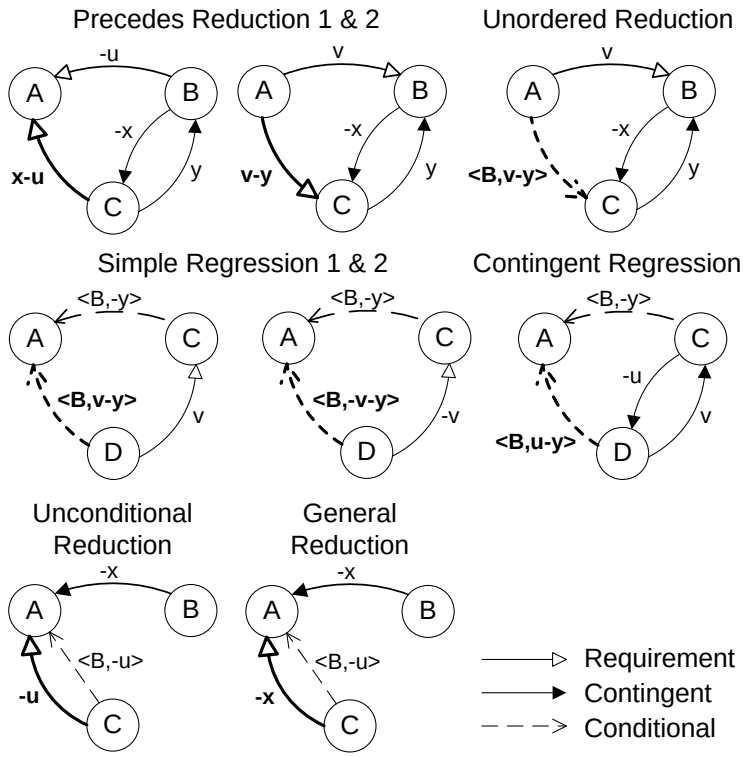

Figure 3.6: MMV derivations in EDG format.

decreasing distance to the *temporal reference* (*TR*), a node always executed before all other nodes at time zero. Therefore edges connecting to nodes closer to the "end" of the STNU will be dequeued before edges connecting to nodes closer to the "start". This will to some extent prevent duplication of effort by the algorithm, but is not essential for correctness or for understanding the derivation process.

In each iteration an edge e_i is dequeued from Q.

A positive loop (an edge of positive weight from a node to itself) represents a trivially satisfied constraint that can be skipped. A negative loop entails that a node must be executed before itself, which violates DC and is reported.

If e_i is not a loop, FastIDC determines whether one or more of the derivation rules in Figure 3.7 can be applied with e_i as focus. In the figure the topmost edge is always the focus edge and the leftmost is the derived edge.

For example, consider rule D1. This rule will be matched if e_i is a pos-

Algorithm 8: BackPropagate-Tighten [32]

```
function BackPropagate-Tighten(G, e_1, ..., e_n)
Q ← sort e_1, ..., e_n by distance to temporal reference
      (order important for efficiency, irrelevant for correctness)
for each modified edge e_i in ordered Q do
    if IS-POS-LOOP(e_i) then SKIP e_i
    if IS-NEG-LOOP(e_i) then return false
    for each rule (Figure 3.7) applicable with e_i as focus do
        if edge z_i in G is modified or created then
            if G is squeezed then return false
            if not BackPropagate-Tighten(G, z_i) then return false
        end
    end
end
return true
```

itive requirement edge, there is a negative contingent edge from its target B to some other node C, and there is a positive contingent edge from C to B. Then a new edge (the AC conditional edge) can be derived. This edge is only added to the EDG if it is strictly tighter than any existing edge between the same nodes.

More intuitively, D1 represents the situation where an action is started at C and ends at B, with an uncontrollable duration in the interval $[x, y]$. The focus edge AB represents the fact that B, the end of the action, must not occur more than v time units after A. This can be represented more explicitly with a conditional constraint AC labeled $\langle B, v - y\rangle$: If B has occurred (the action has ended), it is safe to execute A. If at most $v - y$ time units remain until C (equivalently, at least $y - v$ time units have passed *after* C), no more than v time units can remain until B occurs, so it is also safe to execute A.

Whenever a new edge is created or an existing edge is tightened, a check is done to see if the new edge *squeezes* the graph. This test is different from pseudo-controllability test that MMV uses to detect squeezes. FastIDC considers the graph squeezed if there is a local negative cycle or a contingent edge is squeezed. Since FastIDC works in the EDG there can be several edges of the five different types going in both directions between two nodes. This means that a contingent edge may even be squeezed by a conditional edge. When a new edge is added this means that it must be checked against all edges, in both directions, between the nodes it connects.

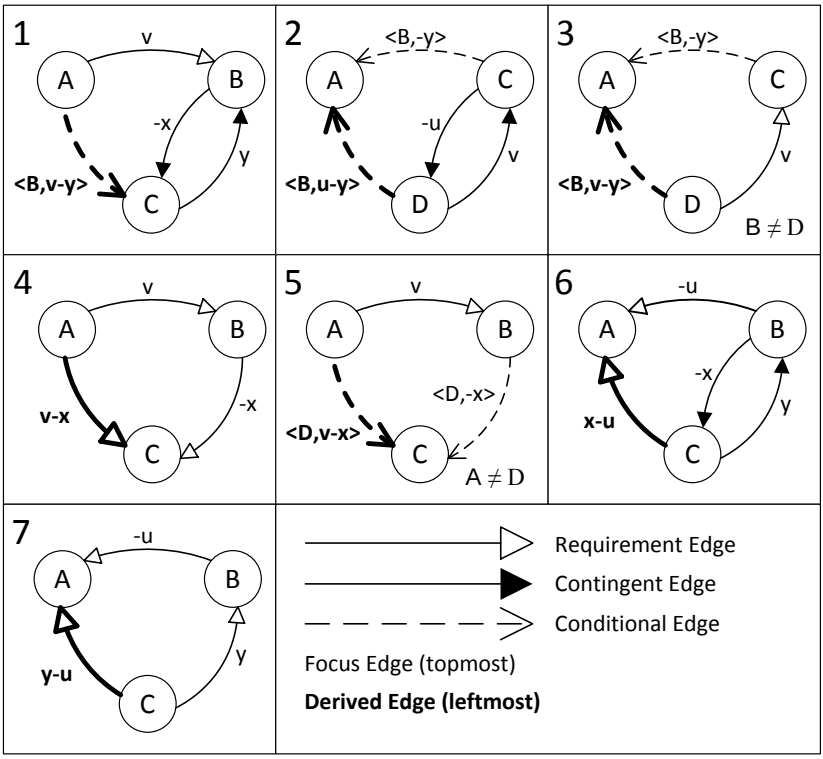

Figure 3.7: BackPropagate-Tighten Derivation Rules.

As an example, suppose FastIDC derives a requirement edge BA of weight w, for example $w = -12$, representing the fact that B must occur at least 12 time units after A. Suppose there is also a contingent edge BA of weight $w' > w$, for example $w' = -10$, representing the fact that an action started at A and ending at B may in fact take as little as 10 time units to execute. Then there are situations where nature may violate the requirement edge constraint, and the STNU is not DC. Scenarios like this example are detected in the local squeeze test.

If the tests are passed and the derived edge is tighter than any existing edges in the same position, FastIDC is called recursively to take care of any derivations caused by this new edge. Although perhaps not easy to see at a first glance, all derivations lead to new edges that are closer to the temporal reference. Derivations therefore have a direction and will eventually stop. We will discuss this fact in detail in Chapter 5. When no more derivations can be done the algorithm returns true to testify that the STNU is DC. If

FastIDC returns true after processing an EDG this EDG can be dispatched directly by the dispatcher in algorithm 5.

3.2.1 Comparing MMV and FastIDC

In Chapter 5 we will relate the work performed by MMV to that of FastIDC. Here we point out the three main differences between the two algorithms.

1. **Representation.** FastIDC does not work in the standard STNU representation but uses the EDG graph (see Section 2.4.3) instead.
2. **Derivation rules.** Partly due to the new representation, FastIDC uses different derivation rules.
3. **Traversal order.** FastIDC uses a significantly different graph traversal order. MMV traverses a graph iteratively, and in each iteration, it considers *all* "triangles" in a graph in arbitrary order. FastIDC, in contrast, uses the concept of *focus edges*. A focus edge is an edge that was tightened and may lead to other constraints being tightened. FastIDC only applies derivation rules to focus edges. If this leads to new tightened edges it will recursively continue to apply the derivation rules until quiescence. Intuitively, this guarantees that all possible consequences of any tightening are covered by the algorithm.

3.3 Conclusion

In this chapter we presented the original versions of the MMV and Back-Propagate-Tighten algorithms that the thesis results are based on. We also presented a proof that MMV must use all edges from the APSP graph which was very unclear from previous publications of the algorithm.

Chapter 4

FastIDC Analysis and Correction

During early integration of the FastIDC algorithm with a planner we discovered that the algorithm was not correct. In some cases it would classify an STNU as DC when it was not. In this chapter we first give an example to show that FastIDC is unsound. We then analyze the algorithm, pinpoint the cause, and show how the algorithm can be modified to correctly detect uncontrollable networks.

4.1 A Misclassified STNU

We now show an STNU that BackPropagate-Tighten fails to classify as non-DC.

Figure 4.1 shows an example network where U occurs uncontrollably between 5 and 50 time units after A. The network is dispatchable and DC.

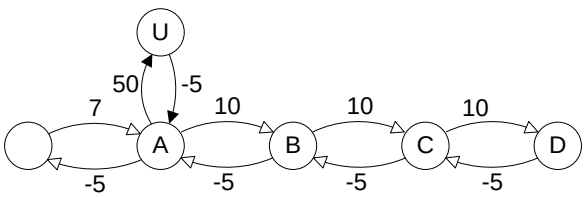

Figure 4.1: Original graph, dispatchable and dynamically controllable.

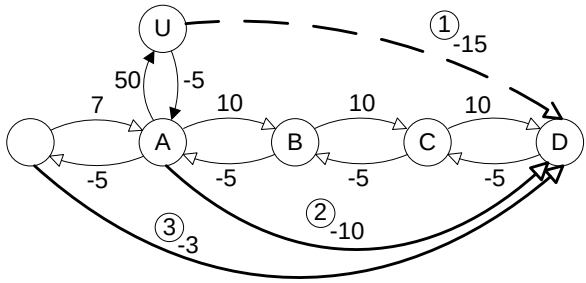

Figure 4.2: After BackPropagate-Tighten is executed.

Figure 4.2 shows the same network after requirement edge ① has been added by a planner, stating that U must occur at least 15 time units after D, and BackPropagate-Tighten has been executed. Rule 6 is matched with UD as its focus, resulting in a new edge ②: Nature *could* decide there will be as few as 5 time units between A and U, so there must be at least 10 time units between D and A. The only remaining match is for rule 7, resulting in edge ③.

No negative self-loop is generated. Will the algorithm consider G to be squeezed? In MMV this was tested globally using an all-pairs shortest path (APSP) algorithm. This would be extremely inefficient in an incremental algorithm, and indeed Stedl and Williams [33] and Shah et al. [32] state that APSP algorithms are not used. Then a local check for squeezing must be used, and there are no edges that locally imply that the bounds on the distance between A and U are squeezed. Therefore, BPT will return true – but the graph is not DC: The path UDCBA shows U must be at least 30 after A, while the edge UA shows U may be observed as little as 5 time units after A, violating the DC requirement.

4.2 Problem Analysis

To determine *why* BackPropagate-Tighten can miss the fact that a graph is not DC, we take a closer look at the steps indicated in Figure 4.2. It is clear that the graph was initially DC and dispatchable. When an edge ① was added, the distance graph remained globally consistent. However, since this edge involved a contingent event (U), we could derive an additional edge ② that would not have been entailed in an ordinary STN. This resulted in a *negative cycle* and an inconsistent graph.

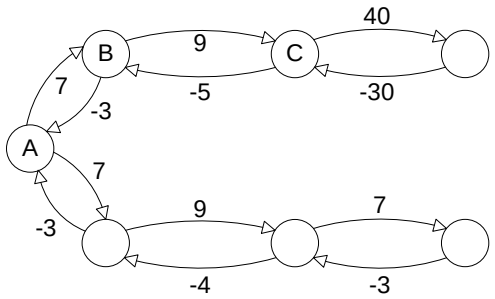

Figure 4.3: Initial consistent and dispatchable STN.

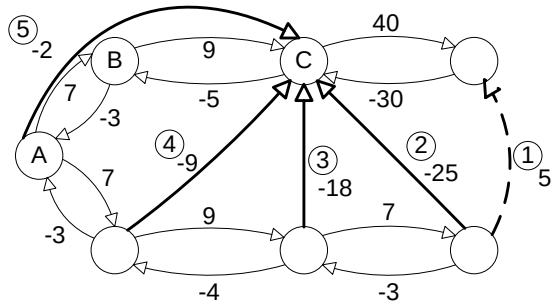

Figure 4.4: Graph where an STN inconsistency is missed.

BackPropagate-Tighten should detect such cycles. In fact, Shah et al. [32] explicitly states that if we have a dispatchable distance graph for an ordinary STN, then tighten or add an edge, and recursively apply *only rules 4 and 7* and check for negative self-cycles, this "*will either expose a direct inconsistency or result in a dispatchable graph*".

Below we will call the recursive application of rules 4 and 7 Incremental Dispatchability (ID), as it was originally called in Stedl [34]. To demonstrate more clearly how ID works without reference to STNUs, we turn to the pure STN in Figure 4.3, which is dispatchable.

Suppose the dashed constraint ① is added as shown in Figure 4.4, leading to a negative cycle and a non-dispatchable graph. Incremental Dispatchability will use rules 4 and 7 to derive edges ②–⑤, after which no further edges can be derived except for positive self-loops, which we omit as they neither indicate inconsistency nor lead to the generation of additional edges. As ID fails to find a negative self-loop it considers the re-

sulting graph consistent and dispatchable, which it is not. Why does this happen, and what can we do about it while retaining the gained efficiency of BackPropagate-Tighten as compared to full APSP calculations?

4.2.1 Reasons for Failure

To see why ID fails we compare it to MMV, which detects negative cycles by running an APSP algorithm. This is equivalent to repeatedly taking *all* edge pairs $A \xrightarrow{x} B \xrightarrow{y} C$ and deriving/tightening edges $A \xrightarrow{x+y} C$. In ID, rules 4 and 7 only consider edge pairs where $x > 0$ and $y \leq 0$, which is part of the reason for its efficiency.

Lemma STN-DBP provided by Shah et al. [32] proves that these derivations are *valid*. The motivation for why they should also be *sufficient* is quite intuitive and based on the fact that as long as a dispatcher ensures that each scheduling decision it makes is consistent with the past, it *will* also be possible to consistently schedule any future events. In other words, any constraint that could possibly be inferred from the occurrence of future events has already been explicitly applied to the current event through new edges derived when the graph was made dispatchable.

As ID requires that the graph was dispatchable before the tightening or addition, it is argued that ID also only has to consider consistency with past events: "*To maintain the dispatchability of the STN when a constraint is tightened by a fast re-planner, we only need to make the modified constraint consistent with past scheduling decisions, since during execution, the bounds on events are only influenced by preceding events*" [32]. Thus, when a positive constraint AB is tightened, rule 4 only considers how this will affect nodes C forced to be in the past from B, and similarly for rule 7.

This reasoning presumes that there is a well-defined past at each node, towards which the recursion can proceed. This is true when a graph *known* to be dispatchable is executed, but now we are verifying *whether* a change preserves dispatchability. In Figure 4.4 we violated dispatchability and could deduce both that C must be before A and vice versa: The STN is inconsistent, and so is the concept of "past".

Since ID determines that A must be before B, and B before C, it does not derive a new edge from AB and BC. The reasoning is again that at execution time, A must occur before B, and *then* the dispatcher will propagate the resulting constraints towards C in the future. This holds for all combinations of negative edges, so the negative cycle is never shortened to a negative self-loop and is therefore missed.

4.2.2 Resolving the Problem

One possible means of resolving this problem would be to fall back on detecting negative cycles using for example incremental APSP calculations or incremental directed path consistency [4] algorithms. But this would lead to the same inefficiency that back-propagation was intended to avoid. For the best possible performance, we would prefer to determine whether we can benefit from the work that is already done when new edges are derived by ID and BackPropagate-Tighten.

We therefore observe Figure 4.4 more closely and find that it contains not only a negative cycle but a cycle consisting entirely of negative edges, ACBA. It turns out that this will always be the case when ID fails to discover an inconsistency.

In the following we assume that any path is simple, i.e. does not contain a repeated node, and that any cycle is simple, i.e. contains only one path from each node to itself.

