ETH zürich

ETH Library

Modelling and Verification of the Timely Dataflow Progress Tracking Protocol

Master Thesis

Author(s):
Decova, Sára

Publication date:
2020

Permanent link:
https://doi.org/10.3929/ethz-b-000444762

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 285

Systems Group, Department of Computer Science, ETH Zurich

Modelling and Verification of the Timely Dataflow Progress Tracking Protocol

by

Sara Decova

Supervised by

Prof. Dr. Timothy Roscoe,
Andrea Lattuada,
Dr. Dmitriy Traytel

November 2019 – May 2020

inf | Informatik
Computer Science

Abstract

Timely Dataflow is a distributed, data-parallel, state-of-the-art dataflow runtime. Its core coordination component, termed *progress tracking*, provides dataflow operators with guarantees about pending and completed phases of computation. This allows asynchronous operators to make safe time-based decisions on streams containing delayed and out-of-order data.

The design and implementation of correct distributed protocols are notoriously difficult and error-prone. However, the Timely Dataflow *progress tracking* protocol has never been fully formalised despite its distributed nature and the central role it plays in ensuring the system's correctness.

In this thesis, we extend the existing work of Abadi et al. [2, 1], which only models the exchange of progress information between workers, to a verified model of the complete coordination protocol. We determine the protocol's main safety property and describe its mechanically verified proof. This formalisation provides a clear interface for other components of Timely Dataflow: most of the system interacts or depend on coordination information. Furthermore, we believe that the specification will prove helpful in integrating the protocol in new systems.

We introduce an executable version of the model of the Progress Tracking protocol and a comparative testing framework between the formal model and the Rust implementation of Timely Dataflow to validate our model and to ensure the correctness of the Rust implementation. The framework is flexible enough to work with any valid Timely Dataflow program, including iterative and hierarchical dataflows, and can be extended to arbitrary time and location types satisfying the assumptions made by the model.

Acknowledgements

I would like to thank Prof. Timothy Roscoe for the opportunity to work on this project in the ETH Systems group.

Secondly, I would like to express my deepest gratitude to Andrea Lattuada and Dr. Dmitriy Traytel for offering me the opportunity to work on the formalisation of the Timely Dataflow and for closely supervising every step towards the completion of this thesis. All findings presented in this thesis are the result of long hours of collaboration, countless inspiring conversations and their expert guidance. I am especially grateful to Andrea Lattuada for sharing his detailed knowledge of the Timely Dataflow and other stream-processing frameworks. I would also like to thank Dmitriy Traytel for introducing me to the Isabelle proof assistant and the addictive nature of theorem proving.

This work has been developed in collaboration with a fellow student Matthias Brun who has significantly contributed to the formalisation of the Progress Tracking protocol and has written the majority of the mechanically verified proofs of its invariants. I am especially thankful for his willingness to discuss proof strategies and his patience with me while learning the secrets of Isabelle.

I would also like to thank Dr. David Cock who has taught the course on Informal Methods at ETH and has introduced me to formal verification in the first place.

Contents

Contents iii

1 Introduction 1
1.1 Coordination in dataflow systems . . . 1
1.2 Timely Dataflow and formal verification . . . 2
1.3 Overview and main contributions . . . 3

2 Background 5
2.1 Dataflow . . . 5
2.1.1 Progress tracking in Dataflow Systems . . . 6
2.2 Timely Dataflow . . . 7
2.2.1 Summaries . . . 8
2.2.2 Capabilities . . . 8
2.2.3 Progress Tracking . . . 9
2.3 Modelling systems with finite automata . . . 10
2.4 Existing Models of Timely Dataflow . . . 10
2.4.1 Abadi and Isard's Model of timely dataflow . . . 11
2.4.2 Naiad Clock Protocol . . . 13
2.5 Formal Verification . . . 13
2.5.1 Proof assistants . . . 14

3 Isabelle/HOL Syntax Overview 17
3.1 Types, terms and basic logic . . . 17
3.2 Function definitions and lemmas . . . 18
3.3 Locales . . . 20

4 Propagation Algorithm: Model 23
4.1 Basic definitions . . . 23
4.1.1 Partial order on timestamps . . . 23
4.1.2 Antichain . . . 24

4.1.3 Signed multiset 24
4.1.4 Infimum of signed multiset 24
4.2 Topology 25
4.2.1 Assumptions on basic types 25
4.2.2 Assumptions on the graph topology 25
4.3 State 26
4.4 The algorithm 27
4.4.1 The next_change_multiplicity action 28
4.4.2 The next_propagate action 30
4.5 Specification 31
4.5.1 Propagation Rounds 33
4.6 Chapter Summary 33

5 Propagation Algorithm: Safety **35**
5.1 Safety Property 35
5.2 Informal proof of the safety property 37
5.2.1 Proof of the safety property 38
5.3 Chapter Summary 40

6 Protocol **43**
6.1 The Exchange algorithm 43
6.2 Basic Types 44
6.3 State 45
6.4 The protocol 46
6.4.1 The next_performop action 46
6.4.2 The next_send_update action 46
6.4.3 The next_recv_update action 46
6.4.4 The next_propagate action 48
6.5 Specification 48
6.6 Chapter Summary 49

7 Safety **51**
7.1 Safety Property 51
7.2 Informal proof of the safety property 52
7.2.1 Proof of the safety property 53
7.3 Chapter Summary 54

8 Comparative Testing **55**
8.1 Overview 55
8.2 Logging 56
8.3 Generation of Isabelle theory files 57
8.4 Executable Isabelle model 59
8.5 Supported types 61
8.6 Comparative testing 62

8.7 Retrospection on Isabelle for executable code 63
8.8 Chapter Summary 65

9 Conclusion 67
9.1 Main contributions 67
9.2 Ongoing work 68
9.3 Future work 69

A Example of a non-terminating propagation round 71

B Formal specification of the Exchange algorithm in Isabelle 75

Bibliography 79

Chapter 1

Introduction

A streaming dataflow system is characterised by the absence of a central point of control. Instead, it consists of asynchronous components (*operators*) each acting in response to the arrival of data, and the connections between these components. As new data arrives, gets processed, and leaves these operators, work can take place at multiple operators across the dataflow in parallel. The flexibility of this setting brings large performance benefits. On the other hand, it increases the complexity of the system, which becomes responsible for providing coordination information to many asynchronous actors.

1.1 Coordination in dataflow systems

To demonstrate the need for coordination, consider a *window* operator that computes the number of people who have logged into their e-mail account for every 5 minute interval. To associate data with different time intervals, each data message carries a logical *timestamp*. However, due to network traffic, communication delays or scheduling strategies, data is not guaranteed to arrive at the operator in logical order or at a predictable time. In order for the operator to produce correct output and reclaim any memory associated with a certain 5-minute interval, the operator needs to know if all data for a given 5-minute interval has been received or whether some of it has been delayed.

In general, operators have to be able to reason locally about which phase of computation has been completed for the sake of overall correctness and efficiency. Commonly, dataflow systems achieve this by providing each operator with a *frontier* - a lower bound on all timestamps an operator may receive in future. The operators in turn communicate to the system the lower bounds of timestamps they may yet produce as output. The goal of the system's coordination component is to continuously recompute correct

frontiers for all operators from the sent and received messages as well as the lower bounds provided by the operators.

A verified coordination component provides a sound basis for building correct operator code and dataflow programs. When we combine the guarantees of a verified coordination component with an assumption (or proof) that the operator logic is sound we can ensure the correctness of complete dataflow programs. To illustrate this, consider the *window* operator introduced earlier. If we assume that the operator produces output for a certain 5-minute window only if its *frontier* has passed the associated time interval, then a verified coordination component ensures that this operator will never output premature, possibly inaccurate, results.

1.2 Timely Dataflow and formal verification

Our research is focused around an efficient, state-of-the-art, distributed, data-parallel dataflow runtime, called Timely Dataflow [30] and its core coordination component termed *progress tracking*. The Progress Tracking protocol consists of two components:

1. **Exchange:** A distributed, conservative reference counter that keeps track of outstanding logical timestamps in the system by modelling the exchange of progress information between workers.

2. **Propagation:** A propagation algorithm that continuously recomputes *frontiers* from the outstanding timestamps.

In practice, the protocol is implemented in the Rust [28] programming language. Yet, only the first component - a distributed reference counter - has been formally modelled and verified [2]. Due to the distributed nature of the dataflow paradigm, it is more challenging to devise tests and simulations that effectively catch corner cases potentially resulting in obscure race conditions, deadlocks or livelocks.

Furthermore, we noticed that the protocol makes a series of assumptions which are necessary for the protocol's correctness but are not obvious. The limited understanding of the protocol allows a developer to introduce unnecessary new bugs into the system affecting the system's overall correctness. On the contrary, formalised protocol assumptions and safety properties allow the developer writing custom operator or dataflow code to reason about its correctness and verify its implementation in a sound way.

Finally, we believe that a formal specification makes it easier to adopt the Progress Tracking protocol to other frameworks.

1.3 Overview and main contributions

In this thesis, we present a verified model of the Timely Dataflow Progress Tracking protocol motivated by its fundamental role in Timely Dataflow's reliability and high performance. The main contributions of this work are:

- A formal specification of the protocol's Propagation algorithm and its safety property in the Isabelle proof assistant.
- An executable model of the Propagation algorithm in the Isabelle proof assistant.
- A comparative testing framework between the Rust and the Isabelle executable model of the Propagation algorithm.
- The identification and modelling of the assumptions on the Timely Dataflow Progress Tracking protocol.
- A formal specification of the Timely Dataflow Progress Tracking protocol by combining our model of the Propagation algorithm with the Naiad Clock protocol [2].
- A description of the mechanically-verified safety property of the Timely Dataflow Progress Tracking protocol.

Concretely, the work is presented as follows:

- Chapter 2 gives an overview of the existing progress tracking practice in stream processing frameworks and discusses the specific workings of Timely Dataflow. Next, we summarise past contributions on formalisation of the Timely Dataflow model and discuss how our work builds on existing research. Finally, we discuss formal verification, available tools and argue for the choice of the Isabelle proof assistant.
- Chapter 3 gives an overview of the Isabelle/HOL syntax.
- Chapter 4 presents the formal specification of the Propagation algorithm. Chapter 5 presents the safety property of the Propagation algorithm and describes its mechanically verified proof.
- Chapter 6 presents the formal specification of the complete Timely Dataflow Progress Tracking protocol in terms of the Exchange model [2] and our Propagation algorithm model. Additionally, Chapter 7 states the main safety property of the Progress Tracking protocol and describes its mechanically verified proof.
- Chapter 8 describes the comparative testing framework. We present the executable model of the Propagation algorithm implemented with the Isabelle proof assistant and explain how input to the model is generated from the Timely Dataflow. Finally, we discuss how we

validate the model and test the correctness of the Timely Dataflow Rust implementation.

- Chapter 9 discusses ongoing and future work and presents our conclusions.

Chapter 2

Background and Related Work

This chapter gives an overview of the dataflow paradigm and the existing progress tracking practices in stream processing frameworks. We continue by describing the specific workings of Timely Dataflow. Furthermore, we summarise past contributions to the formalisation of the Timely Dataflow model and discuss how our work builds on the existing research. Finally, we talk about formal verification, the available tools in formal verification and argue the choice of the Isabelle proof assistant.

2.1 Dataflow

Dataflow [35] is a computational model and a software paradigm that expresses a computation as a directed graph consisting of *nodes* representing transformations, and directed *edges* representing communication channels carrying data between pairs of nodes. An example of such a dataflow graph is depicted in Figure 2.1. Whenever data arrives to a node along a channel, the node processes it which may update its local state. The node may also produce a sequence of new data which are sent along its outgoing edges.

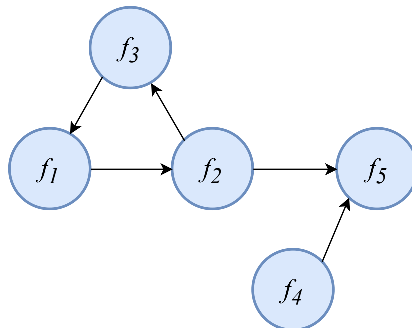


Figure 2.1: Example of a dataflow graph.

The dataflow model is particularly well-suited for concurrent computations, making it a popular choice for data-intensive computations on potentially unbounded streams of unordered data. It has been used in diverse domains ranging from signal processing [25] to data stream processing and analytics. Examples of frameworks profiting from the concurrency opportunities of the dataflow paradigm include Spark Streaming [40], Apache Flink [6], Apache Storm [7] or Naiad [32].

In particular, Spark Streaming supports *task parallelism* which allows multiple transformations to be carried out to disjoint parts of the data stream in parallel. In addition, Naiad and Flink support *pipeline parallelism* with the use of timestamping, discussed in more detail in later sections. In pipeline parallelism, transformations that depend on each other (i.e., are part of the same pipeline) can also be executed concurrently. Looking back at Figure 2.1, task parallelism lets us execute *f1* concurrently with *f4*, while pipeline parallelism allows us to execute *f1* in parallel with any of *f2*, *f3* or *f5*.

2.1.1 Progress tracking in Dataflow Systems

Recent dataflow systems [6, 32] mark data points with *logical time* (also referred to as event time). Logical time refers to the time when the event actually occurred, often even before the data arrives into the system. This is in contrast with *physical time* which refers to the time observed locally. While logical time determines the order in which events have happened, there is no assumed relationship between the logical timestamps and the computer clock at the time of execution.

Even in real-time processing systems, communication delays, scheduling strategies and time spent processing will result in data being delayed and delivered out of order. The general trend, however, is that as data arrives at a particular node in the dataflow graph, the logical times generally increase until eventually the arrival of old timestamps ceases. To ensure that the nodes produce correct output associated with different times and time intervals, the nodes need to know if all data items for a given time has been delivered or if some of it has been delayed.

Stream processing frameworks employ various metrics [4, 39] to bound this delay as a lower bound on future logical timestamps processed by the node. The system's coordination component interacts with nodes and provides them with the lower bound on timestamps they may yet see as input. In turn, the nodes communicate to the system the lower bounds on timestamps they may yet produce as output. The system is responsible for integrating this information with sent and received data messages to continuously recompute correct lower bounds for all nodes.

The following list presents several dataflow frameworks (excluding Timely Dataflow) and shows their approach to dataflow progress tracking.

- **Spark Streaming** [40] models a computation as an acyclic graph with no notion of logical time. Instead, each input to an operator is either complete or incomplete. The system indicates to operators once all of its inputs are complete and in turn, the operators communicate the completion of their outputs.

- **Google Dataflow** [5] and **Apache Flink** [6] model a computation as an acyclic graph. Unlike Spark Streaming, they stamp messages with logical integer timestamps and use watermarking mechanism to signal progress, as introduced in the MillWheel framework [4]. Watermarks are generated at, or directly after, source operators (nodes) and passed to the dataflow. Each operator is tasked with forwarding new watermarks in its output stream based on its internal logic. This leads to watermark ”flowing” through the dataflow, notifying nodes of ongoing progress.

- **Naiad** [32] models a computation as a potentially cyclic dataflow graph. Each data mesage is stamped with a logical partially-ordered timestamp. In Naiad, an operator can request notifications for a specific timestamp. The system delivers a notification to the operator once it is sure that no data bearing this or a lower timestamp will arrive at the operator in future. Operators are themselves tasked with maintaining their views of future timestamps.

2.2 Timely Dataflow

Timely Dataflow [30] is a Rust-based implementation of the dataflow model proposed by Naiad [32]. Like Naiad, it associates each datum with a partially-ordered timestamp, thereby enabling cycles in the dataflow graph. Timely Dataflow achieves data-parallelism by instantiating multiple workers with individual copies of the dataflow graph.

In terms of graph topology, a timely dataflow graph is described by a collection of operators, that each have a number of target (input) and source (output) ports. In our model, we refer to the set of all ports is by *locations*. Source are connected to target ports via external edges. Inside each operator, target and source ports are connected by internal edges. Each internal edge carries a set of mutually incomparable *summaries*. Figure 2.2 presents an example of a timely dataflow graph.

In practise, Timely Dataflow supports nested dataflows such that an operator can be in itself a dataflow graph. For the purposes of this thesis we will

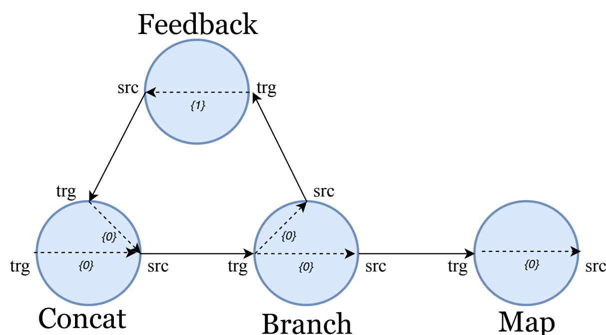


Figure 2.2: Example of timely dataflow graph. Source ports are marked as *src*, and target ports as *trg*. Solid arrows represent external edges while dashed ones represent internal connections. Each internal connection has an associated set of integer summaries.

ignore sub-graphs as they were not incorporated into our model and thus only add unnecessary complexity to our discussions.

