Linköping Studies in Science and Technology
Dissertations, No. 2006

Robust Stream Reasoning Under Uncertainty

Daniel de Leng

Linköping University
Department of Computer and Information Science
Artificial Intelligence and Integrated Computer Systems
SE-581 83 Linköping, Sweden

Linköping 2019

Edition 1:1

Thesis cover: A photo taken in Norrköping near (58.588510°N, 16.183002°W) on July 1st 2018, facing north-west, showing stepped waterfalls representing the incremental transformation of streams.

ISBN 978-91-7685-013-8
ISSN 0345-7524
URL http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-157633

Typeset using XeTeX

Printed by LiU-Tryck, Linköping 2019

Dedicated to the loving memory of Joan Grace de Leng (1921–2019), a strong, brave, and adventurous English lady who gave me this language, the courage to move abroad, an example of perseverance, a love of Star Trek, and an appreciation of birds. She was my grandmother, my nana, and a real-life Captain Janeway. Her spirit has been set free, but she will be sorely missed.

POPULÄRVETENSKAPLIG SAMMANFATTNING

Robust inkrementell slutsatsdragning utifrån osäkra informationsströmmar

Information finns överallt. Mycket av detta produceras och konsumeras som informationsströmmar. Vi har internetsamtal, tittar på video, och live-streamar händelser. Övervakningskameror samlar och skickar bilder kontinuerligt. Sensorer gör att vi kan kolla på hur vädret är just nu. Marknadsinformation gör att vi kan kolla på statusen för världens börser. Våra smartphones kan ge oss positionsinformation live som kan delas med andra. Dessutom observerar robotar sina närområden med hjälp av sensorer, såsom människor observerar sina närområden med sina sinnesorgan. Dessa informationsströmmar ger oss inkompletta ögonblicksbilder av världen där vi befinner oss. Dock kan informationsmängden göra det svårt att förstå världen. Det är därför viktigt för autonoma system att ha förmågan att förstå dessa informationsströmmar, till exempel genom automatisk slutsatsdragning. Inkrementell slutsatsdragning utifrån informationsströmmar, som också kallas för *stream reasoning* på engelska, är särskilt relevant för autonoma robotsystem i den fysiska världen. I den här avhandlingen fokuserar vi på två delar av problemet gällande robust inkrementell slutsatsdragning utifrån osäkra informationsströmmar.

Första delen handlar om hur ett system svarar på tidsrelaterade frågor om informationsströmmar. Vi kan använda en tidslogik för att beskriva händelsen på ett formellt sätt. Dessa händelser kan till exempel representera värden av en särskild aktie, en finansiell transaktion mellan två parter, eller nuvarande status av ett robotsystem. Logiska uttryck är användbara där vi vill kontrollera om logiska specifikationer uppfylls av ett system. En överträdelse av specifikationerna kan till exempel betyda att en särskilt aktie går ner för fort i värde, en suspekt finansiell transaktion har upptäckts, eller ett robotsystem agerar på ett ovanligt och otryggt sätt. Eftersom det ibland saknas information är förmågan att hantera osäkerhet ett viktigt problem.

Andra delen handlar om hur ett sådant system kan generera informationsströmmar på ett robust sätt. Många slutsatsdragningstekniker för logik tar inte hänsyn till ursprunget av de använda symbolernas tolkning i logiska specifikationer. Det är vanligt att man bara antar att informationsströmmar som krävs också finns. Men även om de är direkt tillgängliga så kan tillgängligheten ändras över tid. En potentiell lösning är att beskriva vilken sorts information som krävs, i stället för var information finns. Lösningen gör att det är möjligt för ett system att anpassa sig när informationsströmresurser blir otillgänglig medan de används för slutsatsdragning, genom att fortsätta generera informationsströmmen med hjälp av alternativa resurser.

Dessa två delar integrerades i ett ramverk för robust inkrementell slutsatsdragning utifrån osäkra informationsströmmar. Ramverket stödjer resonemanget om informationen som finns i strömmar, och om strömmarna själva som produkt av en strömsyntesprocess. Dessa förmågor kommer att bli viktigare ju mer informationsströmmar som genereras i vår digitala värld.

POPULAIRWETENSCHAPPELIJKE SAMENVATTING

Robuust automatisch redeneren met onzekere informatiestromen

Informatie is overal. Veel van deze informatie wordt geproduceerd en geconsumeerd in de vorm van informatiestromen. We houden online telefoongesprekken, kijken naar video-afleveringen, en streamen live gebeurtenissen. Toezichtcamera's verzamelen en versturen continu beeldmateriaal. Sensoren zorgen ervoor dat we actuele weersinformatie kunnen opvragen. Daarbij observeren robots hun omgeving met hulp van sensoren, zoals mensen hun omgeving observeren met behulp van zintuigen. Dergelijke informatiestromen geven ons incomplete momentopnamen van de wereld waarin we ons bevinden. Echter kan de hoeveelheid informatie het begrijpen van die wereld bemoeilijken. Het is daarom belangrijk voor autonome systemen om deze informatiestromen te kunnen begrijpen, bijvoorbeeld door middel van automatisch redeneren. Automatisch redeneren met informatiestromen, ook wel *stream reasoning* genoemd in het Engels, is in het bijzonder relevant voor autonome systemen die zich in de fysieke wereld begeven. In deze scriptie concentreren we ons op twee onderdelen van het probleem van robuust automatisch redeneren met onzekere informatiestromen.

Het eerste onderdeel gaat over hoe een systeem antwoorden kan geven op tijdsgerelateerde vragen over informatiestromen. We kunnen een tijdslogica gebruiken om gebeurtenissen op een formele manier te beschrijven. Die gebeurtenissen kunnen bijvoorbeeld gaan over de waarde van een bepaald aandeel, een financiële transactie tussen twee partijen, of de huidige status van een robotsysteem. Logische uitingen zijn handig wanneer we willen controleren of een systeem zich houdt aan een logische specificatie. Een overtreding kan bijvoorbeeld betekenen dat een bepaald aandeel te snel in waarde verliest, een verdachte financiële transactie ontdekt is, of dat een robotsysteem zich op een ongebruikelijke en gevaarlijke manier gedraagt. Omdat er soms informatie ontbreekt is het vermogen om om te gaan met onzekerheid een belangrijk probleem.

Het andere onderdeel gaat over hoe een dergelijk systeem informatiestromen op een robuuste wijze kan genereren. Veel technieken voor automatisch redeneren op basis van logica houden zich niet bezig met de oorsprong van de betekenis van de gebruikte symbolen in een logische specificatie. Het is gebruikelijk dat men simpelweg aanneemt dat de benodigde informatiestromen beschikbaar zijn. Echter, zelfs als ze direct toegankelijk zijn kan die toegankelijkheid over tijd variëren. Een potentiële oplossing is om te beschrijven welk soort informatie benodigd is, in plaats van wáár de informatie is. Dat zorgt ervoor dat het mogelijk is voor een systeem om zich aan te passen wanneer bronnen van informatiestromen ontoegankelijk worden terwijl ze in gebruik zijn voor automatisch redeneren. Dit kan door middel van het genereren van alternatieve informatiestromen met hulp van alternatieve middelen.

Deze twee delen zijn geïntegreerd in een raamwerk voor robuust automatisch redeneren met onzekere informatiestromen. Het raamwerk ondersteunt het redeneren met informatie in de vorm van stromen, en het redeneren over die stromen zelf als product van een syntheseproces. Deze vermogens worden belangrijker naar mate er meer informatiestromen gegenereerd worden in onze digitale wereld.

ABSTRACT

Vast amounts of data are continually being generated by a wide variety of data producers. This data ranges from quantitative sensor observations produced by robot systems to complex unstructured human-generated texts on social media. With data being so abundant, the ability to make sense of these streams of data through reasoning is of great importance. Reasoning over streams is particularly relevant for autonomous robotic systems that operate in physical environments. They commonly observe this environment through incremental observations, gradually refining information about their surroundings. This makes robust management of streaming data and their refinement an important problem.

Many contemporary approaches to stream reasoning focus on the issue of querying data streams in order to generate higher-level information by relying on well-known database approaches. Other approaches apply logic-based reasoning techniques, which rarely consider the provenance of their symbolic interpretations. In this work, we integrate techniques for logic-based stream reasoning with the adaptive generation of the state streams needed to do the reasoning over. This combination deals with both the challenge of reasoning over uncertain streaming data and the problem of robustly managing streaming data and their refinement.

The main contributions of this work are (1) a logic-based temporal reasoning technique based on path checking under uncertainty that combines temporal reasoning with qualitative spatial reasoning; (2) an adaptive reconfiguration procedure for generating and maintaining a data stream required to perform spatio-temporal stream reasoning over; and (3) integration of these two techniques into a stream reasoning framework. The proposed spatio-temporal stream reasoning technique is able to reason with intertemporal spatial relations by leveraging landmarks. Adaptive state stream generation allows the framework to adapt to situations in which the set of available streaming resources changes. Management of streaming resources is formalised in the DyKnow model, which introduces a configuration life-cycle to adaptively generate state streams. The DyKnow-ROS stream reasoning framework is a concrete realisation of this model that extends the Robot Operating System (ROS). DyKnow-ROS has been deployed on the SoftBank Robotics NAO platform to demonstrate the system's capabilities in a case study on run-time adaptive reconfiguration. The results show that the proposed system — by combining reasoning over and reasoning about streams — can robustly perform stream reasoning, even when the availability of streaming resources changes.

This work was funded in part by the National Graduate School in Computer Science, Sweden (CUGS), the Swedish Aeronautics Research Council (NFFP6), the Swedish Foundation for Strategic Research (SSF) project CUAS, the Swedish Research Council (VR) Linnaeus Center CADICS, the ELLIIT Excellence Center at Linköping-Lund for Information Technology, and the Center for Industrial Information Technology CENIIT.

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

ACKNOWLEDGEMENTS

My supervisor once told me that working towards a PhD is like running a marathon; sometimes things move slowly, and sometimes you work all the time. I sometimes also imagine it is a bit like running your own business, where you have to make your own decisions, and where nobody else is going to bail you out. As a PhD student you are responsible for your own progress. You suffer your own setbacks and you reap your own rewards. It can at times be a rollercoaster of highs and lows. Sometimes good enough is good enough, and you have a choice to make in spending your limited time where it matters the most; other times there seems to be a lot of time, and yet it can feel like things are going nowhere. During those times it can be difficult not to compare yourself to others, or to question your own capabilities, but it is important to remember that every PhD is different, both in terms of achievements as well as expectations. What we take away from this experience is different for all of us. For me, it gave me the opportunity to learn a lot from my own experiences as well as those of others. It allowed me to go places, to learn new things, to meet people, to exchange ideas, to grow socially as a person, and many more 'scary things'.

I came to Sweden in the end of November of 2012, a day before winter buried everything under a blanket of ice and snow, with a single suitcase and a backpack containing a laptop and some recent papers by Fredrik Heintz. It got dark after 15:00, and I remember how Marc, who was a fellow student who started his thesis work in Linköping before me, sent me a helpful welcome message assuring me that this was perfectly normal. The first night I slept on a thin mattress in an empty apartment I had signed up for five years prior. That first Christmas, I was invited over by Lotta (my 'Swedish mum') to spend *Jul* with her and my long-time friend Stefan (whose raving about Sweden originally got me interested), which I appreciated tremendously. And over the years, I had the opportunity to learn more about my new home. All these things will stay with me as my PhD student adventure ends and another begins. But this adventure would not have been possible without so many people, and while it is impossible to mention all of you, you know who you are.

I want to start by thanking Fredrik for being willing to take me on as an *exjobb* student back in 2012 (and John-Jules Meyer for being willing to ask on my behalf), despite being an outsider, and for offering me to stay on afterwards as his first PhD student. In a way I also feel lucky to have been his first PhD student because I had the opportunity to see him develop as a supervisor as well! Fredrik's work on the DyKnow stream reasoning framework focused exactly on the problems I found the most interesting, and he let me pursue my own take on those problems from the very beginning. I am grateful for all the supervision support I received over the years. I also want to thank Patrick Doherty for all of the valuable feedback and suggestions

for improvements over the years. I consider myself lucky to have been part of AIICS during my PhD studies, not just for the feedback and support but also for the entertaining Friday *fikas*. I appreciated the support from all of you; Karin, Anna, Patrick, Fredrik H, Jonas, Cyrille, Tommy, Per, Piotr, Mariusz, Karol, Olov, Mikael, Mattias, Fredrik P, Johan, and David.

A special 'thank you' also goes to Anne Moe, who granted me one initial conversation before (thankfully!) forcing me to practice my Swedish, and who is absolutely indispensable to all PhD students at IDA. I also want to thank my good friend Mattias for our frequent discussions about everything research and otherwise, and for really helping me feel at home in Sweden. Our lunches and fikas with Erik, Jon, David and Riley were a great way to relax or to learn new things. My hope is that we can continue our tradition of having some gaming sessions and BBQs over the weekends.

None of this would have been possible without the amazing family support I received these past years. My husband Riley has been part of my journey for almost five years, and I cannot even begin to express how much his love and support has helped me cope with this stressful endeavour. He left behind everything he knew, and moved to a country across the ocean to be here with me. None of this would have been possible if he had not pushed me to aim high and try new things when I was still a Master's student. Thank you so much! I hope you realise the learned lessons listed here — although I am sure you know them by heart — are primarily meant as a reminder for you~

Lastly, I want to thank my extended family across several countries for their support and their patience — my parents Eric and Natasha, my father-in-law Gary and my late mother-in-law Paige, who sadly passed away far too young and who we miss dearly; my sister Samantha, her husband Vincent, and my energetic cousins Thomas and Kevin, whose many adventures I hope to hear more about in our video calls; and my brother Daryl and his fiancée Maaike. Moving abroad is ultimately a selfish act; you end up missing out on baby showers, birthdays, and funerals. I have asked a lot from you, and I am grateful you still welcome me back whenever the opportunity arises. *Dank jullie wel!*

Daniel de Leng
Linköping, October 2019

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

INTRODUCTION AND BACKGROUND

Chapter

1 Introduction

REAL-WORLD robotic systems must be able to interpret and reason about uncertain sensor observations to effectively operate in the physical world in a safe manner. Such observations occur in the context of and across time and space. Consequently, observations are temporally and spatially connected to each other. The discrete observations succeed each other like snapshots that, when taken together, tell us a story about the world we reside in. *Stream reasoning* is a subfield of Artificial Intelligence (AI) that focuses on incremental reasoning over rapidly-available information, which we characterise as *streams* containing situational information. More specifically, stream reasoning is a subfield of Knowledge Representation (KR), which is itself a subfield of AI. The focus of this dissertation is on robust stream reasoning under uncertainty, with applications to adaptive stream processing and path checking. Whereas most pre-existing stream reasoning approaches have considered the stream as a complete and accurate representation of the state of the world, we will make no such assumption. Furthermore, whereas path checking assumes a stream is given, we will additionally consider how such a stream is obtained. The work presented here thus considers the transformations needed for a noisy signal to be used to draw conclusions, resulting in a broad problem domain that reflects the realities an integrated AI-enabled system must be able to cope with.

1.1 Motivation

The world is becoming ever more interconnected. As cities grow and technology advances, we can observe an increase in the number of sensors deployed to monitor our physical environment. These developments are often characterised as *smart cities* and the *Internet of Things* (IoT). But the observations are not necessarily limited to passive sensors. They include people sharing information using mobile devices, as well as more and more affordable unmanned platforms carrying cameras.

Figure 1.1: Synergy effect between reasoning over streams and reasoning about streams.

The research presented here was originally inspired by a discussion of a research project scenario in which unmanned aerial vehicles (UAVs) were to be used for gathering information in the physical world. There is often a disconnect in the way people request information and the way information systems provide that information. Commonly, a client requesting information by default does not care *how* their request is fulfilled, unless specifically mentioned otherwise. If a client wants to obtain a video feed showing the façade of a building, all that matters is that this video feed is obtained under the constraints provided, if any. Stream reasoning can help by providing information on demand.

Increasingly many of these information systems are safety-critical due to their interaction with physical environments, which are often shared with human beings. Such systems include UAVs, and may in the future also include autonomous vehicles sharing the roads with human drivers. Checking whether these systems operate in accordance with their formal specifications is an important problem within AI. Luckcuck et al. (2019) recently provided a survey on techniques for the formal specification and verification of these types of systems, covering both model checking and runtime verification approaches. For many such systems, including autonomous robotic systems, streaming information is generated from sensor observations. Stream reasoning thus plays an increasingly important role as robots are no longer confined to carefully crafted environments and instead have to deal with the highly-dynamic physical world that is shared with other entities. This dynamic and highly complex operational environment makes it difficult or impossible to prove *a-priori* that a system adheres to its specifications. Furthermore, the black box nature of many AI models is problematic when a formal specification of a system is needed to perform safety checks. Stream reasoning can help by reasoning about streaming information during runtime, which is a type of *runtime verification*.

The contributions presented in this dissertation consequently fall under two distinct but adjoining strands; *stream reasoning under uncertainty* and *adaptive stream processing*. Stream reasoning seeks to obtain verdicts (of some kind) from streams of information. In many practical applications, streams are subject to uncertainty, which must be taken into account. Stream reasoning under uncertainty is thus a type of reasoning *over* streams. Conversely, adaptive stream processing utilises reasoning *about* streams, and can be regarded as meta stream reasoning. In this view, the

streams themselves — and by extension, their properties — are of interest for the purpose of reasoning. Both views are complementary and form the basis for the two strands of this dissertation. As illustrated in Figure 1.1, reasoning *about* streams can facilitate and strengthen reasoning *over* streams, and reasoning *over* streams can influence the reasoning *about* streams: the two strands provide a natural synergy effect wherein the whole is greater than its individual parts.