Lemma 3 (Nilsson et al. [24])**.** *Consider all paths of a given weight n between two nodes N and N' in a distance graph that was constructed incrementally by ID. The shortest of these paths, in terms of the number of edges, must have one of the following forms:*

1. *It contains only positive edges.*
2. *It consists of at least one negative edge followed by zero or more positive edges.*

Proof. If a path with the smallest number of edges does not have this form, it must at some point contain a positive edge followed by a negative edge: $N \cdots A \xrightarrow{+} B \xrightarrow{-} C \cdots N'$. Either when the edge AB was added/tightened or when the edge BC was added/tightened, ID would have used rule 4 or 7 to derive a new edge $A \to C$ whose weight was the sum of the weights of $A \to B$ and $B \to C$. There would then exist a path between N and N' with the same weight but fewer edges, leading to a contradiction. □

Theorem 2 (Nilsson et al. [24])**.** *Let G be a consistent and dispatchable distance graph constructed using ID. Assume that the edge $A \xrightarrow{f} B$ is added or tightened in G and that the corresponding STN is then inconsistent. Then after applying ID, there will be a cycle in the distance graph consisting of only negative edges.*

Proof. By induction. Suppose that after adding or tightening an edge in G but before applying ID, G is inconsistent. Then G has at least one negative

cycle according to Dechter et al. [9]. Let C_k be a negative cycle in G with the smallest number of edges. Let $k \geq 1$ be the number of edges in C_k.

Basis: If $k = 1$, there is already a cycle of only one negative edge. This cycle will remain after applying ID, because ID never removes edges and never increases edge weights.

Induction assumption: The theorem holds for $k - 1$.

Induction step: Does the theorem hold for k, where $k > 1$?

First, if all edges in C_k are negative, then they will remain negative after ID and the theorem holds. Otherwise, at least one edge in the cycle is positive, but there must also be at least one negative edge (else C_k could not be negative).

We know C_k consists of the newly tightened or added edge $A \xrightarrow{f} B$ together with some non-empty path from B to A. If there exist other paths from B to A of the same weight, they cannot have fewer edges – otherwise there would have been a shorter negative cycle than C_k, violating the assumption. Thus, the path from B to A included in C_k has the fewest edges among all paths from B to A of the same weight, and Lemma 3 is applicable.

Suppose that the new value of f is negative. As the entire cycle did not consist of negative edges, there must be at least one positive edge on the path from B to A. This together with the lemma shows that there must be a positive edge $X \rightarrow A$ at the end of that path, for some node X. Since the edge $A \rightarrow B$ was altered, ID will apply rule 7 to derive an edge $X \rightarrow B$, yielding a cycle with the same negative weight but with $k - 1$ edges. By the induction assumption, ID will then reduce this cycle to a negative-edge-only cycle.

Suppose instead that the new value of f is positive. As the cycle was negative, the path from B to A must have negative weight, so case 2 of the lemma must hold and the path must begin with a negative edge $B \rightarrow Y$. Since the edge $A \rightarrow B$ was altered, ID will apply rule 4 to derive a new edge $A \rightarrow Y$, again yielding a cycle with the same negative weight but with $k - 1$ edges. By the induction assumption, ID will then reduce this cycle to a negative-edge-only cycle. □

Intuitively this can be seen from the example in Figure 4.5. The example contains a negative weight cycle which includes a positive edge. By the derivation rules the positive edge with weight d will react with the negative $-e$ edge and in doing so creating a "short-cut" with the sum of their weights as weight. This can continue until the derived edge is a negative "short-cut" which we know it will eventually become since the negative

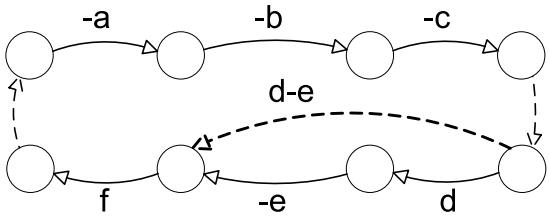

Figure 4.5: A negative weight cycle with at least one positive edge. ID will derive a negative cycle with the same weight and fewer edges.

weight sum of the cycle outweighs the positive weights. If there are several positive edges in different positions along the cycle the same reasoning can be applied to them all.
As shown in Figure 4.4, tightening an edge can indeed yield a cycle of *multiple* negative edges, which is not detected by ID. This is still problematic, but we have now verified that it suffices to detect negative-edge-only cycles rather than arbitrary negative cycles also containing positive edges. To detect these we do not need to take edge weights into account.

We therefore incrementally build an *unweighted* Cycle Checking (CC) graph containing the same nodes as the ID distance graph and with a directed edge exactly where the distance graph has a negative edge. Since edge weights can only be decreased and not increased, edges in the CC graph will never be removed. Also, tightening an already negative edge does not change the CC graph.

We then find cycles in the CC graph using an efficient incremental topological ordering algorithm which does not need to take edge weights into account. Since ID generates many negative edges for propagating time bounds during dispatch, an algorithm for dense graphs appears best suited. One such algorithm has a run-time of $O(n^2 \log n)$ for incrementally cycle-checking a graph with n nodes and a maximum of $O(n^2)$ edges [2]. This is dominated by the run-time of BackPropagate-Tighten, which was empirically shown [33] to be around $O(n^3)$ in practice and in worst case $\Omega(n^4)$ which will be shown in Chapter 6.

4.3 The Sound FastIDC Algorithm

Before we present the sound version of BackPropagate-Tighten, we present the full set of derivation rules as promised in Chapter 3. The derivation

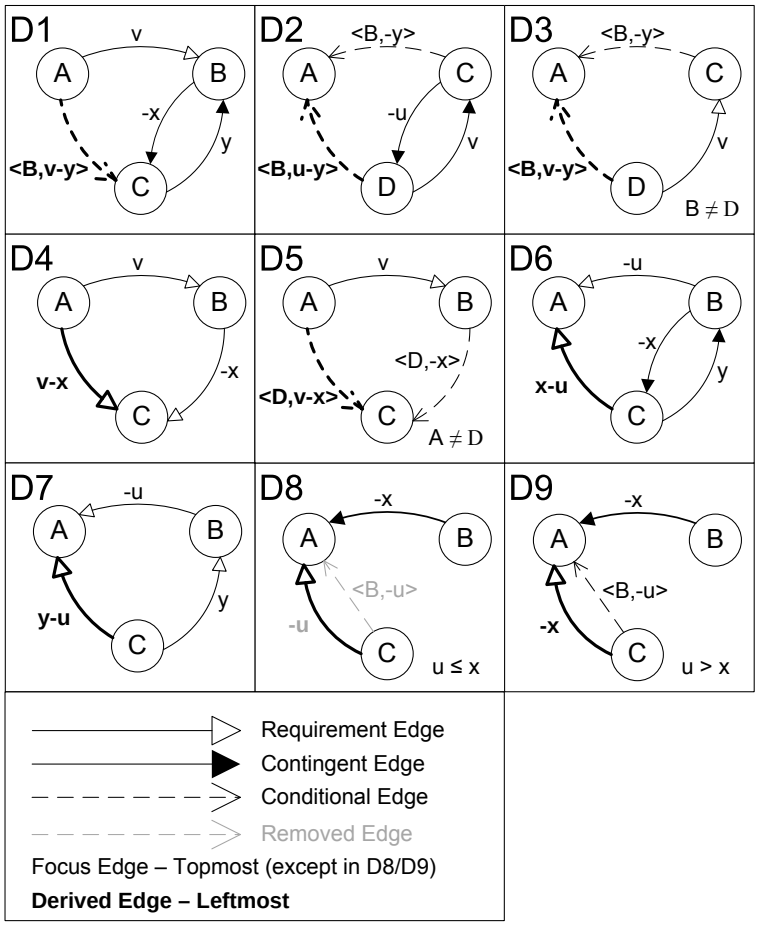

Figure 4.6: FastIDC derivation rules D1-D9.

rules are shown in Figure 4.6 where we have added D8 and D9 which correspond to the identical reduction rules presented by Morris et al. [21] but are here shown in EDG form.

4.3.1 General and Unordered Reductions

In the original FastIDC presentation the use of unconditional/general reductions were confounded. As we will show here, they are both needed in the same situations as for the original MMV algorithm.

First, Figure 4.7 shows what happens if FastIDC (or MMV) would omit

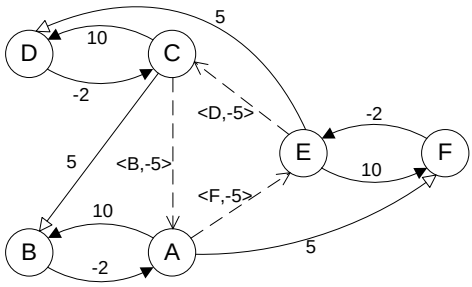

Figure 4.7: Why general reduction is needed.

general reduction. Suppose the graph in the figure is built incrementally. When adding the *CB*, *ED* and *AF* edges, the conditional edges *CA*, *EC* and *AE* will be derived. FastIDC would then terminate with a positive verification of DC. However, the triangle of conditional edges means that all involved nodes (*A/C/E*) need to be executed after each other, an inconsistency which is not discovered. The edge derived by general reduction is entailed by the conditional edge and resolves this problem.

Regarding Unconditional Reduction, suppose the *CB* edge in Figure 4.7 had weight 9, giving the *CA* edge weight -1. Now C needs to execute 1 time unit after A or when B is observed. Since B cannot be observed until at least 2 time units after A, the conditional part of the constraint is of no consequence and a requirement edge of weight -1 can be inferred by Unconditional Reduction. Note that this situation does not trigger general reduction (D9). Therefore, the conditional edge would not cause a negative cycle of requirement edges if it were not for the Unconditional Reduction. We see that in order for the algorithm to work correctly in this situation, Unconditional Reduction is needed.

4.3.2 The Sound Algorithm

Now that we have seen the full set of derivation rules we are ready to present the sound version of the FastIDC algorithm, see Algorithm 9. It is similar to the original BackPropagate-Tighten but additionally contains pseudo-code for managing and making use of a CCGraph for cycle checking (see below). In order to give a complete description of the algorithm we repeat the identical parts of the description for BackPropagate-Tighten here.

Being incremental, FastIDC assumes that at some point a dynamically

Algorithm 9: FastIDC – sound version [24]

```
function FAST-IDC(G, e_1, ..., e_n)
Q ← sort e_1, ..., e_n by distance to temporal reference
      (order important for efficiency, irrelevant for correctness)
for each modified edge e_i in ordered Q do
    if IS-POS-LOOP(e_i) then SKIP e_i
    if IS-NEG-LOOP(e_i) then return false
    for each rule (Figure 4.6) applicable with e_i as focus do
        if edge z_i in G is modified or created then
            Update CCGraph
            if Negative cycle created in CCGraph then return false
            if G is squeezed then return false
            if not FAST-IDC(G, z_i) then
            return false
        end
    end
end
return true
```

controllable STNU was already constructed (for example, the empty STNU is trivially DC). Now one or more requirement edges $e_1, \ldots, e_n$ have been added or tightened, together with zero or more contingent edges and zero or more new nodes, resulting in the graph G. FastIDC should then determine whether G is DC.

The algorithm works in the EDG of the STNU. First it adds the newly modified or added requirement edges to a queue, Q (a contingent edge must be added before any other constraint is added to its target node and is then handled implicitly through requirement edges). The queue is sorted in order of decreasing distance to the *temporal reference* (*TR*), a node always executed before all other nodes at time zero. Therefore edges connecting to nodes closer to the "end" of the STNU will be dequeued before edges connecting to nodes closer to the "start". This will to some extent prevent duplication of effort by the algorithm, but is not essential for correctness or for understanding the derivation process.

In each iteration an edge e_i is dequeued from Q.

A positive loop (an edge of positive weight from a node to itself) represents a trivially satisfied constraint that can be skipped. A negative loop entails that a node must be executed before itself, which violates DC and is reported.

If e_i is not a loop, FastIDC determines whether one or more of the derivation rules in Figure 4.6 can be applied with e_i as focus. The topmost edge in the figure is the focus in all rules except D8 and D9, where the focus is the conditional edge $\langle B, -u \rangle$. Note that rule D8 is special: The derived requirement edge represents a stronger constraint than the conditional focus edge, so the conditional edge is removed.

An example of how D1 is applied can be found in section 3.2 on page 57.

Whenever a new edge is created, the corrected FastIDC tests whether a cycle containing only negative edges is generated. The test is performed by keeping the nodes in an incrementally updated topological order relative to negative edges. The unlabeled graph which is used for keeping the topological order is called the *CCGraph*. It contains the same nodes as the EDG and has an edge between two nodes if and only if there is a negative edge between them in the EDG. See [24] for further information.

After this a check is done to see if the new edge *squeezes* a contingent constraint. Suppose FastIDC derives a requirement edge BA of weight w, for example $w = -12$, representing the fact that B must occur at least 12 time units after A. Suppose there is also a contingent edge BA of weight $w' > w$, for example $w' = -10$, representing the fact that an action started at A and ending at B may in fact take as little as 10 time units to execute. Then there are situations where nature may violate the requirement edge constraint, and the STNU is not DC. The squeeze test also checks for local consistency in the same way as the original FastIDC algorithm does. This means that it tests all different edges coexisting between the involved nodes to see if there is any inconsistency.

If the tests are passed and the edge is tighter than any existing edges in the same position, FastIDC is called recursively to take care of any derivations caused by this new edge. Although perhaps not easy to see at a first glance, all derivations lead to new edges that are closer to the temporal reference. Derivations therefore have a direction and will eventually stop. When no more derivations can be done the algorithm returns true to testify that the STNU is DC. If FastIDC returns true after processing an EDG this EDG can be dispatched directly by the dispatcher in Algorithm 5.

4.4 FastIDC Correctness

Now that the correct pseudo-code and complete set of derivation rules have been shown we give a proof that the sound algorithm is in fact sound. In

Chapter 5 we perform an analysis of the similarities and differences between MMV and FastIDC. There we also give the MMV tightening rules in EDG form. It might help to look at the first section in this chapter when reading this quite long proof.

Theorem 3 (**FastIDC Correctness [25]**). *Given a dynamically controllable STNU and a set new constraints that are added to the STNU, the sound version of FastIDC, shown in Algorithm 9, correctly verifies dynamic controllability of the resulting STNU.*

Proof. First we prove that FastIDC does not derive stronger constraints than MMV and then that it derives enough to be correct.

First: FastIDC cannot derive stronger constraints than MMV does. This follows since MMV applies its derivations and shortest path calculations to *all* triangles of nodes until quiescence, and thus the recursive traversal performed by FastIDC clearly cannot process a focus edge that MMV does not process. Further, every derivation *rule* applied by FastIDC is also used by MMV: D4 and D7 are implicitly performed through APSP calculations, while the other rules are directly applied.

Second: FastIDC derives enough constraints to be able to classify STNUs identically to how MMV would classify them. There are two cases of classification for an STNU: Non-DC or DC.

Case 1: FastIDC indicates that the STNU is not DC. Then applying the derivation rules has resulted in the detection of a negative cycle or a local squeeze. The constraints generated by MMV would be at least as strong and would therefore also result in a negative cycle or local squeeze. The pseudo-controllability test used by MMV would detect this, signaling that the STNU is not DC. Since MMV is correct, FastIDC was also correct in this case.

Case 2: FastIDC indicates that the STNU is DC. We will show that it is dispatchable by the dispatcher in algorithm 5, which in turn entails that there must exist a dynamic execution strategy (the one applied by the dispatcher). Thus, the STNU is DC and FastIDC is correct.
Proving this requires some knowledge of the dispatcher (algorithm 5). When the dispatcher executes or observes the execution of a node, execution bounds are propagated to all neighboring nodes. Upper bounds are propagated along positive edges, while lower bounds are propagated "backwards" along negative edges, which includes all conditional edges.

To unify the cases in the following discussion we assume that when an uncontrollable event is observed, an execution window for the event is propagated to it containing only the observed time. This approach lets

us compare propagated bounds from both controllable and uncontrollable nodes.

Now, let G be a DC STNU constructed through repeated applications of FastIDC. Add one or more edges $e_1, \ldots, e_n$, and assume that FastIDC $(G, e_1, \ldots, e_n)$ classifies G as DC. We then know that:

1. It does not contain a cycle consisting only of negative requirement edges, as this would have been detected by the CCGraph.

2. It does not contain a cycle consisting only of negative requirement edges and conditional edges, since general reduction (D9) would have created a cycle of negative requirement edges from this.

Therefore it is not possible for the dispatcher to end up in a deadlock where no nodes are executable. Theoretically there could, however, be one or more combined outcomes of the uncontrollable events for which execution will fail because the propagation of execution bounds results in an empty execution window for some event.

Assume that this happens: At least one node receives an empty execution window. Let X be the first node for which this happens during the propagation procedure. The execution window was initially $[0, \infty]$, and must have been intersected with at least two propagated execution windows that do not overlap, so that the upper bound of X is below its lower bound. The upper bound and lower bound must then be caused by propagation from distinct nodes. Thus we have a triangle AXB in the EDG where an incoming edge AX has constrained the upper bound of X and an outgoing edge XB has constrained the lower bound of X.

We will now consider all possible edge types for these incoming and outgoing edges and show that in each case, FastIDC would in fact have derived an additional constraint ensuring that the execution window for X could not have become empty. First, suppose the upper bound for X was propagated from a contingent constraint AX. The lower bound might then have originated in:

1. A negative requirement edge XB. Then rule D6 would have generated a constraint AB constraining the relative timing between the execution of A and that of B. This constraint would have prevented the intervals propagated from A and B to X from having an empty intersection.

2. A conditional edge XB, in which case X would be "protected" in a similar way by a constraint generated by D2.

Second, suppose that the upper bound for X was propagated from a positive requirement edge AX. The lower bound might have originated in:

1. A negative requirement edge XB: X protected by D4 or D7.
2. A conditional edge XB: X protected by D3 or D5.
3. A contingent constraint XB: X protected by D1.

Note that we treat contingent edges as a whole constraint since they collapse the interval to a point and therefore it does not matter if the positive or negative edge is considered as propagating the time value.