2.2.1 Summaries

A *summary* describes an increment to a timestamp. It guarantees a minimal increment along the connection from a target to a source port: i.e., by how much the operator promises to increment each timestamp that flows through this connection.

Most operators have internal connections with summary of zero, which simply indicates that certain target and source port are connected. On the other hand, the *Feedback* operator has one target port t and one source port s, and it guarantees that it will increment any timestamp that flows along it by at least one. Therefore, if a datum with timestamp 3 flows along this connection, it must emerge at the source port with timestamp at least 4.

Summaries become crucial in iterative dataflows. They allow operators to make progress while a part of the data is cycling through a loop an indefinite number of times.

2.2.2 Capabilities

Timely Dataflow *capabilities* (also referred to as *pointstamps* by Naiad [32]) give their owners the ability to create messages with certain timestamps at certain locations. They can be dropped, downgraded, or even generate new capabilities.

Concretely, a *capability* is a location-timestamp pair *(loc,t)*, which gives its owner the ability to produce messages or create new capabilities *(loc′,t′)* such that *(loc,t) could result in (loc′,t′)*. We say that a capability *(loc,t) could-result-in* a capability *(loc′,t′)* if there exists a path from *loc* to *loc′* such that adding all

summaries along the path to the timestamp t results in a timestamp smaller or equal to t'.

Capabilities are the only means of creating messages. As a result, the set of all capabilities in the system determines the set of future timestamps at all locations. To bound the future timestamps arriving at a specific location Timely Dataflow uses the notion of *frontiers*.

To demonstrate the effect of *capabilities* on location *frontiers*, consider the dataflow graph in Figure 2.2. Assume that there exists a capability at the source port of the *Concat* operator with timestamp 5. Moreover, assume there are no other capabilities in the system. Figure 2.3 shows the best-approximated *frontiers* at all ports. For example, *frontiers* at source and target ports of operators *Branch* and *Map* are *5*, because all summaries along the shortest paths are zero. On the other hand, the frontier of the *Feedback*'s source port is *6*, since any message with timestamp 5 produced at the source port of *Concat* must be incremented by at least one before it arrives at the port.

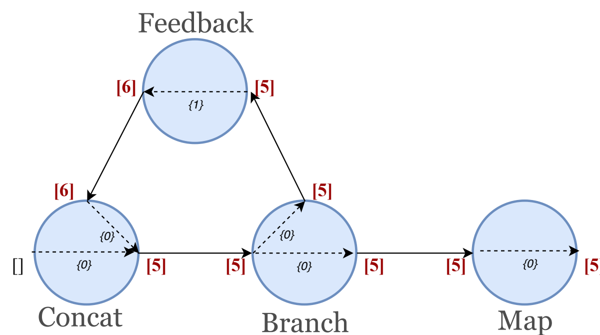


Figure 2.3: Frontiers. Each port in the graph shows an associated best-approximated frontier, given a single capability at the source port of operator *Contact* with timestamp 5.

At the beginning of the dataflow computation, each location is initialised with a set of capabilities bearing the minimal "zero" timestamp. As the computation progresses, locations generate new capabilities, downgrade existing capabilities or drop old capabilities. These are the events that drive computation progress.

2.2.3 Progress Tracking

The goal of the Timely Dataflow Progress Tracking protocol is to recompute locations' frontiers in response to newly created, dropped and downgraded capabilities.

The algorithm computes location frontiers in two phases as stated in the Introduction. In the *Exchange* phase, all workers exchange progress messages

with information about the changes to capabilities they hold. This way, each worker obtains a conservative view of all active capabilities in the system. Secondly, during the *Propagation* phase, each worker recomputes each location's frontier from its own view of system's capabilities.

Unlike Flink, which continually asks operators if their output contains certain timestamps, or Naiad, making operators explicitly request notifications about completed progress phases, this approach minimises the interaction between the system and the operators while still allowing the user to define fine-grained time-based behaviour.

2.3 Modelling systems with finite automata

A finite automaton (or a finite state machine) describes a system as a set of *states* and the transitions (or *actions*) between those states. The state of the system is determined by the values of the *state variables*. Actions are expressed as next-state relations describing the conditions on and the changes to the state variables.

As an example, consider the finite state machine in Figure 2.4 describing the eating habits of a person living exclusively on apples. It consists of fours states, each defined by the values of two state variables, namely *apples* (describing how many apples the person has) and a Boolean flag *hungry*. The finite state machines defines three actions:

- *Buying an apple* (red) increasing the number of apples a person has by one.
- *Eating an apple* (green) decreasing the number of apples a person has by one. This action is conditioned on the person begin hungry and having at least one apple.
- *Getting hungry* (blue) flipping the *hungry* flag to false. This action is conditioned on the person being *not* hungry in the first place.

Finite automata provide an exhaustive and mathematically sound way of modelling and analysing systems. We use a TLA-style state-machine model [22] to formally specify the Progress Tracking protocol. A similar approach has been taken by the existing models of Timely Dataflow introduced in the next section.

2.4 Existing Models of Timely Dataflow

Despite the more formal Naiad model, the development and presentation of Timely Dataflow has been relatively informal, documented partially in code [30], an online book [37] and blog posts [29]. Due to the vast range of applications of Timely Dataflow striving for low-latency and high-throughput

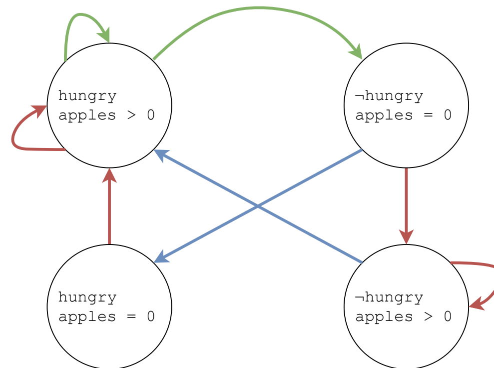


Figure 2.4: Example of a finite state machine.

data-parallel computations, there has been previous work in formalising the Timely Dataflow model.

In this section, we focus on the prior art. Firstly, we discuss the "Timely Dataflow: A Model" [1] by Abadi and Isard which aims to provide a general rigorous definition of timely dataflow. Secondly, we review the Naiad Clock Protocol [2], a formally specified and verified model of the *Exchange* phase of the Timely Dataflow Progress Tracking protocol. In both cases we discuss how the work relates to our objective of modelling Timely Progress Tracking protocol.

2.4.1 Abadi and Isard's Model of timely dataflow

The goal of Aabadi and Isard's model was to provide a "general rigorous definition of timely dataflow" [1] based on the original Naiad dataflow model. It models the computation as an arbitrary - possibly cyclic - graph with *nodes* representing ports, internal edges and external edges. The nodes receive and output messages, receive notifications and produce notification requests. A node that requests a notification asks to be notified when it has received all messages for a given logical time. The model associates each message and notification with a logical timestamp.

The model draws conclusions about the timely dataflow in terms of *pointstamps*, the *could-result-in* relation, and a list of allowed actions.

Pointstamps

A *pointstamp* is a timestamp-location pair *(t,x)* which tells us that a message with timestamp *t* is currently at location *x* (port, external or internal edge).

Could-result-in relation

We say that pointstamp *(x, t) could-result-in* pointstamp *(x′, t′)* if a message at location x and time t may lead to a message at location x' and time t'. Note that our definition of *could-result-in* relation from Section 2.2.1 satisfies this condition. This relation is transitive and anti-symmetric. Moreover, this relation imposes a partial ordering on the set of pointstamps.

Actions

There are four possible actions the algorithm can take at any point. Note that each node in the graph has an associated queue of incoming messages.

1. A message or a notification is enqueued at a source node.
2. A message or a notification is dequeued at a sink node.
3. A node dequeues a message. In turn it can enqueue a set of messages at any of the outgoing neighbouring nodes, and produce notification requests.
4. A node receives a notification. In turn it consumes an outstanding notification request and produces new messages or notification requests.

Naturally, the model imposes further restrictions on the actions, such as what messages can appear at the source nodes or what constitutes producing a valid message and notification request.

Finally, combining the notion of pointstamps, *could-result-in* relation and the list of allowed actions, this work leads to a series of informal proofs of models' properties. One of its main results is the proof of the monotonicity of pointstamps. It states that if a pointstamp p and all pointstamps smaller or equal to p are not present in the system, they will not be introduced again by further actions.

Our work relates to this model in three ways. First, we refine the *could-result-in* relation in terms of timely dataflow graph topology and internal summaries. While the *could-result-in* relation is powerful enough to give strong guarantees about the system, it is unclear how one designs an efficient coordination protocol at this level of abstraction.

Secondly, we omit notifications and notification requests as they are no longer a part of Timely Dataflow. Instead, all nodes passively receive information about future timestamps through *frontiers* and can decide whether to use the information if their internal logic depends on it. Furthermore, while the above model does not explain how the system produces notifications, our model describes how the system computes correct *frontiers*.

Finally, we relax the conditions on receiving a message by a node. This model is too restrictive in a sense that a new message can only be produced

as response to consumed messages. Moreover, the model states that the timestamp of the consumed message and a timestamp of a produced message in each step must satisfy *could-result-in* relation. This is, once again, not representative of Timely Dataflow.

In conclusion, while Abadi and Isard present a neat and powerful model, it has diverged from the Timely Dataflow implementation, imposes a too high level of abstraction to be easily applied and does not describe how notification (or frontiers) are computed by the system. We believe that our model addresses all of these issues.

2.4.2 Naiad Clock Protocol

The Naiad Clock Protocol is a distributed, conservative reference counter that keeps track of outstanding capabilities in the system by modelling the exchange of progress information between workers. It guarantees that each worker holds a safe approximation of the active capabilities in the system at any point during a dataflow computation.

This protocol models the *Exchange* phase of Timely Dataflow Progress Tracking protocol. Our work complements this model by inferring location frontiers at each worker from the approximated views of system's capabilities computed by the Naiad Clock Protocol. Moreover, by combining this protocol with our model of the *Propagation* algorithm, we obtain a complete model of the Timely Dataflow Progress Tracking protocol.

Given that this protocol is a core component of the Progress Tracking protocol modelled in this thesis, we describe it in greater detail later in Section 6.1.

2.5 Formal Verification

Due to the growing scale of present-day software and hardware systems and the complexity of interactions between the systems' components, it has become increasingly more challenging to sufficiently exercise these interactions by a set of tests or simulations. Indeed, the last few decades have seen some alarming software and hardware failures. Gaining in popularity after the famous Pentinum bug [13], formal verification methods have set to tackle this problem.

Formal verification is a form of static analysis involving the construction of rigorous mathematical models of the systems to prove their correctness. In general, creating such a model improves the programmer's understanding of the system while detecting bugs early on in the design cycle. Moreover, incremental changes to the design often require minimal changes to the proofs, allowing for a safer and faster long-term maintenance.

Over the years, we have witnessed a number of breakthroughs in formal mechanical verification of real-life systems. These systems include verified compilers [26, 21], the verified seL4 microkernel [20], the verified distributed system IronFleet [17], or the Roissy airport shuttle wayside control unit [8] generated from a formal model. These systems have significant roles outside of the research context. For instance, CompCert is a commercial product while seL4 microkernel is used, among others, in medicine, aerospace, autonomous aviation, and defence [36].

2.5.1 Proof assistants

Proof assistants are software tools that prove statement about formal structures using deductions rules (e.g.: resolution, induction, etc.). While a formal proof can be easily checked by computers, constructing such a proof is nontrivial. A proof assistant therefore interacts with a programmer who guides it in the search of a formal proof.

We list some of the most popular interactive theorem provers:

- **ACL2** [3] is a highly automated proof assistant used predominantly in hardware verification. It only uses first-order logic making it challenging to describe more complex systems.
- **Coq** [11] is a generic purpose proof assistant with a very expressive type system making it popular in areas of computer science [26] and mathematics [14, 15]. It provides code extraction of programs in OCaml [33] and Haskell [16] directly from Coq functions or Coq proofs and specification.
- **Isabelle** [18] is a generic purpose, higher-order-logic (HOL) theorem prover implemented in Standard ML. It provides high degree of proof automation and a relatively modern user interface. Compared to Coq, it has a more restricted logic that does not support dependent types.
- **TLAPS** [38] is a proof system for the TLA+ [22] formal specification language particularly useful for describing concurrent and distributed systems. Proofs are described in declarative style. TLAPS works by transforming a proof into individual proof obligations and sending them to backend provers such as Isabelle or Zenon [10]. TLAPS has been used to prove correctness of the Pastry Distributed Hash Table [27] or Byzantine Paxos [23].
- **Dafny** [12] is a programming language and an automated program verification system with built-in specification constructs such as function pre-conditions, post-conditions, loop variant and loop invariants. It is based around the concept of dynamic frames [19]. This approach attempts to prove program correctness of individual program parts

locally and from that imply the correctness of the full program. Additionally, Dafny is also capable of produce .NET executable.

- **Lean** [24] is one of the newer proof assistants developed by Microsoft Research. It offers a very expressive type system and good automation. However, due to its young age, its library is limited and in the early stages of development.

We picked Isabelle as our proof assistant choice due to its expressiveness, level of automation, and our familiarity with the tool.

Chapter 3

Isabelle/HOL Syntax Overview

In this chapter, we give an overview of the Isabelle/HOL syntax that will be used throughout the thesis.

3.1 Types, terms and basic logic

The syntax of Isabelle/HOL resembles that of a functional programming language. In terms of types, it supports:

- Basic types: *bool*, *int*, *nat*, ...
- Type variables: *'a*, *'b*, *'loc*, ...
- Function types denoted by $\Rightarrow$: $nat \Rightarrow nat$, $'a \Rightarrow 'b \Rightarrow 'c$ ...
- Type constructors: *list*, *set*, *zmultiset*, etc. The type constructors are written postfix such that *nat set* is a type of sets of elements in $\mathbb{N}$ and *'a list* is a type of lists of elements in type *'a*.

Terms can be formed as:

- Variables
- Constants
- Function applications, for example $f\ x$ or $plus\ x\ y$, ...
- Lambda abstractions, examples of which are the identity function $\lambda x.\ x$, or the addition on two elements $\lambda x\ y.\ x + y$.

All terms in Isabelle must be well-typed. Isabelle tries to automatically infer these types using *type inference*. However, due to overloaded functions, terms must be occasionally explicitly annotated by the user. For this purpose, we write $t{::}\tau$ to mean that the term t is of type τ.

Terms of type *bool* are called formulas. Isabelle uses basic logic constructs for Boolean and logical expressions such as $\vee$, $\wedge$, $=$, $\neq$, $\neg$ or $\longrightarrow$. Quantifiers are denoted by $\exists$ and $\forall$. Additionally, Figure 3.1 lists functions and constructs used throughout the thesis.

3.2 Function definitions and lemmas

The *definition* command provides a way of defining a non-recursive function without pattern-matching. The following function definition computes the square of any natural number.

```
definition sq :: nat ⇒ nat where
sq x ≡ x * x
```

In turn, the *fun* command provides a convenient way of defining functions that do make use of pattern matching or recursion. A simple example is the *sum_list* function computing the sum of all elements in a list of natural numbers.

```
fun sum_list :: nat list ⇒ nat where
  sum_list [] = 0
| sum_list (x#xs) =  x + (sum_list xs)
```

When using the *fun* command, Isabelle tries to solve all the necessary proof obligations automatically. In the case of recursion, this involves the proof of termination. In the case of pattern matching, Isabelle tries to prove that the list of provided patterns is exhaustive and in some cases non-overlapping. If any of these proofs fails, the definition is rejected. This can either mean that the definition is faulty, or that the default proof procedures are insufficient to complete the proofs. If the user believes the latter to be the case, they can define the function using a *function* command rather than *fun* and prove all obligations manually.

```
function sum_list :: nat list ⇒ nat where
  sum_list [] = 0
| sum_list (x#xs) =  x + (sum_list xs)
  by (auto, meson neq_Nil_conv)

(* Proof of termination of the function sum_list *)
termination sum_list
  apply (relation measures [λxs. length xs])
  using wf_measures apply blast
  by (auto simp add: not_less)
```

The *lemma* command consists of a lemma *name*, a *statement* we wish to prove and an associated *proof* as shown below. When the proof is completed,

```
(* LISTS *)
[]        (* an empty list *)
hd L      (* head of list L*)
tl L      (* tail of list L*)
L ! n     (* nth element of list L*)
L1 @ L2   (* concatenation of lists L1 and L2 *)
x # L     (* L appended with element x at front *)

(* STREAMS *)
shd S           (* head of stream S*)
stl S           (* tail of stream S*)
holds P S       (* True iff predicate P holds for the head of stream S *)
alw φ S          (* True iff predicate φ holds for all suffixes of stream S *)
(φ aand ψ) S    (* True iff predicates φ and ψ hold for stream S *)

(* ZMULTISETS *)
{}z              (* an empty zmultiset *)
zcount Z x       (* the multiplicity of x in Z, can be negative *)
x ∈z Z           (* True iff x has a non-zero multiplicity in Z *)
set_zmset Z      (* a set containing all elements of Z
                    with non-zero multiplicity in Z *)

(* RECORD TYPES *)
record point =               (* Definition of a point record type. *)
  Xcoord :: int              (* Records of type point have two fields named*)
  Ycoord :: int              (* Xcoord and Ycoord both of type int *)

⦇ Xcoord = 10, Ycoord = 5 ⦈::point     (* A constant of type point *)
Xcoord p                     (* The value of field Xcoord in point p *)
p ⦇ Xcoord = -81 ⦈           (* A functional update operation on records.
                                This term returns a constant of type point with Xcoord
                                set to -81 and values of other fields copied from p *)

(* BASIC CONSTRUCTS *)
if P then t1 else t2       (* standard if-statement *)
case t of case1 ⇒ t1       (* standard case-statement *)
        |...
        | casen ⇒ tn
let x = a in t             (* local assignment *)
SOME x . P x               (* an arbitrary x satisfying predicate P *)
THE x . P x                (* the unique x satisfying predicate P *)

type_synonym date = (nat × nat)        (* A type alias called date *)
datatype colour = Red | Green | Blue   (* An numeration type with three elements *)
```

Figure 3.1: Useful constructs and operations.