Stream reasoning under uncertainty. Stream reasoning seeks to draw conclusions from streams of information, for example to check whether an information system is behaving in accordance with safety specifications. A stream reasoning system needs to handle the incremental nature of streaming information, where information cannot be assumed to be available immediately, and where the total amount of information in a complete stream may be arbitrarily large. Furthermore, the streaming information may be uncertain, and therefore cannot be assumed to accurately represent an observed environment. This dissertation focuses on the problem of stream reasoning with multiple hypotheses, each of which has a probability associated with it. This is done by considering runtime verification for streams under uncertainty, where we also consider qualitative spatial information.

Adaptive stream processing. In many cases, distributed information systems have streams of information flow between their nodes. At the same time, the number of sources for streams — such as sensors or Internet of Things (IoT) devices — is increasing. Yet most research assumes that the sources of streams as well as their transformation services within distributed information systems are fixed and known. While it is important to reason about which streaming resources to subscribe to, most of today's systems lack the capability to do so. It can therefore be argued that it is unreasonable to assume that the streaming resources are fixed and known, and that being able to reason about these dynamics is important for autonomous systems in order to effectively operate in the real world. This dissertation focuses in particular on the problem of reliably generating a stream of interest, as indicated by a user or an information system, where the computational resources may change over time. This is done by reasoning about streams, and in particular how streams can be generated.

Figure 1.2 illustrates a contextual representation of stream reasoning by using a waterfall model, inspired by the well-known (revised) JDL fusion model (Steinberg and Bowman, 2008). The goal of an agent implementing this model is to respond to a dynamic environment. To do so, the agent needs to produce *verdicts* about the environment. For example, an agent may want to continuously check whether its formal model of the environment holds. As long as the agent produces verdicts that confirm that the model holds, the agent can keep operating normally. However, as soon as there is a verdict that represents a violation, the agent can use this verdict as a trigger to adjust its behaviour in order to maintain safety. Of course, verdicts are

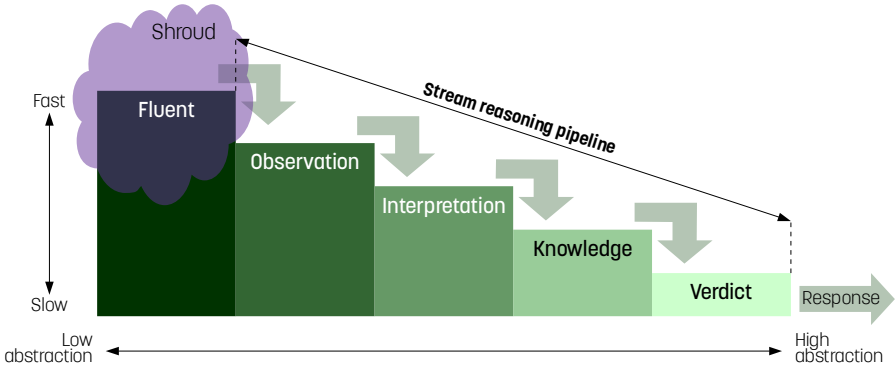

Figure 1.2: The stream reasoning waterfall model showing the incremental transformation of fast streams at a low abstraction level into slow streams at a high abstraction level, which can elicit a response from an agent that implements this model.

highly abstract and the result of multiple steps of reasoning. They are consequently also generated at a relatively slow pace. In the model, verdicts are produced as the result of *knowledge*. Knowledge combines factual information with models that can be based on formal theories or past experience. These models can for example be used to compile past observational information into a compact representation. Knowledge is obtained from *interpretations* of *observations*. An interpretation is a representation of observations, whereas observations are for example streams of raw sensor readings. Observations are often imprecise, but do not have to be. For example, one can have precise observations of social media activity, and the facts that follow from such observations correspond to states. Observations can be obtained from *fluents*, which represent continuous, time-variant (physical) properties. The fluents themselves are *shrouded*, meaning that we cannot read fluents directly as they represent the ground truth of Nature itself. Because they are shrouded, the act of obtaining observations from fluents introduces noise and uncertainty. If the specifications and properties of the sensing device that produces observations are known, however, it is possible to compensate by explicitly representing these properties using probabilistic tools, as is for example common within the area of signal processing. The *stream reasoning pipeline* deals with explicit streams, and therefore starts with observations to eventually produce verdicts. Throughout this dissertation, we will regularly come back to this stream reasoning waterfall model when considering the various subcomponents of such a stream reasoning pipeline.

1.2 Scope and delimitations

The aim of this dissertation is

> **to formally model, develop, and analyse methods and algorithms for incorporating uncertain information in logic-based spatial and tem-**

poral stream reasoning; and to formally model, develop, and analyse methods and algorithms for the adaptive generation of state streams needed to perform this type of reasoning.

This dissertation investigates the following research questions in the pursuit of this aim:

- **[RQ1]**: How can uncertainty be formally modelled for the purpose of logical stream reasoning?
- **[RQ2]**: How can a spatio-temporal logic be constructed by combining spatial and temporal formalisms, and how can statements in such a logic be tested for satisfaction given a stream?
- **[RQ3]**: How can a stream be generated for the purpose of symbol grounding?
- **[RQ4]**: How can the procedure for generating a stream for the purpose of runtime verification be made robust to changes that affect its ability to keep generating such a stream?
- **[RQ5]**: How can the techniques developed towards answering the aforementioned research questions be leveraged in a concrete middleware framework such as the Robot Operating System?

Adaptive stream processing. The waterfall model from Figure 1.2 starts with the problem of adaptively generating streams needed for path checking, i.e. from observations, via interpretations, to knowledge. This is referred to as *adaptive stream processing* (covered in Part III), which is necessary to ground symbols such that they can be given an interpretation. One important delimitation here is that the focus is on how to robustly generate such a stream, rather than the development of sophisticated methods for connecting its contents to symbols. Another delimitation is that we will not consider the generation of *new* knowledge. Rather, we focus on using pre-existing knowledge in the form of logical theories to support the reasoning process.

Stream reasoning under uncertainty. The waterfall model then considers the problem of drawing conclusions from this information. In the work presented here, we specifically focus on drawing such conclusions from uncertain information streams (covered in Part II), i.e. *stream reasoning under uncertainty*. Here we will assume that the uncertainty is explicitly given, rather than attempting to model the uncertainty based on streaming information. The impact of this explicit uncertainty on the reasoning process is the topic of interest. Further, the scope of this dissertation limits itself to the unidirectional support from adaptive stream processing to stream reasoning under uncertainty. The described (bidirectional) synergy effect by allowing the stream reasoning to affect the adaptive stream processing is left to future work.

Integration. The above contributions are integrated into a single architecture (covered in Part IV) for the purpose of checking the behaviour of autonomous robots, called DyKnow. The focus is on the usability of the resulting system towards research into safe autonomous robots. A system integrating DyKnow with the Robot Operating System (ROS) is called DyKnow-ROS. The system restricts itself to producing verdicts, but does not provide any functionality to act on those verdicts, as that ability is left outside of the scope of this dissertation.

1.3 Methodology

The methodology followed for this dissertation is designed to allow for the discovery and investigations of new problems that arise as the result of ongoing research. It can be categorised into three categories; *theory*, *engineering*, and *deployment*.

Theory. First, theoretical contributions were developed and proposed, providing a solid foundation that doubles as a clear design specification. These theoretical contributions are based on and extend previous work in the various fields. The strand for stream reasoning under uncertainty is closely related to research in the field of knowledge representation and reasoning, for example.

Engineering. The different theoretical results were verified empirically as software artefacts. While the contributions themselves are general and could be implemented in a variety of ways, the goal of this work was to provide a stream reasoning framework implementation that integrates these results in a useful manner. This presented a number of engineering problems that were resolved as part of the integration work. The engineering work focused in part on the applicability of the resulting software artefacts. Special care was taken to make sure that the software was easy to use by other developers, decreasing the cost of adoption. The engineering efforts often highlighted potential theoretical problems which had to be resolved.

Deployment. Where suitable, the resulting software artefacts were deployed on the SoftBank Robotics NAO robot platform. Since the work on state stream generation relies on underlying implemented functionality, software under development for the RoboCup Standard Platform League (SPL) was used and adapted to work with the stream reasoning framework. This presented an interesting test-bed for testing the ease of integration, and highlighted various engineering problems that required solving. The result of deployment often also yields or highlights interesting theoretical questions and problems.

The theoretical foundation thus provide a basis upon which the proposed system is built. While some of the presented results are purely theoretical in nature, the focus lies on robotics-related application domains. By providing a formal model of the system, the results can therefore be reproduced in other system realisations

than the one presented in this dissertation, using different platforms than those used here. This demonstrates that the results are general.

1.4 Contributions

The contributions presented in this dissertation benefit from a long line of prior stream reasoning works, albeit under different names, spurred from requirements in the WITAS UAV project (1997–2005) towards the development of technology for autonomous unmanned aerial vehicles, as well as subsequent developments post-WITAS. An overview of the WITAS project's second phase was given by Doherty et al. (2000), and indicated a need for reasoning about streams as follows: "In order to understand the observed ground scenarios, to predict their extension into the near future, and for planning the actions of the UAV itself, the system needs a declarative representation of actions and events." Heintz and Doherty (2001) describe the integration of chronicle recognition into the WITAS system for the purpose of recognising event sequences such as overtakes by vehicles, using the CRS chronicle recognition system by France Telecom (Dousson and Le Maigat, 2007). A Dynamic Object Repository (DOR) was responsible for storing fluent information pertaining to objects that was needed by the UAV to perform chronicle recognition. The chronicle recognition engine itself was a passive component that could be controlled by the UAV control and active vision systems. Given today's description of the field, the WITAS project was one of the first systems to successfully employ what is today known as stream reasoning — a term that would be coined years later by Della Valle et al. (2009) — in a real-world setting.

DyKnow[1] was first introduced by Heintz and Doherty (2004c) and integrated in the *Distributed Autonomous Robotics Architecture* (DARA), by Heintz and Doherty (2004a). To perform chronicle recognition, DyKnow needed to cognise objects of interest, hypothesise their class, and reason continuously about their dynamics. In order for DyKnow to do so, Heintz and Doherty (2004c) recognised that "Consequently, autonomous agents must be able to declaratively specify and re-configure the character of the data received." The *character* of the data was described in terms of *rate* and *form*, which included the way changes were modelled and approximations were handled for time-points without observations. This led to the introduction of the *fluent stream* concept, inspired by Erik Sandewall's work on fluents. A fluent stream could be generated by a computational unit from a particular location and according to a provided policy which described the character of the data.

DyKnow introduced *object linkage structures* (also described as *dynamic object structures*) to realise the ability to hypothesise object classes and, in part, to reason about their dynamics (Heintz and Doherty, 2004c,e,b,d; Heintz et al., 2009, 2013). Object linkage structures made it possible to make and retract class hypothe-

[1] Pronounced 'dino', as in '*dino*saur'. Initially DyKnow was an acronym for *Dynamic Knowledge Processing*. This was later extended to *Dynamic Knowledge Processing and Object Management*. The term has since evolved into a pseudo-acronym.

ses based on observed object dynamics, and to adjust the expected behaviour of these objects based on the currently hypothesised class. This provided bi-directional (bottom-up and top-down) reasoning utilising potentially many levels of abstraction as hypotheses built upon each other. The adopting and retracting of hypotheses is a form of reasoning under uncertainty that is complementary to the contributions presented in Part II. As is the case in this work, high-level reasoning requires a suitable stream to perform the reasoning over. The handling of the character of streams thus required means to perform what was referred to as *knowledge processing* (Heintz and Doherty, 2004b,d). This was used in applications where low-level information was continuously transformed to perform high-level reasoning, for example for chronicle recognition tasks for traffic monitoring (Heintz et al., 2007b,a, 2008b), diagnosis (Heintz et al., 2008a; Krysander et al., 2008, 2010), and execution monitoring (Kvarnström et al., 2008; Doherty et al., 2009, 2013). The *knowledge processing language* (KPL) was introduced by Heintz et al. (2009, 2010) and formalised knowledge processing. The concepts introduced in the formalisation of KPL form the basis of much of the work presented in Part III.

A multi-agent version of DyKnow was considered by Heintz and Doherty (2008, 2010). To achieve this, a Federated DyKnow was introduced, using proxies and speech acts to facilitate the sharing of information between instances. This work also introduced the concept of *semantic labels* for interoperability between agents, stating: "These semantic labels can then be translated by each agent to local DyKnow labels using whatever procedure necessary." (Heintz and Doherty, 2008) These semantic labels represent the precursor to later work towards semantic information integration (Heintz and Dragisic, 2012; Heintz and de Leng, 2013; de Leng and Heintz, 2014), which this dissertation is a continuation of. The DyKnow system has been described in terms of the JDL Fusion Model in Heintz and Doherty (2005b,c,a, 2006), and plays an important role in the *HDRC3 Distributed Hybrid Deliberative/Reactive Architecture* by Doherty et al. (2014).

The work presented in this dissertation is a continuation of these earlier research efforts. In particular, this work continues from the aforementioned efforts towards semantic information integration, and applies them to a new proof-of-concept DyKnow stream reasoning architecture that is separate from HDRC3 at the time of writing. The main contributions presented in this dissertation are as follows:

1. A formal model of a distributed stream reasoning framework was developed, along with the formalisation of its dynamics in terms of changes to the computational environment. Reconfiguration of the computational environment allows for the generation of streams based on requests, for example to support the evaluation of a logic formula. An adaptive reconfiguration algorithm is presented. To support adaptive reconfiguration planning, the cost of using the framework's components is assumed to be estimated during run-time. **[RQ3, RQ4]**

2. The problem of stream reasoning with uncertain state information is considered and applied in conjunction with qualitative spatial reasoning. Specifi-

cally, we consider the problem of path checking over infinite-length streams where each uncertain state is represented by a discrete probability distribution over fully-known states. By keeping track of all possible hypothetical complete streams we are able to incrementally keep track of the satisfaction probability of a temporal logic formula. **[RQ1, RQ2]**

3. The DyKnow-ROS dynamically reconfigurable stream reasoning framework was implemented as an extension to the Robot Operating System (ROS). The required reconfigurability strengthens ROS, which by default does not support this ability. ROS visualisation tools were enhanced with the ability to visualise the dynamically-changing environment. **[RQ5]**

1.5 Publications

These contributions are the result of a number of publications. The complete listing of publications covered in this dissertation is as follows:

- ***D. de Leng** and F. Heintz. Stream reasoning with probabilistic state information using progression-based path checking. Journal article under review.*
- **D. de Leng** and F. Heintz. Approximate Stream Reasoning with Metric Temporal Logic under Uncertainty. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, 2019.
- **D. de Leng** and F. Heintz. Partial-State Progression for Metric Temporal Logic. In *Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning*, 2018.
- **D. de Leng** and F. Heintz. Towards Adaptive Semantic Subscriptions for Stream Reasoning in the Robot Operating System. In *Proceedings of the 30th IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2017.
- **D. de Leng** and F. Heintz. DyKnow: A Dynamically Reconfigurable Stream Reasoning Framework as an Extension to the Robot Operating System. In *Proceedings of the 5th IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, 2016.
- **D. de Leng** and F. Heintz. Qualitative Spatio-Temporal Stream Reasoning With Unobservable Intertemporal Spatial Relations Using Landmarks. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, 2016.
- **D. de Leng** and F. Heintz. Ontology-Based Introspection in Support of Stream Reasoning. In *Proceedings of the 13th Scandinavian Conference on Artificial Intelligence*, 2015.
- **D. de Leng** and F. Heintz. Ontology-Based Introspection in Support of Stream Reasoning. In *Proceedings of the 1st Joint Ontology Workshops held at the 24th International Joint Conference on Artificial Intelligence*, 2015.

Part I	Part II	Part III	Part IV	Part V
Introduction	Stream reasoning	Synthesis	DyKnow-ROS	Conclusions
Preliminaries	Uncertainty	Composition	Case studies	Appendices
	Space	Perturbations	Related work	

Table 1.1: An outline of this dissertation.

- F. Heintz and **D. de Leng**. Spatio-Temporal Stream Reasoning with Incomplete Spatial Information. In *Proceedings of the 21st European Conference on Artificial Intelligence*, 2014.
- **D. de Leng** and F. Heintz. Towards On-Demand Semantic Event Processing for Stream Reasoning. In *Proceedings of the 17th International Conference on Information Fusion*, 2014.
- F. Heintz and **D. de Leng**. Semantic Information Integration with Transformations for Stream Reasoning. In *Proceedings of the 16th International Conference on Information Fusion*, 2013.

Additionally, the following publications were also produced but will be excluded from this dissertation because they are unrelated to the research questions or were not peer-reviewed:

- **D. de Leng**, M. Tiger, M. Almquist, V. Almquist, and N. Carlsson. Second Screen Journey to the Cup: Twitter Dynamics during the Stanley Cup Playoffs. In *Proceedings of the 2nd Network Traffic Measurement and Analysis Conference*, 2018.
- **D. de Leng**. Querying Flying Robots and Other Things: Ontology-supported stream reasoning. In XRDS: Crossroads (popular science magazine), 2015.

Lastly, the material in this dissertation is a continuation of the following Licentiate thesis:

- **D. de Leng**. Spatio-temporal stream reasoning with adaptive state stream generation. Licentiate thesis No. 1783, Linköping University, 2017.

1.6 Dissertation outline

This dissertation is subdivided into five separate parts, as shown in Table 1.1, with each chapter covering a subset of the waterfall model shown in Figure 1.2. Part I covers an introduction and background for this dissertation. Part II covers spatio-temporal stream reasoning under uncertainty. This is followed by Part III covering adaptive stream processing. Part IV covers applied stream reasoning and presents

the DyKnow-ROS stream reasoning framework alongside case studies and related approaches. Finally, Part V concludes the dissertation.