Thus, for X to receive an empty execution window, A or B (or both) must also have received an empty execution window from the propagation of AB together with the other constraints in the EDG. Furthermore, since they propagated constraints to X, they must have been dispatched before X. This contradicts the assumption that X was the first node to receive an empty execution window. Since no additional assumptions were made about X, no node can receive an empty time window during dispatch. The dispatcher together with the processed STNU therefore constitute a dynamic execution strategy, and the STNU is DC. □

4.5 Conclusion

In this chapter we proved that the FastIDC algorithm (see Chapter 3) was unsound. We also showed that it is possible to correct the flaw in the original algorithm without impacting the run-time complexity. We provided a corrected algorithm as well as additional derivation rules which we proved are needed. The final contribution in this chapter was a full proof that FastIDC as given here is correct.

Chapter 5

A Tighter Complexity Result for the MMV Algorithm

In this chapter we re-analyze MMV and prove that with a small modification it is in fact $O(n^4)$ – the algorithm merely needs to stop earlier. The intuition behind the analysis is that not all of MMV's derivations and tightenings are necessary: Only a certain *core* of derivations actually matters for verifying dynamic controllability, and when the STNU is DC, this core is free of cyclic derivations. This can be exploited through a small change to MMV. Stopping at the right time also preserves another aspect of MMV: the result is *dispatchable*, unlike the result of Morris' algorithm.

5.1 A Deeper Comparison between FastIDC and MMV

The property of dynamic controllability is "monotonic" in the sense that if an STNU is not DC, it can never be made DC by further adding or tightening constraints. Therefore, the *non-incremental* verification performed by MMV is equivalent to starting with an empty STNU (which is trivially DC) and *incrementally* adding one edge at a time, verifying with MMV at each step that the STNU remains DC. This procedure allows a comparison between MMV and FastIDC even though MMV only work with full STNUs and FastIDC only work with incrementally built STNUs.

To compare the derivation rules used by MMV to those of FastIDC, we first need a translation into EDG format. This is shown in Figure 5.1 where

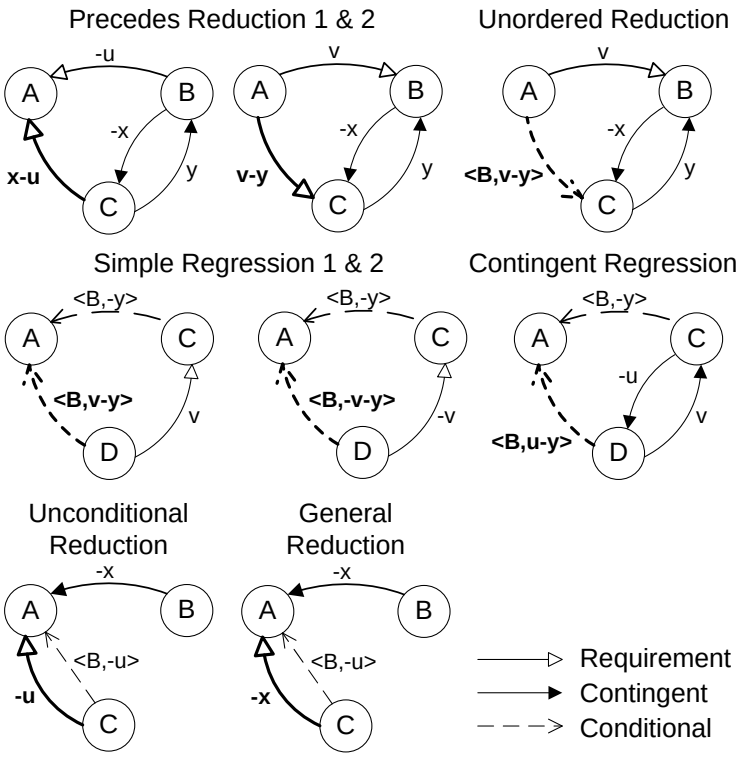

Figure 5.1: Classical derivations in EDG format.

similar to before the bold edges are derived. *Precedes Reduction (PR)* is split in two since it adds two edges. *Simple Regression (SR)* is also split in two, one version regressing over a positive edge and one regressing over a negative edge. All variables used as weights are considered positive, i.e., $-u$ is a negative number (with Unconditional Reduction as an exception since it also matches when $-u$ is positive). The additional requirements from Figure 3.3 still apply but are omitted for clarity. Most are encoded by the edge types – for instance in unordered reduction, only a positive requirement edge can match the rule, making the $v > 0$ requirement implicit. We now see the following similarities between figures 4.6 and 5.1:

- Precedes Reduction 1 (PR1) is identical to D6.
- Unordered Reduction is equivalent to D1. However without the extra requirement ($u \geq 0$) used by MMV to distinguish between applying PR2 and unordered reduction, FastIDC will always apply Unordered

Reduction, even when MMV instead would apply PR2. It can be shown that if the situation calls for an application of PR2, FastIDC derives the same edge as MMV through conversion of the conditional edge resulting from D1 into a requirement edge (via Unconditional Reduction, D8). If the application of PR2 directly leads to non-DC detection, FastIDC also detects this directly. So PR and Unordered Reduction are handled by D1, D6 and D8 together.

- Simple Regression 1 is equivalent to D3 and D5. The only difference between D3 and D5 is which edge is regarded as focus.
- Contingent Regression is identical to D2.
- Unconditional Reduction is identical to D8.
- General Reduction is identical to D9.

Thus, the only significant differences are:

- FastIDC derivations has no counterpart to Simple Regression 2.
- D4 and D7 have no counterpart rules in MMV. These derive shortest path distances towards earlier nodes in the STNU. This derivation is present and handled by the APSP calculation in MMV.

We conclude that MMV and FastIDC derivations, with the exception of SR2, are identical. They are however applied in a different order. MMV uses unordered triangle selection and global APSP calculations whereas FastIDC only follow derivation stemming from focus edges. We also want to remind that MMV is a full DC verification algorithm whereas FastIDC is incremental, but by building the full STNU incrementally their work and results can be directly compared.

If we examine the differences closer it can be seen that SR2 is not needed, not even by MMV. Figure 5.2 shows the situation where a conditional edge CA is regressed over an incoming negative requirement edge DC. Adding a constraint DA to "bridge" two consecutive negative edges is always redundant both for execution and for DC verification. From an execution perspective this is easily seen since C is always executed before D which ensures that the chain of constraints is respected without the addition of DA. From a verification perspective this can be seen since the derived constraint is in fact weaker than the two original constraints. If B is executed before C the DA constraint "forgets" about the $-v$ constraint which must still be fulfilled. So the original two constraints are not only sufficient to

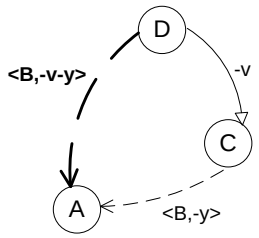

Figure 5.2: Simple regression when the edge is negative.

Table 5.1: The derived edges compared to the focus edges.

Rule	**Effect**
D1	The target of the derived edge is an earlier node.
D2,D6	The source of the derived edge is an earlier node.
D3,D7	The source of the derived edge is an earlier or unordered node.
D4,D5	The target of the derived edge is an earlier node.
D8,D9	The derived edge connects the same nodes.

guarantee the DA constraint: They are tighter and so the DA constraint can be skipped. We will come back to the implications of leaving SR2 out in the final section (Section 5.4).

5.2 Focus Propagation in FastIDC Derivations

If we apply rules D1–D9 in Figure 4.6, every derived edge has a uniquely defined "parent": The focus edge of the derivation rule. Unless this edge was already present in the original graph, it (recursively) also has a parent. This leads to the following definition.

Definition 12 (Derived Chain, Nilsson et al. [25]).
Edges that are derived through Figure 4.6 derivations are part of a ***derived chain****, where the parent of each edge is the focus edge used to derive it.*

We observe the following:

- A contingent constraint orders the nodes it constrains. In EDG form we see this by the fact that the target of a negative contingent edge is always executed before its source.
- Either D8 or D9 is applicable to any conditional edge. Thus there will always be an order between its nodes set by the negative requirement

edge from D8/D9: The target node of a conditional constraint is always executed before its source.

This leads directly to the facts in Table 5.1. Here, node n_1 is considered *earlier* than n_2 if n_1 must be executed before n_2 in every dynamic execution strategy and for all duration outcomes. Similarly, node n_1 is considered *unordered* relative to n_2 if their order can differ depending on strategy or outcome.

We now consider the structure of derived chains in DC STNUs. The focus will be on the direction and weight of each derived edge, ignoring whether edges are negative, positive, requirement or conditional (but still keeping track of contingent edges).

Lemma 4 (Nilsson et al. [25])**.** *Suppose all rules in Figure 4.6 are applied to the EDG of a dynamically controllable STNU until no more rules are applicable. Then, all derived chains are* ***acyclic****: No derivation rule has generated an edge having the same source and target as an ancestor of its parent edge along the current chain.*

Proof. Note that in the definition of acyclic, we allow "cycles" of length 1. These can only be created by applications of D8–D9 in a DC STNU.

For D1–D7, each derived edge shares one node with its parent focus edge, but has another source or target. We can then track how the source and target of the focus edge changes through the chain.

Table 5.1 shows that only derivation rules D1, D4 or D5 result in a different *target* for the derived edge compared to the focus edge. The new target has always "moved" along a negative edge, so it must be executed earlier than the target of the focus edge. Since the STNU is DC, its associated STN cannot have negative cycles. Thus, if the target changes along a chain, it cannot "cycle back" to a previously visited target.

Rules D2, D3, D6 and D7 result in a different *source* for the derived edge. This source may be earlier *or* later than the source of the focus edge, so these rules can be applied in a sequence where the source of the focus edge "leaves" a node n and eventually "returns". Suppose that this happens and the target n' has not changed. This must occur through applications of rules D2, D3, and/or D6–D9. No such derivation step decreases the weight of the focus edge. Therefore, when the source returns to n, the new edge to be derived between n and n' cannot be tighter than the one that already exists. No new edge is actually derived. Thus, if the source changes along a chain, it cannot "cycle back" to a previously visited source. □

This fact together with the previous lemma limits the length of a derived chain to $2n^2$ since we have at most n^2 distinct ordered source/target pairs

and can at most have one application of D8/D9 in-between source/target movements. The use of chains to reach an upper bound on iterations is inspired by Morris and Muscettola [20] where an upper bound of $O(n^5)$ is reached for the MM algorithm.

Note that FastIDC derivations together with local consistency checks and global cycle detection is sufficient to guarantee that all implicit constraints represented by a chain of negative edges are respected, or non-DC is reported. There is no need to add these implicit constraints but the next proof will make use of the fact that they exist.

Some derivations carried out by FastIDC can be proven not to affect the DC verification process, and hence we would like to avoid doing these. These can both be derivations of weaker constraints and constraints that are implicitly checked even if they are not explicitly present in the EDG. In order to single out the needed derivations we define *critical chains*.

Definition 13 (Critical Chain, Nilsson et al. [25]).
*A **critical chain** is a derived chain in which all derivations are needed to correctly classify the STNU. If any derivation in the chain was missing, a non-DC STNU might be misclassified as DC.*

Given a focus edge, one or more derivations may be applicable. Those that would extend the current critical chain into a non-critical one can be skipped without affecting classification. We therefore identify some criteria that are satisfied in all critical chains.

Lemma 5 (Nilsson et al. [25])**.** *Given a DC STNU:*

1. *A D1 derivation for a specific contingent constraint C can only be part of a critical chain once.*
2. *At most one derivation of type D2 and D6 involving a specific contingent constraint C can be part of a critical chain.*

Proof sketch: Part 1 is shown as in the proof of Lemma 4: The target cannot come back for another D1 application to the same contingent node.

We use Figure 5.3 to illustrate the situation when D2 or D6 is applied over the contingent *ab* constraint. The rightmost part of this figure is an arbitrary triangle *abc* where one of the rules is applicable, while the leftmost part is motivated by the proof below.

In the following we do not care if the edges are conditional or requirement: Only the weights of the derived edges are important. We follow

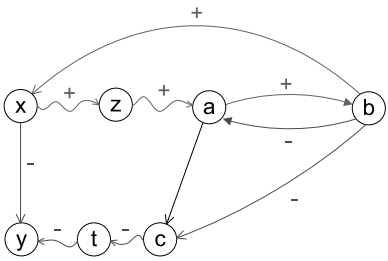

Figure 5.3: Situation where D2 or D6 is applied.

a critical chain and see how the source and target change as we continually derive new edges. Applying D2 or D6 gives a new edge ac where the source changes from b to a. We now investigate how derivations can move the source back to b and show that all derivations using the edge which resulted from moving the source back to b are redundant. We already know that the source can only move back to b if the target moves from c. Otherwise there would be a cycle contradicting Lemma 4. So there must be a list, $\langle c, \ldots, y \rangle$ of one or more nodes that the target moves along. Since the source moves only over positive edges (using the weight of the negative in case of contingent) there must be another list $\langle a, \ldots, x \rangle$ that the source moves over before reaching b again. The final edge derived before reaching b is xy, whose edge will be a sum of negative weights along $\langle c, \ldots, y \rangle$ where negative requirement edges and positive contingent edges contribute, and positive weights along $\langle a, \ldots, x \rangle$ where positive requirement edges and negative contingent edges contribute. For the source to return to b, the weight of xy must be negative and there must be a positive edge bx. Then we can apply a rule deriving the edge by. We can determine that this edge is redundant by applying derivations to it. If by is positive it is redundant since there is a tighter implicit constraint along the strictly negative bcy path, as discussed before the lemma. If by is negative we apply derivation to move the source towards x. In this way we continue to apply derivations until we get a positive edge zy or the source reaches x. If this happens the derived edge must have a larger value than the already present xy edge, and be redundant, or we have derived a cycle contradicting Lemma 4. This can also be seen by observing that derivations start with the weight of xy, which can only increase along the derivation chain.

If we instead get a positive edge zy along the derivations we can show that there is a tighter constraint implicit here. We know $z \neq x$. When first

Algorithm 10: The GlobalDC Algorithm

```
function GLOBAL-DC(G - STNU)
Interesting ← {All edges of G}
repeat
    for each edge e in G do
        Interesting ← Interesting\{e}
        for each rule (Figure 4.6) applicable with e as focus do
            Derive new edges z_i
            for each added edge z_i do
                Interesting ← Interesting ∪ {z_i}
                if not locally consistent then return false
                if negative cycle created then return false
            end
        end
    end
until Interesting is empty
return true
```

deriving xy there was a negative edge from z to some node t in the $\langle c, \ldots, y \rangle$ list. If $t = y$ we arrive with a larger weighted edge (positive) ty this time and it is redundant. If $t \neq y$ there is an implicit tighter negative constraint zty. So again the zt edge is redundant.

So by is already explicitly or implicitly covered and hence redundant for DC-verification. Therefore it is not part of a critical chain. □

This entails that along a critical chain each contingent constraint can only be part of at most two derivations: One using D1 and one using D2 or D6.

5.3 The GlobalDC Algorithm

We will apply the lemma above to the new algorithm GlobalDC (Algorithm 10). Given a full STNU this algorithm applies the derivation rules of Figure 4.6 globally, i.e., with all edges as focus in all possible *triangles* (giving an inner iteration $O(n^3)$ run-time). It does this until there are no more changes detected over a global iteration, i.e. the *Interesting* set is empty. The structure of GlobalDC is hence directly inspired by the Bellman-Ford algorithm [6]. Non-DC STNUs are detected in the same way as FastIDC, by checking locally for inconsistencies and squeezed contingent constraints, and globally for negative cycles.

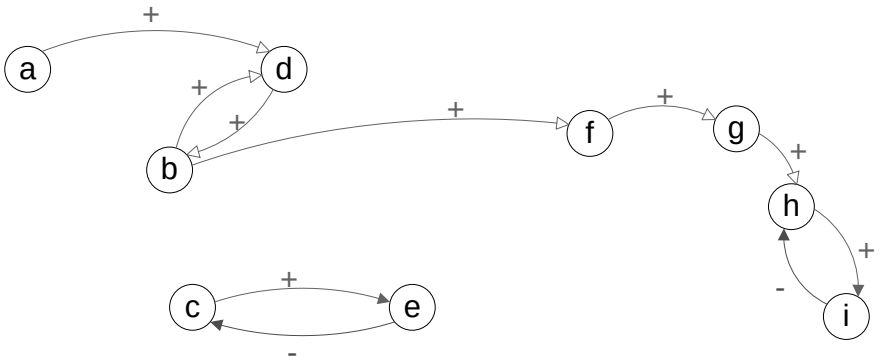

Figure 5.4: Example graph in quiescence.

This full DC algorithm can be compared with how an incremental algorithm (FastIDC) could be used to verify full DC, i.e., by adding edges from the full graph one at a time and doing derivations until done. Note that the order in which the derivation rules are applied to edges does not affect the correctness of FastIDC, only its run-time.

Given a DC STNU, GlobalDC will use the same derivation rules as FastIDC and therefore cannot generate tighter constraints. Since the same mechanism is used for detecting non-DC STNUs, both FastIDC and GlobalDC will indicate that the STNU is DC.

Given a non-DC STNU, there exists a sequence of derivations that will let FastIDC decide this. Since GlobalDC performs all possible derivations in each iteration, it will do all derivations that FastIDC does in the same sequence. Again, the same mechanism is used for detecting non-DC STNUs, and both FastIDC and GlobalDC will indicate that the STNU is non-DC.