Isabelle associates the name of the lemma with the statement which can now be used in the proofs that follow.

```
lemma sum_list_concat :
  sum_list xs + sum_list ys = sum_list (xs@ys)
  apply (induction xs)
   apply simp
  by simp
```

3.3 Locales

In Isabelle, the locales are a module system which allows us to reason from axioms locally in a sound way. A locale defines a *context* containing a set of fixed *parameters* and *assumptions*. Inside the locale, we can make definitions and prove theorems based on these parameters and assumptions.

The following code snippet shows a specification of a locale defining partial order. It contains one parameter *le* (also written infix ⊑), the binary predicate declared using the *fixes* keyword. The locale also contains three assumptions defined in terms of the *le* predicate describing the three properties of partial order: reflexivity, antisymmetry, and transitivity. The assumptions are declared using the *assumes* keyword.

```
locale partial_order =
    fixes le :: 'a ⇒ 'a ⇒ bool (infixl ⊑ 50)
  assumes refl:     x ⊑ x
      and antisym:  ⟦ x ⊑ y; y ⊑ x ⟧ ⟹ x = y
      and trans:    ⟦ x ⊑ y; y ⊑ z ⟧ ⟹ x ⊑ z
```

The locales can be further extended by introducing more parameters and assumptions. For example, we can define a new locale *total_order* by extending the *partial_order* with an assumption that any two elements are comparable with respect to *le*:

```
locale total_order = partial_order +
    assumes comp:     x ⊑ y ∨ y ⊑ x
```

Moreover, concrete examples can be proved to be valid instances of a locale. For example, we can instantiate the *partial_order* locale with the total order on integers:

```
interpretation int: partial_order (≤) :: int ⇒ int ⇒ bool
    by unfold_locales auto
```

In summary, locales provide a sound way of reasoning locally about axiomatic theories and are highly suitable for formalisation of abstract math-

ematical concept and hierarchical structures. The code snippets in this subsection were taken from the *Tutorial to Locales and Locale Interpretation* [9].

Chapter 4

Propagation Algorithm: Model

This chapter presents the formal specification of the Timely Dataflow progress tracking protocol's Propagation algorithm. This algorithm aims to compute location frontiers from a worker's approximated view of system's capabilities.

We start by introducing basic concepts and definitions used in the formalisation. Next, we model the graph topology and list the necessary assumptions on the graph. Finally, we present the model of the Propagation algorithm which consists of the computation state and the set of allowed actions.

4.1 Basic definitions

The Propagation algorithm is parametrised by three type variables:

- *'loc*: the location type,
- *'t*: the timestamp type, and
- *'sum*: the summary type.

4.1.1 Partial order on timestamps

We define partial order as a binary relation that satisfies reflexivity (each timestamps is comparable to itself), antisymmetry (no two different timestamps precede each other), and transitivity (if a first element precedes a second element, and, in turn, that element precedes a third element, then the first element precedes the third element).

For example, *(int64, int64)* is a valid timestamp type with following partial order:

$$(a1, b1) \leq (a2, b2) \iff a1 \leq a2 \land b1 \leq b2. \tag{4.1}$$

4.1.2 Antichain

Having to deal with partially ordered timestamps causes some complications. For instance, given a set of timestamps, there can be multiple minimal or maximal (mutually incomparable) elements in the set. To reason about partially ordered elements we use a standard mathematical concept of an *antichain*.

Definition 4.1 *A subset $A \subseteq S$ is an antichain in S if every pair of different elements in A is incomparable. We define $\{\}_A$ to be the empty antichain and $x \in_A A$ to denote that x is a member of antichain A.*

As an example, let $S = \{(1,2),(2,3),(4,1),(3,1)\}$ be a set of tuples with the partial order defined by Equation (4.1). Then subsets $A = \{(1,2),(3,1)\}$ and $B = \{(2,3),(3,1)\}$ are both antichains in S.

4.1.3 Signed multiset

In later sections, we will see that the state of the computation is expressed in terms of signed multisets (`zmultiset` in Isabelle). In a signed multiset, every element has an associated positive or negative multiplicity. The domain of a signed multiset S (i.e., the set of elements with non-zero multiplicity in S) must be finite.

To effectively talk about signed multisets, we describe concrete signed multisets using a set of element-multiplicity pairs. For example, $S = \{(2,+2), (3,+1),(5,-10)\}$ is a signed multiset of integers with two copies of 2, one copy of 3, and minus ten copies of 5. All other integers have implied multiplicity 0 in S.

We define $\{\}_z$ to be the empty signed multiset and $x \in_z S$ to denote that x has a non-zero multiplicity in S.

4.1.4 Infimum of signed multiset

Next, we define the *infimum* (or greatest lower bound) as a function over a signed multiset S returning all minimal incomparable elements in S with multiplicity 1.

Definition 4.2 *infimum(S) is an antichain in S consisting of elements of S that are either smaller or incomparable to any other element in S. Formally,*

$$\mathit{infimum}(S) = \{x \mid x \in S \wedge \nexists y \in S \,.\, y < x)\}$$

Definition 4.3 *zmset_infimum(S) is signed multiset consisting of elements of infimum(S) with multiplicity $+1$. Formally,*

$$\mathit{zmset_infimum}(S) = \{(x,+1) \mid x \in \mathit{infimum}(S)\}$$

4.2 Dataflow graph topology

The *dataflow_topology* locale describes the timely dataflow computation graph and contains two fixed parameters: *results_in* and *summary*.

The *results_in* relation describes how timestamps are incremented with summaries. The *summary* function defines the edges of the dataflow graph. Concretely, it associates every ordered pair of locations with a set of weights – an antichain *'sum* type. An empty antichain implies there is no edge between the two locations.

```
locale dataflow_topology =
    fixes results_in :: 't ⇒ 'sum  ⇒ 't
    and   summary    :: 'loc ⇒ 'loc ⇒ 'sum antichain
...
```

4.2.1 Assumptions on basic types

In addition, we impose a few restrictions on *'loc*, *'t* and *'sum* type:

- The set of locations *'loc* is enumerable and finite. The finitness is required to show termination of the Propagation algorithm.
- Sets *'t* and *'sum* are partially ordered. We will use $<$ and $\leq$ as the partial order operations.
- The set of summaries *'sum* is an additive monoid. In other words, there is a binary function $(+)$ on *'sum* that is associative, with an identity element 0 in *'sum*.

4.2.2 Assumptions on the graph topology

Having defined the basic types and the set of relations on them, we present a list of necessary assumption on the dataflow graph topology.

```
...
assumes results_in_zero: results_in t 0 = t
    and results_in_mono:      t1 ≤ t2
                              ⟶ results_in t1 s ≤ results_in t2 s
                              s1 ≤ s2
                              ⟶ results_in t s1 ≤ results_in t s2

    and followed_by_mono:     (s1::'sum) ≤ s2 ∧ s3 ≤ s4
                              ⟶ s1 + s3 ≤ s2 + s4
    and followed_by_summary: results_in (results_in t s1) s2
                              = results_in t (s1 + s2)
```

```
  and summary_self:        summary loc loc = {}_A
  and summary_cycle:       path loc loc p ∧ p ≠ []
                           ∧ s = sum_path_weights p
                           ⟶ t < result_in t s.
...
```

The assumption *results_in_zero* states that adding a zero-summary to a timestamp has no effect on the timestamp. The assumption *results_in_mono* states that the relation is monotone in both of its arguments.

Similarly, *followed_by_mono* requires that the addition on summaries is monotone in both arguments. The assumption *followed_by_summary* claims that applying two summaries one after another to a timestamp results in the same timestamp as if we start by adding the two summaries and only then apply them to the timestamp.

The last two assumptions talk about cycles in the graph. The first assumption, *summary_self*, is self-explanatory, requiring no loops (self-cycles). The second assumption, *summary_cycle*, claims that every cycle has to increment a timestamp. Concretely, it states, that if p is a non-empty path from *loc* to *loc*, and s is the accumulated summary along the path p (computed by the function *sum_path_weights*) then applying s to any timestamp t always leads to an increased timestamp.

4.3 State

The state of the algorithm is defined by a *configuration* record. It describes a single worker's view of the system's capabilities and the effect they carry on the location *frontiers*. The *configuration* record is specified using three state variables: *pointstamps*, *implications*, and *worklist*.

```
record (overloaded) ('loc, 't) configuration =
  pointstamps  :: 'loc ⇒ 't zmultiset (* local view of the caps *)
  implications :: 'loc ⇒ 't zmultiset (* implications of the caps *)
  worklist     :: 'loc ⇒ 't zmultiset (* not-yet-applied updates *)
```

The configuration *pointstamps c* is a worker's view of the system's capabilities. For any timestamp t, *(pointstamps c loc)* approximates the number of capabilities *(loc, t)* in the system in configuration c. This multiset does not provide the true representation of the system because capabilities are dropped or created independently by all workers. Each worker is eventually notified of ongoing changes, but in the meantime the worker's view of system's capabilities may be delayed.

The configuration *implications c* is a worker's view of possible future capabilities computed from the worker's view of present capabilities - *pointstamps*

c. In particular, for any location *loc*, a timestamp *t* with strictly positive multiplicity in *implications c loc* suggests that there exists a capability (loc', t') such that (loc', t') *could-result-in (loc, t)*.

The variable *worklist c loc* stores local (temporary) changes that should be applied to *(implications c loc)*.

The *implications* are the output of the protocol because they provide locations with *frontiers*. What we want is for all possible future timestamps to be reflected in *implications*. Clearly, in practice, *implications* cannot store an infinite set of timestamps. However, what we can do is to store a valid lower-bound. In fact, we can define a *frontier* of location *loc* in configuration *c* as the *infimum (implications c loc)*.

4.4 The algorithm

This section starts with an informal description of the Propagation algorithm to build some intuition into how the algorithm works while the following subsections give formal definitions of the algorithm actions.

We know that each worker holds capabilities which allow it to produce data with certain timestamps at certain locations. Additionally, every location is aware of its *frontier* - a lower bound of possible future timestamp that may yet arrive at the location as a result of system's capabilities. The *frontiers* are recorded by *implications*.

As the computation progresses, smaller timestamps cease to arrive while higher timestamps emerge. In other words, locations drop and receive new capabilities. The goal of the algorithm is to repeatedly update *implications* for all locations given a list of changes to the system's capabilities.

To demonstrate the approach of the algorithm, consider a newly created capability *(loc,t)*. We start by updating *implications* at location *loc*. A "message" about the timestamp *t* then starts its journey from location *loc*, passing along the edges of the graph and being transformed by corresponding summaries. Every time it arrives at some location *loc'*, the location takes this recomputed timestamp into consideration and updates its own *implications*. The timestamp then continues along the outgoing edges. In the end, the *implications* at every location should contain a valid lower bound of future timestamps. What is more, the new *implications* may contain a more accurate lower bounds than those before the start of the propagation.

In reality, we do not send actual messages along the graph's edges. Instead, the system which is aware of the graph topology, performs this graph traversal and updates *implications* accordingly. The algorithm does so using two actions:

1. *next_change_multiplicity*
2. *next_propagate*

Formally, actions are *next-state relations* describing how the computation state changes in each step that the algorithm takes. As a result, an action is expressed as a function of two configuration states - the original configuration *c0* and the next-state configuration *c1* - and in some cases additional arguments. The function returns *True* if the two configurations satisfy the next-state relation specified by the action.

The actions can be carried out in any order, as long as all the conditions on the actions are satisfied. The next two sections present a formal specification of both action and discuss them in detail.

4.4.1 The next_change_multiplicity action

```
definition next_change_multiplicity :: ('loc, 't) configuration
                                      ⇒ ('loc, 't) configuration
                                      ⇒ 'loc ⇒ 't ⇒ int ⇒ bool where
next_change_multiplicity c0 c1 loc t n =
    unchanged implications c0 c1 ∧
    n ≠ 0 ∧
    (∃t'. t' ∈A infimum (implications c0 loc) ∧ t' ≤ t) ∧
 - < add n copies of t to pointstamps c0 loc >
    (∀loc'. pointstamps c1 loc' = (
        if loc' = loc
        then update_zmultiset (pointstamps c0 loc') t n
        else pointstamps c0 loc')
 - < add changes of pointstamps infimum into worklist >
    (∀loc'. worklist c1 loc' =
        worklist c0 loc' +
        infimum_changes (pointstamps c1 loc') (pointstamps c0 loc'))
```

Figure 4.1: Formal specification of the *next_change_multiplicity* action.

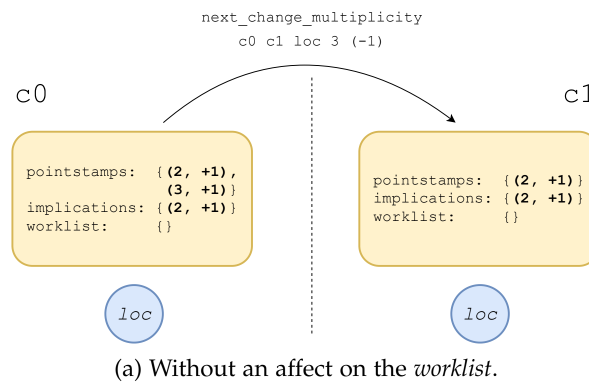
The *next_change_multiplicity* action takes five arguments: the original configuration *c0*, the next-state configuration *c1*, a location *loc*, a timestamp *t* and an integer *n*. The formal specification of the action is presented in Figure 4.1.

In this action, the worker records information about *n* created or destroyed *(loc, t)* capabilities. A positive *n* implies that capabilities were created while a negative *n* suggest that capabilities were destroyed. The change is recorded

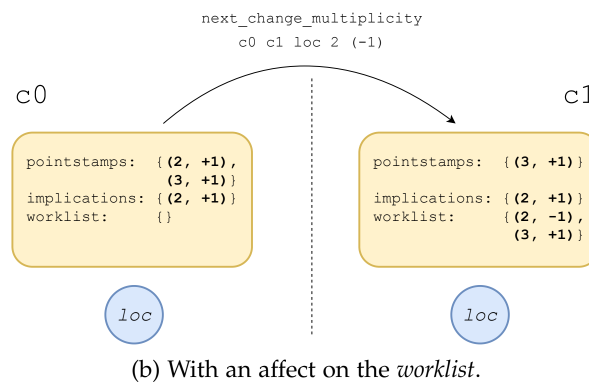
by adding *n* copies of timestamp *t* to *pointstamps c0 loc*. Next, *worklist c0 loc* is updated with the changes to *infimum (pointstamps c0 loc)*.

Consider the example in Figure 4.2a showing a single location *loc* in the original configuration *c0* (left) and next-state configuration *c1* (right). The action (arrow) removes one copy of timestamp 3 from *pointstamps* at location *loc*. This change has no effect on *infimum (pointstamps c0 loc)*, which remains at 2. Thus, the *worklist* also does not change.

Now, consider the second example in Figure 4.2b. The only difference from the previous example is that the action acts on timestamp 2 instead of timestamp 3. We can see the action remove one copy of timestamp 2 from *pointstamps* at location *loc*. This time, however, *infimum (pointstamps c1 loc)* increases to 3. As a consequence, the *worklist* is updated with the changes to *infimum (pointstamps c1 loc)*.



(a) Without an affect on the *worklist*.



(b) With an affect on the *worklist*.

Figure 4.2: A demonstration of the *next_change_multiplicity* action with different effects on the *worklist*.

Lastly, this action has to satisfy two requirements. First, *n* cannot be zero, since it would have no effect on the configuration state, letting us repeatedly perform this action without making any progress.