Chapter 2, titled 'Preliminaries', further elaborates on the concept of a stream by considering the two different views used in this work and relates streams to the concepts of stream processing and stream reasoning. The purpose of this chapter is to clarify these concepts for the context of this dissertation, because there have been various interpretations for these concepts in the literature due to the stream reasoning research area still being fairly young.

Chapter 3, titled 'Reasoning about time', focuses on traditional stream reasoning tasks where streaming information is used in conjunction with reasoning capabilities to yield verdicts. This chapter introduces a well-known incremental path checking procedure and suggests improvements.

Chapter 4, titled 'Reasoning under uncertainty', enhances the path checking procedure from the preceding chapter to also consider uncertainty. Here, uncertainty is represented by assigning probabilities to different hypotheses, all of which are commonly kept track of for the purpose of yielding verdicts.

Chapter 5, titled 'Reasoning about space', presents an extension from temporal reasoning to qualitative spatio-temporal reasoning. Concretely, the Region Connection Calculus (RCC-8) is utilised to support qualitative spatio-temporal stream reasoning.

Chapter 6, titled 'State stream synthesis', discusses what is needed in order to synthesise state streams and how to ground logical symbols in those state streams.

Chapter 7, titled 'Reasoning about composition', takes the view of streams as objects which are the product of potentially many stream processing steps. It illustrates how a configuration manager can adapt the configuration of stream processing components to produce a stream in accordance with a semantic specification.

Chapter 8, titled 'Reasoning about perturbations', follows up on the preceding chapter by also considering adaptive behaviour in the face of changes to the availability of stream transformations. It does so both for cases where a processing pipeline 'breaks', as well as for cases where switching to a different pipeline is beneficial to the overall system.

Chapter 9, titled 'DyKnow-ROS', takes the formal contributions from the preceding chapters and combines them into a stream reasoning framework called DyKnow-ROS. The chapter does so by connecting computations to services provided by the framework.

Chapter 10, titled 'Case-studies', utilises the stream reasoning framework from the preceding chapter in a case study. The intention is to show the applicability of the proposed approaches on a real robot as a proof of concept.

Chapter 11, titled 'Related work', relates the contributions of this dissertation to a number of other stream reasoning systems.

Chapter 12, titled 'Conclusions and future work', discusses some of the limitations of the presented contributions, lists some of the remaining open problems, and concludes this dissertation by reiterating the contributions made and discussing potential future work.

Chapter

2 Preliminaries

STREAMS form the foundation for the work presented in this dissertation. This chapter considers the nature of streams; what they are and where they originate from, and how one can model and interpret them in an information system. We focus on different views of streams and discuss the relationship between streams and stream reasoning relative to the stream reasoning model.

2.1 Introduction

Classical database approaches tend to only operate on what is stored and always on everything that is stored. In contrast, stream reasoning puts constraints on how much can be stored and always assumes to only have a fragment of the entire stream to operate on. In this chapter, we therefore seek to describe the nature of streams, i.e. what a stream is, how it can be represented, and how it related to stream reasoning. It is important to be aware of the different views that exist for stream reasoning. In particular, streams are represented in different ways in the literature, using different assumptions and constraints. This occurs at both the data level, i.e. what is contained within a stream, and the temporal level, i.e. how time plays a role in the description of a stream. Furthermore, streams can themselves be represented as objects with their own properties, which can be useful in applications that focus on the generation and transformation of streams.

2.2 Views of streams

We consider two views of streams; streams as *data sequences*, and streams as *objects*.

Streams as data sequences

Streams are commonly regarded as data sequences, using what we refer to as an *internal view*. In the internal view, we consider the properties of the samples that make up a stream. These samples could for example be noisy discrete observations of continuous fluents, or even data generated from social media postings or system logs. The samples can be used to represent instantaneous events, time periods, or simply a logical ordering between samples. Streams as data sequences have a lot in common with *Big Data*, which is a term that generally focuses on large volumes of data and the challenges pertaining to the processing of such data. Laney (2001) originally described the terms volume, velocity and variety as important properties for describing data, and these properties were subsequently extended to define the Big Data concept. The following stream properties originate from the 'four Vs of big data' applied to a stream reasoning context:

Volume. One can no longer assume that the data can be collected in its entirety prior to processing it. The volume of data may simply be too large for any practical storage to take place. Streaming data is therefore generally assumed to be accessed once and then lost, unless explicitly and only partially stored.

Velocity. The incremental nature of streams invokes the property of velocity, i.e. how quickly data becomes available. Depending on the source of a data stream one can or cannot make assumptions about its velocity. For example, user-generated content could be highly irregular and bound to human behavioural patterns, whereas sensor data in a real-time system could be assumed to have a fixed frequency. A general stream reasoning system must be able to cope with differences in velocity, and high velocity in particular.

Variety. Streaming data can originate from many heterogeneous sources in various data formats and as various data types. Examples of different data types are text, images, and speech. Being able to interpret the data from streams in various formats and types is important in order to effectively work with this data.

Veracity. The trustworthiness and accuracy of data is another important factor to consider when dealing with streaming data. The trustworthiness of data is in part based on who produced the data and who provided it; some sources may be of poor quality or (purposely or not) misrepresent information. This may also be a consequence of low accuracy of data.

Different stream reasoning systems focus on different aspects. For social media tools, variety and veracity may be far less important than dealing with volume and velocity, as the focus is user-generated unstructured data. In robot systems, veracity

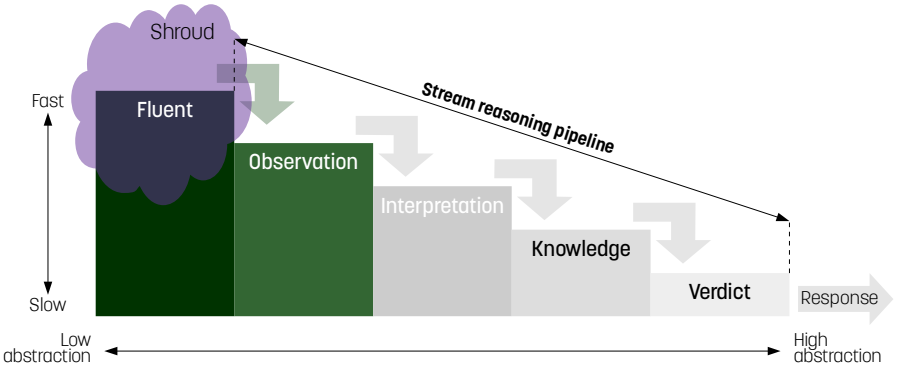

Figure 2.1: The stream reasoning waterfall model with the transformation of shrouded fluents into observations highlighted.

and velocity are especially important in order to deal with a rapidly-changing environment. Figure 2.1 shows the observation of fluents highlighted, which is where a lot of uncertainty enters the reasoning pipeline.

Streams as objects

An alternative *external view* of streams is also possible. In the external view, we consider streams as objects with their own properties and labels. This is particularly useful in cases where we want to consider streams as a product of computations. Streams can be transformed, combined, or subscribed to. In this dissertation, we consider the following properties of streams as objects:

Syntactic label. When considering streams as objects, they can be *named* or *anonymous*. A named stream is a stream that has one or more labels associated with it. These labels can then be used to refer to a particular stream in a system, such that they can be subscribed to by a program, allowing the samples in the stream to be used for processing.

Type. Streams in practice often have a type. This type provides a constraint on the data type of the samples. By knowing the type of a stream, a program is able to interpret the samples using the correct data type. This particular property is closely related with the 'variety' property from earlier.

Semantic annotation. A semantic annotation for a stream is an additional specification that can be associated with a stream in order to describe the semantic meaning of the samples contained in the stream. Commonly a semantic annotation of a stream is inherited from the process that led to the generation of the stream.

Provenance. Provenance information for streams conveys the origin of a stream; how, where, and by whom it was created. This type of information provides a context which can be important in order to correctly interpret the information in a stream. For example, it is possible for a stream to be generated from transformations applied to an external source, in which case it can be useful to know more about said source when considering the veracity of the streaming data.

Policy. A policy for a stream is also inherited from the process that led to the generation of the stream, and describes the conditions under which a transformation is applied. This includes properties like the frequency of a stream (which can be regular or irregular), and how missing or late samples are handled using for example different methods of interpolation.

The above properties treat a stream as an object that can be reasoned with. While treating streams as objects is in itself not a new idea, stream reasoning commonly considered only the internal view for streams (see e.g. de Leng (2017); Dell'Aglio et al. (2019)). Several of the listed properties inherit from an underlying stream processing process, which we cover in more detail in Part III.

2.3 Anatomy of a stream

The internal and external views of streams both hold simultaneously, and while they give a general idea of what a stream looks like, we have not yet considered the anatomy of a stream that combines these two views. We use the term *timed data stream* to represent a named discrete instantiation of the concept of a stream wherein each *sample* is a set of time-stamped strictly-typed key-value pairs. The time-stamps can for example be used to describe the *available time*, meaning the time at which the data sample was received. An alternative time-stamp is the *valid time*, which represents the time-point for which the key-value pairs hold. A formal definition for timed data streams is given in Chapter 7. For now, however, we limit ourselves to an informal overview.

Figure 2.2 illustrates the anatomy of a stream. It shows a graphical representation of a stream along two dimensions. The horizontal dimension represents time, with time-point 0 representing the present. A stream can theoretically be infinitely long; we may simply not know when the stream ends, so the relative time-points run up to infinity. Along the temporal axis, we can see samples represented by vertical black lines. The distance between these samples may vary, which allows us to represent a time-line using reals. The samples are intersected by red horizontal lines. The horizontal axis represents the stream's bandwidth, and each horizontal red line represents a field within a stream. Such a field can in practice be named. The simplest form of stream however only contains one field. The intersections are then values for observations over time. As the stream progresses, the latest value in

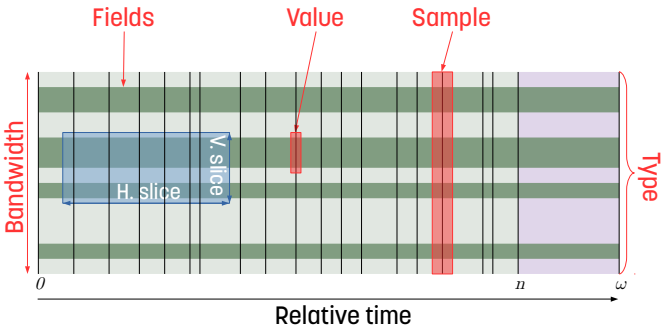

Figure 2.2: Anatomy of an irregular-timed data stream showing key concepts in red and primitive operations in blue.

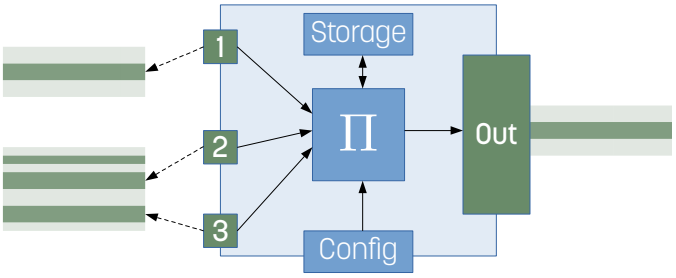

Figure 2.3: Anatomy of a transformation, showing its structure and its relationship to streams.

a field may change over time. Finally, the combination of fields along the bandwidth axis represents the type of the stream.

Because a stream is a composite entity, it is possible to consider a subset of a stream in the two different axes. We call these subsets *slices*, and distinguish between horizontal slices and vertical slices. A horizontal slice corresponds to a temporal subset of a stream, which is commonly referred to as a *window*. Similarly, a vertical slice corresponds to a volumetric subset of a stream, which we call a *substream*[2].

2.4 Anatomy of a transformation

Transformations are functions that, given some data streams, produce a new data stream. They therefore need to consider both the internal and external views on data streams.

[2]Not to be confused with the LARS definition of a substream as per Beck et al. (2014, 2015), which corresponds to a horizontal slice here.

Figure 2.3 shows a graphical representation of a transformation and how it connects to streams. The light-blue box marks the components that make up an active transformation, also known as a *computation unit*. A *source* is a specific type of computation unit which does not take any streams as input. To the left, we can see two streams. There are dashed arrows originating from the transformation and pointing to fields in the two streams, although not all of them. These dashed lines represent *subscriptions* for input arguments one through three of a transformation denoted by $\amalg$. This represents that whenever a new sample is observed, the most recent samples for all of the subscriptions are sent to the transformation. They are joined by a *configuration* which can be set externally, and a small storage which the transformation can read from and write to. The configuration can be changed dynamically, and controls properties such as which streams the unit is subscribed to. The result, if any, is then sent out to the *stream generator* marked by 'out', which, over time, generates a resulting stream. By default, a transformation is set to respond to every change as characterised by the observation of samples from one of its subscribed-to streams. By considering a clock stream, which sends out a time value at a regular interval, the transformation can adopt a policy in which it only published new samples whenever the time is updated.

2.5 Stream reasoning

In recent years definitions of stream reasoning have started to slowly converge. In this dissertation, we informally define stream reasoning as follows.

Definition 2.1 (Stream reasoning)**.** *Stream reasoning is the incremental reasoning over and about rapidly-changing information.*

The intuition behind stream reasoning is that there is some potentially-infinite length sequence over which reasoning is performed with finite computational resources, commonly including storage as a bottleneck. There is also a time dimension; because the information changes rapidly, the stream reasoning process needs to either keep up with the stream or handle any dropped samples through alternative means. The incremental nature of reasoning is also important, since it forces any reasoning process to deal with parts of the stream rather than to consider the stream as a whole, as is common in traditional reasoning approaches. As a logical extension of an informal theory of streams, we consider some of the ontology (in the metaphysical sense of the word) for stream reasoning here.

Stream reasoning has been studied for some time now, and even the definition used here slightly deviates from the one used in publications this dissertation is based on. Other researchers have characterised stream reasoning through different lenses; a characteristic attributable to the multidisciplinary nature of stream reasoning. Cugola and Margara (2012a) collectively refer to stream reasoning systems as *Information Flow Processing* (IFP) systems, and provide a thorough survey of the various approaches. The following is a brief contrast between two classes of stream

reasoning systems they identified; the *Data Stream Management* (DSM) systems[3], and the *Complex Event Processing* (CEP) systems. The boundaries between DSM and CEP systems can be blurry at times, but generally speaking, DSM systems originate from the area of databases and Database Management Systems (DBMS) and take continuous queries that produce results for the duration that they are active by constructing relational tables based on time windows. This is in contract with CEP systems, where CEP is sometimes defined as methods, techniques and tools for the continuous and timely processing of events as they occur (Eckert and Bry, 2009)[4]. Whereas DSM systems make use of windows, CEP systems tend to make use of temporal orderings. The detection of a queried temporal ordering of events can itself be seen as a complex event. Early CEP techniques include *chronicle recognition* systems, which were introduced by Ghallab (1996). Chronicles are represented by (complex) events and metric temporal constraints on those events. Chronicles can be detected in a stream by checking for the occurrence of their composite events relative to the metric temporal constraints. A more in-depth discussion of these types of systems is presented as part of related work in Chapter 11.

It is important to point out that some stream reasoning efforts have chosen to characterise themselves as *stream processing* efforts. While no agreed-upon formal distinction has been developed thus far — and given that some may argue that such a distinction does not even exist to begin with — this dissertation *does* also refer to stream processing and stream reasoning. Here we consider stream reasoning to deal with the obtaining of verdicts through reasoning processes, by combining streaming information with background knowledge bases. Stream reasoning produces either single verdicts or slow-moving streams of verdicts, and these verdicts can be regarded as informative conclusions about an environment that aid in the decision-making process of an agent. On the other hand, stream processing focuses on the transformation of streaming data from one format into another by combining data resources. If the transformations are simple, this can be done at a fast rate. Stream processing is therefore process-centric; it focuses on how data is transformed, without necessarily needing to consider the meaning of this data. Simple window-based aggregation jobs are an example where the meaning of the values has no bearing on the information system performing the stream processing. But the distinction between the two is not a perfect one; there is no clear separation since the processing needed to obtain a verdict in stream reasoning can be explained as a stream processing task, and if the goal of a stream processing task is to obtain verdicts it is often regarded as stream reasoning. This phenomenon can be observed with RSP research, which often characterises itself as stream reasoning research due to the combination of RDF streams with a background knowledge base in the form of an ontology.

[3]The term 'Data Stream Management Systems' is commonly written as DSMS, but when contrasted with 'CEP systems' it is also written as 'DSM systems'.

[4]Loosely translated from a German-language definition of CEP by Eckert and Bry (2009): *"Complex Event Processing (CEP) ist ein Sammelbegriff für Methoden, Techniken und Werkzeuge, um Ereignisse zu verarbeiten während sie passieren, also kontinuierlich und zeitnah."*

2.6 Summary

In this chapter, we considered an informal theory of streams. Streams can be viewed in different ways. The interval view of streams can be aligned with the 4 V's of Big Data; volume, velocity, variety, and veracity. In this view, streams are regarded as sequences of data. An alternative view is the external view of streams, which considers streams as objects. In this view, streams are their own objects with associated properties, including a label for named streams, a type related to variety, a possible semantic annotation describing the meaning of a stream, a provenance covering the stream's origin, and a generation policy determining properties such as frequency for regular streams. Both views can play a role when considering stream processing and stream reasoning. However, these two concepts as of yet have no agreed-upon distinction. We covered various types of stream reasoning and stream processing in the literature to give an overview of the breadth and commonalities between the approaches, and explained how this dissertation distinguishes between stream reasoning and stream processing.