The key to analyzing the complexity of GlobalDC is the realization that we can stop deriving new constraints as soon as we have derived all critical chains: These are the only derivations that are required for detecting whether the STNU is DC or not.

From the discussion after Lemma 4 we see that any derived chain has a length bounded by $2n^2$. This is then true also for the longest critical chain derived by the algorithm.

An example will illustrate how we can shrink the length of critical chains. Figure 5.4 shows a graph where no more derivations can be made. In Figure 5.5 a negative edge *ie* is added to the graph and GlobalDC is used to update the graph with this increment.

Figure 5.6 shows the critical chain of edge *ac* at this point. Here we see

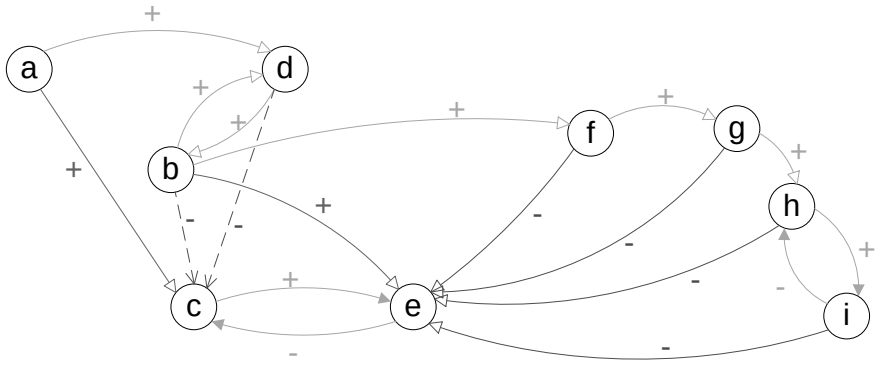

Figure 5.5: Derivations resulting from adding the $i \to e$ edge.

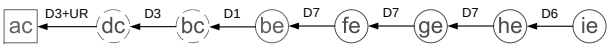

Figure 5.6: The critical chain of edge *ac*, derived in Figure 5.5.

as mentioned before that the source of the derived edge can move many times in sequence without the target moving in-between. In the example chain this is shown by the sequential D7 derivations. For requirement edges in general such a sequence may also include D4 derivations. Conditional edges can also induce sequences of moving sources through derivation rules D3 and D5.

All these derivations (D4/D7 and D3/D5) leading to sequential movement of the source require it to pass over requirement edges. If we had access to the shortest paths along requirement edges all these movements could in fact be derived in one global iteration. The source would be moved to all destinations at once and would not be replaced later since it had already followed a shortest path making the derived edge as tight as possible. Of course derivation rules may change the shortest paths, but if we added an APSP calculation to every global iteration we would compress the critical chains so that there would be no repeated application of sources moving along requirement constraints.

Figure 5.7 shows how several applications of D7 and two D3s are compressed by the availability of shortest path edges.
GlobalDC with the addition of APSP calculations in each inner iteration is still sound and complete since the APSP calculations only make more implicit constraints explicit. The run-time complexity is also preserved since

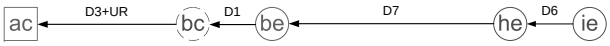

Figure 5.7: Critical chain compressed using shortest paths.

each inner iteration was already $O(n^3)$ (applying rules to all focus edges). We now give an upper bound of the critical chain length:

Lemma 6 (Nilsson et al. [25])**.** *The length of the longest critical chain in GlobalDC with APSP is* $\leq 7n$.

Proof. To be able to prove this we need the results of Lemma 5. We will refer to derivations that can only occur once along a critical chain, i.e. D1, D2 and D6, as *limited derivations*.

What is the longest sequence in a critical chain consisting only of requirement edges such that it does not use any limited derivations? The only non-limited derivation rules that result in a requirement edge are D4, D7 and D8/D9. The last two require a conditional edge as focus, and can therefore only be at the start of such a sequence. We know that due to APSP there can only be one of D4/D7 in a row. Therefore the longest requirement-only sequence not using limited derivations starts with D8/D9 which is followed by D4/D7 for a total length of 2.

The longest sequence consisting of only conditional edges not using limited derivations must start with D5. It can then be continued only by D3. As we have access to shortest paths there can be at most one D3 in any sequence of only conditional edges.

In summary the longest sequences of the same type, requirement or conditional, not using limited derivations, are of length 2.
It is not possible to interleave the length-2 sequences of conditional edges with requirement edges more than once without changing the conditioning node of the conditional edges. To see this suppose we have a requirement edge which derives a conditional edge conditioned on B. This means that the edge is pointing towards A being the start of the contingent duration ending in B. If derivations now takes this edge into a requirement edge this edge must point towards A as well since the only way of going from conditional to requirement is via D8/D9 which preserves the target. If the target of the requirement edge later were to move (such targets only move forwards) it would become impossible to later invoke D5 for going back to conditional, because D5 requires the requirement edge to point towards a node that is after A. So in order for derivations to come back to a conditional edge again by D5 the target must stay at A. But then D5 cannot be

applicable, for the same reason: It must point towards a node after A. So it is not possible to interleave these sequences.

This gives us the longest possible sequence without using limited derivations. It starts with a requirement sequence followed by a conditional sequence again followed by a requirement sequence. Such a sequence can have a length of at most 6. An issue here is that if a conditional edge conditioned on for instance B is part of the chain a D1 derivation involving B cannot also occur in the chain since this contingent constraint has already been passed. This means that it does not matter which of derivation D1 or D5 is used to introduce a conditioning node into the chain. The limitation applies to them both.

In conclusion this lets us construct an upper bound on the number of derivations in a critical chain. We have sequences of length 6 and these are interleaved with the n derivations of type D2 and D6 for a total of at most $7n$ derivations. □

Therefore all critical chains will have been generated after at most $7n$ iterations of GlobalDC. If we can iterate $7n$ times without detecting that an STNU is non-DC, it must be DC. With a limitation of $7n$ iterations, GlobalDC verifies DC in $O(n^4)$.

5.4 A Revised MMV Algorithm

We have described a new algorithm called GlobalDC and seen that it is $O(n^4)$. Compared to MMV, the following similarities and differences exist.

1. GlobalDC and MMV both interleave the application of derivation rules with the calculation of APSP distances and the detection of local inconsistencies and negative cycles. In MMV some of this is hidden in the pseudo-controllability test, but the actual conditions being tested are equivalent.

2. GlobalDC works in an EDG whereas MMV works in an STNU extended with wait constraints. These structures represent the same underlying constraints and the difference is not essential.

3. Third, GlobalDC lacks SR2, which is half of the original Simple Regression (SR) rule. As discussed before SR2 can be removed from MMV. This change will greatly speed it up in practice. Since MMV runs in an APSP graph it is reasonable to expect, on average, half of

the nodes to be after a derived wait. This change will then cut the needed regression in MMV to half of that of the original version.

4. Fourth, GlobalDC stops after $7n$ iterations.

These similarities lead to the following theorem:

Theorem 4 (Nilsson et al. [25])**.** *The classical MMV algorithm for deciding dynamic controllability of an STNU can, with the small modifications shown in Algorithm 11, decide dynamic controllability in time $O(n^4)$.*

Proof. According to the four points just listed, the differences between MMV and GlobalDC that affect the resulting STNU are: 1) the use of SR2 by MMV and that 2) MMV may continue to apply derivations after all critical chains have been derived. The theorem about critical chains carries directly over from FastIDC derivations to MMV tightenings. Therefore, it never takes MMV more than 7n iterations to derive all critical chains of an STNU. It is also possible to remove SR2 as discussed earlier without affecting the correctness of the algorithm. Algorithm 11 is a revised version of MMV which does not apply SR2 and stops after at most 7n iterations. It is correct and since the inner loop takes $O(n^3)$ time the whole algorithm has a run-time in $O(n^4)$. □

Algorithm 11: The revised MMV Algorithm

```
function Revised-MMV(G - STNU)
Interesting ← {All edges of G}
iterations ← 0
repeat
    if not pseudo-controllable (G) then
        return false
    Compare edges and add all edges which were changed since last
    iteration to Interesting
    for each edge e in Interesting do
        Interesting ← Interesting\{e}
        for each triangle ABC containing e do
            tighten ABC according to figure 5.1 except SR2
        end
    end
    iterations ← iterations + 1
until Interesting is empty or iterations = 7n
return true
```

A further improvement in algorithm 11 as compared to the original MMV is that it will only process triangles which can lead to derivations of new edges. This is facilitated by keeping track of edges that have been tightened in previous iterations, through use of the *Interesting* set. If a triangle could not be used to derive new edges when tried in one iteration, this cannot happen in later iterations unless at least one of its edges is tightened.

5.5 Conclusion

We have proven that with a small modification the classical "MMV" dynamic controllability algorithm, which in its original form is pseudo-polynomial, finishes in $O(n^4)$ time. The modified algorithm is an excellent and viable option for determining whether an STNU is dynamically controllable. Compared to other algorithms, it offers a simpler and more intuitive theory. We also showed indirectly that there is no reason for MMV to regress over negative edges, a result that can be used to improve performance further.

Chapter 6

The EfficientIDC Algorithm

In this chapter we start with analyzing the FastIDC algorithm seen in Chapter 3. We show that a small incremental change may result in the algorithm traversing part of a temporal network multiple times, with constraints slowly tightening towards their final values. We find that in the worst case it takes $\Omega(n^4)$ time for FastIDC to handle one incremental change. We then present a new algorithm that uses additional analysis together with a different traversal strategy to avoid this behavior. The new algorithm has a time complexity of amortized $O(n^3)$, and we prove that it is sound and complete.

6.1 Complexity of FastIDC

The complexity of FastIDC was not known when we started to work with it. Instead we performed the analysis in this section, which together with other experiences lead to the EfficientIDC algorithm. We first discuss the edge processing order, then propose an improvement, and finally show that even with the improvement FastIDC has a bad run-time complexity.

Edge processing order. Following [32], the initial list of modified edges is processed in order of distance to the temporal reference, but all edges derived by FastIDC itself are handled recursively and depth-first. The small example in Figure 6.1 shows why this is a suboptimal strategy for selecting focus edges. In this example the positive edges are present in the initial graph. All edges in the graph are requirement edges.

The negative IA edge is added as the only edge in this example increment. Thus FastIDC is called and Q will contain only $e_1 = IA : -100$.

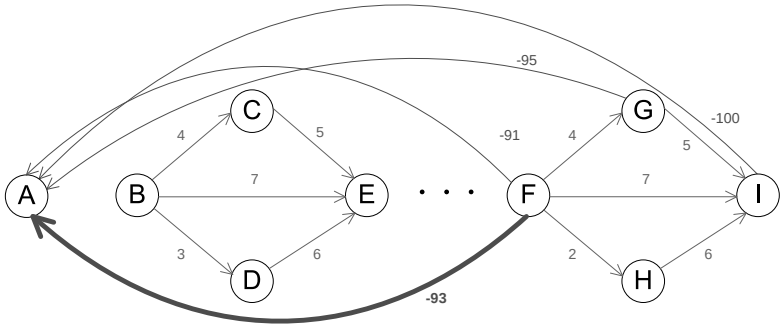

Figure 6.1: Why depth first is a suboptimal strategy.

This leads to derivation of the $GA : -95$. The depth first strategy then derives $FA : -91$, and additional edges moving toward the start of the STNU. However at a later step the IA and FI edges will be used to derive a strictly smaller weight $FA : -93$ (shown by the bold edge in the figure). This derivation will then propagate to decrease the weights of all the previously derived edges. In the worst case, a number of paths of positive edges from A to I, proportional to the total number of paths, may be traversed in reverse by FastIDC as new negative weights are incrementally derived. There is an exponential number of different paths in a graph which makes this worst case suboptimal.

An Improved Search Strategy. As noted above, the algorithm as published sorts the initial list of modified edges but processes newly derived edges depth first. This can be improved by keeping a *global* priority queue Q of modified edges. When a new edge is derived, it is not processed recursively but added at the proper place in this queue. The algorithm then iterates until the queue of modified edges is empty. The effect is that in each iteration the algorithm chooses among *all* known modified edges the one that is the furthest from the temporal reference, as was perhaps intended by the authors but not realized in the pseudo-code.

A Lower Bound on Time Complexity. An example will now show that even with the improved search strategy, the worst case run-time complexity of FastIDC is still $\Omega(n^4)$ when processing the tightening of one edge.

The left part of Figure 6.2 shows a part of an EDG created by FastIDC when incrementally adding constraints to an STNU. The figure contains three categories of nodes: A, B and C nodes. All B nodes are connected in sequence by edges of weight 1 as illustrated in the figure. So are the A

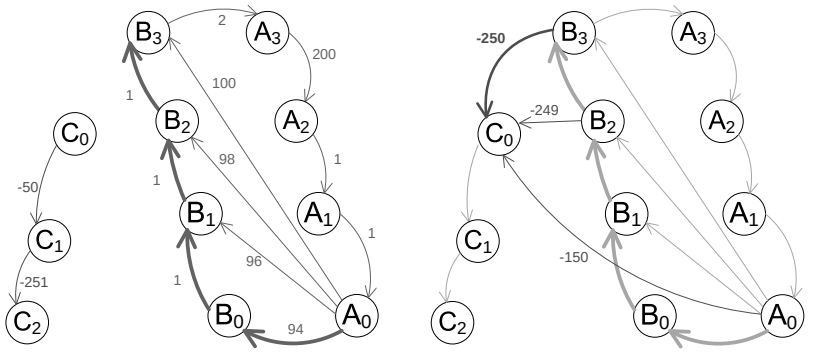

Figure 6.2: High complexity scenario part 1.

nodes, except the one with highest index. A_0 is connected to all B nodes by edges whose weights increase with the indices of B nodes. There is also one edge of weight 100 from each B node to each A node. These $|A| \cdot |B|$ edges are omitted in the figure for clarity.

The nodes in the figure are ordered from left to right by path distance to the temporal reference node (TR), which is not shown in the figures. This means that there are negative edges or paths from the nodes to the TR and that these are more negative the further to the right in the figure a node is placed. Recall that negative edges are sorted in the FastIDC queue by the distance from their source node to the TR.

FastIDC derives edges by giving higher priority to negative edges whose source nodes are closer to the end of the EDG. The example in Figure 6.2 contains a shortest path A_0, B_0, B_1, B_2, B_3 from the end towards earlier nodes. However this order works against FastIDC since derivation rule D7 derives tighter constraints in the *opposite* direction (the source of the derived edge is that of the positive edge). We will exemplify this now.

Suppose the $C_0 \leftarrow B_3$ edge shown in bold in the right part of Figure 6.2 is added or tightened to a weight of -250 and that FastIDC is called with this edge as e_1. FastIDC will find two applicable derivations where this is the focus edge. Both derivations are instances of D7, resulting in $C_0 \leftarrow B_2$ and $C_0 \leftarrow A_0$ being created and placed in the queue.

In the next iteration, $C_0 \leftarrow A_0$ has the highest priority (because A_0 is farther from the TR than B_2 is), and will be taken from the queue. When processing this edge, derivations using D7 will combine it with the "hidden" edges $B_i \rightarrow A_0$ with weight 100 to derive $|B|$ edges $C_0 \leftarrow B_i$. These are all put in the queue. An edge $C_0 \leftarrow A_1$ with weight -149 will also be

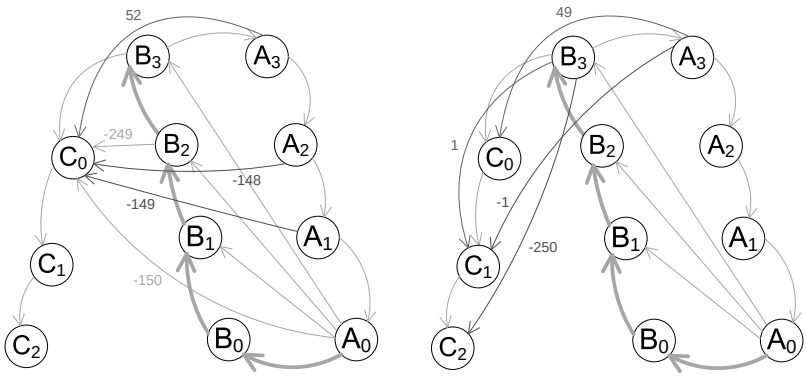

Figure 6.3: High complexity scenario part 2.

generated and ends up in front of the queue (Figure 6.3).

When $C_0 \leftarrow A_1$ is taken for processing, $|B|$ new edges are derived through combination with $B_i \rightarrow A_1$, but these are discarded since they have higher weights than the previously derived edges in their positions. An edge $C_0 \leftarrow A_2$ with weight -148 is also derived and will be processed first. Processing this leads to a similar procedure, again with $|B|$ edges discarded. The edge $C_0 \leftarrow A_3$ is among those derived. Since it has a positive weight it is sorted by its target node's (C_0) distance to the TR and therefore ends up last in the queue. The left part of Figure 6.3 shows the current situation.