Secondly, the timestamp of the created or destroyed capabilities cannot be smaller than all elements of *(implications c0 loc)*. Remember that *implications* store information about which timestamps the location mat still see in future. Removing this condition would allow us to create or destroy capabilities with timestamps that the location considers to be "expired". The condition is thus necessary for the algorithm's correctness.

4.4.2 The next_propagate action

```
definition next_propagate :: ('loc, 't) configuration
                          ⇒ ('loc, 't) configuration
                          ⇒ 'loc ⇒ 't ⇒ bool where
next_propagate c0 c1 loc t =
    unchanged pointstamps c0 c1 ∧
    t ∈z worklist c0 loc ∧
    (∀t' loc'. t' ∈z worklist c0 loc' ⟶ t' ≮ t) ∧
 - < add all copies of t in worklist c0 loc to implications c0 loc >
    (∀loc'. implications c1 loc' = (
        if loc' = loc
        then implications c0 loc' +
             (filter_zmset (λt'. t' = t)(worklist c0 loc))
        else implications c0 loc')
 - < 1. Remove all copies of t from worklist c0 loc >
 - < 2. Add changes of implications infimum to worklists at all out-
        going locations taking into account summaries along the edges. >
    (∀loc'. worklist c1 loc' =
         if loc' = loc
         then (filter_zmset (λt'. t' ≠ t)(worklist c0 loc'))
         else worklist c0 loc' +
              after_summary
                (infimum_changes
                    (implications c1 loc) (implications c0 loc))
                (summary loc loc')))
```

Figure 4.3: Formal specification of the *next_propagate action*.

The *next_propagate* action takes four arguments: the original configuration *c0*, the next-state configuration *c1*, a location *loc* and a timestamp *t*. The formal specification of the action is presented in Figure 4.3.

In this action, all copies of timestamp *t* in *worklist c0 loc* are moved to *implications c0 loc*. Then, the *worklists* at all outgoing locations are updated with changes to *infimum (implications c0 loc)*. Since edges connecting locations

carry summaries, the timestamps are incremented with edge summaries before they are added to the worklist.

Once again, we try to demonstrate the action with a use of a simple example. The first example in Figure 4.4a shows three connected locations loc_1, loc_2 and loc_3, their original configuration state *c0* (left) and their next-state configuration *c1* (right). All copies of timestamp 2 at location loc_1 are moved from *worklist c0* loc_1 to *implications c1* loc_1. The action does not change *infimum (implications c0* loc_1*)*. Therefore, *worklists* at locations other than loc_1 remain the same.

In the second example in Figure 4.4b, the action moves all copies of timestamp 2 at loc_1 from *worklist c0* loc_1 to *implications c1* loc_1, this time causing a change to *infimum (implications c0* loc_1*)*. We can see that the *worklist* at neighbouring outgoing locations is modified with these changes and that they satisfy the increments enforced by edge summaries.

The action must satisfy two requirements. First, timestamp *t* at location *loc* must have a non-zero, otherwise the action would have no effect on the state.

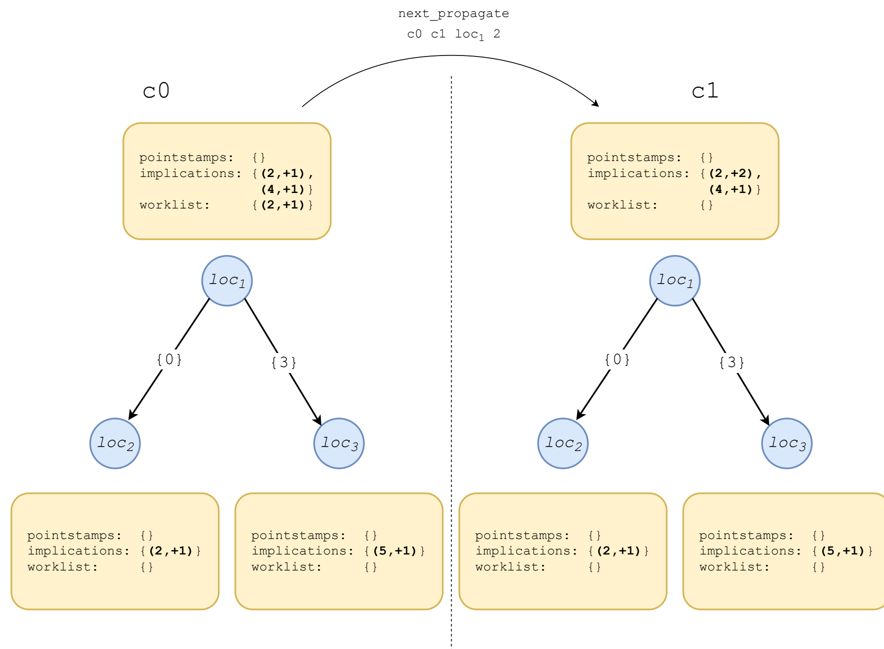
Secondly, *t* must be a smallest such timestamp in *worklist*. This condition may not be obvious and we have to build some intuition into why it is at all necessary. The *next_propagate* action broadcasts information about new capability implications to neighbouring locations. Those locations can in turn broadcast this information further along the graph edges. Since the graph can contain cycles, it is possible that the information will travel along the edges indefinitely. Always picking the smallest timestamp ensures that positive and negative entries in the *worklist* eventually cancel out and consequently prevents this endless loop from happening.

Appendix A shows an example of a graph state, in which *next_propagate* step always picks a largest timestamp instead of a smallest, and the state never converges. We have implemented the same scenario in the Rust implementation of Timely Dataflow and got identical results.

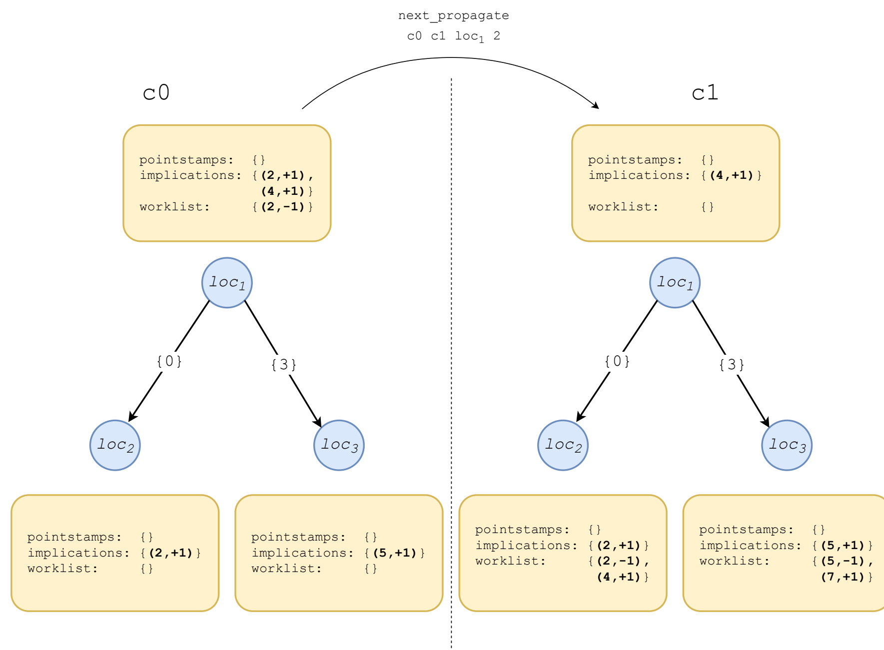
4.5 Specification

Finally, the specification of the Propagation algorithm, formalised in Figure 4.5, is defined in terms of an initial configuration *Init* and next-state relation *Next*. A complete specification *Spec* is an infinite stream of configurations starting with *Init*, and satisfying the *Next* relation for every pair of consecutive configurations.

In the initial configuration *Init*, the *implications* at all locations are empty, thus, the locations are not yet aware of their *frontiers*. The configuration may however contain pointstamps, termed *default pointstamps*, as long as they are reflected in the *worklist*.



(a) Without an affect on the *worklists* of neighbouring locations.



(b) With an affect on the *worklists* of neighbouring locations.

Figure 4.4: A demonstration of the *next_propagate* action with different effects on the *worklists* of neighbouring locations.

```
definition Init where
  Init c ≡ ∀loc.
      implications c loc = {}z ∧
      worklist c loc = zmset_infimum (pointstamps c loc)

definition Next :: ('loc, 't :: order) configuration stream
                 ⇒ bool where
  Next s ≡ (
    let c0 = shd s;
        c1 = shd (stl s)
    in
    ∃loc t n. next_change_multiplicity c0 c1 loc t n ∨
    ∃loc t. next_propagate c0 c1 loc t
 )

definition Spec :: ('loc, 't) configuration stream
                 ⇒ bool where
  Spec ≡ holds Init aand alw Next
```

Figure 4.5: The algorithm specification.

In practice, Timely Dataflow initialises the *default pointstamps* at all locations with a signed multiset $\{(0, +m)\}$ where m is the number of workers in the system.

4.5.1 Propagation Rounds

The definition of *Spec* does not impose any ordering on actions. However, in practice, this ordering is not arbitrary. Instead, Timely Dataflow preforms a sequence of *next_propagate* actions in a batch until *worklists* at all locations are empty. We call such a sequence of *next_propagate* actions a *propagation round*. Completing a propagation round guarantees that all changes to system's capabilities are reflected in all frontiers. The details on how this guarantee is enforced are shown in Chapter 5 when we discuss the algorithm's safety property.

4.6 Chapter Summary

We have introduced a formal model of the Timely Dataflow Propagation algorithm the goal of which is to continuously re-compute locations' *frontiers* from a local view of the system's capabilities. The model represents computation as a sequence of configurations consisting of *pointstamps* - the local view of system's capabilities, *implications* - the implications of the view

of system's capabilities and *worklist* - the temporary progress updates. The stream evolves with respect to two next-state relations: *next_change_multiplicity* and *next_propagate*.

Additionally, the model of the Propagation algorithm is constrained by a set of assumptions on the basic data-types and the dataflow topology. The purpose of these assumptions is to prove the safety property of the Propagation algorithm as well as the safety property of the overall Timely Dataflow Progress Tracking protocol. We have put great effort into making our model general by minimising the number of constraints.

Chapter 5

Propagation Algorithm: Safety

This chapter presents the main safety property of the Propagation algorithm presented in Chapter 4 and gives the proof overview of the safety property.

5.1 Safety Property

The Propagation algorithm aims to compute location *frontiers* from a worker's approximated view of system's capabilities. Each *frontier* represents a lower-bound of possible future timestamps arriving at a given location. Each location can use its *frontier* to reason about which phases of computation have passed and which are still in progress allowing it to produce correct output for the individual phases.

The safety property of the Propagation algorithm is defined in Figure 5.1. It states that for every known capability *(loc1, t1)*, if this capability *could result in (loc2, t2)*, then the *frontier* at location *loc2* is *not* ahead of *t2*.

It is important to note that the safety property is *not* an invariant of the Propagation algorithm. Indeed, the *spec_implies_safe* lemma in Figure 5.1 claims the safety property is satisfied for those configurations in *s* whose *worklist* is empty at all locations. We know that the *worklists* become empty at the end of a propagation round.

It is indeed possible that while the Propagation algorithm processes *worklist* items, it temporarily advances *implications* past valid *frontiers*. As a result, the operator should not make time-depended decisions such as creating new messages or draining its state before the Propagation algorithm clears all *worklists*.

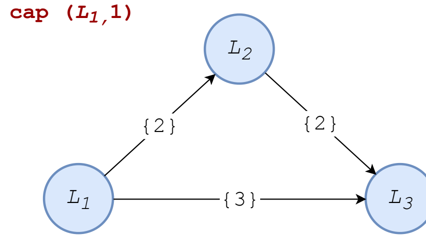
The safety property is defined in terms of *could result in* relation. We say that a capability *(loc1, t1) could result in (loc2, t2)* if and only if there exists a path from *loc1* to *loc2* such that applying the accumulated path summary *s* along the path to *t1* does not exceed *t2*.

```
definition supported_by where
    supported_by t S ≡ ∃t'. t' ∈_A infimum S ∧ t' ≤ t

definition safe :: ('loc, 't) configuration ⇒ bool where
safe c ≡ ∀loc1 loc2 t1 t2.
    zcount (pointstamps c loc1) t1 > 0
    ∧ could_result_in (loc1, t1) (loc2, t2)
    ⟶ supported_by t2 (implications c loc2)

lemma spec_implies_safe:
    assumes Spec s
    shows   alw (holds (λc. ∀loc. worklist c loc = {}_z ⟶ safe c)) s
```

Figure 5.1: Safety property of the Propagation algorithm

```
definition could_result_in :: ('loc × 't) ⇒ ('loc × 't) ⇒ bool where
could_result_in (loc1,t1) (loc2,t2) ≡ ∃s.
    s ∈_A (path_summaries loc1 loc2) ∧ results_in t1 s ≤ t2
```

Example 5.1 *To visualise how the frontiers and the could result in relation, consider the simple dataflow graph below with a single capability cap = (L_1, 1)*[1]*. We will reason about frontiers per location:*



- L_1: *There exists a unique path from L_1 to L_1, namely the empty path, with an accumulated path summary 0. Thus, given the definition of could result in relation, the capability cap could result in (L_1, t) where t ≥ 1. Finally, the frontier at L_1 is at most $\{1\}$.*
- L_2: *Similarly, there exists a unique path from L_1 to L_2 with path summary 2. Thus, the capability cap could result in (L_2, t) where t ≥ 3 and the frontier at L_2 is at most $\{3\}$.*
- L_3: *There are two paths form L_1 to L_3, an upper path with path summary 4 and a lower path with path summary 3. Since we try to compute the lower-bound*

[1]In this example, we use integer timestamps and integer addition as the additive relation on summaries and the operation *results_in*.

of future timestamps at location L_3 only the smaller summary matters. Thus, cap could result in (L_3, t) where $t \geq 4$ and the frontier at L_3 is at most $\{4\}$.

5.2 Informal proof of the safety property

The proof of the safety property relies on two propositions, the proofs of which are omitted.

Definition 5.2 *union_frontier c loc =*

$$\sum_{loc'} \left[\textit{after_summary} \ (\textit{infimum (implications c loc')}) \ \textit{(summary loc' loc)} \right]$$

The *union frontier* is the union of all frontiers at incoming locations of *loc*, incremented by summaries along the edges to *loc*. We could say that *union frontier* computes the implications, or the effect, that the frontiers one level upstream of *loc* have on the location *loc*.

The first protocol property describes the relationship between the *worklist*, *pointstamps* and *implications* at some location *loc*, and its *union frontier*.

Proposition 5.3 *Fix a configuration c from the computation s satisfying Spec s. Then, the union of the implications and the worklist at location loc is the same as the union of the pointstamps infimum and the union frontier at location loc.*

$$\begin{aligned} &implications\ c\ loc\ +\ worklist\ c\ loc\ = \\ &zmset_infimum\ (pointstamps\ c\ loc)\ +\ union_frontier\ c\ loc \end{aligned}$$

While this may not be obvious at the first sight, the property is intuitive. Remember that the *implications* and the *worklist* at location *loc* store information about the timestamps that may arrive at location *loc* in future. The equality claims, that a timestamp t can be present in the *implications* or *worklist* at location *loc* for one of two reasons. First reason is that there exists a capability with timestamp t at location *loc* and consequently, t $\in_z$ *pointstamps c loc*. The second reason is that one of the incoming locations of *loc* knows about a capability upstream that *could result in* (loc, t) and must have notified *loc* about it. Hence, t is reflected in the *union frontier* of location *loc*.

The second property of the model talks about the existence of *prefix-optimal* paths.

Definition 5.4 *A path from location loc_1 to location loc_2 with a path summary s is* ***optimal*** *if and only if there is no other path from loc_1 to loc_2 with path summary s' such that $s' < s$. In other words, s is a smallest path summary from loc_1 to loc_2.*

Definition 5.5 *A path p from location loc_1 to location loc_2 is* ***prefix-optimal*** *if and only if every prefix p' of the path p from location loc_1 is optimal.*

Proposition 5.6 *If there exists a path from loc_1 to loc_2 with a path summary s, then there exists a prefix-optimal path from loc_1 to loc_2 with path summary $s' \leq s$.*

While this statement is obvious for integer timestamps and integer summaries, it is highly non-trivial for the general setting we consider. Indeed, the proof of the Proposition 5.6 took up majority of our time in proving the model's safety property.

5.2.1 Proof of the safety property

Theorem 5.7 *Fix a configuration c from the computation s satisfying Spec s. Then,*

$$\forall\ loc\ .\ worklist\ c\ loc\ = \{\}_z \longrightarrow\ safe\ c$$

Proof Figure 5.1 defines *safe* in therms of all pairs of capabilities (loc_1, t_1) and (loc_2, t_2), such that t_1 has a strictly positive multiplicity in *pointstamps c* loc_1 and (loc_1, t_1) *could result in* (loc_2, t_2). A configuration is *safe* if t_2 is *supported by implications* at location loc_2 in configuration c for all such pairs of capabilities.