Part II

STREAM REASONING UNDER UNCERTAINTY

Chapter

3 Reasoning about time

TIME represents the core of stream reasoning. Being able to make temporal statements and to determine the truth value of such a statement is of great importance to many applications, including areas such as intelligent robotics. We refer to the evaluation of temporal statements through a temporal logical language by using the term 'logic-based stream reasoning'. In particular, we focus on the problem of model checking a stream, also known as *path checking*.

3.1 Introduction

Temporal logics can be a powerful tool for the formal verification of programs and systems. Given an information system, temporal logics such as *Linear Temporal Logic* (LTL) (Pnueli, 1977) or *Metric Temporal Logic* (MTL) (Koymans, 1990) allow us to make statements describing the correct behaviour of these systems over time. As shown in Figure 3.1, by checking whether these statements are upheld — given a stream of states containing truth values for the propositions used in making such a statement — we can then issue a verdict or even a stream of verdicts. The generation of verdicts can be an expensive process, and the rate at which verdicts are produced is therefore generally much lower than for example the rate at which observations can be produced. Once obtained, a verdict can be used to trigger an information system to respond accordingly.

Traditional approaches for checking the correctness — and, consequently, safety — of a system include automata-based *model checking* (Wolper et al., 1983; Vardi and Wolper, 1994). For LTL, these automata can for example be Büchi (1990) or Muller (1963) ω-automata when we consider non-deterministic or infinite-length runs. However, we are instead interested in those cases where a model of the system is unavailable, or where constructing one is infeasible. *Runtime verification* is the verification of a system relative to a formal specification during run-time, and is

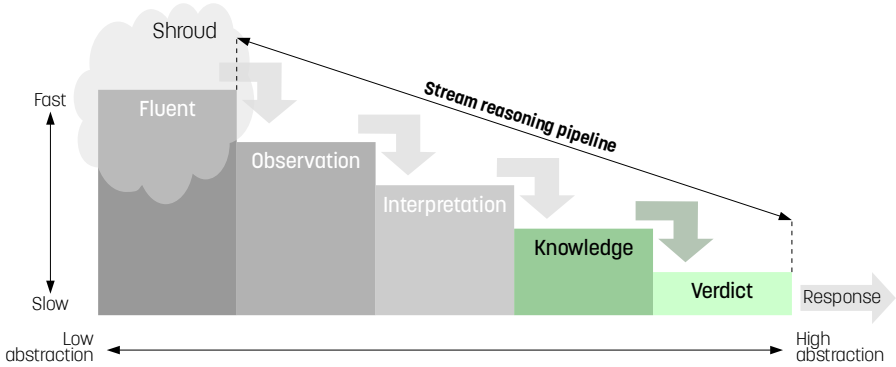

Figure 3.1: The stream reasoning waterfall model with the transformation of knowledge into verdicts, also known as logic-based stream reasoning, highlighted.

suitable in application domains where model checking a system *a priori* is infeasible, such as the domain of autonomous robotic systems (Adolf et al., 2017; Desai et al., 2017). This can often be stated as a *path checking problem* (Markey and Schnoebelen, 2003), which is computationally simpler than satisfiability or model checking. Concretely, a path checking problem is a decision problem in which we receive a path and a temporal formula, and have to determine whether the provided path satisfies the provided formula. Path checking ω-words has previously been applied to MTL in different forms, for example covering dense-time continuous semantics over transition sequences (Baldor and Niu, 2012), trace-length independent monitoring over timed words under pointwise semantics by rewriting into LTL (Ho et al., 2014), path checking of data words under pointwise semantics (Feng et al., 2015, 2017), and almost event-rate independent path checking of timed words under pointwise semantics (Basin et al., 2017).

In this chapter we focus on reasoning about time, by evaluating the truth value of logical statements. First, we introduce some temporal logics which allow us to make statements about propositions over time. Then, we consider how these logics are traditionally used to verify the correctness of a system *a priori*, under the assumption that a model of that system is available. This is different from the case where such a model is missing, or where we wish to do so during runtime, which is discussed next. In particular, this dissertation opts to focus on a well-known incremental syntactic rewriting procedure, and extends it with a formula simplification technique. Finally, a small empirical evaluation is provided for this syntactic rewriting procedure.

3.2 Temporal models and logics

Logic-based approaches to stream reasoning are often related to temporal (modal) logics. LTL is commonly used when dealing with linear time, and its models can be represented using a linear time-line composed of discrete points. In contrast,

Computation Tree Logic (CTL) (Clarke and Emerson, 1981) is sometimes referred to as a branching-time logic, where the possible time-lines fan out like a tree structure. We are primarily concerned with linear time, and therefore focus on LTL and its extensions towards dealing with real time. In particular, we consider MTL and its sublanguage *Metric Interval Temporal Logic* (MITL) (Alur et al., 1996). These logics make it possible to more precisely specify temporal constraints. The following is an introduction to the syntax and semantics of these logics, and the temporal models over which those semantics are defined.

Linear Temporal Logic

The syntax for LTL determines what statements constitute well-formed formulas (wffs), as shown below.

Definition 3.1 (LTL syntax)**.** *The syntax for propositional LTL is as follows for atomic propositions p and well-formed formulas (wffs) ϕ and ψ:*

$$\top \mid \bot \mid p \mid \neg\phi \mid \phi \vee \psi \mid \phi \; U \, \psi \mid X\phi \tag{3.1}$$

The operators U and X represent the temporal operators for 'until' and 'next'. A formula $\phi \mathsf{U} \psi$ holds iff ϕ is true until ψ is true. Similarly, a formula $\mathsf{X}\phi$ holds iff ϕ is true at the next time-point, meaning the tail of the ω-word under consideration.

LTL also commonly makes use of syntactic sugar, which allows for abbreviations of wffs. In this dissertation, we make use of $\phi \wedge \psi \equiv_{def} \neg(\neg\phi \vee \neg\psi)$ for conjunctions, $\phi \rightarrow \psi \equiv_{def} \neg\phi \vee \psi$ for implications, $\phi \leftrightarrow \psi \equiv_{def} (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$ for bi-implications, $\phi \; \mathsf{R} \; \psi \equiv_{def} \neg(\neg\phi \; \mathsf{U} \; \neg\psi)$ for the 'release' temporal operator, $\mathsf{F}\phi \equiv_{def} \top \; \mathsf{U} \; \phi$ for the 'eventually' ('finally') temporal operator, $\mathsf{G}\phi \equiv_{def} \neg\mathsf{F}\neg\phi$ for the 'always' ('globally') temporal operator, $\top \equiv_{def} p \vee \neg p$ for truth, and $\bot \equiv_{def} \neg\top$ for contradiction. Their semantics are thus defined in terms of the semantics for wffs.

Example 3.1 (LTL statement)**.** *Consider the following statement: "If I am in my room, it is always the case that if the light condition of my surroundings is poor, then the surroundings will eventually be well-lit." This statement can be approximated as*

$$\mathsf{inRoom} \rightarrow G(\mathsf{poorLight} \rightarrow F(\mathsf{goodLight})), \tag{3.2}$$

where propositions inRoom, poorLight *and* goodLight *stand for "I am in my room", "the light condition of my surroundings is poor", and "the surroundings are well-lit" respectively.*

The semantics for LTL are defined in terms of ω-words. To this end, let Σ denote an alphabet based on a set of propositions denoted by $\mathcal{P}$, i.e. $\Sigma = 2^{\mathcal{P}}$. An ω-word is then denoted by $\sigma = (\sigma_0\sigma_1 \dots)$, i.e. $\sigma \in \Sigma^\omega$. We can take a *suffix* of any ω-word by writing $\sigma_{\geq i}$ for $i \in \mathbb{N}$, yielding $\sigma_{\geq i} = (\sigma_i\sigma_{i+1} \dots)$. We call a suffix for $i > 0$ a *strict suffix*, which in turn is an ω-word. We can now formally define the semantics of LTL.

Definition 3.2 (LTL semantics)**.** *Let ϕ, ψ stand for well-formed LTL formulas, and σ_i for an ω-word. The semantics of LTL are defined recursively for any choice of time-point $i \in \mathbb{N}$:*

$$\sigma, i \models p \text{ iff } p \in \sigma_i \text{ for } p \in \mathcal{P} \tag{3.3}$$
$$\sigma, i \models \neg\phi \text{ iff not } \sigma, i \models \phi \tag{3.4}$$
$$\sigma, i \models \phi \vee \psi \text{ iff } \sigma, i \models \phi \text{ or } \sigma, i \models \psi \tag{3.5}$$
$$\sigma, i \models \phi \; U \, \psi \text{ iff there is an } j \geq i \text{ such that } \sigma, j \models \psi \text{ and } \sigma, k \models \phi \text{ for all } i \leq k < j \tag{3.6}$$
$$\sigma, i \models X\phi \text{ iff } \sigma, i+1 \models \phi \tag{3.7}$$

Lastly, we use the short-hand $\sigma \models \phi$ to represent $\sigma, 0 \models \phi$.

Metric Temporal Logic

MTL extends LTL with temporal intervals for the modal operators, restricting them to a specific time-period. This allows for concise temporal statements such as "ϕ is true for the next 10 time-points," or "ψ becomes true within the next 10 time-points." Crucially, MTL considers real-time events where every letter of an ω-word is associated with a time-stamp. An MTL-formula is said to be well-formed iff it adheres to the MTL syntax, which is similar to that of LTL.

Definition 3.3 (MTL syntax)**.** *The syntax for (future-restricted) MTL for atomic propositions p, temporal (natural) intervals $I \subseteq \mathbb{R}^+$, and well-formed formulas (wffs) ϕ and ψ is as follows:*

$$\top \mid \bot \mid p \mid \neg\phi \mid \phi \vee \psi \mid \phi \; U_I \; \psi \mid \phi \; S_I \; \psi \mid X_I \phi \tag{3.8}$$

Just as for LTL, we make use of mostly the same syntactic sugar. Only the temporal operators G_I, F_I, and R_I have been modified to incorporate the temporal interval I introduced for U_I and X_I. Additionally, the temporal operator intervals may be omitted for cases where $I = [0, \infty]$.

Example 3.2 (MTL statement)**.** *Recall the example statement: "If I am in my room, it is always the case that if the light condition of my surroundings is poor, then the surroundings will be well-lit within 10 seconds." In MTL, this statement can be written as follows:*

$$\mathsf{inRoom} \rightarrow (G(\mathsf{poorLight} \rightarrow F_{[0,10]}(\mathsf{goodLight}))), \tag{3.9}$$

where propositions inRoom, poorLight *and* goodLight *stand for "I am in my room", "the light condition of my surroundings is poor", and "the surroundings are well-lit" respectively.*

The semantics of MTL takes into account the real-time nature of its logical statements by considering *timed ω-words* rather than the untimed ω-words used for the

LTL semantics. A timed ω-word annotates each letter of the word with a *time-stamp*. The index of each letter is referred to as a *time-point*. Concretely, let σ denote a timed ω-word; then, $\sigma = ((\sigma_0, \tau_0)(\sigma_1, \tau_1)\dots)$, where σ_0 represents a letter in Σ for time-point 0 and is annotated with a time-stamp $\tau_0 \in \mathbb{R}$. The same holds for all other time-points. Time-stamps are assumed to be non-decreasing, so for any pair of time-points $i < j$ it is the case for the associated time-stamps that $\tau_i \leq \tau_j$. As before, a suffix $\sigma_{\geq i}$ is defined to be a timed ω-word $((\sigma_i, \tau_i)(\sigma_{i+1}, \tau_{i+1})\dots)$. We can now formally define the semantics of MTL.

Definition 3.4 (MTL semantics)**.** *Let ϕ, ψ stand for well-formed MTL formulas, σ for a timed ω-word, $\tau \in \mathbb{R}^+$ for a time-stamp, and $i \in \mathbb{N}$ for a time-point. The semantics of MTL are defined recursively:*

$$\sigma, \tau_i \models p \text{ iff } p \in \sigma_i \text{ for } p \in \mathcal{P} \tag{3.10}$$

$$\sigma, \tau_i \models \neg\phi \text{ iff not } \sigma, \tau_i \models \phi \tag{3.11}$$

$$\sigma, \tau_i \models \phi \vee \psi \text{ iff } \sigma, \tau_i \models \phi \text{ or } \sigma, \tau \models \psi \tag{3.12}$$

$$\sigma, \tau_i \models \phi \; U_I \; \psi \text{ iff there is a } \tau' \in I + \tau_i \text{ such that } \sigma, \tau' \models \psi \text{ and } \sigma, \tau_i'' \models \phi \text{ for all } \tau_i < \tau'' < \tau' \tag{3.13}$$

$$\sigma, \tau_i \models \phi \; S_I \; \psi \text{ iff there is a } \tau' \in I - \tau_i \text{ such that } \sigma, \tau' \models \psi \text{ and } \sigma, \tau_i'' \models \phi \text{ for all } \tau' < \tau'' < \tau_i \tag{3.14}$$

Note that the 'next' operator X from LTL by default is undefined in MTL because there is no trivial 'next' time-stamp. However, sometimes the interval-bounded 'next' operator, denoted by X_I, is added as syntactic sugar, where $\mathsf{X}_I\phi$ is defined as $\bot \mathsf{U}_I \phi$.

Metric Interval Temporal Logic

The use of intervals allows for more precise logical statements in MTL. Because of its undecidability for satisfiability and model checking problems (Alur et al., 1996), several restrictions to MTL were proposed, among them MITL. MITL disallows 'punctuality' constraints — in which temporal intervals are points — and temporal intervals to subsets $I \subseteq \mathbb{N}$ to get around the undecidability. We will be using MITL alongside LTL in the remainder of this dissertation.

3.3 Formal verification

Formal verification techniques are used to check whether a system's possible behaviours are in line with the formal specifications for that system, the latter of which can be given using the previously-introduced logics. Automata are commonly used to represent the systems for which the correctness is to be proven. Figure 3.2 illustrates the relationship between specifications and system descriptions. Automata-theoretic model checking makes use of ω-automata to describe a

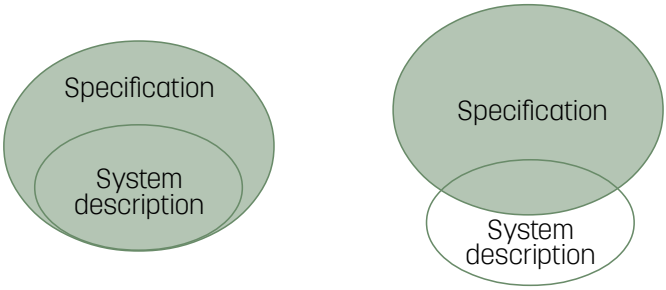

Figure 3.2: Left: All models of the system description are also models of the formal specification, showing correctness. Right: Some models of the system description are not models of the formal specification, indicating that the specification is violated by some system traces.

program in terms of possible state sequences, which can be regarded as streams. These finite automata operating on (infinite-length) ω-*words* are therefore sometimes called 'stream automata'. For infinite-length words we instead use the set Σ^ω of ω-words, from which languages of infinite words can be constructed. Just as regular languages can be described by regular expressions, ω-regular languages can be described by ω-regular expressions.

Example 3.3 (Finite and infinite regular languages)**.** *Suppose we have a finite alphabet $\Sigma = \{a, b\}$. The regular expression $a(a|b)^*$ describes any finite sequence of a's or b's following a single a. These sequences describe finite-length words. We can describe ω-words with ω-regular expressions. As an example, consider the ω-regular expression a^*b^ω, which describes all ω-words which start with a finite sequence of a's followed by an infinite sequence of b's.*

We can use acceptors to recognise finite-length inputs in Σ^*. These finite-state automata use accept (or final) states to determine the acceptability of words: if a word ends in an accept state of a given finite-state automaton, the word is considered to be *accepted* by that finite-state automaton.

Definition 3.5 (Finite-state automaton)**.** *A deterministic finite-state automaton (FSA) $\mathcal{A}$ is denoted by a tuple $(\Sigma, Q, q_0, \delta, F)$, where Σ denotes the alphabet of $\mathcal{A}$, Q denotes the set of states, $q_0 \in Q$ denotes the initial state, $\delta : Q \times \Sigma \to Q$ denotes the transition function, and $F \subseteq Q$ denotes the set of final (accepting) states.*

An FSA $\mathcal{A}$ can then be used to check for the acceptance of finite-length words. These are part of the language $\mathcal{L}(\mathcal{A})$.

Definition 3.6 (FSA acceptance)**.** *An FSA $(\Sigma, Q, q_0, \delta, F)$ accepts a word $(\sigma_0, \ldots, \sigma_{n-1})$ iff there exists a sequence of states $r_0, \ldots, r_n$ such that $r_0 = q_0$ for the initial state, $r_n \in F$ for the final state, and $r_{i+1} = \delta(r_i, \sigma_i)$ for all $0 \leq i < n$.*

ω-automata are extensions of FSA that can detect ω-words. Because ω-words are of infinite length, the acceptance conditions of ω-automata differs from those of FSA. Different types of ω-automata consequently exist with varying semantics in terms of acceptance conditions, but *Büchi ω-automata* are commonly used.

Definition 3.7 (Büchi automata)**.** *A deterministic Büchi automaton $\mathcal{B}$ is a type of ω-automaton over an alphabet Σ denoted by a tuple $(\Sigma, Q, q_0, \delta, F)$, where Q denotes a finite set of states, $q_0 \in Q$ denotes an initial state, $\delta : Q \times \Sigma \to Q$ denotes the transition relations, and $F \subseteq Q$ denotes the set of accepting states.*

An ω-automaton can be said to encode a language $\mathcal{L}(\mathcal{B})$, which represents the set of ω-words accepted by $\mathcal{B}$. The concept of a *run* is used to formally describe such ω-words.