At this point the edges from $C_0 \leftarrow B_i$ will be taken from the queue and not lead to any derivations until $C_0 \leftarrow B_2$ is processed. This is used to derive tighter $C_0 \leftarrow B_1$ and $C_0 \leftarrow A_0$ edges which in turn follow the pattern just described leading to tightenings of the $C_0 \leftarrow A_i$ edges. This happens again when $C_0 \leftarrow B_0$ and $C_0 \leftarrow A_0$ are tightened. At this point the $C_0 \leftarrow A_3$ edge reaches its final weight 49. The queue is then processed until $C_0 \leftarrow A_3$ is removed as the last edge in the queue. This leads to derivation of $C_1 \leftarrow A_3$ with weight -1 which in turn leads to $C_1 \leftarrow B_3$ of weight 1 and $C_2 \leftarrow B_3$ with weight -250. Here we again have an edge from B_3 with weight -250. FastIDC will then continue the exact same sequence as before but now deriving edges toward C_2 instead of C_0.

It is possible that $|A|$, $|B|$ and $|C|$ are all $O(n)$ and follow the same pattern as in the example. Then there are $O(n)$ spins around the $A - B$ cycle as the target of the negative $C \leftarrow B$ edges traverses all C nodes. Each spin around the cycle takes $O(n^3)$ time: There are $O(n)$ updates to $C_x \leftarrow A_0$

and each of these updates the $O(n)$ $C_x \leftarrow A_i$ edges, each of which tries to update the $O(n)$ $C_x \leftarrow B$ edges. The worst case complexity of one call to FastIDC must therefore be at least $O(n^4)$.

Unfortunately, even though the structure in the example requires the addition of many edges that are handled quickly by FastIDC, the complexity cannot be amortized to reach a lower value. The problem is that FastIDC will always pay the full $O(n^4)$ price each time the $C_0 \leftarrow B_3$ edge in the example is tightened. This may happen as part of other tightenings or by direct change many times as the final STNU is built. Therefore, it cannot be assumed that there are cheaper increments that can pay for the more expensive ones.

One cause of this high complexity is the existence of a region of nodes (the A and B nodes) where there is at least initially no forced ordering between the nodes. In the next section we present the EfficientIDC algorithm. This algorithm presents a novel way of handling these regions, among other things.

6.2 The EfficientIDC Algorithm

We now present the Efficient Incremental Dynamic Controllability checking algorithm (Algorithm 12, EfficientIDC or EIDC for short). The key to EIDC's efficiency is the use of focus *nodes* instead of focus edges. When EIDC tightens an edge, it adds the target and sometimes also the source of this edge as new focus nodes to be processed. When EIDC processes a focus node n, it applies all derivation rules that have an incoming edge to n as focus edge, guaranteeing that no tightenings are missed.

The use of a focus node allows EIDC to use a modified version of Dijkstra's algorithm to efficiently process parts of an EDG in a way that avoids the repetitive intermediate edge tightenings performed by FastIDC that we just saw in the previous section. The key to understanding this is that derivation rules essentially calculate shortest distances. For example, rule D4 states that if we have tightened edge AB and there is an edge BC, an edge AC may have to be tightened to indicate the length of the shortest path between A and C. Dijkstra's algorithm cannot be applied indiscriminately, since there are complex interactions between the different kinds of edges, but can still be applied in certain important cases.

The *final* tightening performed for each edge will be identical in EIDC and FastIDC, which is required for correctness. An extensive example will be provided below.

As in FastIDC, the EDG is associated with a *CCGraph* used for detecting cycles of negative edges. The graph also helps EIDC determine in which order to process nodes: In reverse temporal order, from the "end" towards the "start", taking care of incoming edges to one node in each iteration. A difference compared to FastIDC is that EIDC keeps the transitive closure of negative edges in the CCGraph. This facilitates the correct ordering of nodes that are indirectly ordered.

The EDG is also associated with a *Dijkstra Distance Graph* (*DDG*), a new structure used for the modified Dijkstra algorithm as described below. To simplify the presentation, EIDC will be given one new or tightened requirement edge e at a time. In practice, an outer loop would be used to handle a set of changes. This set could then be sorted before presented to EIDC in a way similar to how FastIDC handles a set of changes.

The EfficientIDC algorithm. The EIDC algorithm is shown in Algorithm 12. First, the target of e is added to *todo*, a set of focus nodes to be processed.

If e is a *negative* requirement edge, a corresponding edge is added to the *CCGraph* C which keeps track of all negative edges. If this causes a negative cycle, G is not DC. Otherwise, the source of e is also added to *todo* for processing.

Iteration. As long as there are nodes to process:

A node to process, *current*, is selected and removed from *todo*. Incoming negative edges e to the chosen node n must not originate in a node also marked as *todo*. In that case, $\mathsf{Source}(e)$ should be processed first, since this has the potential of adding new incoming edges to n and we must make sure we have found all these at the time we start processing n in order to be correct.

As long as *todo* is not empty there is always a *todo* node satisfying this criterion, or there would be a cycle of negative edges which would have been detected.

Then it is time to process all existing incoming edges to *current*. This may derive more incoming edges and push some towards earlier nodes. *current* is processed by three helper functions that are shown in Algorithms 13 to 15.

Incoming conditional edges are processed similarly to FastIDC focus edges using ProcessCond. This is equivalent to applying rules D2, D3, D8 and D9, but is done for a larger part of the graph in a single step compared to FastIDC.

There are only $O(n)$ contingent constraints in an EDG and hence only $O(n)$ conditioning nodes (nodes that are the target of a contingent con-

Algorithm 12: The EfficientIDC Algorithm

```
function EfficientIDC(EDG G, DDG D, CCGraph C, edge e)
todo ← {Target(e)}
if e is negative and e ∉ C then
    add e to C
    if negative cycle detected then return false
    todo ← todo ∪{Source(e)}
end
while todo ≠ ∅ do
    current ← pop some n from todo where
        ∀e ∈ Incoming(C, n) : Source(e) ∉ todo
    ProcessCond(G, D, current)
    ProcessNegReq(G, D, current)
    ProcessPosReq(G, current)
    for each edge e added or modified in G in this iteration do
        if Target (e) ≠ current then
            todo ← todo ∪{Target(e)}
        end
        if e is a negative requirement edge and e ∉ C then
            add e to C
            if negative cycle detected then return false
            todo ← todo ∪{Target(e), Source(e)}
        end
    end
    if G is squeezed then return false
end
return true
```

straint). All times in conditional constraints/edges are measured towards the source of the contingent constraint. Therefore, all conditional constraints conditioned on the same node have the same target.

It is important to note that EIDC processes conditional edges conditioned on the same node separately. This is possible because FastIDC does not "mix" conditional edges with different conditioning nodes in any of the rules, so they cannot be derived "from each other".

For each conditioning node c, the function finds all edges that are conditioned on c and have *current* as target. We now in essence want to create a single source shortest path tree rooted in *current*. Derivations over positive requirement edges traverse the edges in reverse order, and so the DDG contains these edges in reverse order. Derivations over contingent edges

Algorithm 13: Process Conditional Edges

```
function ProcessCond(EDG G, DDG D, Node current)
allcond ← IncomingCond(current, G)
condnodes ← {n ∈ G | n is the conditioning node of some e ∈ allcond}
for each c ∈ condnodes do
    edges ← {e ∈ allcond | conditioning node of e is c}
    minw ← |min{weight(e) : e ∈ edges)}|
    add minw to the weight of all e ∈ edges
    for e ∈ edges do
        add e to D with reversed direction
    end
    LimitedDijkstra(current, D, minw)
    for all nodes n reached by LimitedDijkstra do
        e ← cond. edge (n → current), weight Dist (n) - minw
        if e is a tightening then
            add e to G
            apply D8 and D9 to e
    end
    Revert all changes to D
end
return
```

follows the negative contingent edge, but the distance used in the derivation is the positive weight of this, so this is also contained in the DDG. The section of the graph which can be traversed contains only positive weight edges and so Dijkstra's algorithm can be used to find the shortest paths. The only remaining issue is that the edges connecting the source of the tree we want to build are negative and in reverse order. Since only one of these edges will be used by each path, there is no risk of negative cycles so they could be used directly. However, when EIDC reverses the edges it also adds a positive weight to them to make all edges used by the Dijkstra calculation positive. The added weight, *minw*, is the absolute value of the most negative edge weight of the incoming conditional edges. This value also serves as a cut-off for stopping the Dijkstra calculation. Once the distance is longer than *minw* the derived result will be a positive edge which cannot further react to cause more derivations. Running Dijkstra calculations will in a single call derive a final set of shortest distances that FastIDC might have had to perform a large number of iterations to converge towards. An example in the next section shows how this is carried out.

The function checks whether any calculated shortest distance leads to the derivation of a tighter edge, corresponding to applying D2 and D3 over the processed part of the graph. If so, it directly applies the "special" derivation rules D8 and D9, which convert conditional edges to requirement edges, or adds a requirement edge parallel to the conditional edge.

Because of D8/D9, ProcessCond may generate new incoming *requirement* edges for *current*, which is why it must be called before incoming requirement edges are processed.

Incoming negative requirement edges are processed using ProcessNegReq. This function is almost identical to ProcessCond with the only differences being that the edges are negative requirement instead of conditional and because of this there is no need to apply the D8 and D9 derivations or to handle different conditioning nodes. Applying the calculated shortest distances in this case corresponds to applying the derivation rules D6 and D7.

Algorithm 14: Process Negative Requirement Edges

```
function ProcessNegReq(EDG G, DDG D, Node current)
edges ← IncomingNegReq(current, G)
minw ← |min{weight(e) : e ∈ edges)}|
add minw to the weight of all e ∈ edges
for e ∈ edges do
    add e to D with reversed direction
end
LimitedDijkstra(current, D, minw)
for all nodes n reached by LimitedDijkstra do
    e ← req. edge (n → current) of weight Dist (n) - minw
    if e is a tightening then add e to G
end
Revert all changes to D
return
```

This function may generate new incoming *positive requirement* edges for *current*, which is why it must be called before incoming requirement edges are processed.

Incoming positive requirement edges are processed using ProcessPosReq, which applies rules D1, D4 and D5. These are the rules that may advance derivation towards earlier nodes. By deriving a new edge targeting an earlier node, the node is put in *todo* by the main algorithm.

After processing incoming edges. These are the only possible types of fo-

Algorithm 15: Process Positive Requirement Edges

```
function ProcessPosReq(EDG G, Node current)
for each e ∈ IncomingPosReq(current, G) do
    apply derivation rule D1, D4 and D5 with e as focus edge
    for each derived edge f do
        if f is conditional edge then
            apply derivations D8-D9 with f as focus edge
        end
        if derived edge is a tightening then
            add it to G
        end
    end
end
return
```

cus edge in FastIDC derivations. Therefore all focus edges that could possibly have given rise to the *current* focus node have now been processed.

EIDC then checks all edges that were derived by the helper functions. Edges that do not have *current* as a target need to be processed, so their targets are added to *todo*. If there is a negative requirement edge that is not already in the *CCGraph*, this edge represents a new forced ordering between two nodes. It must then update the *CCGraph* and check for negative cycles. If a new edge is added to the *CCGraph* both the source and the target of the edge will be added to *todo*.

Finally, EIDC verifies that there is no local squeeze when a new edge is added, precisely as FastIDC does.

Updating the CCGraph. A novel feature of EIDC as compared to FastIDC is that the CCGraph now contains the *transitive closure* of all edges added to it. This prevents reprocessing when new orders are found through ProcessPosReq. How the transitive closure is derived will be discussed later.

Updating the DDG graph. The DDG graph contains weights and directions of edges that FastIDC derivations use to derive new edges, and is needed to process edges effectively. Edges in the DDG have no type, only weights that are always positive. The DDG contains:

1. The positive requirement edges of the EDG, in reverse direction
2. The negative contingent edges of the EDG, with weights replaced by their absolute values

To make the algorithm easier to read, updates to the DDG have been omitted. Updating the DDG is straight forward and quite simple. When a positive edge is added to the EDG it is added to the DDG in reversed direction. Negative contingent edges also have to be added to the DDG (with the absolute value of their weight as new weight). In case a positive requirement edge disappears from the EDG, because it was tightened to a negative weight, it is removed from the DDG.

Before we process an example STNU we want to point to a specific aspect of the algorithm. We can see that D4 and D7 are the same rule with different focus. Both ProcessNegReq and ProcessPosReq apply this rule. The responsibility of ProcessNegReq is to efficiently find all incoming edges to *current* over areas of positive weights (the problem regions of FastIDC) whereas ProcessPosReq is used to find those nodes that are affected by finding new edges targeting *current*. So ProcessPosReq main responsibility is to find those nodes that must be processed in coming iterations.

6.3 EfficientIDC Processing Example

We now go through a detailed example of how EIDC processes the three kinds of incoming edges. Like before, dashed edges represent conditional constraints, filled arrowheads represent contingent constraints, and solid lines with unfilled arrowheads represent requirement constraints.

Fig. 6.4 shows an initial EDG constructed by incrementally calling EIDC with one new edge at a time. We will initially focus on the nodes and edges marked in black, while the gray part will be discussed at a later stage.

In the example we add a new requirement edge $Y \xleftarrow{-10} Z$ as shown in the rightmost part of Fig. 6.5. When we call EIDC for this edge, both Y and Z will be added to *todo*. Z must be processed first because of the ordering between Z and Y. Since Z has no incoming conditional or negative requirement edges only ProcessPosReq will be applied. This results in the bold requirement edges $a \xrightarrow{25} Y$ and $b \xrightarrow{30} Y$. The node Y is then selected as *current* in the next iteration. Even though Y has an incoming negative edge, no new derivations are done by ProcessNegReq. However, Y also has two incoming positive requirement edges that are processed (using D1) to generate the conditional edges $X \xleftarrow{\langle Y,-25\rangle} a$ and $X \xleftarrow{\langle Y,-20\rangle} b$. Two negative requirement edges, $X \xleftarrow{-9} a$ and $X \xleftarrow{-9} b$, are also derived alongside the conditional edges due to D9 but these are not stronger than the already existing identical edges. Since there were already edges from $X \longleftarrow a$ and

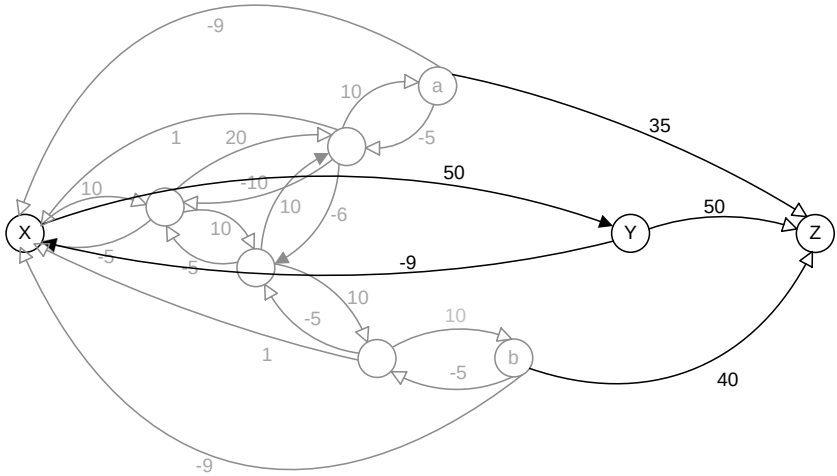

Figure 6.4: Initial EDG.

$X \longleftarrow b$ in the CCGraph, a and b are not added to *todo*. However, X is added as the target of a newly derived edge is always added to *todo*. Since the derived edges are not incoming to Y they require no further processing at the moment. This leaves only X in the *todo* set for the next iteration.

In the next iteration, X is selected as *current*. No more edges will be derived in the rightmost black part of the example EDG, so we focus on the previously gray part of the EDG shown in Fig. 6.6. We see that X has two incoming conditional edges with the same conditioning node Y. These edges are processed together, resulting in a *minw* value of 25. After adding edges corresponding to the reversed conditional edges, each with a weight increase of *minw*, we get the DDG that is used for Dijkstra calculations when processing X. The DDG is shown in Fig. 6.7. Recall that in the DDG all positive edges are present with reversed direction and all negative contingent edges are present with positive weight. Note that the weight 1 edges from X are left out of the DDG in Fig. 6.7. These are present in the DDG but cannot be used when X is *current* since using them would require that the source and target of the conditional edge used for derivation was the same. This is a degenerate case which cannot occur in the EDG. Such an edge would either be removed before addition or responsible for non-DC of the STNU. In Fig. 6.7 we have labeled each node with its shortest distance from X in the DDG.

Processing $current = X$ gives rise to the bold edges in Fig. 6.8. We

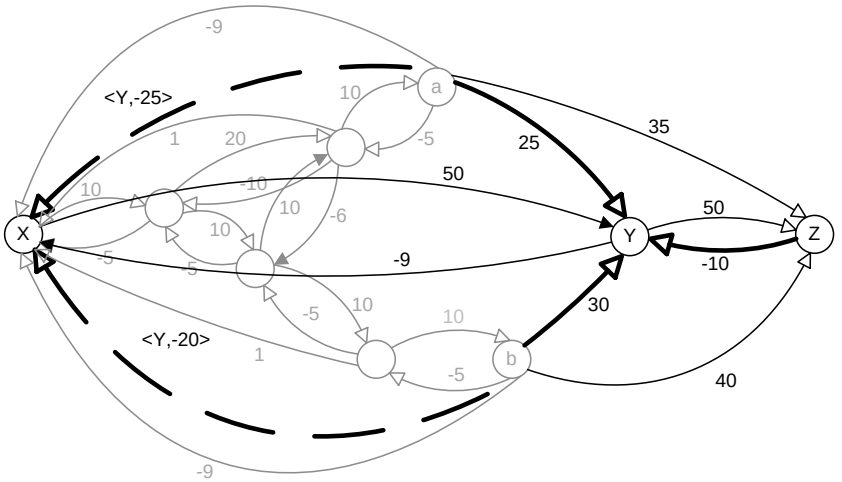

Figure 6.5: Derivation of the smaller scenario.

consider how the -9 edge is created. First the distance from X to the source node of the -9 edge is calculated by Dijkstra's algorithm. This is 16 (see Fig. 6.7). Subtraction of 25 gives a conditional edge with weight -9. However, since the lower bound of the contingent constraint involving X is 9, D8 is then applied to remove the conditional edge and create a requirement edge with weight -9. The distance calculation corresponds in this case to what FastIDC would derive by applying first D3 and then D6, starting with the conditional $X \overset{\langle Y,-25\rangle}{\longleftarrow} a$ edge as focus.