We prove the above theorem for an arbitrary pair of such capabilities by induction on the length of a *prefix-optimal* path from loc_1 to loc_2. The length is defined in terms of the number of edges in the path.

Base case

In the base case, p has length 0 so $loc_1 = loc_2$. Then, from the assumption of the Theorem 5.7 and the assumptions on the pair of capabilities $(loc_1, t_1), (loc_2, t_2)$ we have:

$$\forall loc\ .\ worklist\ c\ loc = \{\}_z \tag{5.1}$$
$$\wedge\ zcount\ (poinstatmps\ c\ loc_2)\ t_1 > 0 \tag{5.2}$$
$$\wedge\ t_1 \leq t_2 \tag{5.3}$$

From the assumption 5.2 and the definition of *infimum*, we get:

$$\exists t'.\ t' \leq t_1 \wedge t' \in_A\ infimum\ (pointstamps\ c\ loc_2) \tag{5.4}$$

Note that the *worklist* at all locations including loc_2 is empty and all timestamps in the *union_frontier* have non-negative multiplicity. Then, by the

Proposition 5.3 we get:

$$\begin{aligned}
& \textit{zcount (implications c loc}_2\textit{)}\ t' > 0 \\
\overset{\textit{infimum}\text{ def.}}{\Longrightarrow}\ & \exists t''.\ t'' \le t' \wedge t'' \in_A \textit{infimum (implications c loc}_2\textit{)} \\
\overset{\text{equation (5.4)}}{\Longrightarrow}\ & \exists t''.\ t'' \le t_1 \wedge t'' \in_A \textit{infimum (implications c loc}_2\textit{)} \\
\overset{\text{equation (5.3)}}{\Longrightarrow}\ & \exists t''.\ t'' \le t_2 \wedge t'' \in_A \textit{infimum (implications c loc}_2\textit{)} \\
\overset{\textit{supported_by def.}}{\Longrightarrow}\ & \textit{supported_by}\ t_2\ \textit{(implications loc}_2\textit{)}
\end{aligned}$$

Inductive case

$$\begin{aligned}
& \forall loc\ .\ \textit{worklist c loc} = \{\}_z \\
& \wedge\ \textit{zcount (pointstamps c loc}_1\textit{)}\ t_1 > 0 \\
& \wedge\ (loc_1, t_1)\ \textit{could-result-in}\ (loc_2, t_2) \\
& \wedge\ \exists p\ .\ \textit{prefix_optimal p loc}_1\ loc_2 \wedge \textit{path_length}\ p \le n \\
\longrightarrow\ & \textit{supported_by}\ t_2\ \textit{(implications c loc}_2\textit{)}
\end{aligned} \tag{5.5}$$

The inductive hypothesis in the equation (5.5) states, that while the *worklist* at all locations is empty, if an existing capability (loc_1, t_1) *could result in* a capability (loc_2, t_2), and there exists a prefix-optimal path p from loc_1 to loc_2 no longer than n, then timestamp t_2 must be supported by the *frontier* at location loc_2. Specifically, there must exists a timestamp $t' \le t_2$, such that $t' \in_A$ *infimum (implications c* loc_2*)*.

Assume that for a pair of capabilties (loc_1, t_1) and (loc_2, t_2) we have:

$$\begin{aligned}
& \forall loc\ .\ \textit{worklist c loc} = \{\}_z \\
& \wedge\ \textit{zcount (pointstamps c loc}_1\textit{)}\ t_1 > 0 \\
& \wedge\ (loc_1, t_1)\ \textit{could-result-in}\ (loc_2, t_2) \\
& \wedge\ \exists p\ .\ \textit{prefix-optimal p loc}_1\ loc_2 \wedge \textit{length}\ p = n + 1
\end{aligned} \tag{5.6}$$

Let s be the path summary along path p, then from the definition of *could result in*

$$t2 \ge \textit{results_in}\ t_1\ s \tag{5.7}$$

Now, obtain a new path p^* by removing the last edge from path p. Figure 5.2 helps visualise this scenario. Let s^* be the path summary of p^* and $s_\rightarrow$ be the summary along the edge from loc^* to loc_2. Then we get:

$$s = s^* + s_\rightarrow \tag{5.8}$$

Combining the above equation with equation (5.7), we get:

$$\begin{aligned}
& t2 \ge \textit{results_in}\ t_1\ (s^* + s_\rightarrow) \\
\overset{\textit{followed_by_summary}\text{ assum.}}{\Longrightarrow}\ & t2 \ge \textit{results_in (results_in}\ t_1\ s^*)\ s_\rightarrow
\end{aligned} \tag{5.9}$$

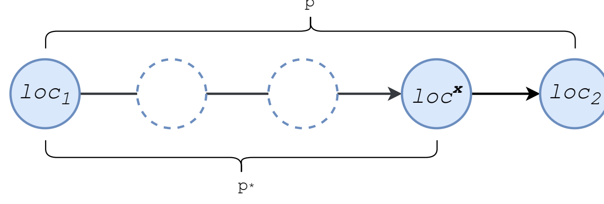


Figure 5.2: Proof of the safety property: visualisation of the inductive case.

Therefore, we can obtain a timestamp t^*, such that

$$\begin{aligned} & t^* = results_in\ t_1\ s^* \\ \wedge\ & (loc_1, t_1)\ could\text{-}result\text{-}in\ (loc^*, t^*) \\ \wedge\ & t2 \geq results_in\ t^* s_\rightarrow \end{aligned} \tag{5.10}$$

Since p is *prefix-optimal*, so must be its prefix p'. Moreover, we know that p^* has length n. Then, by the inductive hypothesis we get:

$$supported_by\ t^*\ (implications\ c\ loc^*) \tag{5.11}$$

$$\overset{supported_by \text{ def.}}{\Longrightarrow} \exists t'.\ t' \leq t^*\ \wedge\ t' \in_A infimum\ (implications\ c\ loc^*) \tag{5.12}$$

$$\overset{union_frontier \text{ def.}}{\Longrightarrow} zcount\ (union_frontier\ c\ loc_2)\ (t' + s_\rightarrow) > 0 \tag{5.13}$$

Finally, since the *worklist* is empty at all locations, and *zmset_infimum* is always positive, Proposition 5.3 tells us that

$$\begin{aligned} & zcount\ (implications\ loc_2)\ (t' + s_\rightarrow) > 0 \\ \overset{infimum \text{ def.}}{\Longrightarrow}\ & \exists t''.\ t'' \leq t' + s_\rightarrow\ \wedge\ t'' \in_A infimum\ (implications\ c\ loc_2) \\ \overset{\text{equation (5.12)}}{\Longrightarrow}\ & \exists t''.\ t'' \leq t^* + s_\rightarrow\ \wedge\ t'' \in_A infimum\ (implications\ c\ loc_2) \\ \overset{\text{equation (5.10)}}{\Longrightarrow}\ & \exists t''.\ t'' \leq t_2\ \wedge\ t'' \in_A infimum\ (implications\ c\ loc_2) \\ \overset{supported_by \text{ def.}}{\Longrightarrow}\ & supported_by\ t_2\ (implications\ loc_2) \end{aligned}$$

□

5.3 Chapter Summary

In this chapter, we have defined the safety property of the Timely Dataflow Propagation algorithm. Specifically, a worker in a configuration c is defined to be *safe* if each location's *frontier* is a lower bound for all possible future timestamps arriving at that location given the worker's approximated view of system's capabilities.

Finally, we have given an overview of a formal proof of the safety property. In particular, we have shown that the algorithm reaches a *safe* state after each propagation round, in other words, when all *worklists* become empty. While the induction argument is quite straightforward, we have found the proof of the existence of the *prefix-optimal* paths to be highly non-trivial and have since then constructed a more concise proof of the safety property that does not rely on this assumption. However, we believe that if we assume the existence of the *prefix-optimal* paths, the proof presented in this chapter is easier to follow.

Chapter 6

Timely Dataflow Progress Tracking protocol

This chapter presents the formal specification of the complete Timely Dataflow Progress Tracking protocol. Our model combines the models of the Propagation algorithm specified in Chapter 4 and the Exchange algorithm (also know as Naiad Clock Protocol) originally introduced by Abadi et al. [2]. We begin the chapter with a short overview of the Exchange algorithm. Then we continue with the introduction of basic types used in the formalisation. Finally, we present the model by defining its state, the set of actions and the initial state.

6.1 The Exchange algorithm

The Exchange algorithm, also called the Naiad Clock Protocol [2], is a conservative reference counter that keeps track of outstanding capabilities in the system. It describes how the capabilities are dropped and created by the workers and how the information about these new and dropped capabilities is exchanged between them.

We build on an existing translation of the Naiad Clock Protocol model and proofs to Isabelle/HOL. The formal specification of the protocol can be found in Appendix B.

The state of the Exchange algorithm is defined in terms of four state variables as shown in Figure 6.1:

- *nrec* – the true view of system's capabilities.
- *glob* p – a conservative view of system's capabilities at worker p.
- *msg* p q – a message queue from worker p to worker q with information about created and dropped capabilities at worker p.

- *tmp p* – a local (temporary) change to capabilities at worker *p* that has not yet been broadcasted to other workers.

```
record ('p, 't) configuration =
    nrec :: ('loc × 't) zmultiset
    temp :: 'p ⇒ ('loc × 't) zmultiset
    msg  :: 'p ⇒ 'p ⇒ ('loc × 't) zmultiset list
    glob :: 'p ⇒ ('loc × 't) zmultiset
```

Figure 6.1: The state of the Exchange algorithm model.

The Exchange algorithm also defines three transitions.

- *Exchange.next_performop c0 c1 p* Δ_m Δ_p

  In this action, worker *p* performs an operation that creates capabilities Δ_p and destroys capabilities Δ_m. These changes are recorded in *nrec c1* – the true view of all system's capabilities – and *temp c1 p* – the temporary changes to capabilities made by worker *p*.

- *Exchange.next_send_update c0 c1 p tt*

  In this action, worker *p* selects a set of capabilities *tt* and broadcasts all local changes to these capabilities, recorded by *temp c0 p*. A worker *p* broadcasts the information by removing the local changes from *temp c1 p* and enqueuing them on all messages queues from worker *p*.

- *Exchange.next_recv_update c0 c1 p q*

  In this action, worker *p* selects a worker *q* and processes the oldest update on the message queue *msg c0 p q*. The worker dequeues the message and uses its information to update *glob c0 p* – its local view of system's capabilities.

Finally, the formal specification of the Exchange algorithm guarantees that for all configurations *c* the value of *glob c p* always holds a conservative view of system's capabilities *nrec c*. In other worded, a worker's local view of system's capabilities is never ahead of the global view of the system's capabilities.

6.2 Basic Types

The Progress Tracking protocol is parametrised by five type variables:

- *'loc* - the location type,
- *'t* - the timestamp type,
- $('loc \times 't)$ - the pointstamp type,

- *'sum* - the summary type, and
- *'p* - the worker type

In addition to the constraints of the Propagation algorithm model, we require the set of workers *'p* to be finite. Moreover, the Exchange algorithm calls for a partial order on the set of pointstamps $('loc \times 't)$. We use the *could result in* partial order for this purpose.

```
definition (≤) :: ('loc × 't) ⇒ ('loc × 't) ⇒ bool where
    (≤) x y = could_result_in x y

definition (<) :: ('loc × 't) ⇒ ('loc × 't) ⇒ bool where
    (<) x y = x ≤ y ∧ x ≠ y
```

6.3 State

The state of the computation is specified in terms of the Exchange model and the Propagation model. The Exchange model describes how the information about created and destroyed system's capabilities is exchanged between the workers. On the other hand, the Propagation model describes how each worker infers the location frontiers from this exchanged progress information.

```
record ('p, 't, 'loc) configuration =
    c_exchange  :: ('p, ('loc × 't)) Exchange.configuration
    c_propagate :: 'p ⇒ ('loc, 't) Propagate.configuration
    init        :: p ⇒ bool

type_synonym ('p, 't, 'loc) computation =
             ('p, 't, 'loc) configuration stream
```

The configuration *c_exchange c* is the state of the progress-exchange between all workers introduced earlier in the chapter. It describes the workers' queued progress updates and the workers' approximated view of system's capabilities. The *Exchange.configuration* is defined in terms of the set of workers *'p*, and the set of pointstamps $('loc \times' t)$.

The configuration *c_propagate c p* is the state of the propagation at worker *p*. Since all the workers carry out the Propagation algorithm independently using their local view of the computation, we define the propagation state per-worker. Each *Propagate.configuration* is defined in terms of the set of locations *'loc* and the set of timestamps *'t*.

The Boolean flag *init p* determines whether the initialisation phase at worker *p* has been completed. The *init* flags are initialised with *False* and get flipped

after the first propagation round as discussed in upcoming sections. In Chapter 7, we show that the worker that has completed the initialisation phase, thus has *init* flag set to *True*, will always maintain safe location *frontiers*.

We further define *computation* as a stream of *configurations*.

6.4 The protocol

The Timely Dataflow Progress Tracking protocol combines all actions of the *Exchange* and *Propagation* algorithm into four global actions:

1. *next_performop*
2. *next_send_update*
3. *next_recv_update*
4. *next_propagate*

The actions can be once again carried out in any order, as long as all the conditions on the actions are satisfied.

The following subsections present a formal specification of each action and discuss them in detail. Each action is expressed as a function of two configuration states - the original configuration *c0* and the next-state configuration *c1* - and additional arguments.

6.4.1 The next_performop action

In the *next_performop* action, the worker p performs an operation that consumes and produces some number of capabilities as defined by the *Exchange* algorithm. The values of *c_propagate* and *init* do not change while the value of *c_exchange* changes according to the *Exchange.next_performop*. Figure 6.2 gives the formal specification of the *next_performop* action.

6.4.2 The next_send_update action

In the *next_send_update* action, the worker p sends messages to all workers (including itself) carrying local progress updates as defined by the *Exchange* algorithm. The values of *c_propagate* and *init* remain the same while *c_exchange* changes according to *Exchange.next_send_update*. The formal specification of the action is presented in Figure 6.3.

6.4.3 The next_recv_update action

In the *next_recv_update* action, the worker q chooses a worker p and receives the oldest *message* on the message queue from p to q, where the *message*

```
definition next_performop :: ('p, 't, 'loc) configuration
                          ⇒ ('p, 't, 'loc) configuration
                          ⇒ 'p
                          ⇒ ('loc × 't) multiset
                          ⇒ ('p × ('loc × 't)) multiset
                          ⇒ bool where
next_performop c0 c1 p Δm Δp =
    Exchange.next_performop (c_exchange c0) (c_exchange c1) p Δm Δp ∧
    unchanged c_propagate c0 c1 ∧
    unchanged init c0 c1
```

Figure 6.2: Formal specification of the *next_performop* action.

```
definition next_send_update :: ('p, 't, 'loc) configuration
                            ⇒ ('p, 't, 'loc) configuration
                            ⇒ 'p ⇒ ('loc × 't) set ⇒ bool where
next_send_update c0 c1 p tt =
    Exchange.next_send_update (c_exchange c0) (c_exchange c1) p tt ∧
    unchanged c_propagate c0 c1 ∧
    unchanged init c0 c1
```

Figure 6.3: Formal specification of the *next_send_update* action.

is a signed multiset of pointstamps. Worker q uses the *message* to update its local view of the system's capabilities. The formal specification of the *next_recv_update* action is shown if Figure 6.4.

This action has affect on the *c_propagate* and the *c_exchange* configuration since both of them record the worker's approximated view of the system's capabilities.

The change to the *c_exchange* configuration is modelled by the *Exchange.next_recv_update* action.

The change to the *c_propagate* configuration is however more complicated. The *c_propagate* configuration of worker q records the worker's view of system's capabilities in the *pointstamps* record as explained in Chapter 4. The Propagation model lets us modify this record using the *Propagate.next_change_multiplicity* action. However, this action can be only applied to a single location-timestamp pair, as opposed to a signed multiset of such pairs. As a result, we express the change to the *c_propagate* configuration as a series of *Propagate.next_change_multiplicity* actions.

```
definition next_recv_update :: ('p, 't, 'loc) configuration
                            ⇒ ('p, 't, 'loc) configuration
                            ⇒ 'p ⇒ 'p ⇒ bool where
next_recv_update c0 c1 p q =
    Exchange.next_recv_update (c_exchange c0) (c_exchange c1) p q ∧
    unchanged init c0 c1 ∧
    (let message = hd (msg (c_exchange c0) p q)
     in
     c_propagate c1 p' = (
        if p' = q then change_multiplicity_all (c_propagate c0 p') message
        else c_propagate c0 p')

definition change_multiplicity_all where
change_multiplicity_all c Δ =
    (let f = (λ(loc, t) c . THE c'.
        Propagate.next_change_multiplicity
            c c' loc t (zcount Δ (loc,t)))
     in
     Finite_Set.fold f c0 (set_zmset Δ))
```

Figure 6.4: Formal specification of the *next_recv_update* action.