Definition 3.8 (Büchi run)**.** *Let $r = (r_0, \dots)$ denote an infinite sequence of states $r_i \in Q$. A run on a Büchi automaton is then an ω-word σ such that $r_0 = q_0$ for the initial state, and $r_{i+1} = \delta(r_i, \sigma_i)$ for any $i \geq 0$.*

Not all ω-words that can be described by a run on an automaton $\mathcal{B}$ are also *accepted* by $\mathcal{B}$, but those that do are part of the language $\mathcal{L}(\mathcal{B})$ encoded by $\mathcal{B}$.

Definition 3.9 (Büchi acceptance)**.** *Let $\mathcal{B}$ denote a Büchi automaton $(\Sigma, Q, q_0, \delta, F)$, and let the function $\inf : \sigma \to Q$ denote a set of states that occur infinitely often. An ω-word $\sigma \in \Sigma^\omega$ is accepted by $\mathcal{B}$, i.e. $\alpha \in \mathcal{L}(\mathcal{B})$, iff it is the case that $\inf(\sigma) \cap F \neq \varnothing$, i.e. at least one of the accept states is encountered infinitely often.*

These automata are often used in the context of *model checking* and *satisfiability checking*. In the case of model checking, we can describe a system in terms of a Büchi automaton $\mathcal{B}_{sys}$ such that the set of ω-words that are accepted by $\mathcal{B}_{sys}$ correspond to the set of possible system traces. A Büchi automaton thus describes a language $\mathcal{L}(\mathcal{B}) \subseteq \Sigma^\omega$. LTL is commonly used to describe the properties of a system which we want to verify. These properties can then be translated into equivalent Büchi automata in various ways, by converting an LTL specification ϕ into a Büchi automaton $\mathcal{B}_\phi$. There exist many techniques for the construction of Büchi automata, and a survey is presented by Vardi (2007). As illustrated in Figure 3.2, if we can determine that $\mathcal{L}(\mathcal{B}_{sys}) \subseteq \mathcal{L}(\mathcal{B}_\phi)$, we prove that the system adheres to the formal LTL specifications. This is done by checking for the emptiness property $\mathcal{L}(\mathcal{B}_{sys}) \cap \mathcal{L}(\mathcal{B}_{\neg\phi}) = \varnothing$, meaning there are no ω-words which are part of the system's language while also being in violation of the LTL specifications denoted by ϕ. In the case of satisfiability checking, the problem is whether there exists an ω-word that satisfies a formula ϕ, which requires determining whether $\mathcal{L}(\mathcal{B}_\phi) \neq \varnothing$.

3.4 Runtime verification

Model checking has as an advantage that it allows for formal correctness proofs for systems without needing to test the system during run-time. However, to do so, it

requires a system model $\mathcal{B}_{sys}$, which may not always be explicitly available. Instead, *runtime verification* considers the case where the behaviour of a system needs to be checked on-line during runtime. These incremental approaches commonly utilise the equivalence

$$\phi \mathsf{U} \psi \equiv \psi \vee (\phi \wedge \mathsf{X}(\phi \mathsf{U} \psi)). \tag{3.15}$$

This dissertation therefore focuses on the case where we can observe the system's runs during run-time, also called *traces* or *paths*. The problem of *path checking*, which is a type of runtime verification, is to determine whether such a trace σ is a model of a wff ϕ, i.e. whether $\sigma \models \phi$. Since runtime verification practically only deals with finite time, this affects the choice of formula ϕ. In particular, ϕ is picked based on the ability of a runtime verification system to determine whether it is violated by a finite-length prefix. Kupferman and Vardi (2001) refer to these prefixes as *bad prefixes* (in addition to *good prefixes*), and the LTL formulas for which these prefixes exist *safety properties*. This leads us to the concept of *safety languages*, defined as follows.

Definition 3.10 (Safety language)**.** *A a safety language is any language $\mathcal{L} \subseteq \Sigma^\omega$ for which each word $\alpha \notin \mathcal{L}$ has a bad prefix.*

Runtime verification techniques have for example been used by Doherty et al. (2009) for execution monitoring in autonomous UAV applications, in which path checking of MITL formulas was used to check whether the execution of a plan is in accordance with expectations. Path checking is sufficient for any application which only needs to check whether a given wff is true or false for a given path, for example to check for adherence to safety requirements. The techniques for path checking can be roughly divided into three categories; using *automata-based model checking techniques*, using *proof systems*, or through the use of *derivatives-based syntactic rewritings*.

Automata

From an automata-theoretic perspective, it is not possible to check whether $\sigma \in \mathcal{L}(\mathcal{B}_\phi)$ by simulating a run over $\mathcal{B}_\phi$, because the acceptance condition would require an infinitely-long simulation. However, one important observation is that there sometimes exist finite-length prefixes for which no extension exists that would allow the resulting ω-word to be accepted by $\mathcal{B}_\phi$. For example, let $\phi = \mathsf{G}p$. Any finite-length prefix containing $\neg p$ cannot be extended into an ω-word that would be accepted by $\mathcal{B}_\phi$. If a formula ϕ is a safety property, then we can construct a finite-state automaton for $\neg\phi$ and check whether a prefix $\sigma_{<n}$ for some $n \in \mathbb{N}$ is accepted by that automaton.

Proof systems

Alternatively, LTL *proof systems* can be used for performing path checking. Cini and Francalanza (2015) in particular present a local proof system which can be ap-

Algorithm 3.1: Simplified progression

```
1  function PROGRESS(φ, s_i):
2  if φ = φ_1 ∨ φ_2 then
3  |  return PROGRESS(φ_1, s_i) ∨ PROGRESS(φ_2, s_i)
4  else if φ = ¬φ_1 then
5  |  return ¬PROGRESS(φ_1, s_i)
6  else if φ = φ_1 U_I φ_2 then
7  |  if I < 0 then
8  |  |  return ⊥
9  |  else if 0 ∈ I then
10 |  |  return PROGRESS(φ_2, s_i) ∨ (PROGRESS(φ_1, s_i) ∧ φ_1 U_{I−1} φ_2)
11 |  else
12 |  |  return PROGRESS(φ_1, s_i) ∧ φ_1 U_{I−1} φ_2
13 |  end
14 else
15 |  if φ ∈ s_i then
16 |  |  return ⊤
17 |  else
18 |  |  return ⊥
19 |  end
20 end
```

plied during run-time, by focusing on individual points rather than sets of points. Their proof system allows for the construction of both satisfaction ($\vdash^+$) and violation ($\vdash^-$) proofs from corresponding sets of rules. By incrementally and concurrently constructing proofs towards both satisfaction and violation judgements, the proof process can be in one of three modes at any given time. If the proof system finds a satisfaction proof for ϕ, then we have detected a good prefix and can terminate. Conversely, if the proof system finds a violation proof for ϕ, then we have detected a bad prefix and can terminate. Finally, if the proof system has found neither a satisfaction proof nor a violation proof, the prefix is too short to make any judgementt. Upon termination, we furthermore have an explicit proof that can be used to explain why a certain verdict was reached, although this proof can become very large.

Derivatives

In this dissertation, we consider as an alternative the *derivatives approach* to path checking. These approaches work by applying a state to a formula to obtain a new formula that has been evaluated for the state applied to it (see e.g. Havelund and Roşu (2001)). This is similar to the proof system discussed above, but does not keep track of an explanation. All of the information stored is contained within the formula that is continuously being rewritten.

An incremental procedure for real-time path checking MITL has previously been proposed by Bacchus and Kabanza (1998). Progression assumes complete states but allows the temporal distance between states to vary, i.e. they support positive val-

ues for delay $\Delta \in \mathbb{Z}^+$ between iterations. An adapted version of the *progression* procedure, called PROGRESS, is shown in Algorithm 3.1. It deviates from the original progression procedure by fixing the delay parameter to $\Delta = 1$, which corresponds to the assumption that the stream is synchronised at a regular time interval. A procedure for this type of synchronisation is described in more detail in Chapter 6.

Progression is a syntactic rewriting procedure which takes a formula together with a state and a delay, and produces a new formula obtained from evaluation the temporal interval in the input formula which is covered by the input state. This resulting output formula can then be used as the input formula for the next iteration of progression. This means that every iteration of progression has a time-complexity which is linear in the size of the formula, but the repeated application of progression can result in exponential formula growth due to the handling of temporal intervals on lines 6–13 in Algorithm 3.1. Bacchus and Kabanza (1998) further prove that their incremental approach is correct given the semantics of MITL. We provide our adapted version below.

Lemma 3.1: Correctness of simplified progression

The PROGRESS procedure is *correct*, i.e.

$$\sigma, i \models \phi \text{ iff } \sigma, i+1 \models \texttt{PROGRESS}(\phi, \sigma_i) \tag{3.16}$$

for traces σ, time-points $i \in \mathbb{N}$, and wffs ϕ.

Proof. Follows trivially from the correctness proof for progression on MITL per Bacchus and Kabanza (1998) by restricting timepoints to $\mathbb{N}$ and fixing the delay to $\Delta = 1$. ■

3.5 Formula simplification

One problem caused by progression is that it naively grows a formula without regard for the size-compactness of the resulting statement. To illustrate this, we can represent formulas by formula trees, which are a special case for abstract syntax trees.

Definition 3.11 (Formula tree)**.** *A formula tree $\mathcal{T}(\phi)$ is an abstract syntax tree for a wff ϕ such that*

1. *$\wedge, \vee, \leftrightarrow$ are represented by commutative binary vertices;*
2. *$\rightarrow$, U_I for intervals I are represented by non-commutative binary vertices;*
3. *$\neg$, G_I, F_I for intervals I are represented by unary vertices; and*
4. *propositions in $\mathcal{P}$ and verdicts $\top, \bot$ are represented by leaf vertices.*

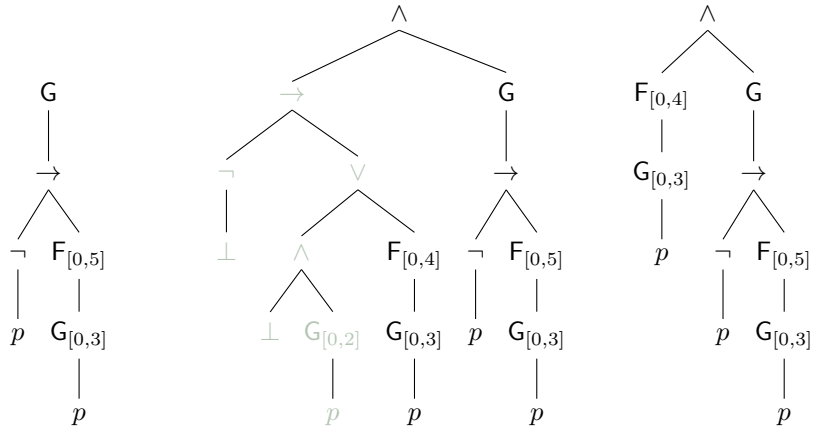

Figure 3.3: Formula trees $\mathcal{T}(\mathsf{G}(\neg p \rightarrow \mathsf{F}_{[0,5]}\mathsf{G}_{[0,3]}p))$ (left), and its progressed versions $\mathcal{T}(\texttt{PROGRESS}(\mathsf{G}(\neg p \rightarrow \mathsf{F}_{[0,5]}\mathsf{G}_{[0,3]}p, \varnothing)))$ before (middle) and after (right) formula simplification. The tree nodes in light green can be eliminated.

The size of a formula tree, denoted by $|\mathcal{T}(\phi)|$ for the formula tree of wff ϕ, is determined by the number of vertices in $\mathcal{T}(\phi)$.

Consider a wff $\mathsf{G}(\neg p \rightarrow \mathsf{F}_{[0,5]}\mathsf{G}_{[0,3]}p)$. Figure 3.3 (left) shows the formula tree $\mathcal{T}(\mathsf{G}(\neg p \rightarrow \mathsf{F}_{[0,5]}\mathsf{G}_{[0,3]}p))$. The space complexity can be expressed in terms of the size of the associated formula tree. If we were to progress this formula with a complete state $\varnothing$ (i.e. p is false), we would obtain the formula represented by the formula tree in Figure 3.3 (right) after *formula simplification*. However, prior to simplification, strict application of the `PROGRESS` procedure can produce large trees as illustrated by Figure 3.3 (middle). The unsimplified formula tree has some obvious redundancies (in light green) that could be eliminated to obtain a more concise tree. The key however is to perform these simplifications during construction of the progressed tree as an optimisation approach. This is done during progression by checking the current subtree against formula patterns and performing rewritings as shown in Table 3.1. Application of the rewriting rules then yields the formula simplification shown in Figure 3.3, which are used by the simplification procedure `SIMPLIFY` shown in Algorithm 3.2. A similar approach was taken by for example Shen et al. (2014) for LTL, in which they call the combination of progression and formula rewriting *convergent formula progression*.

Theorem 3.1: Correctness of formula simplification

The formula simplification procedure `SIMPLIFY` as shown in Algorithm 3.2 is *correct*, meaning that each rule ℓ in Table 3.1 is sound and `SIMPLIFY` terminates after a finite number of rewritings.

ℓ	Pattern	$\Rightarrow_\ell$	Product
1.	$\neg\top$		$\bot$
2.	$\neg\bot$		$\top$
3.	$\top \vee \phi$		$\top$
4.	$\bot \wedge \phi$		$\bot$
5.	$\top \wedge \phi$		ϕ
6.	$\bot \vee \phi$		ϕ
7.	$\phi \wedge \phi$		ϕ
8.	$\phi \vee \phi$		ϕ
9.	$\neg(\neg\phi)$		ϕ
10.	$(\phi \mathsf{U}_{[i,j]} \psi) \wedge (\phi \mathsf{U}_{[i,k]} \psi)$		$\phi \mathsf{U}_{[i,\min(j,k)]} \psi$
11.	$(\phi \mathsf{U}_{[i,j]} \psi) \vee (\phi \mathsf{U}_{[i,k]} \psi)$		$\phi \mathsf{U}_{[i,\max(j,k)]} \psi$
12.	$\phi \mathsf{U}_{[0,0]} \psi$		ψ

Table 3.1: Rewriting rules for wffs ϕ, ψ, χ where we assume $i \neq j \neq k$. Symmetric relationships are implicit for commutative vertices. Rules for syntactic sugar (i.e. $\mathsf{G}_I, \mathsf{F}_I, \rightarrow, \leftrightarrow$) follow implicitly from the rules listed.

Algorithm 3.2: Formula rewriting

```
1  function SIMPLIFY(φ):
2  while True do
3      φ' ← φ
4      foreach Rule ℓ in Table 3.1 do
           ▷ Apply rule ℓ to φ if applicable, otherwise keep φ unchanged:
5          φ ← ℓ(φ)
6      end
7      if φ' = φ then
8          return φ
9      end
10 end
```

Proof. For each pattern-product pair $x \Rightarrow_\ell y$ in Table 3.1, it is the case that $x \rightarrow y$ is true for all interpretations. Additionally, because each applicable rule ℓ reduces the size of a formula it is applied to, formula rewriting terminates after a finite number of rewritings. ■

While formula simplification does not change the conclusions progression would otherwise yield, it does provide a deterministic method for compacting formula representations, resulting in lower space requirements. Furthermore, the deterministic transformations also make it possible to check the equality of two formulas by comparing their respective simplified formula trees, which is a feature we make use of when considering repeated progression of formulas.

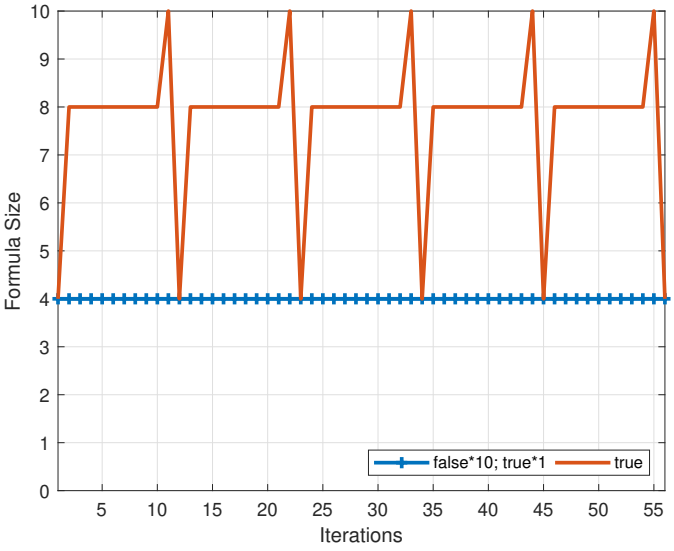

Figure 3.4: Formula size over time when progressing $\mathsf{GF}_{[0,10]}p$ over regular state sequences.

3.6 Empirical evaluation

The adapted progression procedure from Algorithm 3.1 combined with the application of rewriting rules from Table 3.1 was empirically evaluated. First, a baseline evaluation is given for the standard progression procedure. This is then followed up with a contrasting experiment where the impact of the rewriting rules is measured. The performance of progression has been studied previously (Doherty et al., 2009), but is included for the sake of completeness using our new implementation. The performance of progression is closely tied to the formula being progressed and the stream used for the progression. The evaluation of progression is therefore done through two separate experiments with different formulas and different streams. We primarily focus on the change in formula size over time, bearing in mind that the time complexity of progression is linear in the size of the formula.