The example shows how EIDC adds *minw* to the negative edges from the source to get positive edges for Dijkstra's algorithm to work with. It also shows why outgoing DDG edges from *current* cannot be used when calculating the Dijkstra distances from *current*.

Note that there is no reason to follow incoming DDG edges to *current* since these only create loops from *current* to *current*. Any such positive loop could be removed and a negative loop would be discovered as a local inconsistency in the step immediately before it would be added.

Finally, all new derived edges need to be checked so they do not squeeze existing edges, and negative edges should be added to the cycle checking graph when needed.

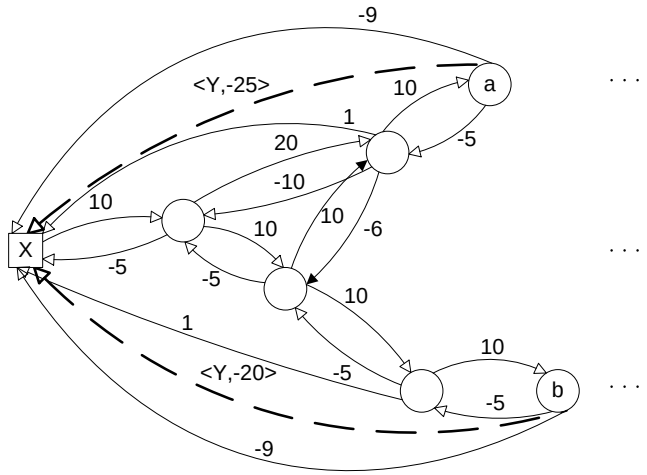

Figure 6.6: Example scenario for conditional edges.

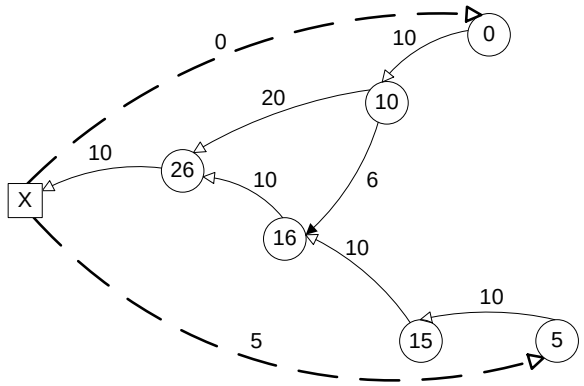

Figure 6.7: Dijkstra Distance Graph of the small scenario.

6.4 Correctness of the EfficientIDC Algorithm

The following lemma states the correctness of the algorithm based on the corrected version of FastIDC as proven in Chapter 4. Since FastIDC may update the same edge more than once when processing an increment the lemma compares the EDG of EfficientIDC against the final result of FastIDC. In the proof it is assumed for simplicity that each call consists of only one edge tightening/addition. Any increment consisting of more changes can be broken down to several without affecting the end results, thus preserv-

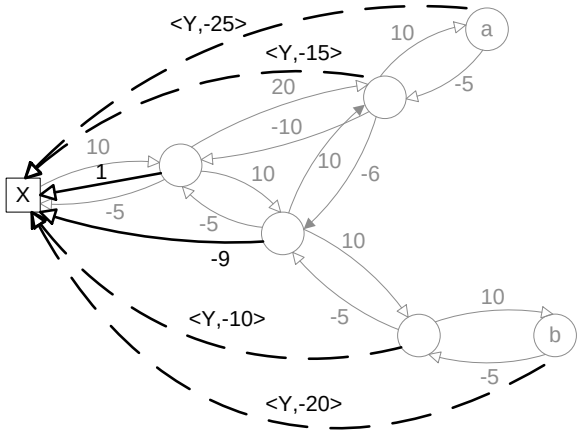

Figure 6.8: Result of processing *current* $= X$.

ing correctness.

Lemma 7 (Nilsson et al. [26])**.** *Let G be an EDG of a DC STNU and e be a single tightened edge in G. Let G' be the graph produced by* FastIDC (G, e) *and let G'' be the graph produced by* EfficientIDC (G, e)*. Then $G' = G''$. Additionally, the algorithms agree on whether the corresponding STNU is dynamically controllable.*

Proof. First, the derivation rules of FastIDC only generate sound conclusions. The derivations performed by EIDC are either through direct use of FastIDC derivation rules or through the use of Dijkstra in a way that corresponds directly to repeated application of derivation rules. Therefore EIDC is sound in terms of edge generation.

Second, completeness requires that for every tightened edge, all applicable derivation rules are applied. When an edge is tightened, EIDC always adds the target node to *todo*. All nodes in *todo* will eventually be processed, and when a node *current* is removed from *todo*, all derivation rules applicable with any incoming edge as focus are applied.

This means that any tightened edge that would be a focus edge of FastIDC has its target node added to *todo* and is then later processed as *current*. At this time all the same derivation rules as FastIDC are applied to the edge, although some may be applied through the use of Dijkstra's algorithm. So all edges derived by FastIDC are derived by EIDC, although perhaps in different order, and some intermediate edges that are overwritten by FastIDC will be skipped by EIDC. Thus, the algorithms eventually derive the same edges. Since they both check the DC property in the same way they also

agree on which STNUs are DC and which are not. □

6.5 Run-time Complexity of EfficientIDC

We first discuss the complexity of deriving the transitive closure of the negative edges in the CCGraph. This can be done in $O(n^2)$ when processing a node. First, all negative requirement edges that was derived in this iteration are added to the CCGraph. Then, for each of the edges that targets *current*, all their sources and via them, all their CCGraph predecessors are collected. These are then connected in the CCGraph to all CCGraph successors of *current*. This simple algorithm is enough to guarantee that the transitive closure is found. The complexity is within $O(n^2)$ since there are $O(n)$ incoming edges and sources, each which requires at most $O(n)$ time to find the predecessors. Connecting these possible $O(n)$ predecessors to the possible $O(n)$ successors of *current* takes at most $O(n^2)$ time.

We now present the run-time of EfficientIDC in a theorem.

Theorem 5 (Nilsson et al. [26])**.** *The run-time of EfficientIDC when processing one tightened or added edge is $O(n^4)$ in worst case but $O(n^3)$ amortized, where n is the number of nodes.*

Proof. When EIDC adds a negative requirement edge e, it checks whether this is already represented in the CCGraph. If not ($e \notin C$), the edge previously had positive weight (possibly ∞), and its new negative weight represents a new forced ordering.

First, assume this does *not* happen: Whenever a new negative requirement edge e is created, it is already in C. This means that no node is added to *todo* as the source of a derived edge (with the exception of the externally added edge). Therefore, the only derivations which cause nodes to be added to todo are those which add them as targets, namely those handled by ProcessPosReq (derivation rules D1, D4 or D5). It can be seen from Figure 4.6 that these are only applicable if there is a negative edge between the previous focus edge target and the derived edge target.

Since we assumed no new negative edges are derived, all negative edges along which nodes can be added to *todo* are present from the start. If a node X is added to *todo* and selected for processing, there can be no other node in *todo* which has a path of negative edges to X. This would be caught by the transitive closure in the CCGraph and X would have not have been selected in that case. Therefore, once X is processed it cannot be added to

todo again. We now continue to analyze the complexity of EIDC under this assumption.

Complexity 1. Since each node can be selected as *current* at most once, the main **while** loop iterates $O(n)$ times.

In each iteration, the incoming positive requirement edges for *current* can be processed in $O(n^2)$ time: Each derivation is $O(1)$ and there are at most $O(n)$ incoming positive edges which can find at most $O(n)$ outgoing edges for derivations.

Processing incoming conditional and negative requirement edges is more complicated, due to the use of Dijkstra's algorithm. Conditional edges require slightly more work than negative requirement edges and therefore provide an upper bound for both types. The cost of updating the DDG used for Dijkstra calculations is $O(1)$ per edge change which is hidden in the normal cost of adding edges. The following list shows the complexity of the different steps done when processing conditional edges conditioned on *one* node.

1. Add conditional edges to the DDG, $O(n)$
2. Find *minw* among these, $O(n)$
3. Replace weights on the negative contingent edges, $O(n)$
4. Run the limited Dijkstra's algorithm $O(n^2)$
5. Add new conditional/requirement edges to the EDG, $O(n)$
6. Remove conditional edges from the DDG, $O(n)$
7. Update the transitive closure in the CCGraph, $O(n^2)$

This sums to $O(n^2)$ for processing all conditional edges conditioned on *one* node. Taking care of all conditioning nodes throughout the EDG causes the procedure to be carried out $O(n)$ times and incurs an $O(n^3)$ aggregated cost.

It follows from the described procedure that processing negative requirement edges for *current* takes $O(n^2)$ time.

Each outer loop adds $O(n^2)$ new edges. Checking local consistency of these takes $O(n^2)$ time. Adding the new edges to the *CCGraph* takes accumulated $O(n^3)$ time over the whole increment.

The final step is to choose the next *current* node for processing and this is done by picking any node from *todo* that has no predecessors in *todo*. In practice a list of candidates is kept which is updated every time a node has

been removed from *todo*. This is done in $O(n^2)$ and is done once in each outer iteration, for a total within $O(n^3)$.

Second, we consider what happens when new orderings are found and added to C while processing an increment.

Let X be the node that was found to be ordered after *current*. Finding all incoming edges to *current* depends on the fact that all nodes ordered after it, including X, must have been processed before *current*. If X was processed after *current*, edges targeting *current* that could be derived via X may be missed and the algorithm would not be complete. These are however the only edges targeting *current* that would be missed. So the algorithm goes back to process X and then reprocesses *current* to find these edges.

An order such as the one just discovered can only be found when processing a node that is ordered after *current* or when processing *current*. The new edge must be derived through interaction of a positive edge and a negative edge targeting *current*, i.e. it would be found at *current* or when processing the source of the negative edge which is by definition ordered after *current*. If the new order is found when processing any other node ordered after *current* there is no need for reprocessing as the requirement for finding all edges at *current* is satisfied.

Complexity 2. New orderings that lead to reprocessing of nodes are detected when the node needing reprocessing is being processed as *current*. Therefore, the cost for reprocessing is only that of one iteration in the algorithm per new ordering found. Over the course of constructing an STNU there can be $O(n^2)$ new orderings found, however each may only affect the same node $O(n)$ times. This is important since the cost of processing a node is $O(n^2)$ plus any processing of conditional edges for up to a total of $O(n^3)$, for instance if this node is the target of a maximum of conditional edges in the STNU. Regardless of how the cost of processing conditional nodes is spread throughout the STNU they may be involved in reprocessing $O(n)$ times in the worst case. Therefore, in the worst case, all $O(n^2)$ orderings are found in the same iteration, leading to a worst case complexity of $O(n^4)$ for one increment.

However, considering that the algorithm is $O(n^3)$ if no new orders are detected, it is possible to amortize the cost on the number of nodes added to the STNU. For the amortized analysis, each time a node is added we first save the $O(n^3)$ cost that may later be needed for reprocessings related to the new node. One node may be the cause of at most $O(n)$ reprocessings, each costing $O(n^2)$ for positive and negative requirement edges. There is also the possibility of an accumulated cost of $O(n^3)$ for all possible conditional

reprocessings. Therefore, the total reprocessing cost that can be caused by the addition of one node is bounded by $O(n^3)$. If we now instead save the $O(n^3)$ cost per added or tightened edge, to amortize the incremental cost, there will be even more accumulated resources since there must be at least one added edge per node. We conclude that the amortized cost of adding or tightening an edge is $O(n^3)$. □

6.6 Conclusion

A new way of incrementally testing dynamic controllability is presented. It is more efficient than FastIDC but provides the same result, both in form of EDG and DC classification. Higher efficiency is gained by observing that FastIDC is inefficient when deriving constraints over unordered sections in the EDG. EIDC overcomes this by applying Dijkstra's algorithm to quickly derive all constraints over such sections. The EDG processed by EIDC is dispatchable since it contains the same constraints as FastIDC.

Chapter 7

Related and Future Work

Recently an algorithm for full DC verification which is $O(n^3)$ was published by Morris [18]. We wish to clarify the relation between this work and his. Though these algorithms have similar complexity results and share certain concepts, they were developed independently. They also use different graph representations and different rules for updating the graphs, and the key ideas underlying EIDC [26] were submitted and finalized before the publication of Morris' paper.

Future work includes a rigorous comparison of the algorithms and comparison of their relative performance in practice. To do this a good set of benchmarks is needed. Finding this is a large study in itself since benchmarks would need to include random STNUs, STNUs from planners and STNUs with certain properties (for instance magic loops [14]).

It is possible to do both full and incremental DC verification in $O(n^3)$. Can we do better? For STNs $O(n^2)$ incremental algorithms exist. There are also several results for STNs by Planken [27, 28, 30] which may be transferable to STNUs.

Among related work we further find the MM and Morris algorithms which we only mentioned in the introduction chapter since all our work was based on the other track of algorithms (MMV $\rightarrow$ FastIDC). Both the MM and Morris algorithm are outdated now as they are subsumed by the new algorithm from Morris.

A lot of related work has gone into executing the networks. For this thesis we focused on DC verification, but after this is done execution is the natural next step. Hunsberger has published several interesting results on execution [12, 13].

Recently several papers [3, 5] have examined the use of Timed Game

Automata (TGA) for both verification and execution of STNUs. These solutions work on a smaller scale and do not exploit the inherent structure of STNUs as distance graphs. Therefore they are more useful in networks that are small in size but involve choice and resources which cannot be handled by pure STNU algorithms.

Chapter 8

Conclusion

In this thesis we have pointed out a flaw in the FastIDC algorithm which makes the algorithm unsound. We have then analyzed its cause and provided a fix, correcting it. This included adding a novel structure for incremental cycle checking while making use of work done by the FastIDC algorithm in order to keep the efficiency while making the algorithm sound. We also showed that two additional derivation rules are needed and provided a correctness proof for the sound version of the FastIDC algorithm.

We continued to use insights from FastIDC and applied those to the MMV algorithm. An algorithm which we showed had no need of one of its derivations, SR2. We also showed that the algorithm must work with all APSP edges. The final result related to the MMV algorithm consisted of a small change which allowed it to become $O(n^4)$, which is on par with the best existing algorithm at the time our paper were published.

We then turned back to FastIDC and analyzed the run-time which was found to be in $\Omega(n^4)$. The analysis inspired us to come up with the EfficientIDC algorithm which has an amortized run-time of $O(n^3)$. This is the final contribution of the thesis.

Bibliography

[1] James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[2] M.A. Bender, J.T. Fineman, S. Gilbert, and R.E. Tarjan. A new approach to incremental cycle detection and related problems. *arXiv preprint arXiv:1112.0784*, 2011.

[3] Amedeo Cesta, Alberto Finzi, Simone Fratini, Andrea Orlandini, and Enrico Tronci. Analyzing Flexible Timeline-based Plans. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI)*, pages 471–476, 2010. URL `http://dx.doi.org/10.3233/978-1-60750-606-5-471`.

[4] N. Chleq. Efficient algorithms for networks of quantitative temporal constraints. In *Proceedings of CONSTRAINTS-95*, pages 40–45, 1995.

[5] Alessandro Cimatti, Luke Hunsberger, Andrea Micheli, and Marco Roveri. Using Timed Game Automata to Synthesize Execution Strategies for Simple Temporal Networks with Uncertainty. In *Proceedings AAAI*, 2014.

[6] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001. ISBN 0070131511.

[7] G.B. Dantzig. *Linear programming and extensions*. Princeton University Press, 1998. ISBN 978-0-69105-913-6.

[8] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003. ISBN 978-1-55860-890-0.

[9] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991. URL `http://dx.doi.org/10.1016/0004-3702(91)90006-6`.

[10] Patrick Doherty and Jonas Kvarnström. TALPLANNER - A temporal logic-based planner. *The AI Magazine*, 22(3):95–102, 2001. ISSN 0738-4602.

[11] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, Amsterdam, 2004. ISBN 978-1-55860-856-6.

[12] Luke Hunsberger. A fast incremental algorithm for managing the execution of dynamically controllable temporal networks. In *Proceedings of the 17th International Symposium on Temporal Representation and Reasoning (TIME)*, pages 121–128, 2010.

[13] Luke Hunsberger. A faster execution algorithm for dynamically controllable STNUs. In *Proceedings of the 20th International Symposium on Temporal Representation and Reasoning (TIME)*, 2013.

[14] Luke Hunsberger. Magic loops in simple temporal networks with uncertainty. In *Fifth International Conference on Agents and Artificial Intelligence (ICAART-2013). SciTePress*, 2013.