6.4.4 The next_propagate action

The *next_propagate* action preforms one complete propagation round at a worker *p* and sets the *init* flag at worker *p* to *True*, stating that the worker has by now completed the initialisation phase. A propagation round is defined by sequence of *Propagate.next_propagate* actions on a single worker terminated by an empty *worklist*.The state of other workers remains unchanged, and so does the *c_exchange* state. Figure 6.5 gives the formal specification of the *next_propagate* action.

6.5 Specification

Finally, the specification of the Timely Dataflow Progress Tracking protocol is defined in terms of an initial configuration *Init* and a next-state relation *Next*. A complete specification *Spec* is a *computation* (i.e., a *configuration stream*) starting with configuration *Init*, and satisfying the *Next* relation for every pair of consecutive configurations. The formal specification is shown in Figure 6.6.

In the initialisation configuration *Init*, the *init* flag is set to *False* for every worker, implying that the workers have not yet completed the initialisation phase. Furthermore, *c_exchange c* and *c_propagate c* must satisfy the require-

```
definition next_propagate :: ('p, 't, 'loc) configuration
                          ⇒ ('p, 't, 'loc) configuration
                          ⇒ 'p ⇒ bool where
next_propagate c0 c1 p =
    unchanged c_exchange c0 c1 ∧
    Some (c_propagate c1 p') = (
        if p' = p then Some (propagate_all (c_propagate c0 p'))
        else c_propagate c0 p') ∧
    init c1 = (init c0)(p := True)

definition propagate_all :: ('t, 'l) Propagate.configuration
                         ⇒ ('t, 'l) Propagate.configuration option
                           where
propagate_all c0 =
    while_option
        (λc. ∃loc. (c_worklist c loc) ≠ {}_z)
        (λc. SOME c'. ∃loc t. Propagate.next_propagate c c' loc t)
        c0
```

Figure 6.5: Formal specification of the *next_propagate* action.

ments on the initial configurations in their respective models. This allows us to take advantage of the models' safety properties and invariants.

Finally, *Init* imposes a relationship between *pointstamps (c_propagate c p)* and *glob (c_exchange c) p*. Both of these fields store the view of the system's capabilities at the worker *p* in their respective models. The initial configuration *Init* thus requires that they are equivalent.

6.6 Chapter Summary

We have introduced a formal model of the Timely Dataflow Progress Tracking protocol the goal of which is to continuously re-compute location's *frontiers* from the true set of system's capabilities. Our model combines the model of the *Propagation* algorithm introduced in Chapter 4 with the model of the *Exchange* algorithm (also known as the Naiad Clock Protocol [1]).

The model defines a computation as a stream of configurations consisting of per-worker *Propagate.configurations* and a single *Exchange.configuration*. The computation evolves with respect to four next-state actions: *next_performop*, *next_send_update*, *next_recv_update* and *next_propagate*. We have given a formal definition of each of these actions.

```
definition Init where
Init c  =
    Exchange.Init (c_exchange c) ∧
    (∀p . c_init c p = False ∧
          Propagate.Init (c_propagate c p)) ∧
    (∀p loc t . zcount (pointstamps (c_propagate c p) loc) t =
                zcount (glob (c_exchange c) p) (loc, t))

definition Next :: ('loc, 't :: order) configuration stream
                ⇒ bool where
  Next s ≡ (
    let c0 = shd s;
        c1 = shd (stl s)
    in
    ∃p Δm Δp . next_performop c0 c1 p Δm Δp ∨
    ∃p tt . next_send_update c0 c1 p tt ∨
    ∃p q . next_recv_update c0 c2 p q ∨
    ∃p . next_propagate c0 c1 p
 )

definition Spec :: ('loc, 't :: order) computation ⇒ bool where
  Spec ≡ holds Init aand alw Next
```

Figure 6.6: The complete specification of the Progress Tracking protocol.

Chapter 7

Safety

This chapter states the main safety property of the Timely Dataflow Progress Tracking protocol, a set of additional protocol invariants, and gives an informal proof of the main safety property.

7.1 Safety Property

The Progress Tracking protocol aims to continuously re-compute location *frontiers* as the lower-bound of possible future timestamps arriving at a given location. As mentioned before, conservative frontiers are critical for the correctness of the computation. They provide locations with information about which phases of computation have passed and which ones are still in progress. The locations thus know when it is safe to output results and reclaim any memory associated with a certain time interval.

The set of future timestamps is determined by the active capabilities and the *could-result-in* relation. The set of active capabilities is tracked by the *nrec* field of the *Exchange* protocol. The safety property of the Timely Dataflow Progress Tracking protocol formally describes the relationship between location *frontiers* and the set of system's capabilities *nrec*.

The safety property defined in Figure 7.1 is very similar to the local safety property described in Chapter 5. It states that for every system's capability $(loc1, t1)$, if this capability *could result in* $(loc2, t2)$, then the *frontier* at location *loc2* computed by *any* worker is *not* ahead of $t2$.

Intuitively, the safety property guarantees that the *frontiers* at all locations at all workers reflect the full set of system's capabilities. Any new capability that can be generated from the current set of system's capabilities cannot be smaller than the *frontier* of a location it refers to.

The *spec_implies_safe* lemma states that in any configuration c in the computation s satisfying the specification *Spec*, all initialised workers are in a *safe*

```
definition safe :: ('p, 't, 'loc) configuration ⇒ bool where
safe c p ≡ ∀loc1 loc2 t1 t2.
    zcount (nrec (c_exchange c)) (loc1, t1) > 0
    ∧ could_results_in (loc1, t1) (loc2, t2)
    ⟶ supported_by t2 (implications (c_prop c p) loc2)

lemma spec_implies_safe:
    assumes Spec s
    shows   alw (holds (λc. ∀p. init c p ⟶ safe c p)) s
```

Figure 7.1: Safety property of the Timely Dataflow Progress Tracking protocol.

state.

7.2 Informal proof of the safety property

The proof of the safety property relies on three propositions, the proofs of which are omitted.

Proposition 7.1 *Fix a configuration c from the computation s satisfying Spec s. Then, every initialised worker p is in a safe configuration with respect to its local view of the system's cpabilties recorded by pointstamps. Formally,*

$$c_init\ c\ p \longrightarrow Propagate.safe\ (c_propagate\ c\ p)$$

A worker's local safety property is defined in Chapter 5 and describes a worker as locally safe if the *frontiers* at all locations reflect the approximated view of system's capabilities recorded by *pointstamps (c_prop c)*. The above proposition claims that a worker p is locally safe if it has been initialised (i.e., completed at least one propagation round).

Definition 7.2 *Let S be a signed multiset of pointstamps. Then S is* ***nonpositive up to x****, if and only if all pointstamps smaller or equal to x have a non-positive multiplicity in S. Formally,*

$$nonpos_upto\ S\ x \longleftrightarrow \left(\forall y\ .\ y \leq x \longleftrightarrow\ zcount\ S\ y\ \leq 0\right)$$

Proposition 7.3 *Fix a configuration c from the computation s satisfying Spec s. Then, if glob (c_exchange c) p — the approximated view of system's capabilities at worker p — is nonpositive upto a pointstamp x, then so is nrec (c_exchange c). Formally,*

$$nonpos_upto\ (glob\ (c_exchange\ c)\ p)\ x \longrightarrow nonpos_upto\ (nrec\ (c_exchange\ c))\ x$$

The Proposition 7.3 is a stronger version of the *InvGlobVacantUptoImplies-Nrec* – a Naiad Clock Protocol invariant [2] stating that the worker's approximated view of system capabilities *glob c p* is a conservative view of system's capabilities *nrec c*. In particular, it states that if *glob (c_exchange c) p* is nonpositive upto some pointstamp x, then there are no active system's capabilities that *could result in x*.

Proposition 7.4 *Fix a configuration c from the computation s satisfying Spec s. Then, for all workers p, locations loc and timestamps t we have:*

$$\begin{aligned}&\textit{zcount (pointstamps (c_propagate c p) loc) t} =\\&\textit{zcount (glob (c_exchange c) p) (loc, t)}\end{aligned}$$

The Proposition 7.4 describes the correspondence between the *Propagation* model and the *Exchange* model. It states that *pointstamps (c_prop c p)* are equivalent to *glob (c_exchange c) p* with the only distinction that the former is defined in terms of timestamps and the latter one in terms of pointstamps.

7.2.1 Proof of the safety property

Lemma 7.5 *Fix a configuration c from the computation s satisfying Spec s. Then, if a pointstamp x has a strictly positive multiplicity in (nrec (c_exchange c)) then, at any worker p, there exists a pointstamp y ≤ x with a strictly positive multiplicity in (glob (c_exchange c) p). Formally,*

$$\begin{aligned}&\textit{zcount (nrec (c_exchange c))}\ x > 0\\\longrightarrow\ &\exists\, y\,.\, y \le x \wedge\ \textit{zcount (glob (c_exchange c) p)}\ y > 0\end{aligned}$$

Proof

$$\begin{aligned}&\textit{zcount (nrec (c_exchange c))}\ x > 0\\\overset{\textit{nonpos_upto}\ \text{def.}}{\Longrightarrow}\ &\neg \textit{nonpos_upto (nrec (c_exchange c))}\ x\\\overset{\text{Proposition (7.3)}}{\Longrightarrow}\ &\neg \textit{nonpos_upto (glob (c_exchange c) p)}\ x\\\overset{\textit{nonpos_upto}\ \text{def.}}{\Longrightarrow}\ &\exists y.\ y \le x \wedge \textit{zcount (glob (c_exchange c) p)}\ y > 0\end{aligned}$$

□

Theorem 7.6 *Fix a configuration c from the computation s satisfying Spec s. Then, a worker that has completed the initialisation phase is in the safe state with respect to all system's capabilities.*

$$\textit{init c p} \longrightarrow \textit{safe c p}$$

Proof Figure 7.1 defines *safe* in terms of all pairs of capabilities (loc_1, t_1) and (loc_2, t_2) such that (loc_1, t_1) has a strictly positive multiplicity in *nrec* and (loc_1, t_1) *could result in* (loc_2, t_2). Then, a configuration is *safe* if for every initialised worker p, t_2 is *supported by implications* at location loc_2.

We prove the above theorem for an arbitrary initialised worker p and pair of such capabilities (loc_1, t_1) and (loc_2, t_2):

$$\begin{aligned} & \textit{init c p} \\ \wedge\ & \textit{zcount (nrec (c_exchange c))}\ (loc_1, t_1) > 0 \\ \wedge\ & (loc_1, t_1) \leq (loc_2, t_2) \end{aligned} \tag{7.1}$$

From equation 7.1, we get:

$$\textit{zcount (nrec (c_exchange c))}\ (loc_1, t_1) > 0 \tag{7.2}$$

$$\overset{\text{Lemma (7.5)}}{\Longrightarrow} \exists (loc_0, t_0)\ .\ (loc_0, t_0) \leq (loc_1, t_1)\ \wedge \\ \textit{zcount (glob (c_exchange c) p)}\ (loc_0, t_0) > 0 \tag{7.3}$$

$$\overset{\text{Propositon (7.4)}}{\Longrightarrow} \exists (loc_0, t_0)\ .\ (loc_0, t_0) \leq (loc_1, t_1)\ \wedge \\ \textit{zcount (pointstamps (c_propagate c p)}\ loc_0\ )\ t_0) > 0 \tag{7.4}$$

From Theorem 7.6 assume that the worker p is initialised. Then, by Proposition 7.1, worker p is locally safe. Furthermore, from the transitive property of partial order on pointstamps, we get that $(loc_0, t_0) \leq (loc_2, t_2)$. Finally, the definition of local safety and Equation 7.4 yields:

$$\textit{supported_by}\ \big(\textit{implications (c_propagate c p)}\ loc_2\big)\ t_2. \qquad \square$$

7.3 Chapter Summary

In this chapter, we have defined the safety property of the Timely Dataflow Progress Tracking protocol and given an informal proof of this safety property. The configuration is said to be in a *safe* state if for all workers, a location's *frontier* holds a valid lower-bound of all possible future timestamps arriving at that location. The set of possible future timestamps is determined by all the system's capabilities, recorded by the *nrec* field, and the *could-result-in* relation.

At the time of writing, we have a partial formal proof of the above theorem. The missing piece is showing that *Spec s* implies Proposition 7.1 and Proposition 7.3.

Chapter 8

Comparative Testing of the Timely Dataflow Propagation algorithm

In this chapter, we describe our approach of comparing the existing Rust-based implementation of the Timely Dataflow with the new specification and invariants to check the validity of the Propagation model and to test the correctness of the implementation. For the sake of conciseness, in this chapter, we will use *Timely dataflow* to refer to the Rust-based implementation of the Timely Dataflow model.

8.1 Overview

The framework compares the location frontiers computed by *Timely dataflow* with those computed by the formal model at the end of each propagation round for an arbitrary dataflow graph and input stream of data. The framework consists of multiple components shown in Figure 8.1.

First, an arbitrary dataflow program is executed in *Timely dataflow*, producing an output log-file. By arbitrary, we mean any program that can already be implemented in *Timely dataflow*, including hierarchical dataflows. A hierarchical dataflow contains operators which are in themselves dataflow graphs. We call each individual dataflow graph a *scope*.

The output log-file contains information about the dataflow graph topology, all created and dropped capabilities, and points at which workers initiate propagation rounds. The log-file is in turn processed by a *Propagation* executable, a wrapper around the Propagation component of *Timely dataflow*, also implemented in Rust. The executable parses the file, and for each *scope-worker* combination, it runs the Propagation algorithm of *Timely dataflow* in isolation. Finally, the executable generates a pair of Isabelle theory files for each *scope-worker* pair. These theory files record location frontiers

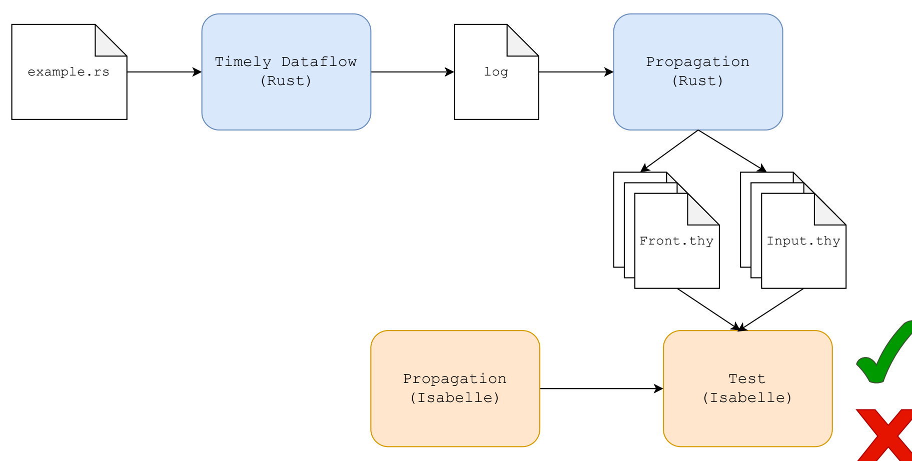


Figure 8.1: Comparative testing framework of the Propagation algorithm

computed by *Timely dataflow*, and other information necessary to execute the Propagation algorithm in Isabelle.

Furthermore, we have implemented a generic executable version of the Propagation algorithm in Isabelle and shown that it is a refinement of the formal model presented in Chapter 4.

Finally, for every *scope-worker* pair, the framework executes the Propagation algorithm in Isabelle and compares the location frontiers computed by the two implementations, yielding a *Success* or *Fail* if any of the computed frontiers do not match. In the following subsections we discuss each component in more detail and eventually summarise our experience with code extraction from Isabelle.

8.2 Timely dataflow logging

This section is concerned with the output log-file generation. The first purpose of the log-file is to be able to reconstruct any dataflow graph topology, and must thus contain information about the operators, the channels between them, the internal operator connections, and their associated summaries. Secondly, to execute identical runs of the Propagation algorithm in Rust and Isabelle, the output file must record any dropped and created capabilities and the beginning of each propagation round.

Fortunately, *Timely dataflow* has an extendable logging infrastructure already supporting a set of useful loggers. The logger we found particularly useful for our purposes is the *TimelyLogger* recording amongst others the following two events: *OperatesEvent* marking the creation of an operator and *ChannelsEvent* recording the creation of a channel between a source and a target port.

To record all necessary data about the graph topology, we have extended the *Operates* event with a record of the operator's internal summaries, and added a new *ScopeEvent* which determines the summary and timestamp type implemented by a given *scope* (e.g., `usize`, `i64`, etc.).

Finally, we have implemented a new logger called *PropagationLogger* to record events driving the propagation algorithm. The *PropagationLogger* logs three types of events:

- *UpdateSourceEvent* recording created or destroyed capabilities at a given source port, for a given scope and worker.
- *UpdateTargetEvent* recording created or destroyed capabilities at a given target port, for a givens cope and worker.
- *StartPropagationEvent* marking the beginning of a propagation round for a given scope and worker.

With respect to the formal model of the Propagation algorithm, the first two events correspond to a *Propagate.next_change_multiplicity* action.