In the first experiment we measure the formula growth over successive progressions for the formula

$$\mathsf{GF}_{[0,10]}p, \tag{3.17}$$

where the truth value of p is determined by a regular pattern. The the first pattern is illustrated by a crossed (blue) line; the second pattern is illustrated by an uninterrupted (red) line. The first pattern shows a sequence wherein p is false for 10 time-steps and becomes true for one time-step, before repeating. This means that the formula must grow in order to keep track of the eventually operator, for which the interval allows a delay of up to 10 time-steps before p has to be true in order for the formula to not be evaluated to false. The pattern uses the full duration allowed,

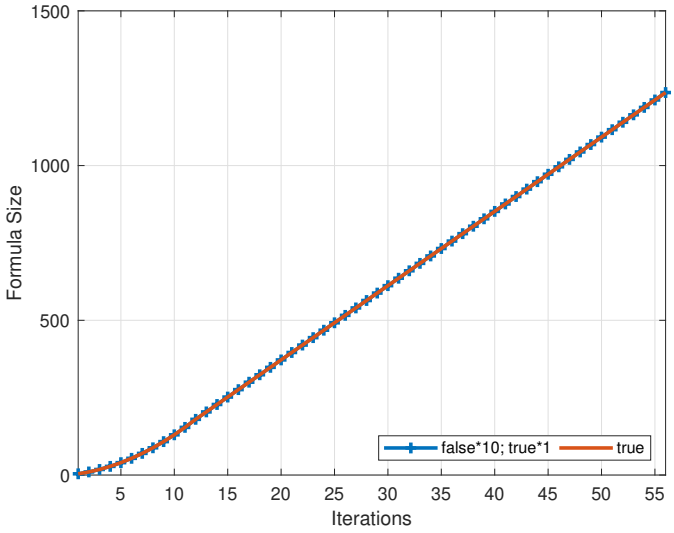

Figure 3.5: Formula size over time when progressing $\mathsf{GF}_{[0,10]}p$ without formula simplification.

and once p becomes true the formula shrinks again. This shrinking and growing behaviour can be correctly observed in Figure 3.4, where the shrinking occurs every 10 time-steps. For the second pattern, p is set to always be true. This corresponds to a state stream in which for every state p is set to true. The nesting of temporal operators is important here. Since p is always true, the eventually operator immediately evaluates to true as well, so the formula does not grow in size.

It is important to perform formula rewritings because they allow us to reduce the formula size where possible, resulting in the growing and shrinking patterns observed in Figure 3.4. If we however disable formula rewriting, the formula's growth becomes unbounded, as is illustrated in Figure 3.5. Progression will evaluate the formula $\mathsf{GF}_{[0,10]}p$ by rewriting it into $\mathsf{GF}_{[0,10]}p \wedge \mathsf{F}_{[0,9]}p \wedge p$. Since proposition p is not scoped by a temporal interval, it is then replaced with the truth value for p as specified by the state stream. Even if p is false, without simplification rules we cannot collapse the formula. To make matters worse, each subsequent iteration will again progress the original formula, which remains part of the progressed formula, resulting in the unbounded growth observed in Figure 3.5.

In the second experiment we similarly measure the formula growth over successive progressions for the formula

$$\mathsf{G}(\neg p \rightarrow \mathsf{F}_{[0,10]}\mathsf{G}_{[0,9]}p), \tag{3.18}$$

where the truth value of p is again determined by a regular pattern. Figure 3.6 shows the formula size for different stream patterns with formula rewriting enabled. Due to the semantics of implication, this formula only grows whenever p is false. The different state sequences show different degrees growth accordingly. In the best

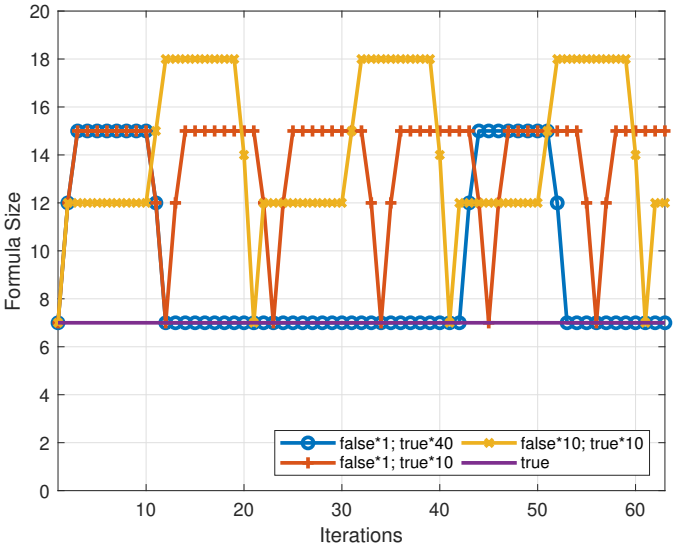

Figure 3.6: Formula size over time when progressing $\mathsf{G}(\neg p \rightarrow \mathsf{F}_{[0,10]}\mathsf{G}_{[0,9]}p)$ over regular state sequences.

case, p is never false and the formula is never expanded. If p does become false, the formula is expanded, and progression steps consequently take more time.

3.7 Summary

In this chapter we started with an introduction to the temporal logics, which are useful tools for describing logical specifications and requirements of information systems. To check the correctness of such a system, different techniques can be used. When the logical specification of a system is known, automata-based model checking techniques can be used. However, such a system specification is not always available, for a variety of reasons. In those cases, path checking techniques can be applied to check the system during runtime. We looked at the syntactic formula rewriting technique by Bacchus and Kabanza (1998) called progression, which can be used to check whether a trace satisfies a formula. We showed that it can however also be used to check whether a stream satisfies a formula under certain constraints. We then considered a number of rewriting rules, which make it possible to deterministically reduce the size of a formula, which in turn speeds up progression since its complexity depends on the size of the formula being progressed. The rewriting technique also makes it possible to check for formula equality, which is a useful property as we shall see in Chapter 4.

Chapter

4 Reasoning under uncertainty

HANDLING uncertainty is a vital ability for systems operating in the physical world. Just like humans, these systems only observe the world through inherently imprecise sensors that measure the shrouded fluents of reality. Furthermore, these sensors will only be able to observe a small part of the world, whereas many tasks require an agent to know facts about the unobserved world. We call this type of reasoning *reasoning under uncertainty*, which is a problem of high importance within the stream reasoning community and beyond. This chapter borrows from and extends previous work on progression-based path checking with incomplete states (de Leng and Heintz, 2018) under uncertainty (de Leng and Heintz, 2019).

4.1 Introduction

While a common problem within AI, the problem of reasoning with uncertain information was only recently identified as a problem of high importance in the area of stream reasoning (Dell'Aglio et al., 2017a, 2019). We therefore again consider a future-restricted MITL incremental path checking procedure under pointwise semantics. This time, however, we adapt the technique for application to *probabilistic* streams. Informally, these streams are represented using sets of states rather than single states for each time-point. Each state in a set of states has a probability associated with it, corresponding to the likelihood of the hypothetical state being the true state.

While our focus is on uncertainty represented by distributions over states, there have been many approaches where the uncertainty is directly represented in the logic itself. P-MTL (Tiger and Heintz, 2016) is a probabilistic temporal logic which allows for logical differentiation between observations and predictions. TLD (Kovtunova and Peñaloza, 2018) is a probabilistic extension of LTL which allows for tem-

poral uncertainty of event occurrences to be modelled through probabilistic distributions. LTL_4-C (Medhat et al., 2016) extends LTL by introducing absolute and relative 'counting quantifiers', allowing for the expression and monitoring of constraints pertaining to a (absolute or relative) lower or upper bound on instances. The query language TPQ (Koopmann, 2019) extends a temporal query language are based on LTL for ontology-based data access (OBDA) by introducing a probability operator ranging over statements in the TPQ language. Further, the work by Sato (1995); Sato and Kameya (2001) introduces distribution semantics for probabilistic logic programs.

In this chapter we first formally define what a probabilistic stream is and how it can be represented formally. We then consider the application of progression to these streams, which requires a different approach from the usual syntactic rewriting of single formulas. Concretely, we show how to construct so-called *progression graphs* that can be used to represent many formulas in an efficient manner. We then conclude by considering an incremental construction and maintenance process for these graphs, in line with the incremental nature of streams.

4.2 Prefix progression under uncertainty

In many practical applications, an agent is constrained by partial observability of its environment. Nevertheless, while certain facts may not be perceived (or even perceivable) directly, an agent can often, through reasoning, infer partial state information. Dealing with incomplete states is an important problem within the area of stream reasoning, and one which has been identified as being in need of further study (Dell'Aglio et al., 2017a, 2019). A good example of the occurrence of incomplete states is qualitative spatial reasoning, which is discussed in more detail in Chapter 5. Given a partial model of spatial relations between regions, qualitative spatial reasoning allows an agent to infer — for example through composition table based reasoning — a set of possible complete models consistent with the partial model. This set represents the set of possible hypotheses, each of which could be the 'true' model. Note that this set is disjunctive; we know beforehand that one of the hypothetical models must be true, but we cannot be certain which one. Likewise, to relax the constraint on complete states for progression, we consider disjunctive sets of states called *incomplete states*. This is a generalisation; we could model the case of complete states by considering only singleton incomplete states.

Model for incomplete prefixes

In the case of *incomplete state streams*, we want to be able to represent all of the different hypothetical complete streams it could represent. We therefore represent an incomplete state stream as a set of complete state streams, which are in agreement for complete states but are in disagreement for incomplete states.

Definition 4.1 (Incomplete state stream)**.** *An incomplete state stream of size $N \in \mathbb{Z}^+$, representing N hypothetical complete streams $\sigma \in \rho$, is denoted by*

$$\rho = \left\{ \sigma^{(1)}, \sigma^{(2)}, \ldots, \sigma^{(N)} \mid \sigma^{(i)} \in \Sigma^\omega \right\} \subset \Sigma^\omega. \tag{4.1}$$

Definition 4.2 (Incomplete state)**.** *An incomplete state is denoted by a set of states*

$$\rho_n = \left\{ \sigma_n^{(1)}, \sigma_n^{(2)}, \ldots, \sigma_n^{(|\rho|)} \mid n \in \mathbb{N} \wedge \sigma^{(i)} \in \rho \right\} \subseteq \Sigma. \tag{4.2}$$

Our goal is to extend progression to be able to process incomplete state streams in a meaningful way. Due to its incremental nature, progression produces results for (finite) sequences of states, and in many cases does not require the entirety of a stream before returning a verdict (i.e. $\top, \bot$). Such a finite sequence of states corresponds to a stream *prefix*.

Definition 4.3 (Prefix)**.** *A prefix for complete state streams σ and incomplete state streams ρ for time-points $n \in \mathbb{N}$ is denoted by*

$$\sigma_{<n} = (\sigma_0 \ldots \sigma_{n-1}), \tag{4.3}$$

$$\rho_{<n} = \left\{ \sigma_{<n}^{(1)}, \ldots, \sigma_{<n}^{(|\rho|)} \right\}. \tag{4.4}$$

The problem with prefixes is that they are incompatible with the semantics of temporal logics that assume infinite-length streams. Yet progression is able to produce verdicts prematurely given the right conditions. This happens when every possible infinite-length extension of a finite-length prefix satisfies certain constraints. Following the example of Kupferman and Vardi (2001), we call such prefixes *good prefixes* — and, for the converse case in which all possible infinite-length extension of a finite-length prefix satisfies the negation of the constraints, *bad prefixes*.

Definition 4.4 (Prefix truth)**.** *Let $\sigma_{<n}$ and $\rho_{<n}$ be prefixes. We define prefix truth using the short-hands*

$$\sigma_{<n} \models \phi \text{ iff } \sigma_{<n} + \sigma' \models \phi \text{ for all } \sigma' \in \Sigma^\omega, \tag{4.5}$$

$$\rho_{<n} \models \phi \text{ iff } \sigma_{<n} \models \phi \text{ for all } \sigma_{<n} \in \rho_{<n}, \tag{4.6}$$

where we denote the concatenation of words by the '+' operator.

In particular, progression is able to terminate when every possible infinite-length extension of a finite-length prefix is a model of the formula being progressed. We can connect progression reaching a verdict after n iterations to the satisfaction relation for those n-length-prefix extensions.

Definition 4.5 (Prefix progression)**.** *We denote the repeated application of* PROGRESS *to an initial formula ϕ over an n-length prefix $\sigma_{<n}$ of complete state stream σ, called prefix progression, by*

$$\textit{PROGRESS}^n(\phi, \sigma) = \textit{PROGRESS}(\textit{PROGRESS}^{n-1}(\ldots), \sigma_{n-1}), \tag{4.7}$$

where $n \in \mathbb{N}$. For base-case $n = 0$ we have a fix-point; $\textit{PROGRESS}^0(\phi, \sigma) = \phi$.

Theorem 4.1: Soundness of prefix progression

The application of progression over prefixes is *sound* wrt the semantics of MITL for any wff ϕ and prefix $\sigma_{<n}$, i.e. $\texttt{PROGRESS}^n(\phi, \sigma_{<n}) = \top$ implies that $\sigma_{<n} + \sigma' \models \phi$ for all $\sigma' \in \Sigma^\omega$.

Proof. Assume that $\texttt{PROGRESS}^n(\phi, \sigma_{<n}) = \top$. Since $\sigma_{<n} + \sigma', n \models \top$ for all $\sigma' \in \Sigma^\omega$, it is therefore the case that $\sigma_{<n} + \sigma', n \models \texttt{PROGRESS}^n(\phi, \sigma_{<n})$ for all $\sigma' \in \Sigma^\omega$. From the definition of prefix progression (Definition 4.5), this is equivalent to $\sigma_{<n} + \sigma', n \models \texttt{PROGRESS}(\texttt{PROGRESS}^{n-1}(\dots), \sigma_{n-1})$ for all $\sigma' \in \Sigma^\omega$. Applying Lemma 3.1, the correctness of simplified progression, n times then yields $\sigma_{<n} + \sigma', 0 \models \texttt{PROGRESS}^0(\phi, \epsilon)$ for all $\sigma' \in \Sigma^\omega$ and word terminator (i.e. empty word) ϵ, which according to Definition 4.5 is equivalent to $\sigma_{<n} + \sigma' \models \phi$. ■

Theorem 4.2: Incompleteness of prefix progression

The application of progression over prefixes is *incomplete* wrt the semantics of MITL for any wff ϕ and prefix $\sigma_{<n}$, i.e. $\sigma_{<n} + \sigma' \models \phi$ for all $\sigma' \in \Sigma^\omega$ does *not* imply that $\texttt{PROGRESS}^n(\phi, \sigma_{<n}) = \top$.

Proof. It suffices to show a counter-example to completeness. One such example is for $\phi = \mathsf{G}p \vee \mathsf{F}\neg p$ and an arbitrary prefix $\sigma_{<n}$. The formula ϕ is true for any infinite-length ω-word in Σ^ω, so $\sigma_{<n} + \sigma' \models \phi$ for all $\sigma' \in \Sigma^\omega$. However, since n is finite, there exists no n such that $\texttt{PROGRESS}^n(\phi, \sigma_{<n}) = \top$, demonstrating incompleteness. ■

Corollary 4.1: Partial completeness of prefix progression

The application of progression over prefixes is complete for the safety fragment of MITL for any wff ϕ and prefix $\sigma_{<n}$.

Proof. The counter-examples to (full) completeness hinge on the choice of a wff ϕ which has no good prefixes and no counter-models. Therefore, let ϕ be a safety property as described in Definition 3.10. Due to the safety restriction it is guaranteed for an $n \in \mathbb{Z}^+$ to exist such that $\texttt{PROGRESS}^n(\phi, \sigma_{<n}) = \top$ whenever $\sigma_{<n} + \sigma' \models \phi$ for all $\sigma' \in \Sigma^\omega$. ■

Satisfaction probability

We have thus far considered purely disjunctive sets of states as incomplete states. This can be further enhanced by providing a probabilistic grounding for such disjunctive sets. In the absence of priors, one may reasonably assign a uniform probability distribution to the set of complete states making up an incomplete state. However, if prior information exists, we may wish to change the probabilities of the complete states such that some become more likely while others become less likely. These probabilities are assumed to be given — for each time-point — by the incomplete state stream. The set of states Σ is associated with a time-varying, discretely-distributed stochastic variable $\mathbf{S}_n \sim \text{Discrete}(\theta_n)$ for time-points $n \in \mathbb{N}$, where $\theta_\mathbf{n} = \{\theta_{n,s}\}_{s\in\Sigma}$ represents a probability mass function (pmf) for states $s \in \Sigma$ and time-points $n \in \mathbb{N}$. While the discrete distribution for $\mathbf{S}_n$ could under certain conditions be learned, we will assume it is given. We write $Pr\left(\mathbf{S}_n = s\right) = \theta_{n,\sigma}$ to denote the probability of observing a state s at time-point $n \in \mathbb{N}$. The conditional probability of a complete state s given an observed incomplete state ρ_n is denoted by

$$Pr\left(\mathbf{S}_n = s \mid \rho_n\right) = \frac{\theta_{n,s} \cdot [s \in \rho_n]}{\sum_{s' \in \rho_n} \theta_{n,s'}}, \tag{4.8}$$

where the notation $[\cdot]$ represents Iverson brackets[5]. Similarly, we denote the (conditional) probability of observing prefixes by

$$Pr\left(\mathbf{S}_{<n} = \sigma_{<n}\right) = \prod_{i=0}^{n-1} Pr\left(\mathbf{S}_i = \sigma_i\right), \tag{4.9}$$

$$Pr\left(\mathbf{S}_{<n} = \sigma_{<n} \mid \rho_{<n}\right) = \prod_{i=0}^{n-1} Pr\left(\mathbf{S}_i = \sigma_i \mid \rho_i\right). \tag{4.10}$$

This notation makes it possible to refer to the posterior probability distribution at time-point n by combining the prior $\mathbf{S}_n$ with an observation of an incomplete state ρ_n.