[15] K. M. Kahn. Mechanization of temporal knowledge. Technical report, Cambridge, MA, USA, 1975.

[16] P. Kim, B.C. Williams, and M. Abramson. Executing reactive, model-based programs through graph-based temporal planning. In *Proceedings of the 17th international joint conference on Artificial intelligence (IJCAI)*, pages 487–493, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-812-5, 978-1-558-60812-2.

[17] Paul Morris. A structural characterization of temporal dynamic controllability. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 4204 of *Lecture Notes in Computer Science*, pages 375–389. Springer, 2006. ISBN 3-540-46267-8.

[18] Paul Morris. Dynamic Controllability and Dispatchability Relationships. In *Proceedings of the 11th Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*, pages 464–479. Springer, 2014. ISBN 978-3-319-07045-2. URL `http://dx.doi.org/10.1007/978-3-319-07046-9_33`.

[19] Paul Morris and Nicola Muscettola. Managing temporal uncertainty through waypoint controllability. In *Proceedings of the 16th international*

joint conference on Artificial intelligence (IJCAI), pages 1253–1258, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[20] Paul Morris and Nicola Muscettola. Temporal dynamic controllability revisited. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI)*, pages 1193–1198. AAAI Press / The MIT Press, 2005.

[21] Paul Morris, Nicola Muscettola, and Thierry Vidal. Dynamic control of plans with temporal uncertainty. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 494–499, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-812-5, 978-1-558-60812-2.

[22] Paul H. Morris and Nicola Muscettola. Execution of temporal plans with uncertainty. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 491–496. AAAI Press / The MIT Press, 2000. ISBN 0-262-51112-6.

[23] Nicola Muscettola, Paul H. Morris, and Ioannis Tsamardinos. Reformulating temporal plans for efficient execution. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 444–452, 1998.

[24] Mikael Nilsson, Jonas Kvarnström, and Patrick Doherty. Incremental dynamic controllability revisited. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS)*, 2013.

[25] Mikael Nilsson, Jonas Kvarnström, and Patrick Doherty. Classical Dynamic Controllability Revisited: A Tighter Bound on the Classical Algorithm. In *Proceedings of the 6th International Conference on Agents and Artificial Intelligence (ICAART)*, pages 130–141, 2014. doi: 10.5220/0004815801300141.

[26] Mikael Nilsson, Jonas Kvarnström, and Patrick Doherty. EfficientIDC: A Faster Incremental Dynamic Controllability Algorithm. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS)*, 2014.

[27] L. Planken, M. de Weerdt, and R. van der Krogt. P^3C: A new algorithm for the simple temporal problem. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 256–263, 2008.

[28] Léon Planken, Mathijs de Weerdt, and Neil Yorke-Smith. Incrementally solving stns by enforcing partial path consistency. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 129–136, 2010. URL `http://www.aaai.org/ocs/index.php/ICAPS/ICAPS10/paper/view/1447`.

[29] Léon R. Planken. New algorithms for the simple temporal problem. Master's thesis, Delft University of Technology, January 2008. URL `http://www.st.ewi.tudelft.nl/~planken/Papers/mscthesis.pdf`.

[30] L.R. Planken, M.M. de Weerdt, and C. Witteveen. Optimal Temporal Decoupling in Multiagent Systems. In *Proceedings of the Ninth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IFAAMAS, 2010.

[31] Eddie Schwalb and Lluís Vila. Temporal constraints: A survey. *Constraints*, 3(2/3):129–149, 1998.

[32] Julie A. Shah, John Stedl, Brian C. Williams, and Paul Robertson. A fast incremental algorithm for maintaining dispatchability of partially controllable plans. In Mark S. Boddy, Maria Fox, and Sylvie ThiÃ©baux, editors, *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 296–303. AAAI Press, 2007. ISBN 978-1-57735-344-7. URL `http://dblp.uni-trier.de/db/conf/aips/icaps2007.html#ShahSWR07`.

[33] John Stedl and Brian Williams. A fast incremental dynamic controllability algorithm. In *Proceedings of the ICAPS Workshop on Plan Execution: A Reality Check*, 2005.

[34] John L. Stedl. Managing temporal uncertainty under limited communication: A formal model of tight and loose team coordination. Master's thesis, Massachusetts Institute of Technology, 2004.

[35] K. Stergiou and M. Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence*, 120(1):81–117, 2000.

[36] I. Tsamardinos. Reformulating temporal plans for efficient execution. *Master's thesis, University of Pittsburgh*, 2000.

[37] I. Tsamardinos, M.E. Pollack, and S. Ramakrishnan. Assessing the probability of legal execution of plans with temporal uncertainty. In

Proceedings of ICAPS'03 Workshop on Planning Under Uncertainty and Incomplete Information, 2003.

[38] I. Tsamardinos, T. Vidal, and M.E. Pollack. CTP: A new constraint-based formalism for conditional, temporal planning. *Constraints*, 8(4): 365–388, 2003.

[39] Ioannis Tsamardinos, Nicola Muscettola, and Paul Morris. Fast transformation of temporal plans for efficient execution. In *Proceedings of the 15th National Conference on Artificial Intelligence / 10th Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI)*, pages 254–261, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence. ISBN 0-262-51098-7.

[40] K. Brent Venable, Michele Volpato, Bart Peintner, and Neil Yorke-Smith. Weak and dynamic controllability of temporal problems with disjunctions and uncertainty. In *COPLAS 2010: ICAPS Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*, 2010.

[41] K.B. Venable and N. Yorke-Smith. Disjunctive temporal planning with uncertainty. In *19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1721–22, 2005.

[42] T. Vidal and H. Fargier. Contingent durations in temporal CSPs: From consistency to controllabilities. In *Proceedings of the 4th International Workshop on Temporal Representation and Reasoning (TIME)*, page 78, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7937-9.

[43] Thierry Vidal and Hèléne Fargier. Handling contingency in temporal constraint networks: from consistency to controllabilities. *Journal of Experimental and Theoretical Artificial Intelligence*, 11:23–45, 1998.

[44] Thierry Vidal and M. Ghallab. Dealing with uncertain durations in temporal constraints networks dedicated to planning. In *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI)*, pages 48–52, 1996.

[45] L. Vila. A survey on temporal reasoning in artificial intelligence. *Ai Communications*, 7(1):4–28, 1994.

[46] Marc B Vilain and Henry A Kautz. Constraint propagation algorithms for temporal reasoning. In *AAAI*, volume 86, pages 377–382, 1986.

[47] Lin Xu and Berthe Y. Choueiry. A new efficient algorithm for solving the simple temporal problem. In *10th International Symposium on Temporal Representation and Reasoning / 4th International Conference on Temporal Logic (TIME-ICTL 2003), 8-10 July 2003, Cairns, Queensland, Australia*, page 212, 2003. doi: 10.1109/TIME.2003.1214898. URL http://dx.doi.org/10.1109/TIME.2003.1214898.

Department of Computer and Information Science
Linköpings universitet

Licentiate Theses

Linköpings Studies in Science and Technology
Faculty of Arts and Sciences

No 17 **Vojin Plavsic:** Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)

No 28 **Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.

No 29 **Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.

No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.

No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.

No 60 **Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.

No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.

No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.

No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs, 1986.

No 74 **Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.

No 104 **Shamsul I. Chowdhury: S**tatistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.

No 108 **Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.

No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.

No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.

No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.

No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.

No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.

No 139 **Christer Bäckström:** Reasoning about Interdependent Actions, 1988.

No 140 **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.

No 146 **Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.

No 150 **Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.

No 165 **Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.

No 166 **Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.

No 174 **Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.

No 177 **Peter Åberg:** Design of a Multiple View Presentation and Interaction Manager, 1989.

No 181 **Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.

No 184 **Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.

No 187 **Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.

No 189 **Magnus Merkel:** Temporal Information in Natural Language, 1989.

No 196 **Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.

No 197 **Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.

No 203 **Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.

No 212 **Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.

No 230 **Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.

No 237 **Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.

No 250 **Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.

No 253 **Torbjörn Näslund:** SLDFA-Resolution - Computing Answers for Negative Queries, 1990.

No 260 **Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.

No 283 **Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge- Bases, 1991.

No 298 **Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.

No 318 **Lena Strömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.

No 319 **Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.

No 326 **Andreas Kågedal:** Logic Programming with External Procedures: an Implementation, 1992.

No 328 **Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.

No 333 **Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.

No 335 **Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Systems, 1992.

No 348 **Ulf Cederling:** Industrial Software Development - a Case Study, 1992.

No 352 **Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.

No 371 **Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.

No 378 **Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.

No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.
No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.
No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.
No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.
No 398 **Johan Boye:** Dependency-based Groudness Analysis of Functional Logic Programs, 1993.
No 402 **Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
No 406 **Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
No 414 **Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agent-teoretiskt perspektiv, 1994.
No 417 **Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
No 436 **Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.
No 437 **Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
No 440 **Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
FHS 3/94 **Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
FHS 4/94 **Karin Pettersson:** Informationssystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
No 441 **Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
No 446 **Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
No 450 **Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
No 451 **Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
No 452 **Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
No 455 **Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
FHS 5/94 **Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
No 462 **Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
No 463 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
No 464 **Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
No 469 **Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
No 473 **Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
No 475 **Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
No 476 **Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. - Fallstudier i tre byggföretag, 1995.
No 478 **Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
FHS 7/95 **Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylprogram, 1995.
No 482 **Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
No 488 **Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
No 489 **Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
No 497 **Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
No 498 **Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
No 503 **Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
FHS 8/95 **Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
FHS 9/95 **Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
No 513 **Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
No 517 **Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
No 518 **Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
No 522 **Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
No 538 **Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 545 **Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
No 546 **Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.
FiF-a 1/96 **Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
No 549 **Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
No 550 **Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.
No 557 **Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
No 558 **Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
No 561 **Anders Ekman:** Exploration of Polygonal Environments, 1996.
No 563 **Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.

No 567 **Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
No 575 **Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
No 576 **Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
No 587 **Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
No 589 **Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
No 591 **Niclas Wahllöf:** A Default Extension to Description Logics and its Applications, 1996.
No 595 **Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
No 597 **Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.
No 598 **Rego Granlund:** C^3Fire - A Microworld Supporting Emergency Management Training, 1997.
No 599 **Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997.
No 607 **Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997.
No 609 **Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997.
FiF-a 4 **Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997.
FiF-a 6 **Tommy Wedlund**: Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997.
No 615 **Silvia Coradeschi**: A Decision-Mechanism for Reactive and Coordinated Agents, 1997.
No 623 **Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997.
No 626 **David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997.
No 627 **Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997.
No 629 **Gunilla Ivefors:** Krigsspel och Informationsteknik inför en oförutsägbar framtid, 1997**.**
No 631 **Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997
No 639 **Jukka Mäki-Turja:**. Smalltalk - a suitable Real-Time Language, 1997.
No 640 **Juha Takkinen**: CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997.
No 643 **Man Lin**: Formal Analysis of Reactive Rule-based Programs, 1997.
No 653 **Mats Gustafsson**: Bringing Role-Based Access Control to Distributed Systems, 1997.
FiF-a 13 **Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997.
No 674 **Marcus Bjäreland:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998.
No 676 **Jan Håkegård**: Hierarchical Test Architecture and Board-Level Test Controller Synthesis, 1998.
No 668 **Per-Ove Zetterlund**: Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
No 675 **Jimmy Tjäder**: Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998.
FiF-a 14 **Ulf Melin**: Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998.
No 695 **Tim Heyer**: COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998.
No 700 **Patrik Hägglund:** Programming Languages for Computer Algebra, 1998.
FiF-a 16 **Marie-Therese Christiansson:** Inter-organisatorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
No 712 **Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998.
No 719 **Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998.
No 723 **Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999.
No 725 **Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998.
No 730 **Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998.
No 731 **Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998.
No 733 **Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998.
No 734 **Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998.
FiF-a 21 **Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999.
FiF-a 22 **Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998.
No 737 **Jonas Mellin:** Predictable Event Monitoring, 1998.
No 738 **Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998.
FiF-a 25 **Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998.
No 742 **Pawel Pietrzak:** Static Incorrectness Diagnosis of CLP (FD), 1999.
No 748 **Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999.
No 751 **Anders Ferntoft:** Elektronisk affärskommunikation - kontaktkostnader och kontaktprocesser mellan kunder och leverantörer på producentmarknader, 1999.
No 752 **Jo Skåmedal:** Arbete på distans och arbetsformens påverkan på resor och resmönster, 1999.
No 753 **Johan Alvehus:** Mötets metaforer. En studie av berättelser om möten, 1999.

No 754 **Magnus Lindahl:** Bankens villkor i låneavtal vid kreditgivning till högt belånade företagsförvärv: En studie ur ett agentteoretiskt perspektiv, 2000.
No 766 **Martin V. Howard:** Designing dynamic visualizations of temporal data, 1999.
No 769 **Jesper Andersson:** Towards Reactive Software Architectures, 1999.
No 775 **Anders Henriksson:** Unique kernel diagnosis, 1999.
FiF-a 30 **Pär J. Ågerfalk:** Pragmatization of Information Systems - A Theoretical and Methodological Outline, 1999.
No 787 **Charlotte Björkegren:** Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
No 788 **Håkan Nilsson:** Informationsteknik som drivkraft i granskningsprocessen - En studie av fyra revisionsbyråer, 2000.
No 790 **Erik Berglund:** Use-Oriented Documentation in Software Development, 1999.
No 791 **Klas Gäre:** Verksamhetsförändringar i samband med IS-införande, 1999.
No 800 **Anders Subotic:** Software Quality Inspection, 1999.
No 807 **Svein Bergum**: Managerial communication in telework, 2000.
No 809 **Flavius Gruian:** Energy-Aware Design of Digital Systems, 2000.
FiF-a 32 **Karin Hedström:** Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000.
No 808 **Linda Askenäs:** Affärssystemet - En studie om teknikens aktiva och passiva roll i en organisation, 2000.
No 820 **Jean Paul Meynard:** Control of industrial robots through high-level task programming, 2000.
No 823 **Lars Hult:** Publika Gränsytor - ett designexempel, 2000.
No 832 **Paul Pop:** Scheduling and Communication Synthesis for Distributed Real-Time Systems, 2000.
FiF-a 34 **Göran Hultgren:** Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000.
No 842 **Magnus Kald:** The role of management control systems in strategic business units, 2000.
No 844 **Mikael Cäker:** Vad kostar kunden? Modeller för intern redovisning, 2000.
FiF-a 37 **Ewa Braf**: Organisationers kunskapsverksamheter - en kritisk studie av ”knowledge management”, 2000.
FiF-a 40 **Henrik Lindberg:** Webbaserade affärsprocesser - Möjligheter och begränsningar, 2000.
FiF-a 41 **Benneth Christiansson:** Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000.
No. 854 **Ola Pettersson:** Deliberation in a Mobile Robot, 2000.
No 863 **Dan Lawesson**: Towards Behavioral Model Fault Isolation for Object Oriented Control Systems, 2000.
No 881 **Johan Moe:** Execution Tracing of Large Distributed Systems, 2001.
No 882 **Yuxiao Zhao:** XML-based Frameworks for Internet Commerce and an Implementation of B2B e-procurement, 2001.
No 890 **Annika Flycht-Eriksson:** Domain Knowledge Management in Information-providing Dialogue systems, 2001.
FiF-a 47 **Per-Arne Segerkvis**t: Webbaserade imaginära organisationers samverkansformer: Informationssystemarkitektur och aktörssamverkan som förutsättningar för affärsprocesser, 2001.
No 894 **Stefan Svarén:** Styrning av investeringar i divisionaliserade företag - Ett koncernperspektiv, 2001.
No 906 **Lin Han**: Secure and Scalable E-Service Software Delivery, 2001.
No 917 **Emma Hansson:** Optionsprogram för anställda - en studie av svenska börsföretag, 2001.
No 916 **Susanne Odar:** IT som stöd för strategiska beslut, en studie av datorimplementerade modeller av verksamhet som stöd för beslut om anskaffning av JAS 1982, 2002.
FiF-a-49 **Stefan Holgersson:** IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.
FiF-a-51 **Per Oscarsson:** Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001.
No 919 **Luis Alejandro Cortes:** A Petri Net Based Modeling and Verification Technique for Real-Time Embedded Systems, 2001.
No 915 **Niklas Sandell:** Redovisning i skuggan av en bankkris - Värdering av fastigheter. 2001.
No 931 **Fredrik Elg:** Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002.
No 933 **Peter Aronsson:** Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002.
No 938 **Bourhane Kadmiry**: Fuzzy Control of Unmanned Helicopter, 2002.
No 942 **Patrik Haslum**: Prediction as a Knowledge Representation Problem: A Case Study in Model Design, 2002.
No 956 **Robert Sevenius:** On the instruments of governance - A law & economics study of capital instruments in limited liability companies, 2002.
FiF-a 58 **Johan Petersson:** Lokala elektroniska marknadsplatser - informationssystem för platsbundna affärer, 2002.
No 964 **Peter Bunus:** Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.
No 973 **Gert Jervan:** High-Level Test Generation and Built-In Self-Test Techniques for Digital Systems, 2002.
No 958 **Fredrika Berglund:** Management Control and Strategy - a Case Study of Pharmaceutical Drug Development, 2002.
FiF-a 61 **Fredrik Karlsson:** Meta-Method for Method Configuration - A Rational Unified Process Case, 2002.
No 985 **Sorin Manolache:** Schedulability Analysis of Real-Time Systems with Stochastic Task Execution Times, 2002.
No 982 **Diana Szentiványi:** Performance and Availability Trade-offs in Fault-Tolerant Middleware, 2002.
No 989 **Iakov Nakhimovski:** Modeling and Simulation of Contacting Flexible Bodies in Multibody Systems, 2002.
No 990 **Levon Saldamli:** PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations, 2002.
No 991 **Almut Herzog:** Secure Execution Environment for Java Electronic Services, 2002.