The third event marks the start of a propagation round for a given scope and worker. You may notice that we do not record every single propagation step modelled by the Propagation algorithm. Instead, *Timely dataflow* and our executable Isabelle implementation are tasked with inferring a sequence of steps that satisfy the *next_propagate* relation leading to an empty *worklist*. It is completely possible that each of the two implementations will choose a valid yet different sequence from each other. However, we have also shown that any valid sequence of *next_propagate* actions resulting in an empty *worklist* always computes the same location frontiers. As we are only interested in the state of location frontiers at the end of a propagation round, recording the start of a propagation round at this point is sufficient.

8.3 Generation of Isabelle theory files

This section is concerned with a Rust-based *Propagation* executable, that executes the propagation algorithm inside *Timely dataflow* in isolation from the parsed log file, and generates valid Isabelle theory files.

The input log-file is initially parsed using Rust's `serde_json` library to determine the number of workers, the list of existing scopes and their corresponding timestamp and summary types. The executable then instantiates one `PropagationMocker`[1] for each *scope-worker* pair. Each `PropagationMocker` parses the log-file once again, picking up the graph topology and the list of

[1] In the code we refer to this by `TrackerMocker` to adhere to the naming conventions of Timely Dataflow Rust implementation.

actions relevant to the given scope. Subsequently, each `PropagationMocker` reconstructs the dataflow graph topology of the scope, and executes the propagation algorithm from the list of recorded actions.

We say that the propagation is run in isolation because the operators are completely inactive while the algorithm is running. The operators neither send and receive messages, nor create and destroy capabilities, nor keep any internal state. Instead, capability changes are deduced from the list of logged actions, bypassing the interactions between the system and the operators.

At the end of these isolated propagation runs, each `PropagationMocker` produces two Isabelle theory files: *Input.thy* and *Frontiers.thy*.

Input.thy

The *Input.thy* file acts as an "input" for the executable Isabelle implementation of the Propagation algorithm. It starts by instantiating the timestamp and summary types. The following example shows the case when both summary and timestamps are of type $\mathbb{N}$.

```
type_synonym t = nat
type_synonym sum = nat
```

Next, the file contains the description of the graph topology defined by the `summary` function (see Section 4.2) and the set of default capabilities for every location. Finally, the file records a list of actions driving the propagation algorithm, termed *trace*. The code snippet below shows the start of such *trace* defined in terms of locations ($op \times port$ pairs) and the set of timestamps t. The *PR* entries mark the start of a propagation round while the *CM* entries mark created (positive) or destroyed (negative) capabilities.

```
definition trace :: (op × port, t) Action list where
trace = [
    (PR),
    (CM (Op 0, src 0) 0 1),    (* one created capability with timestamp 0 *)
    (PR),
    (CM (Op 1, src 0) 0 -1),   (* one dropped capability with timestamp 0 *)
    (CM (Op 5, trg 0) 3 1),    (* one created capability with timestamp 3 *)
    (PR),
    ...
]
```

Frontiers.thy

The *Frontiers.thy* file contains a single function definition called *rust_frontier* which describes the state of frontiers at all locations at the end of each

propagation round. The value of *rust_frontiers* at round n corresponds to the n'th *PR* entry in the definition of the *trace* in the corresponding *Input.thy* file.

8.4 Executable Isabelle model of the Propagation algorithm

The Isabelle model of the Propagation algorithm presented in Chapter 4 is descriptive, defined in terms of two next-state relations. For any pair of configurations, the model tells us if either of the relations is satisfied by the two configurations.

However, to execute the Propagation algorithm with Isabelle we need a prescriptive implementation of the model that tells us how to compute a new configuration from an old configuration. We have implemented such an executable version of our model, called `take_step`, shown in Figure 8.2. Just as in the original model, the *take_step* function can preform one of two actions, specified by the *Action* argument:

```
datatype ('loc :: enum, 't) Action =
  CM 'loc 't int |  (* CM loc t n -> add n (loc, t) capabilities *)
  PR                (* PR -> perform one propagation step*)
```

In the *CM* case, the function creates or removes n copies of timestamp t at location *loc*.

In the *PR* case, the function actively finds a smallest timestamp in *worklist* given the *less_t* argument defining the total order on the set of timestamps *'t*. This total order is not part of the model, but is merely used to resolve the non-determinism of choosing a smallest value from the set of partially ordered timestamps. Subsequently, the function propagates information about the timestamp to neighbouring workers in the same way *next_propagate* action does.

It is helpful to note that a large difference between the executable Isabelle implementation in Figure 8.2 and the formal specification in Chapter 4 comes from the use of universal quantifiers. While the universal quantifiers are often helpful to intuitively describe the workings of a protocol, they are not naturally executable. As a result, we omitted these quantifiers and replaced them with executable constructs. Finally, we have formally proved that our executable Isabelle implementation satisfies the formal specification of the model from Chapter 4.

```
fun take_step :: ('t ⇒ 't ⇒ bool)
              ⇒ ('loc :: {linorder, enum} , 't) Action
              ⇒ ('loc, 't) configuration
              ⇒ ('loc, 't) configuration where
take_step _ (CM loc t delta) c =
    (let pointstamps_old = pointstamps c loc;
         pointstamps_new = (pointstamps c)(loc :=
            (update_zmultiset (pointstamps c loc) t delta))
    in
    c ⦇ pointstamps := pointstamps_new,
        (* update worklist with changes to pointstamps-infimum
           at location loc *)
        worklist := worklist c (loc :=
            worklist c loc +
            (infimum_change pointstamps_old (pointstamps_new loc)))
      ⦈)
| take_step t_less PR c =
    (let (t, loc) = mymin t_less (t_loc_pairs c;
         implications_old = implications c loc;
         (* update implications at location loc with worklist entries
            corresponding to location loc and a smallest timestamp t *)
         implications_new = (implications c) (loc :=
            implications c loc +
            (filter_zmset (λt'. t' = t)(worklist c loc));
         (* remove all copies of timestamp t from worklist at location loc *)
         worklist_removed_loc = (c_worklist c) (loc :=
            (filter_zmset (λt'. t' ≠ t)(worklist c loc)))
    in
    c ⦇
        (* Add any propagated implications to teh final worklist *)
        c_worklist := λ loc'.
            (worklist_removed_loc loc' +
             after_summary
                (infimum_changes (implications_old) (implications_new loc))
                (summary loc loc')),
        implications := implications_new
    ⦈)
```

Figure 8.2: Executable Isabelle implementation of the Propagation model

8.5 Supported types

The *take_step* function is generic, with types classes constrained by the same set of requirements as the original model. In particular, the set of locations *'loc* must be enumerable, while the set of timestamp *'t* must satisfy partial order, and the set of summaries *'sum* is an additive monoid.

When we try to evaluate the function in Isabelle using specific type instantiation, we must make sure that the types satisfy the above requirements. Of course, many existing Isabelle types already do so. For example, the set of natural numbers under addition is an additive monoid and at the same time implements the partial order type class. However, many *Timely dataflow* programs contain more complex types, including lists or products. As a result, we define a series of additional types and shown that they satisfy the necessary properties.

In particular, we have defined the following types to describe locations:

- *op* - an operator type

  ```
  typedef op = {0.. MAX_OP}
  ...
  lift_definition Op :: int ⇒ op
  ...
  ```

  e.g.: `Op 6`

- *pnum* - a port number type

  ```
  typedef pnum = {0.. MAX_PNUM}
  ...
  lift_definition Pnum :: int ⇒ pnum
  ...
  ```

  e.g.: `Pnum 0`

- *port*

  ```
  datatype port = Src pnum | Trg pnum
  ```

  e.g.: `Src (Pnum 0)` which abbreviates to `(src 0)` or `Trg (Pnum 10)` abbreviated to `(trg 10)`

- *loc = (op × port)* - a location type
  e.g.: `(Op 0, src 0), (Op 10, src 5)`

We have shown all these types to be enumerable, and endowed them with linear order.

For the set of timestamps and summaries, we support the product type in addition to the basic Isabelle types which already satisfy the necessary

requirements (e.g., the set of natural numbers *nat*). Addition on a product type is defined as a component-wise addition, while the partial order is defined by the Equation 4.1.

8.6 Comparative testing

In this section, we describe how the generated Isabelle theory files and the executable Isabelle implementation of the Propagation algorithm come together to carry out the comparison test.

First, we construct an initial state satisfying the initial configuration of the Propagation model from the *default_capabilities* supplied in the *Input.thy* file as shown in Figure 8.3.

```
definition initial_state where
initial_state ≡ ⦇ c_worklist =  (λloc . infimum (default_capabilities loc)),
                  c_pointstamps = default_capabilities,
                  c_implications = (λloc . {}z) ⦈

lemma Propagate.Init initial_state
   ...
```

Figure 8.3: The definition of the initial state of the computation and a lemma stating that this state satisfies the conditions set on the initial state by the Propagation algorithm.

Next, we run the program by recursively apply the *take_step* function to the *initial_state* given the *trace* of actions supplied in the *Input.thy* file. The implementation of the *run* is shown in Figure 8.4. For every *CM* entry, we take a single *CM* step. However, for each *PR* entry, we repeatedly take *PR* steps until we reach an empty *worklist* at all locations, or we exceed a predefined upper bound on the number of allowed propagation iterations.

Why is the upper bound necessary? Isabelle asks for every function definition to be terminating (see Section 3.2). At the time of writing, we have not formally proved that the Propagation model guarantees termination of an arbitrary propagation round but strongly believe it to be true. Imposing an upper bound on the number of *PR* steps taken in a single propagation round allows us to execute this function with Isabelle bypassing the actual proof of termination. We have set this maximum sufficiently high (one million iterations) to prevent our comparative testing framework from producing unnecessary false negatives.

The *run* function outputs a list of configurations called *configs*. Each of these configurations corresponds to the state of the algorithm at the end of one propagation round.

The last step left is to compare the location frontiers computed by the *run* function (stored by *configs*) with frontiers computed by *Timely dataflow* and recorded in the *Frontiers.thy* file. The test returns either a positive or a negative value, describing if the location frontiers computed by the two implementations are equivalent.

We have tested our framework on all example programs already included in the *Timely dataflow* repository as well as a set of additional programs specifically designed for iterative dataflows, hierarchical dataflows, out-of-order execution, and a range of partially ordered timestamp and summary types. We have found no deviations between the frontiers computed by *Timely dataflow* and those computed by our model.

```
definition MAX_TRIES:: nat where MAX_TRIES = 1000000

function run :: nat
             ⇒ nat
             ⇒ (op × port, sum) configuration
             ⇒ (op × port, sum) configuration list where
run prop_it step_it c = (
    if MAX_TRIES < prop_it then [] else
    if length trace ≤ step_it then [] else
    (let step = trace ! step_it in
        case step of
            PR ⇒ (
                if (worklist_is_empty c)
                then c # (run 0 (step_it + 1) c)
                else  (run (prop_it + 1) step_it (take_step step c)))
          | _ ⇒ run 0 (step_it + 1) (take_step step c)))
```

Figure 8.4: The definition of a recursive function `run` which performs a complete propagation algorithm from a set of actions and produces a list of configurations, one configuration for each completed propagation round.

8.7 Retrospection on Isabelle for executable code

Isabelle/HOL offers a code generator facility which allows the user to turn a certain class of HOL (higher-order-logic) specifications into corresponding executable code in the programming languages such as SML [31], OCaml [33], Haskell [16] and Scala [34]. Isabelle achieves code generation with a concept of *shallow embedding* which identifies logical entities like constants, types, or classes with concrete entities in the target language such as SML or Haskell. Isabelle guarantees partial correctness of the executable code. In practice, this means that if $f\ x$ terminates and evaluates to y then $f\ x = y$ is provable in Isabelle.

Code Generation

Isabelle was not designed for the sole purpose of code generation and it is in fact more expressive than the above-mentioned programming languages. For example, Isabelle allows us to define a quantified expression over infinite sets, something that cannot be naturally executable.

One of the ways to deal with such situations is to locally derive an executable specification and show that it refines the associated non-executable one. These *code equations* then override the original code definitions upon code generation.

This process is called *program refinement* and it lets a user separate code generation from the original HOL formalisation. The user is allowed to implement any executable version of the original specification as long as it is shown to satisfy the original specification. On the other hand, code generation thus becomes more involved and time consuming.

In our implementation of the executable model, we had to define a number of code equations for creating and manipulating sets and antichains, whose code equations contain quantifiers. While quantifiers have an associated code equations, they only refine finite and enumerable type classes which the generic timestamp and pointstamp types are not instances of. Unfortunately, if a constraint on a type class set by a code equation is not satisfied, Isabelle's error message does not provide details about where in the execution stack the problem occurred but instead raises a generic *Wellsortedness error*. As a result, we have spent a large part of implementation tracking down the missing executable parts.

Verbosity

The second issue we came across is the verbosity of defining new datatypes and proving they are instances of existing type classes, such as the *enum* class or the *finite* class. While all of our type definitions were simple and the proofs quite trivial, the (unoptimised) file which contains the type definitions of the *operator* and *port* types and necessary proofs is almost 200 lines long excluding white-space.

Parsing input

In our current implementation of the comparative testing framework, we generate a series of Isabelle theory files to "pass" the details of dataflow programs to the Isabelle executable. Originally, we tried to by-pass this step by generating a well-typed input to Isabelle directly. While this approach would be more efficient, it is naturally more complex and lacks a good documentation.

Since performance was not our priority, we have decided to implement a Rust-based parser described in the previous sections which auto-generates valid Isabelle theories. We have found it particularly simple to implement and maintain while our model and the executable Isabelle implementation continued to change.

8.8 Chapter Summary

In summary, we have implemented a comparative testing framework which compares location frontiers computed by *Timely dataflow* – the Rust-based implementation of Timely Dataflow – with the location frontiers computed by the formally specified model. The framework consists of multiple components in addition to the *Timely dataflow*. Firstly, a Rust-based executable that performs the Propagation algorithm in isolation and generates a series of Isabelle theory files. Secondly, an executable Isabelle implementation of the formally specified Propagation model, and finally, an Isabelle program that performs the comparison of the frontiers computed by the two implementations.

We have made our framework flexible enough to work with valid *Timely dataflow* program. The framework was tested on the full set of example programs included in the *Timely dataflow* repository as well as a set of additional programs specifically designed for iterative dataflows, hierarchical dataflows, out-of-order execution, and a range of partially ordered timestamp and summary types. We have found no deviations between the locations frontier computed by the *Timely dataflow* and those computed by the formal model.

While this approach cannot guarantee the correctness of the *Timely dataflow* implementation, it strongly suggests that *Timely dataflow* indeed computes correct location frontiers and that our formal model of the Propagation algorithm is valid.

Finally, this framework allows users to check the correctness of their running *Timely dataflow* programs in retrospect. As long as the the program has the two necessary loggers enabled, a user noticing strange results can simply re-run the Propagation algorithm inside the framework to determine if the strange behaviour is caused by incorrectly computed location frontiers.

Chapter 9

Conclusion

In this chapter, we summarise our contributions, give an overview of the ongoing work and propose ways in which this work can be be further extended.

9.1 Main contributions

This thesis set out to model the core coordination component of Timely Dataflow termed the *Progress Tracking protocol*. We have identified two components of the protocol: an *exchange algorithm* and a *propagation algorithm*. The *exchange algorithm*, also known as the Naiad Clock Protocol, has been formerly formalised and verified by Abadi et al. [2].

Moreover, we have combined the individual models of the *exchange* and the *propagation algorithm* and presented a formal specification of the complete Progress Tracking protocol. We have identified the necessary assumptions that the protocol makes on the types and the structure of the dataflow graph. Furthermore, we have defined the protocol's main safety property and given an overview of its mechanically verified proof.

To test the validity of our model and of the existing Rust-based implementation, we have implemented an executable Isabelle specification of the *propagation algorithm* and a comparative testing framework using a combination of Rust and Isabelle. The framework is flexible enough to work with any valid Timely Dataflow program, including iterative and hierarchical dataflows, and can be extended to arbitrary timestamp, summary and location types.

A formally verified model of the Timely Dataflow Progress Tracking protocol provides a clearer abstraction for other components of Timely Dataflow that interact or depend on its progress tracking component. Furthermore, we believe that the specification will prove helpful in integrating the proto-

col in new systems. In particular, there has been recent interest in porting parts of the protocol to FPGAs.

9.2 Ongoing work

In this section, we discuss the ongoing work in extending the *exchange algorithm*. We have by now incorporated these changes into the model and are currently proving that the invariants of the original model continue to hold.

Relaxing the exchange algorithm

The model of the *exchange algorithm* imposes a set of constraints on how the workers create and drop capabilities. Note that Abadi refers to these by *pointstamps*. Having re-implemented and applied the mode of the *exchange algorithm* into our model of the Progress Tracking protocol, we have found these constraints to be too limiting and unfaithful to the implementation of Timely Dataflow.