Definition 4.6 (Prefix satisfaction probability)**.** *The (conditional) probability of a complete prefix $\sigma_{<n}$ to satisfy a wff ϕ is denoted by*

$$Pr(\sigma_{<n} \models \phi) = Pr(\mathbf{S}_{<n} = \sigma_{<n}) \cdot [\sigma_{<n} \models \phi], \tag{4.11}$$

$$Pr(\sigma_{<n} \models \phi \mid \rho_{<n}) = Pr(\mathbf{S}_{<n} = \sigma_{<n} \mid \rho_{<n}) \cdot [\sigma_{<n} \models \phi]. \tag{4.12}$$

Similarly, the probability of an incomplete prefix $\rho_{<n}$ to satisfy a wff ϕ is denoted by

$$Pr(\rho_{<n} \models \phi) = \sum_{\sigma_{<n} \in \rho_{<n}} Pr(\sigma_{<n} \models \phi \mid \rho_{<n}). \tag{4.13}$$

[5] Iverson brackets contain a Boolean statement and resolve to 1 if that statement is true, and 0 otherwise, i.e. for a Boolean statement b, $[b] = 1$ iff b is true; otherwise $[b] = 0$.

Theorem 4.3: Monotonicity of prefix satisfaction probability

Let ρ denote an incomplete state stream and ϕ an arbitrary wff. The probability of prefix satisfaction grows monotonically as $n \in \mathbb{Z}^+$ increases, such that

$$Pr(\rho_{<n} \models \phi) \leq Pr(\rho_{<n+1} \models \phi), \tag{4.14}$$
$$Pr(\rho_{<n} \not\models \phi) \leq Pr(\rho_{<n+1} \not\models \phi). \tag{4.15}$$

Proof. Per Definition 4.6, we can rewrite the prefix satisfaction probability into

$$Pr(\rho_{<n} \models \phi) = \sum_{\sigma_{<n} \in \rho_{<n}} (Pr(\mathbf{S}_{<n} = \sigma_{<n} \mid \rho_{<n}) \cdot [\sigma_{<n} \models \phi]), \tag{4.16}$$
$$Pr(\rho_{<n} \not\models \phi) = \sum_{\sigma_{<n} \in \rho_{<n}} (Pr(\mathbf{S}_{<n} = \sigma_{<n} \mid \rho_{<n}) \cdot [\sigma_{<n} \not\models \phi]). \tag{4.17}$$

Let $\rho_n = \left\{\sigma_n^{(1)}, \ldots, \sigma_n^{(N)}\right\}$. By concatenating this incomplete state with the incomplete prefix $\rho_{<n}$ we obtain

$$\rho_{<n+1} = \left\{\sigma_{<n} + \sigma_n^{(i)} \mid \sigma_{<n} \in \rho_{<n} \text{ and } 1 \leq i \leq N\right\}. \tag{4.18}$$

Per Definition 4.4, if $\sigma_{<n} \models \phi$ then also $\sigma_{<n} + \sigma' \models \phi$ for any choice of $\sigma' \in \Sigma$. Conversely, if $\sigma_{<n} \not\models \phi$ then also $\sigma_{<n} + \sigma' \not\models \phi$ for any choice of $\sigma' \in \Sigma$. Therefore the following also holds;

$$\sigma_{<n} \models \phi \Rightarrow \sigma_{<n} + \sigma' \models \phi, \tag{4.19}$$
$$\sigma_{<n} \not\models \phi \Rightarrow \sigma_{<n} + \sigma' \not\models \phi, \tag{4.20}$$

for all $\sigma' \in \rho_n$, since $\rho_n \subseteq \Sigma$. For these two cases, any extension of $\sigma_{<n}$ neither increases nor decreases the prefix satisfaction probability.

However, if neither $\sigma_{<n} \models \phi$ nor $\sigma_{<n} \models \phi$, then it is possible that $\sigma_{<n} + \sigma' \models \phi$ or $\sigma_{<n} + \sigma' \not\models \phi$ for some $\sigma' \in \rho_n$. In that former case, $Pr(\rho_{<n+1} \models \phi) > Pr(\rho_{<n} \models \phi)$, whereas in the latter $Pr(\rho_{<n+1} \not\models \phi) > Pr(\rho_{<n} \not\models \phi)$. This means that in the general case $Pr(\rho_{<n} \models \phi) \leq Pr(\rho_{<n+1} \models \phi)$ and $Pr(\rho_{<n} \not\models \phi) \leq Pr(\rho_{<n+1} \models \phi)$, which shows that the probability of prefix satisfaction grows monotonically. ■

4.3 Progression graphs

To perform progression with incomplete states, we can apply simplified progression (Algorithm 3.1) for each of the complete states represented by an incomplete state. A graph structure can be used to leverage the property that different formulas progressed with different states can produce the same resulting formula, constraining the potential combinatoric explosion. An example of this is illustrated in

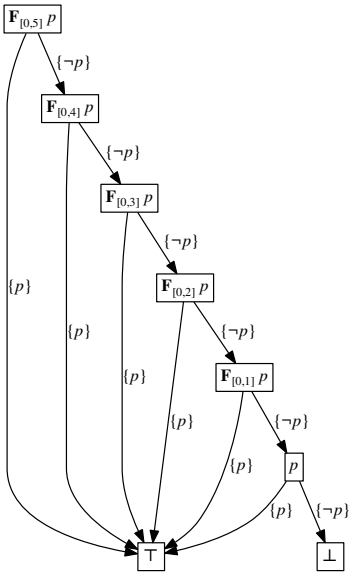

Figure 4.1: Example progression graph for the formula $\mathsf{F}_{[0,5]}p$. Vertices represent formulas; edges are labelled with complete states to illustrate under which logical state a formula progresses into a formula. Reflexive edges for the verdicts are omitted for clarity.

Figure 4.1, which shows how formulas progress into formulas given some complete state. We start with a formula we wish to progress, in this case $\mathsf{F}_{[0,5]}p$. The graph structure follows the PROGRESS procedure by requiring that there exists a directed edge (ϕ, ψ, s) iff $\texttt{PROGRESS}(\phi, s) = \psi$. Hence we can observe $\psi = \mathsf{F}_{[0,4]}p$ is reachable from $\phi = \mathsf{F}_{[0,5]}p$ for complete state $s = \varnothing$, whereas $\psi = \top$ is reachable from ϕ for complete state $s = \{p\}$. Since all vertices have an out-degree equal to $|\Sigma|$ — the figure omits the reflexive edges for verdict nodes for clarity — the graph is a complete encoding of progression for $\mathsf{F}_{[0,5]}p$.

These *progression graphs* can be represented as a type of finite state automaton. Given a wff ϕ, a progression graph $\mathcal{G}(\phi) = (\phi, V, E)$ can distinguish between acceptance of both ϕ and $\neg\phi$, where the latter represents a 'rejection' of ϕ. If $\top \in V$, that means that there exist finite-length accepting runs, which corresponds to the existence of good prefixes. Conversely, if $\bot \in V$, that means that there exist finite-length *rejecting* runs, which corresponds to the existence of *bad* prefixes. So if $\top \in V$, it is an accepting state, with the analogous holding for $\bot$. If neither $\top$ nor $\bot$ are states of $\mathcal{G}(\phi)$, then there exist no good or bad prefixes for ϕ, which means that ϕ is not a safety formula. In addition to encoding the structure of progression — and unlike standard finite state automata — progression graphs also encode probabilistic information in terms of *probability mass* associated with each formula.

Definition 4.7 (Progression graph)**.** *A progression graph is a directed graph* $\mathcal{G}_n(\phi) = (\phi, V, E, m_n)$ *at time-point* n *consisting of a set of wffs* V *such that* $\phi \in V$*, a set of directed labelled transitions*

$$E = \{(v, v', s) \in V \times V \times \Sigma \mid \mathit{PROGRESS}(v, s) = v'\}, \tag{4.21}$$

and a probability mass function $m_n : V \to [0, 1]$ *representing a probability distribution over formulas in* $v \in V$ *defined as*

$$m_n(v) = \sum_{\sigma_{<n} \in \rho_{<n}} \left(Pr\left(\mathbf{S}_{<n} = \sigma_{<n} \mid \rho_{<n}\right) \cdot [\mathit{PROGRESS}^n(\phi, \sigma_{<n}) = v]\right), \tag{4.22}$$

and $m_0(\phi) = 1$ *corresponds to the base-case. We will use the a-temporal shorthand* $\mathcal{G}(\phi) = (\phi, V, E)$ *when referring only to the structure of a progression graph.*

Given a progression graph $\mathcal{G}_n(\phi)$, the probability mass $m_n(\psi)$ for a formula ψ represents the probability that progression of the 'true' stream would have produced the formula ψ at time-point n. At time-point $n = 0$, no part of the stream has yet been observed, and thus all of the probability mass resides in the to-be-progressed source formula ϕ. The exact process involved in incrementally updating a progression graph given these components is presented later; for now it suffices to assume that such a process exists. In that case, we can consider the probabilistic counterpart to the satisfaction relation for incomplete state stream prefixes; *satisfaction probability*.

Theorem 4.4: Correctness of progression graphs

Given a progression graph $\mathcal{G}_n(\phi) = (\phi, V, E, m_n)$ and an incomplete state stream prefix $\rho_{<n}$ for any time-point $n \in \mathbb{N}$. Let $Pr(\rho_{<n} \models^? \phi)$ denote $Pr(\text{not } (\rho_{<n} \models \phi \text{ or } \rho_{<n} \not\models \phi))$. Then it is the case that

$$Pr\left(\rho_{<n} \models \phi\right) = m_n(\top), \tag{4.23}$$

$$Pr\left(\rho_{<n} \not\models \phi\right) = m_n(\bot), \tag{4.24}$$

$$Pr\left(\rho_{<n} \models^? \phi\right) = 1 - (m_n(\top) + m_n(\bot)). \tag{4.25}$$

Proof. Per Definition 4.7, the pmf m_n for verdicts is based on the sum of the probabilities of the complete state stream prefixes progressing to those verdicts by time-point n.

$$m_n(\top) = \sum_{\sigma_{<n} \in \rho_{<n}} \left(Pr\left(\mathbf{S}_{<n} = \sigma_{<n} \mid \rho_{<n}\right) \cdot [\mathtt{PROGRESS}^n(\phi, \sigma_{<n}) = \top]\right), \tag{4.26}$$

$$m_n(\bot) = \sum_{\sigma_{<n} \in \rho_{<n}} \left(Pr\left(\mathbf{S}_{<n} = \sigma_{<n} \mid \rho_{<n}\right) \cdot [\mathtt{PROGRESS}^n(\phi, \sigma_{<n}) = \bot]\right). \tag{4.27}$$

From Theorems 4.1 and 4.2, we know that prefix progression is sound but incomplete.

Consider the case where $n = 0$, from which we know that $V = \{\phi\}$, $E = \varnothing$, and $m_0(\phi) = 1$. This corresponds to not yet having observed any part of the incomplete state stream ρ. In this case, $m_0(\top) = [\phi = \top]$ and $m_0(\bot) = [\phi = \bot]$.

Consider the case where $\top \notin V$ and $n > 0$. That means that there is no $m \leq n$ such that $\texttt{PROGRESS}^m(\phi, \sigma_{<n}) = \top$ for any $\sigma_{<n} \in \rho_{<n}$. This means that there exists no good prefix $\sigma_{<n} \in \rho_{<n}$ for ϕ, so $m_n(\top) = 0$. Conversely, if $\bot \notin V$ and $n > 0$, then there is no $m \leq n$ such that $\texttt{PROGRESS}^m(\phi, \sigma_{<n}) = \bot$ for any $\sigma_{<n} \in \rho_{<n}$. This means that there exists no bad prefix $\sigma_{<n} \in \rho_{<n}$ for ϕ, so $m_n(\bot) = 0$. Both cases match the definitions for $Pr\left(\rho_{<n} \models \phi\right)$ and $Pr\left(\rho_{<n} \not\models \phi\right)$ respectively.

Consider the case where $\top \in V$ (or: $\bot \in V$) and $n > 0$. Then ϕ is guaranteed to have at least one good (or: bad) prefix due to the existence of a sequence of edges leading from ϕ to $\top$ (or: $\bot$). This means that ϕ is a safety property. Based on Theorem 4.1 and Corollary 4.1, we can therefore rewrite $m_n(\top)$ and $m_n(\bot)$ to

$$m_n(\top) = \sum_{\sigma_{<n} \in \rho_{<n}} \left(Pr\left(\mathbf{S}_{<n} = \sigma_{<n} \mid \rho_{<n}\right) \cdot [\forall \sigma' \in \Sigma^\omega (\sigma_{<n} + \sigma' \models \phi)]\right), \tag{4.28}$$

$$m_n(\bot) = \sum_{\sigma_{<n} \in \rho_{<n}} \left(Pr\left(\mathbf{S}_{<n} = \sigma_{<n} \mid \rho_{<n}\right) \cdot [\forall \sigma' \in \Sigma^\omega (\sigma_{<n} + \sigma' \not\models \phi)]\right). \tag{4.29}$$

This corresponds to the satisfaction probability of the observed incomplete state stream wrt ϕ, i.e.

$$\begin{aligned} m_n(\top) &= \sum_{\sigma_{<n} \in \rho_{<n}} \left(Pr\left(\mathbf{S}_{<n} = \sigma_{<n} \mid \rho_{<n}\right) \cdot [\sigma_{<n} \models \phi]\right) \\ &= Pr(\rho_{<n} \models \phi), \end{aligned} \tag{4.30}$$

$$\begin{aligned} m_n(\bot) &= \sum_{\sigma_{<n} \in \rho_{<n}} \left(Pr\left(\mathbf{S}_{<n} = \sigma_{<n} \mid \rho_{<n}\right) \cdot [\sigma_{<n} \not\models \phi]\right) \\ &= Pr(\rho_{<n} \not\models \phi). \end{aligned} \tag{4.31}$$

Then it trivially follows from the definition of $Pr(\rho_{<n} \models^? \phi)$ that

$$Pr\left(\rho_{<n} \models^? \phi\right) = 1 - (m_n(\top) + m_n(\bot)), \tag{4.32}$$

which matches the original claim. ■

Given a formula ϕ, a progression graph can be regarded as automaton that accepts runs σ such that $\sigma \models \phi$ whenever $\top$ is a node in that progression graph. The opposite holds as well; a progression graph can be regarded as automaton that accepts runs σ such that $\sigma \not\models \phi$ whenever $\bot$ is a node in that progression graph.

Theorem 4.5: Progression graphs encode good/bad prefixes

Let ϕ denote a well-formed MITL formula, $\mathcal{G}(\phi) = (\phi, V, E)$ a full progression graph for wffs V such that $\phi \in V$, and

$$E = \{(v, v', s) \in V \times V \times \Sigma \mid \texttt{PROGRESS}(v, s) = v'\}. \tag{4.33}$$

We define a transition function $\delta_E : V, \Sigma \to V$ based on E as follows:

$$\delta_E : (v, s) \mapsto v' \text{ for all } (v, v', s) \in E. \tag{4.34}$$

Then let $\mathcal{A}_\phi = (\Sigma, V, \phi, \delta_E, \{\top\})$ and $\mathcal{A}_{\neg\phi} = (\Sigma, V, \phi, \delta_E, \{\bot\})$ represent deterministic finite-state automata. Then the following two relationships hold:

- if there exists a prefix $\sigma_{<n}$ such that $(\phi, v_0, \sigma_0), \ldots, (v_{n-2}, \top, \sigma_{n-1}) \in E$, then $\mathcal{A}_\phi$ accepts $\sigma_{<n}$;
- if there exists a prefix $\sigma_{<n}$ such that $(\phi, v_0, \sigma_0), \ldots, (v_{n-2}, \bot, \sigma_{n-1}) \in E$, then $\mathcal{A}_{\neg\phi}$ accepts $\sigma_{<n}$.

Proof. Follows directly from Definition 3.6 describing accepting runs over FSAs. ■

These relationships are due to the edges of a progression graph being based on progression. This also means that a progression graph for a formula ϕ that lacks a node $\top$ does not have a good prefix. Likewise, if it lacks a node $\bot$ then it does not have a bad prefix. Consequently, if a progression graph for a formula ϕ lacks both the $\top$ and $\bot$ nodes, then ϕ is not a safety property.

Corollary 4.2: Progression graphs encode safety properties

Let ϕ denote a well-formed MITL formula, and $\mathcal{G}(\phi) = (\phi, V, E)$ a full progression graph for ϕ. Then ϕ is a safety property iff $\top \in V$ or $\bot \in V$.

Proof. If ϕ is a safety property as per Definition 3.10, then there must exist good or bad prefixes for ϕ, which means $\top \in V$ or $\bot \in V$ according to Theorem 4.5. Conversely, if $\top \in V$ or $\bot \in V$, then according to Theorem 4.5 there exist good or bad prefixes respectively, which per Definition 3.10 means that ϕ is a safety property. ■

Thus far we have shown that a progression graph is an accurate snapshot of progression over incomplete state streams for time-points $n \in \mathbb{N}$, where $n = 0$ represents the base-case before having observed any part of the stream. We have also assumed that the structure of a graph $\mathcal{G}_n(\phi)$ — its vertices V and edges E — remain

fixed across time. This is an assumption we will relax shortly, but it does present us with the worst-case space complexity of progression graphs. In the worst-case scenario, progression will repeatedly produce new formulas such that no two non-verdict formulas share a common child. In such a graph, each formula produces $|\Sigma|$ new formulas in accordance with its out-degree. In practice, some of these formulas will be shared, which is a necessary condition for the graph to be of finite size. This is because of the constraints forced upon the graph's structure by the `PROGRESS` procedure from Algorithm 3.1. This too therefore applies to progression graphs.

Formula simplification is beneficial to restricting the space requirements of progression graphs. Note that there is significant overlap between the trees before and after progression, both in terms of the subtrees for the 'eventually' operator as well as the unbounded 'always' operator. This is because the `PROGRESS` procedure only recombines subtrees with logical connectives and decremented-interval temporal operators. We can therefore cache parts of formula trees and use pointers whenever we are about to construct a cached formula tree as part of the `PROGRESS` procedure. The formula cache will in the worst case contain all interval-shifted subformulas of the original formula ϕ for which a progression graph $\mathcal{G}(\phi)$ — meaning the full progression graph for ϕ — was constructed. Basin et al. (2017) refer to this set as the set of *interval-skewed subformulas* (ISF), which was originally introduced by Thati and Roşu (2005).