No 999 **Jon Edvardsson:** Contributions to Program- and Specification-based Test Data Generation, 2002.
No 1000 **Anders Arpteg:** Adaptive Semi-structured Information Extraction, 2002.
No 1001 **Andrzej Bednarski:** A Dynamic Programming Approach to Optimal Retargetable Code Generation for Irregular Architectures, 2002.
No 988 **Mattias Arvola:** Good to use! : Use quality of multi-user applications in the home, 2003.
FiF-a 62 **Lennart Ljung:** Utveckling av en projektivitetsmodell - om organisationers förmåga att tillämpa projektarbetsformen, 2003.
No 1003 **Pernilla Qvarfordt:** User experience of spoken feedback in multimodal interaction, 2003.
No 1005 **Alexander Siemers:** Visualization of Dynamic Multibody Simulation With Special Reference to Contacts, 2003.
No 1008 **Jens Gustavsson:** Towards Unanticipated Runtime Software Evolution, 2003.
No 1010 **Calin Curescu:** Adaptive QoS-aware Resource Allocation for Wireless Networks, 2003.
No 1015 **Anna Andersson:** Management Information Systems in Process-oriented Healthcare Organisations, 2003.
No 1018 **Björn Johansson:** Feedforward Control in Dynamic Situations, 2003.
No 1022 **Traian Pop:** Scheduling and Optimisation of Heterogeneous Time/Event-Triggered Distributed Embedded Systems, 2003.
FiF-a 65 **Britt-Marie Johansson:** Kundkommunikation på distans - en studie om kommunikationsmediets betydelse i affärstransaktioner, 2003.
No 1024 **Aleksandra Tešanovic:** Towards Aspectual Component-Based Real-Time System Development, 2003.
No 1034 **Arja Vainio-Larsson**: Designing for Use in a Future Context - Five Case Studies in Retrospect, 2003.
No 1033 **Peter Nilsson:** Svenska bankers redovisningsval vid reservering för befarade kreditförluster - En studie vid införandet av nya redovisningsregler, 2003.
FiF-a 69 **Fredrik Ericsson:** Information Technology for Learning and Acquiring of Work Knowledge, 2003.
No 1049 **Marcus Comstedt:** Towards Fine-Grained Binary Composition through Link Time Weaving, 2003.
No 1052 **Åsa Hedenskog:** Increasing the Automation of Radio Network Control, 2003.
No 1054 **Claudiu Duma:** Security and Efficiency Tradeoffs in Multicast Group Key Management, 2003.
FiF-a 71 **Emma Eliason:** Effektanalys av IT-systems handlingsutrymme, 2003.
No 1055 **Carl Cederberg:** Experiments in Indirect Fault Injection with Open Source and Industrial Software, 2003.
No 1058 **Daniel Karlsson:** Towards Formal Verification in a Component-based Reuse Methodology, 2003.
FiF-a 73 **Anders Hjalmarsson:** Att etablera och vidmakthålla förbättringsverksamhet - behovet av koordination och interaktion vid förändring av systemutvecklingsverksamheter, 2004.
No 1079 **Pontus Johansson:** Design and Development of Recommender Dialogue Systems, 2004.
No 1084 **Charlotte Stoltz:** Calling for Call Centres - A Study of Call Centre Locations in a Swedish Rural Region, 2004.
FiF-a 74 **Björn Johansson:** Deciding on Using Application Service Provision in SMEs, 2004.
No 1094 **Genevieve Gorrell:** Language Modelling and Error Handling in Spoken Dialogue Systems, 2004.
No 1095 **Ulf Johansson:** Rule Extraction - the Key to Accurate and Comprehensible Data Mining Models, 2004.
No 1099 **Sonia Sangari:** Computational Models of Some Communicative Head Movements, 2004.
No 1110 **Hans Nässla:** Intra-Family Information Flow and Prospects for Communication Systems, 2004.
No 1116 **Henrik Sällberg:** On the value of customer loyalty programs - A study of point programs and switching costs, 2004.
FiF-a 77 **Ulf Larsson:** Designarbete i dialog - karaktärisering av interaktionen mellan användare och utvecklare i en systemutvecklingsprocess, 2004.
No 1126 **Andreas Borg:** Contribution to Management and Validation of Non-Functional Requirements, 2004.
No 1127 **Per-Ola Kristensson:** Large Vocabulary Shorthand Writing on Stylus Keyboard, 2004.
No 1132 **Pär-Anders Albinsson:** Interacting with Command and Control Systems: Tools for Operators and Designers, 2004.
No 1130 **Ioan Chisalita:** Safety-Oriented Communication in Mobile Networks for Vehicles, 2004.
No 1138 **Thomas Gustafsson:** Maintaining Data Consistency in Embedded Databases for Vehicular Systems, 2004.
No 1149 **Vaida Jakonienė:** A Study in Integrating Multiple Biological Data Sources, 2005.
No 1156 **Abdil Rashid Mohamed**: High-Level Techniques for Built-In Self-Test Resources Optimization, 2005.
No 1162 **Adrian Pop:** Contributions to Meta-Modeling Tools and Methods, 2005.
No 1165 **Fidel Vascós Palacios:** On the information exchange between physicians and social insurance officers in the sick leave process: an Activity Theoretical perspective, 2005.
FiF-a 84 **Jenny Lagsten:** Verksamhetsutvecklande utvärdering i informationssystemprojekt, 2005.
No 1166 **Emma Larsdotter Nilsson:** Modeling, Simulation, and Visualization of Metabolic Pathways Using Modelica, 2005.
No 1167 **Christina Keller:** Virtual Learning Environments in higher education. A study of students' acceptance of educational technology, 2005.
No 1168 **Cécile Åberg:** Integration of organizational workflows and the Semantic Web, 2005.
FiF-a 85 **Anders Forsman:** Standardisering som grund för informationssamverkan och IT-tjänster - En fallstudie baserad på trafikinformationstjänsten RDS-TMC, 2005.
No 1171 **Yu-Hsing Huang:** A systemic traffic accident model, 2005.
FiF-a 86 **Jan Olausson:** Att modellera uppdrag - grunder för förståelse av processinriktade informationssystem i transaktionsintensiva verksamheter, 2005.
No 1172 **Petter Ahlström**: Affärsstrategier för seniorbostadsmarknaden, 2005.
No 1183 **Mathias Cöster**: Beyond IT and Productivity - How Digitization Transformed the Graphic Industry, 2005.
No 1184 **Åsa Horzella**: Beyond IT and Productivity - Effects of Digitized Information Flows in Grocery Distribution, 2005.
No 1185 **Maria Kollberg**: Beyond IT and Productivity - Effects of Digitized Information Flows in the Logging Industry, 2005.
No 1190 **David Dinka**: Role and Identity - Experience of technology in professional settings, 2005.

No 1191 **Andreas Hansson**: Increasing the Storage Capacity of Recursive Auto-associative Memory by Segmenting Data, 2005.
No 1192 **Nicklas Bergfeldt:** Towards Detached Communication for Robot Cooperation, 2005.
No 1194 **Dennis Maciuszek:** Towards Dependable Virtual Companions for Later Life, 2005.
No 1204 **Beatrice Alenljung:** Decision-making in the Requirements Engineering Process: A Human-centered Approach, 2005.
No 1206 **Anders Larsson:** System-on-Chip Test Scheduling and Test Infrastructure Design, 2005.
No 1207 **John Wilander:** Policy and Implementation Assurance for Software Security, 2005.
No 1209 **Andreas Käll:** Översättningar av en managementmodell - En studie av införandet av Balanced Scorecard i ett landsting, 2005.
No 1225 **He Tan:** Aligning and Merging Biomedical Ontologies, 2006.
No 1228 **Artur Wilk:** Descriptive Types for XML Query Language Xcerpt, 2006.
No 1229 **Per Olof Pettersson:** Sampling-based Path Planning for an Autonomous Helicopter, 2006.
No 1231 **Kalle Burbeck:** Adaptive Real-time Anomaly Detection for Safeguarding Critical Networks, 2006.
No 1233 **Daniela Mihailescu:** Implementation Methodology in Action: A Study of an Enterprise Systems Implementation Methodology, 2006.
No 1244 **Jörgen Skågeby:** Public and Non-public gifting on the Internet, 2006.
No 1248 **Karolina Eliasson:** The Use of Case-Based Reasoning in a Human-Robot Dialog System, 2006.
No 1263 **Misook Park-Westman:** Managing Competence Development Programs in a Cross-Cultural Organisation - What are the Barriers and Enablers, 2006.
FiF-a 90 **Amra Halilovic:** Ett praktikperspektiv på hantering av mjukvarukomponenter, 2006.
No 1272 **Raquel Flodström**: A Framework for the Strategic Management of Information Technology, 2006.
No 1277 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Embedded Systems, 2006.
No 1283 **Håkan Hasewinkel:** A Blueprint for Using Commercial Games off the Shelf in Defence Training, Education and Research Simulations, 2006.
FiF-a 91 **Hanna Broberg:** Verksamhetsanpassade IT-stöd - Designteori och metod, 2006.
No 1286 **Robert Kaminski:** Towards an XML Document Restructuring Framework, 2006.
No 1293 **Jiri Trnka:** Prerequisites for data sharing in emergency management, 2007.
No 1302 **Björn Hägglund:** A Framework for Designing Constraint Stores, 2007.
No 1303 **Daniel Andreasson**: Slack-Time Aware Dynamic Routing Schemes for On-Chip Networks, 2007.
No 1305 **Magnus Ingmarsson:** Modelling User Tasks and Intentions for Service Discovery in Ubiquitous Computing, 2007.
No 1306 **Gustaf Svedjemo**: Ontology as Conceptual Schema when Modelling Historical Maps for Database Storage, 2007.
No 1307 **Gianpaolo Conte**: Navigation Functionalities for an Autonomous UAV Helicopter, 2007.
No 1309 **Ola Leifler**: User-Centric Critiquing in Command and Control: The DKExpert and ComPlan Approaches, 2007.
No 1312 **Henrik Svensson**: Embodied simulation as off-line representation, 2007.
No 1313 **Zhiyuan He:** System-on-Chip Test Scheduling with Defect-Probability and Temperature Considerations, 2007.
No 1317 **Jonas Elmqvist:** Components, Safety Interfaces and Compositional Analysis, 2007.
No 1320 **Håkan Sundblad:** Question Classification in Question Answering Systems, 2007.
No 1323 **Magnus Lundqvist**: Information Demand and Use: Improving Information Flow within Small-scale Business Contexts, 2007.
No 1329 **Martin Magnusson:** Deductive Planning and Composite Actions in Temporal Action Logic, 2007.
No 1331 **Mikael Asplund:** Restoring Consistency after Network Partitions, 2007.
No 1332 **Martin Fransson:** Towards Individualized Drug Dosage - General Methods and Case Studies, 2007.
No 1333 **Karin Camara:** A Visual Query Language Served by a Multi-sensor Environment, 2007.
No 1337 **David Broman:** Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments, 2007.
No 1339 **Mikhail Chalabine:** Invasive Interactive Parallelization, 2007.
No 1351 **Susanna Nilsson:** A Holistic Approach to Usability Evaluations of Mixed Reality Systems, 2008.
No 1353 **Shanai Ardi:** A Model and Implementation of a Security Plug-in for the Software Life Cycle, 2008.
No 1356 **Erik Kuiper:** Mobility and Routing in a Delay-tolerant Network of Unmanned Aerial Vehicles, 2008.
No 1359 **Jana Rambusch**: Situated Play, 2008.
No 1361 **Martin Karresand:** Completing the Picture - Fragments and Back Again, 2008.
No 1363 **Per Nyblom:** Dynamic Abstraction for Interleaved Task Planning and Execution, 2008.
No 1371 **Fredrik Lantz:** Terrain Object Recognition and Context Fusion for Decision Support, 2008.
No 1373 **Martin Östlund:** Assistance Plus: 3D-mediated Advice-giving on Pharmaceutical Products, 2008.
No 1381 **Håkan Lundvall:** Automatic Parallelization using Pipelining for Equation-Based Simulation Languages, 2008.
No 1386 **Mirko Thorstensson:** Using Observers for Model Based Data Collection in Distributed Tactical Operations, 2008.
No 1387 **Bahlol Rahimi:** Implementation of Health Information Systems, 2008.
No 1392 **Maria Holmqvist:** Word Alignment by Re-using Parallel Phrases, 2008.
No 1393 **Mattias Eriksson**: Integrated Software Pipelining, 2009.
No 1401 **Annika Öhgren:** Towards an Ontology Development Methodology for Small and Medium-sized Enterprises, 2009.
No 1410 **Rickard Holsmark:** Deadlock Free Routing in Mesh Networks on Chip with Regions, 2009.
No 1421 **Sara Stymne:** Compound Processing for Phrase-Based Statistical Machine Translation, 2009.
No 1427 **Tommy Ellqvist**: Supporting Scientific Collaboration through Workflows and Provenance, 2009.
No 1450 **Fabian Segelström:** Visualisations in Service Design, 2010.
No 1459 **Min Bao**: System Level Techniques for Temperature-Aware Energy Optimization, 2010.
No 1466 **Mohammad Saifullah**: Exploring Biologically Inspired Interactive Networks for Object Recognition, 2011

No 1468 **Qiang Liu**: Dealing with Missing Mappings and Structure in a Network of Ontologies, 2011.
No 1469 **Ruxandra Pop**: Mapping Concurrent Applications to Multiprocessor Systems with Multithreaded Processors and Network on Chip-Based Interconnections, 2011.
No 1476 **Per-Magnus Olsson**: Positioning Algorithms for Surveillance Using Unmanned Aerial Vehicles, 2011.
No 1481 **Anna Vapen**: Contributions to Web Authentication for Untrusted Computers, 2011.
No 1485 **Loove Broms:** Sustainable Interactions: Studies in the Design of Energy Awareness Artefacts, 2011.
FiF-a 101 **Johan Blomkvist:** Conceptualising Prototypes in Service Design, 2011.
No 1490 **Håkan Warnquist:** Computer-Assisted Troubleshooting for Efficient Off-board Diagnosis, 2011.
No 1503 **Jakob Rosén:** Predictable Real-Time Applications on Multiprocessor Systems-on-Chip, 2011.
No 1504 **Usman Dastgeer:** Skeleton Programming for Heterogeneous GPU-based Systems, 2011.
No 1506 **David Landén:** Complex Task Allocation for Delegation: From Theory to Practice, 2011.
No 1507 **Kristian Stavåker**: Contributions to Parallel Simulation of Equation-Based Models on Graphics Processing Units, 2011.
No 1509 **Mariusz Wzorek:** Selected Aspects of Navigation and Path Planning in Unmanned Aircraft Systems, 2011.
No 1510 **Piotr Rudol:** Increasing Autonomy of Unmanned Aircraft Systems Through the Use of Imaging Sensors, 2011.
No 1513 **Anders Carstensen:** The Evolution of the Connector View Concept: Enterprise Models for Interoperability Solutions in the Extended Enterprise, 2011.
No 1523 **Jody Foo:** Computational Terminology: Exploring Bilingual and Monolingual Term Extraction, 2012.
No 1550 **Anders Fröberg:** Models and Tools for Distributed User Interface Development, 2012.
No 1558 **Dimitar Nikolov:** Optimizing Fault Tolerance for Real-Time Systems, 2012.
No 1582 **Dennis Andersson:** Mission Experience: How to Model and Capture it to Enable Vicarious Learning, 2013.
No 1586 **Massimiliano Raciti:** Anomaly Detection and its Adaptation: Studies on Cyber-physical Systems, 2013.
No 1588 **Banafsheh Khademhosseinieh:** Towards an Approach for Efficiency Evaluation of Enterprise Modeling Methods, 2013.
No 1589 **Amy Rankin:** Resilience in High Risk Work: Analysing Adaptive Performance, 2013.
No 1592 **Martin Sjölund:** Tools for Understanding, Debugging, and Simulation Performance Improvement of Equation-Based Models, 2013.
No 1606 **Karl Hammar:** Towards an Ontology Design Pattern Quality Model, 2013.
No 1624 **Maria Vasilevskaya:** Designing Security-enhanced Embedded Systems: Bridging Two Islands of Expertise, 2013.
No 1627 **Ekhiotz Vergara:** Exploiting Energy Awareness in Mobile Communication, 2013.
No 1644 **Valentina Ivanova:** Integration of Ontology Alignment and Ontology Debugging for Taxonomy Networks, 2014.
No 1647 **Dag Sonntag:** A Study of Chain Graph Interpretations, 2014.
No 1657 **Kiril Kiryazov:** Grounding Emotion Appraisal in Autonomous Humanoids, 2014.
No 1683 **Zlatan Dragisic:** Completing the Is-a Structure in Description Logics Ontologies, 2014.
No 1688 **Erik Hansson:** Code Generation and Global Optimization Techniques for a Reconfigurable PRAM-NUMA Multicore Architecture, 2014.
No 1715 **Nicolas Melot:** Energy-Efficient Computing over Streams with Massively Parallel Architectures, 2015.
No 1716 **Mahder Gebremedhin:** Automatic and Explicit Parallelization Approaches for Mathematical Simulation Models, 2015.
No 1722 **Mikael Nilsson:** Efficient Temporal Reasoning with Uncertainty, 2015.