Concretely, the *exchange algorithm* specifies that a worker can create a new pointstamps x if and only if it also drops a capability y, such that $y < x$. However, in Timely Dataflow, operators are allowed to mint new capabilities from the capabilities they own without the need to destroy some.

Removing dependencies on a global state in the exchange algorithm

Secondly, the *exchange algorithm* specifies that a worker can drop capabilities owned by any other worker, as long as each capability is dropped at most once. This is achieved by modelling the true global state of all system's capabilities called *nrec*. The workers make their decisions based on the value of *nrec* and directly modify it whenever they create or destroy capabilities.

Of course, in practice, workers do not have access to such a global state. Therefore, we propose that each worker models the set of its own capabilities and uses this information when dropping existing capabilities.

Proofs of the invariants of the Progress Tracking protocol

It remains to show that our model of the Timely Dataflow Progress Tracking protocol is a refined model of the *propagation algorithm* and the *exchange algorithm*. This proof is necessary to show the Progress Tracking protocol preserves the invariants guaranteed by both of the algorithms. While we have described a mechanically verified proof of the protocol's main safety property in Chapter 7, this proof assumes that the invariants of the *propagation* and *exchange* protocol are satisfied. The final proof of the protocol's main safety property is thus incomplete at the time of writing.

9.3 Future work

In this section, we discuss the challenges lying ahead.

Support for hierarchical dataflows

Timely Dataflow supports hierarchical dataflows in which each operator can itself be a dataflow graph, also called a *scope*. This approach provides a level of abstraction by hiding details that the outer dataflow graph does not need to know. In terms of the Progress Tracking protocol, this model can yield large performance benefits since all propagation rounds are carried out inside smaller scopes as opposed to one large and complex dataflow graph.

Extending our model to support hierarchical dataflows would involve modelling the exchange of progress information between outer and inner scopes.

Extending the executable Isabelle implementation

At the moment, the executable Isabelle implementation performs the *propagation algorithm* only. The natural next step would be to implement the exchange of propagation information between workers and the complete Progress Tracking protocol.

We have postponed this work due to the fact that we are currently extending the model of the *exchange algorithm*, which the implementation of the executable Progress Tracking protocol in Isabelle relies on.

Appendix A

Example of a non-terminating propagation round

In this example, all timestamps and summaries are natural numbers. The addition natural numbers implements the *result_in* relationship and the addition on the summary type.

Figure A.1 depicts an intermediate state of a dataflow computation. The dataflow graph consists of four locations loc_1 to loc_4 connected in a cycle. The edges of the graph carry summaries. The edge from loc_3 to loc_4 is the only one with a non-zero summary ensuring that the cycle always increases a timestamp passing through it.

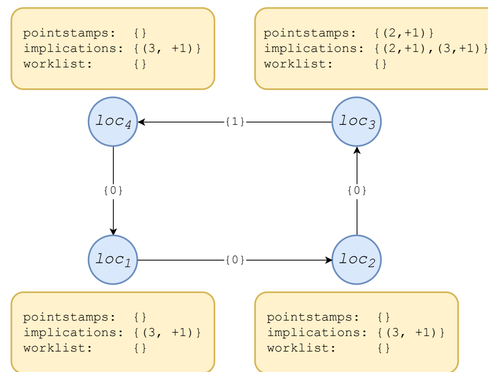


Figure A.1: An intermediate state of a dataflow computation.

In this intermediate state, the system is aware of one capability – $(loc_3, 2)$. This capability is recorded in *pointstamps* at location loc_3, reflected in the locations' *implications* and by extension their frontiers.

Next, the algorithm carries out a *next_change_multiplicity* action removing the capability from the *pointstamps*. The change is recorded in the *worklist* at

location loc_3 as shown in Figure A.2.

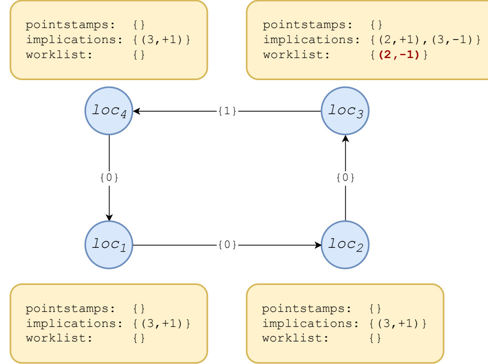


Figure A.2: The state after action *next_change_multiplicity* loc_3 *2 (-1)*.

Now, we start a propagation round always picking a largest timestamp.

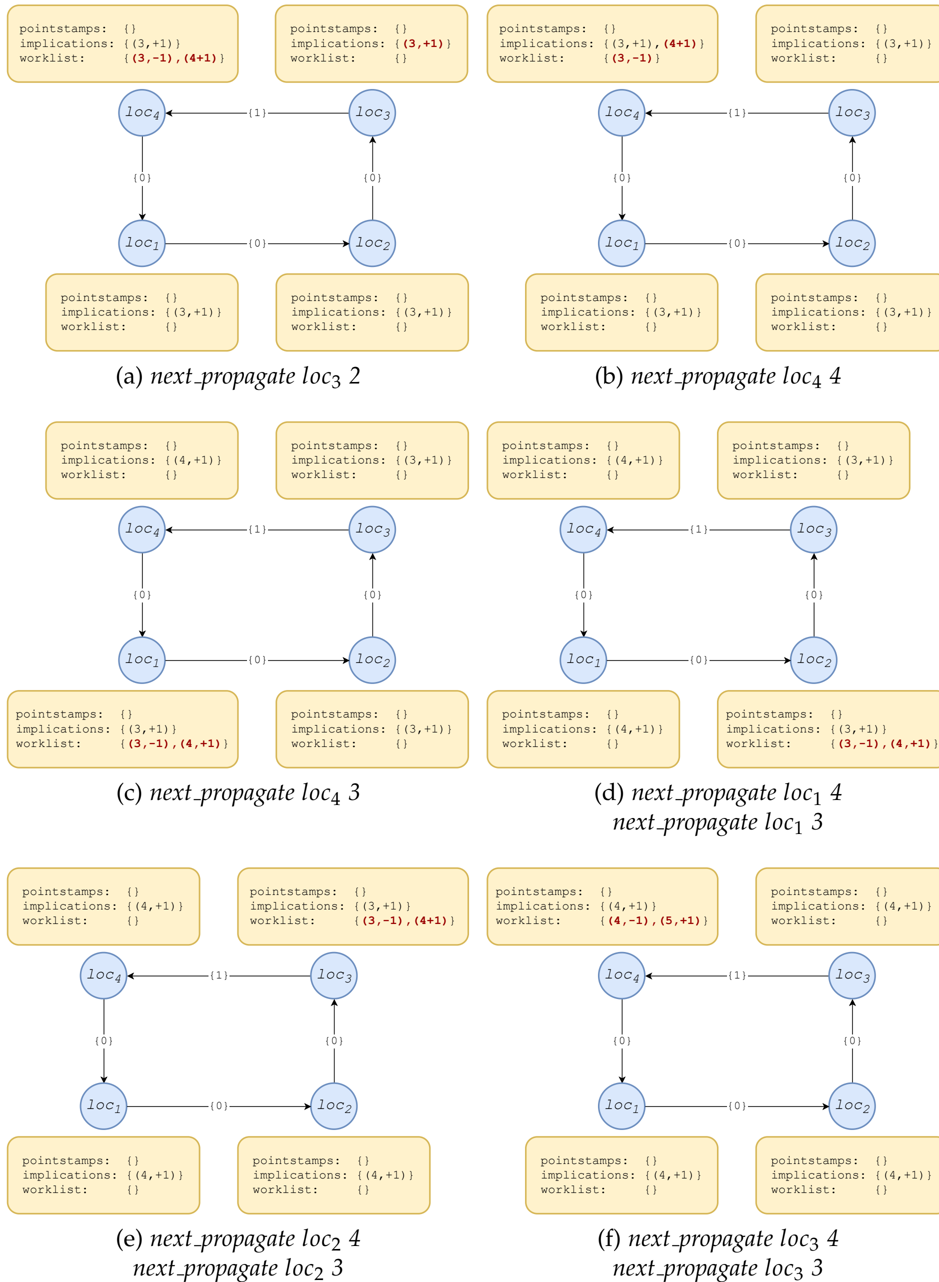
The first action is deterministic as there is exactly one entry across all the *worklists*, namely timestamp 2 at location loc_3. We perform a *next_-propagate* action by applying this entry to the *implications* at location loc_3 and propagating the changes to *implications infimum* to outgoing location loc_4 as shown in Figure A.3a.

At this point, the *worklist* contains two items, a timestamp *3* and a timestamp *4* at location loc_4. Remembering to always pick a largest timestamp, we perform a *next_propagate* loc_4 *4* action leading to the state in Figure A.3b. After the action, the worklist contains only one element. The next *next_-propagate* action thus leads to the state in Figure A.3c.

We continue in a similar fashion always propagating a larger timestamp first. We see that as we progress throughout the cycle, the *infimum* of *implications* at all locations increases from 3 to 4. This is the first sign that something is going wrong. Clearly, by removing the only capability in the system, the propagation round should ensure that the *implications* should gradually become empty, instead of increase.

Indeed, as we complete a full cycle and the implications get propagated back to location loc_4 as shown in Figure A.3f, we reach the state almost identical to one in Figure A.3a, only with all timestamps increased by one.

In conclusion, in this example, a propagation algorithm choosing a largest timestamp instead of a smallest one never manages to clear the *implications* and instead continues to increase the *implications* indefinitely.



(a) *next_propagate* loc_3 *2*

(b) *next_propagate* loc_4 *4*

(c) *next_propagate* loc_4 *3*

(d) *next_propagate* loc_1 *4*
next_propagate loc_1 *3*

(e) *next_propagate* loc_2 *4*
next_propagate loc_2 *3*

(f) *next_propagate* loc_3 *4*
next_propagate loc_3 *3*

Figure A.3: Demonstration of an incomplete non-terminating propagation round.

Appendix B

Formal specification of the Exchange algorithm in Isabelle

The Exchange algorithm is parametrised by two type variables:

- *'p*: the set of workers,
- *'t*: the set of pointstamps.

State

```
record ('p, 't) configuration =
  nrec :: 't zmultiset
  temp :: 'p ⇒ 't zmultiset
  msg  :: 'p ⇒ 'p ⇒ 't zmultiset list
  glob :: 'p ⇒ 't zmultiset
```

The algoirhtm

```
definition next_performop :: ('p, 't) configuration
                           ⇒ ('p, 't) configuration
                           ⇒ 'p
                           ⇒ 't multiset
                           ⇒ 't multiset
                           ⇒ bool where
next_performop c0 c1 p Δm Δp ≡
    (let Δ = Δp − Δm
     in
     ∀t . (count Δm t) ≤ zcount (nrec c0) t ∧
     upright Δ ∧
     nrec c1 = nrec c0 + (Δ) ∧
     temp c1 = (temp c0)(p := temp c0 p + (zmset_of Δ)) ∧
     unchnaged msg c0 c1 ∧
     unchnaged glob c0 c1)
```

```
definition next_send_update :: ('p, 't) configuration
                            ⇒ ('p, 't) configuration
                            ⇒ 'p
                            ⇒ 't set
                            ⇒ bool where
next_send_update c0 c1 p tt ≡
   (let update = {#t ∈z temp c0 p . t ∈ tt#}
    in
    update ≠ 0 ∧
    upright (temp c0 p - udpate) ∧
    temp c1 = (temp c0)(p := temp c0 p - update) ∧
    msg c1 = (λp' q. if p' = p then msg c0 p q @ [update]
                     else msg c0 p' q) ∧
    unchanged nrec c0 c1 ∧
    unchanged glob c0 c1)

definition next_recvupd :: ('p, 't) configuration
                        ⇒ ('p, 't) configuration
                        ⇒ 'p
                        ⇒ 'p
                        ⇒ bool where
next_recvupd c0 c1 p q ≡
   (let κ = hd (c_msg c0 p q)
    in
    msg c0 p q ≠ [] ∧
    unchanged nrec c0 c1 ∧
    unchanged temp c0 c1 ∧
    msg c1 = (λp' q' = if p' = p ∧ q' = q then tl (msg c0 p' q')
                       else msg c0 p' q') ∧
    glob c1 = (glob c0)(q := glob c0 q + κ))
```

Initial configuration and full specification

```
(* Initial configuration *)
definition Init :: ('p, 't) configuration
                ⇒ bool where
Init c ≡
    ∀p. temp c p = {}z ∧
    ∀p1 p2. msg c p1 p2 = [] ∧
    ∀t. 0 ≤ zcount (nrec c) t ∧
    ∀p. glob c p = nrec c

(* The Next-state relation*)
```

```
definition Next where
Next s ≡ (
    let c0 = shd s;
        c1 = shd (stl s)
    in
    ∃p Δm Δp . next_performop c0 c1 p Δm Δp ∨
    ∃p tt . next_send_update c0 c1 p tt ∨
    ∃p q . next_recv_update c0 c2 p q
)

type_synonym ('p, 't) computation = ('p, 't) configuration stream

(* The algorithm specification *)
definition Spec :: ('p, 't) computation ⇒ bool where
Spec s ≡ holds Init s ∧ alw Next s
```

Bibliography

[1] Martín Abadi and Michael Isard. Timely dataflow: A model. In *Formal Techniques for Distributed Objects, Components, and Systems*, pages 131–145. Springer International Publishing, 2015.

[2] Martín Abadi, Frank McSherry, Derek G. Murray, and Thomas L. Rodeheffer. Formal Analysis of a Distributed Algorithm for Tracking Progress. In *Formal Techniques for Distributed Systems*, pages 5–19. Springer Berlin Heidelberg, 2013.

[3] ACL2 Version 8.3. http://www.cs.utexas.edu/users/moore/acl2. Accessed: 2020-04-16.

[4] T. Akidau, A. Balikov, K. Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, August 2013.

[5] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, August 2015.

[6] Apache Flink. https://flink.apache.org. Accessed: 2020-04-16.

[7] Apache Storm. https://storm.apache.org. Accessed: 2020-04-16.

[8] Frédéric Badeau and Arnaud Amelot. Using b as a high level programming language in an industrial project: Roissy VAL. In *ZB 2005: Formal Specification and Development in Z and B*, pages 334–354. Springer Berlin Heidelberg, 2005.

[9] Clemens Ballarin. Tutorial to locales and locale interpretation, 2010.

[10] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs, Oct 2007.

[11] The Coq Proof Assistant. https://coq.inria.fr/. Accessed: 2020-04-16.

[12] Dafny: A Language and Program Verifier for Functional Correctness. https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/. Accessed: 2020-04-22.

[13] Alan Edelman. The Mathematics of the Pentium Division Bug. *SIAM Review*, 39:54–67, 1997.

[14] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In *Computer Mathematics*, pages 333–333. Springer Berlin Heidelberg, 2008.

[15] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving*, pages 163–179. Springer Berlin Heidelberg, 2013.

[16] Haskell language. https://www.haskell.org/. Accessed: 2020-04-16.

[17] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet. In *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP'15*. ACM Press, 2015.

[18] Isabelle. http://isabelle.in.tum.de. Accessed: 2020-04-16.

[19] Ioannis T. Kassios. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions, Aug 2006.

[20] Gerwin Klein, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, and Rafal Kolanski. seL4. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP'09*. ACM Press, 2009.

[21] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL'14*. ACM Press, 2014.

[22] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley, 2003.

[23] Leslie Lamport. Byzantizing Paxos by Refinement, Sep 2011.

[24] LEAN. `https://leanprover.github.io`. Accessed: 2020-04-16.

[25] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[26] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[27] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. Towards Verification of the Pastry Protocol Using TLA+, 2011.

[28] Nicholas D. Matsakis and Felix S. Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, November 2014.

[29] Frank McSherry. Github blog. `https://github.com/frankmcsherry/blog`. Accessed: 2020-04-16.

[30] Frank McSherry. Timely Dataflow. `https://github.com/TimelyDataflow/timely-dataflow`. Accessed: 2020-04-16.

[31] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[32] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad. In *Proceedings of the Twenty-Fourth {ACM} Symposium on Operating Systems Principles SOSP'13*. ACM Press, 2013.

[33] OCaml. `https://ocaml.org`. Accessed: 2020-04-16.

[34] Martin Odersky, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and et al. An overview of the scala programming language, 2004.

[35] James E. Rumbaugh. A parallel asynchronous computer architecture for data flow programs" mit ph, 1975.

[36] About seL4. https://sel4.systems/About/home.pml. Accessed: 2020-04-16.

[37] Timely Dataflow book. https://timelydataflow.github.io/timely-dataflow. Accessed: 2020-04-16.

[38] The tlaps project. https://tla.msr-inria.inria.fr/tlaps/content/Home.html. Accessed: 2020-04-22.

[39] P.A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, May 2003.

[40] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP'13*. ACM Press, 2013.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Modelling and Verification of the Timely Dataflow Progress Tracking Protocol

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Decova	Sara

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	Signature(s)
London, 14/05/2020	Sara Decova

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.