Definition 4.8 (Interval-skewed subformulas)**.** *The (future-restricted[6]) set of interval-skewed subformulas (ISF) is defined for wffs ϕ as*

$$ISF(\phi) = SF(\phi) \cup \{\phi_1 \mathsf{U}_{I-n} \phi_2 \mid \phi_1 \mathsf{U}_I \phi_2 \in SF(\phi) \wedge n \in [1, \max(I)]\}\,, \quad (4.35)$$

where $SF(\phi)$ represents the inclusive set of subformulas of ϕ.

For example, Figure 4.1 shows $\mathcal{G}(\mathsf{F}_{[0,3]}p) = (\mathsf{F}_{[0,3]}p, V, E, m_n)$, and for which $ISF(\mathsf{F}_{[0,3]}p) = V \setminus \{\top, \bot\}$. Additionally, given a formula ϕ, the size of its ISF is proportional to the size of its subformula set, i.e. $|ISF(\phi)| \propto |SF(\phi)|$. Taking into account the predetermined out-degree of formula vertices, the space complexity of a progression graph $\mathcal{G}(\phi)$ is therefore $\mathcal{O}(|SF(\phi)| \cdot |\Sigma|)$.

The use of formula simplification and formula caching in conjunction with the progression procedure yields the `REPROGRESS` procedure shown in Algorithm 4.1. As the name implies, `REPROGRESS` is optimised for use-cases in which progression is performed multiple times. It takes the same input information as Algorithm 3.1, plus a formula tree cache Ω, which can be initialised with $SF(\phi)$. The notation $\Omega[\phi]$ returns a pointer to a cached tree $\mathcal{T}(\phi)$ if one already exists; otherwise one is constructed and added to the cache first. This prevents duplicate subtrees from being stored, and allows for subtrees to be re-used. The `SIMPLIFY`(ψ) operation (see Algorithm 3.2) is performed by repeatedly applying the inference rules from Table 3.1 until no more rules can be applied, and returning the result.

[6] Basin et al. (2017) additionally use the backwards-looking temporal operator 'since', which we do not consider in this work, hence the 'future-restricted' qualifier.

Algorithm 4.1: Repeat-progression

```
1  function REPROGRESS(φ, s_i, Ω):
2  if φ = φ_1 ∨ φ_2 then
3      ψ ← REPROGRESS(φ_1, s_i, Ω) ∨ REPROGRESS(φ_2, s_i, Ω)
4      ψ* ← SIMPLIFY(ψ)
5      return Ω[ψ*]
6  else if φ = ¬φ_1 then
7      ψ ← ¬REPROGRESS(φ_1, s_i, Ω)
8      ψ* ← SIMPLIFY(ψ)
9      return Ω[ψ*]
10 else if φ = φ_1 U_I φ_2 then
11     if I < 0 then
12         return Ω[⊥]
13     else if 0 ∈ I then
14         ψ ← REPROGRESS(φ_2, s_i, Ω) ∨ (REPROGRESS(φ_1, s_i, Ω) ∧ φ_1 U_{I−1} φ_2)
15         ψ* ← SIMPLIFY(ψ)
16         return Ω[ψ*]
17     else
18         ψ ← REPROGRESS(φ_1, s_i, Ω) ∧ φ_1 U_{I−1} φ_2
19         ψ* ← SIMPLIFY(ψ)
20         return Ω[ψ*]
21     end
22 else
23     if φ ∈ s_i then
24         return Ω[⊤]
25     else
26         return Ω[⊥]
27     end
28 end
```

Theorem 4.6: Correctness of REPROGRESS

The REPROGRESS procedure is *correct* wrt simplified progression, meaning

$$\texttt{PROGRESS}(\phi, s) \equiv \texttt{REPROGRESS}(\phi, s, \Omega) \tag{4.36}$$

for any wff ϕ, complete state $s \in \Sigma$, and cache Ω.

Proof. In order for REPROGRESS to be correct, the formulas it produces need to be equivalent to the formulas produced by PROGRESS under the same inputs. If we ignore the caching mechanism in REPROGRESS, the procedure is identical to PROGRESS. Since PROGRESS is correct according to Lemma 3.1, REPROGRESS is also correct wrt simplified progression. ■

4.4 Incremental graph progression

Previously, we looked exclusively at progression graph snapshots; that is, the state of progression graphs at specific time-points. We now focus on the transitions between these snapshots by considering how probability mass flows through a progression graph, and how a progression graph is incrementally constructed, using accurate or approximate strategies. Concretely, the problem we consider here is how to compute a progression graph $\mathcal{G}_{n+1}(\phi)$ given a progression graph $\mathcal{G}_n(\phi)$ for wff ϕ at time-point $n \in \mathbb{N}$, where we assume for the base-case $\mathcal{G}_0(\phi) = (\phi, \{\phi\}, \varnothing, m_0)$.

Probability mass flow

Over time, we will incrementally observe an increasingly-large prefix $\rho_{<n}$ as n increases. The goal of a progression graph is to efficiently encode an arbitrarily-long prefix for all timepoints preceding n, and to only rely on the most recent incomplete state ρ_{n-1} to obtain $\mathcal{G}_n(\phi)$ from $\mathcal{G}_{n-1}(\phi)$. This is achieved through simultaneous updates to the pmf m_{n-1} to obtain m_n.

Lemma 4.1: Incremental updates

Let $\mathcal{G}_n(\phi), \mathcal{G}_{n-1}(\phi)$ be full progression graphs. An *update* from m_{n-1} to m_n given an incomplete state ρ_{n-1}, where $n > 0$, can be characterised by the update

$$m_n(v) \leftarrow \sum_{(v',v,s)\in E} \big(m_{n-1}(v') \cdot Pr\left(\mathbf{S}_{n-1} = s \mid \rho_{n-1}\right)\big). \tag{4.37}$$

Proof. We need to show that the full update Eq. 4.22 from Definition 4.7 for time-point n is equivalent to the full update for time-point $m = n - 1$ followed by an incremental update at time-point n as shown in the above relationship. By plugging Definition 4.7 into the incremental update rule, we get

$$\sum_{(v',v,s)\in E} \Bigg(\sum_{\sigma_{<m}\in\rho_{<m}} \left(Pr\left(\mathbf{S}_{<m} = \sigma_{<m} \mid \rho_{<m}\right) \cdot [\texttt{PROGRESS}^m(\phi, \sigma_{<m}) = v']\right) \cdot Pr\left(\mathbf{S}_m = s \mid \rho_m\right)\Bigg). \tag{4.38}$$

The inner sum ranging over $\sigma_{<m} \in \rho_{<m}$ can be rewritten to instead range over paths in the graph:

$$\sum_{(v',v,s)\in E} \left(\left(\sum_{(\phi,v',\sigma_{<m})\in E^m} Pr\left(\mathbf{S}_{<m} = \sigma_{<m} \mid \rho_{<m}\right) \right) \cdot Pr\left(\mathbf{S}_m = s \mid \rho_m\right) \right). \tag{4.39}$$

Algorithm 4.2: Graph progression

```
function GRAPH-PROGRESS(G_{n-1}, ρ_{n-1}, Ω):
G_n ← (V_{n-1}, E_{n-1}, [])
foreach v ∈ V_{n-1} do
    if m_{n-1}[v] > 0 then
        foreach s ∈ ρ_n do
            v' ← REPROGRESS(v, s, Ω)
            V_n ← V_n ∪ {v'}
            E_n ← E_n ∪ {(v, v', s)}
        end
        foreach (v, v', s) ∈ E_n do
            m_n[v'] ← m_n[v'] + m_{n-1}[v] · Pr(S_{n-1} = s | ρ_{n-1})
        end
    end
end
return (G_n, Ω)
```

We can now collapse the two sums into one sum ranging over paths from ϕ to v, appending the incomplete state ρ_n to the incomplete stream $\rho_{<m}$ to obtain $\rho_{<n}$:

$$\sum_{(\phi,v,\sigma_{<n})\in E^n} Pr\left(\mathbf{S}_{<n} = \sigma_{<n} \mid \rho_{<n}\right). \tag{4.40}$$

Plugging the PROGRESS function back in we can rewrite the mass assignment to

$$m_n(v) \leftarrow \sum_{\sigma_{<n}\in\rho_{<n}} \left(Pr\left(\mathbf{S}_{<n} = \sigma_{<n} \mid \rho_{<n}\right) \cdot \left[\texttt{PROGRESS}^n(\phi, \sigma_{<n}) = v\right]\right), \tag{4.41}$$

which matches Eq. 4.22 from Definition 4.7. ■

We thus only have to consider those vertices with a non-zero probability mass and their immediate neighbours when performing an update. Because the update is performed simultaneously for all vertices, it is possible for a vertex to receive fresh probability mass from multiple parents while distributing its own probability mass, if any, to its children. Finally, since we know what $\mathcal{G}_0(\phi)$ looks like for any wff ϕ, we only need to consider the most recent element of any prefix $\rho_{<n}$ without having to store any of its preceding incomplete states.

Incremental graph construction

Because we only need to keep track of graph vertices with a non-zero probability mass and their direct neighbours, it is not necessary to know the complete set of vertices and edges at time-point $n = 0$. Instead, we can incrementally construct parts of a progression graph when needed, and it is even possible to 'forget' parts of progression graphs when they are no longer required. We therefore propose an incremental graph construction procedure based on REPROGRESS, called

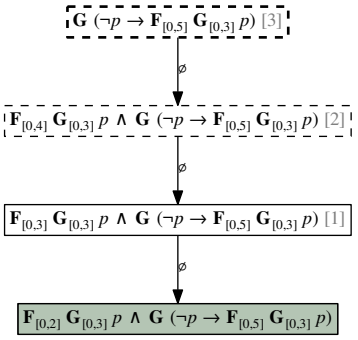

Figure 4.2: Example progression graph $\mathcal{G}_3(\mathsf{G}(\neg p \rightarrow \mathsf{F}_{[0,5]}\mathsf{G}_{[0,3]}p))$ after receiving state $\{\varnothing\}$ three times in a row.

GRAPH-PROGRESS, illustrated in Algorithm 4.2. The procedure starts out with the progression graph of the previous iteration, inheriting its structure as a base-line. It then creates a new but empty pmf for the new graph (line 2). The procedure considers all formulas which had probability mass associated with them according to the old pmf (line 4), and expands those formulas by applying REPROGRESS where needed (lines 5–9). Since the cache is shared between calls to graph progression, REPROGRESS benefits from the overlap between the formulas that make up the progression graph. The procedure then proportionally to ρ_{n-1} redistributes the probability mass from the previous iteration, constructing the new probability mass pmf (lines 10–12). The resulting progression graph and updated cache are then returned (line 15) to serve as input for the next iteration. The GRAPH-PROGRESS procedure can be shown to be correct wrt the definition of progression graphs based on the incremental update mechanism employed in lines 10–12.

Theorem 4.7: Correctness of GRAPH-PROGRESS

For every progression graph $\mathcal{G}_{n-1}$, the procedure GRAPH-PROGRESS produces a pmf $m_n[v]$ such that

$$m_n[v] = \sum_{\sigma_{<n} \in \rho_{<n}} \left(Pr\left(\mathbf{S}_{<n} = \sigma_{<n} \mid \rho_{<n}\right) \cdot \left[\texttt{PROGRESS}^n(\phi, \sigma_{<n}) = v\right]\right), \tag{4.42}$$

meaning GRAPH-PROGRESS is *correct* wrt the definition of a progression graph.

Proof. Follows directly from Algorithm 4.2 and its application of Lemma 4.1 on line 11. The admissibility of REPROGRESS on line 6 follows directly from Theorem 4.6. ■

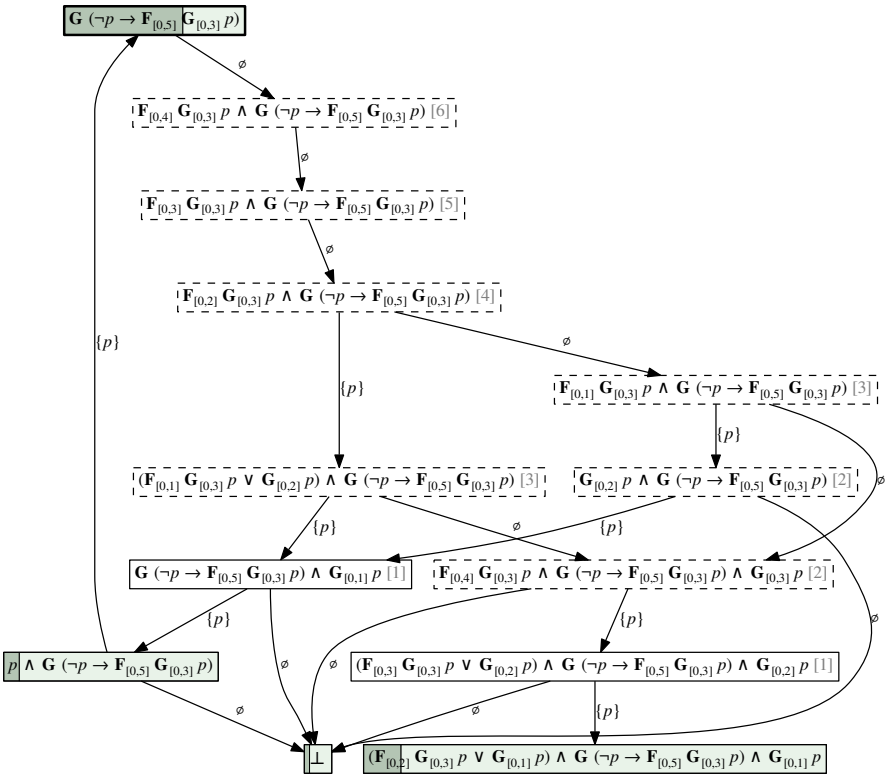

Figure 4.3: Example progression graph $\mathcal{G}_7(\mathsf{G}(\neg p \rightarrow \mathsf{F}_{[0,5]}\mathsf{G}_{[0,3]}p))$.

We can graphically illustrate the incremental construction behaviour of graph progression by considering our previous example formula $\phi = \mathsf{G}(\neg p \rightarrow \mathsf{F}_{[0,5]}\mathsf{G}_{[0,3]}p)$. Let us first consider the case where we receive states $\{\varnothing\}$, i.e. p is known to be false, for three consecutive iterations. Figure 4.2 shows the progression graph for ϕ at the end of the third iteration. The bottom formula, $\mathsf{F}_{[0,2]}\mathsf{G}_{[0,3]}p \wedge (\mathsf{G}(\neg p \rightarrow \mathsf{F}_{[0,5]}\mathsf{G}_{[0,3]}p))$, has a 100% green background, indicating it contains all of the probability mass. This matches the behaviour of the original progression procedure. Its parent has a solid box and an age of 1, illustrating this is a formula which held probability mass during the previous iteration. In general, its ancestors in turn are represented using dashed boxes with corresponding ages, indicating that these parts of the graph are dormant.

If we now consider the case in which we receive incomplete states $\{\{p\}, \varnothing\}$, i.e. completely unknown, where the probabilities for all time-points $n \geq 3$ are $Pr(\mathbf{S}_n = \{p\}) = 0.9$ and $Pr(\mathbf{S}_n = \varnothing) = 0.1$. Figure 4.3 shows the resulting graphs for iteration 7. We can observe how the probability mass has moved through the graph, and by iteration 7 both the verdict node $\bot$ and the original formula ϕ are receiving probability mass. Since mass cannot leave verdict nodes, the mass contained

within them will continue to grow. As more of the progression graph is explored, its size will also continue to grow.

Approximation strategies

One observation when looking at Figure 4.3 is that many formulas have gone stale, meaning that they have not received any probability mass for some time. In the ideal case, the *stale ratio* — being the ratio of stale formulas compared to the total number of formulas — of a progression graph stays close to 0 while only a few formulas get expanded. We can use formula removal strategies to shrink the size of a progression graph, at the cost of having to reacquire them through progression if we need them again. We call a formula removal strategy an *approximation strategy* when it removed formulas that have probability mass associated with them, resulting in *leakage*.

Definition 4.9 (Leakage)**.** *Assume a progression graph $\mathcal{G}_n(\phi)$. The leaked probability mass ℓ_n is defined as*

$$\ell_n = 1 - \sum_{v \in V} m_n(v). \tag{4.43}$$

Definition 4.10 (Removal strategy)**.** *A removal strategy is a function π from progression graphs $\mathcal{G}_n$ to progression graphs $\mathcal{G}'_n$ such that $|\mathcal{G}_n| \geq |\mathcal{G}'_n|$. A removal strategy is called exact iff it guarantees that $\ell'_n = 0$; otherwise it is called approximative.*

In the following, we consider some example removal strategies, both exact and approximate:

- **Default** π_{id}: The default strategy simply returns the same graph it was given. It is therefore exact, but does not do anything to reduce the size of the progression graph it is applied to. It thus serves as a base-case strategy for when no removals are performed.

- **Maximum time-to-live** π_{ttl}: The maximum time-to-live strategy utilises the age of formulas by considering 'stale' formulas for removal. A formula is 'stale' if it exceeds a predetermined `MAX_AGE` value. The strategy is exact since nodes receiving probability mass have their age reset to 0, and are thus never removed. Any formulas removed may be recomputed by applying the progression procedure to their direct neighbours.

- **Maximum size** π_{max}: The maximum size strategy is more aggressive than π_{ttl} because it puts a hard limit `MAX_NODES` on the size of the progression graph, and removes formulas until this hard limit is no longer violated. It prioritises first based on staleness, followed by low probability mass. This means that π_{max} will induce leakage if necessary, starting with formulas containing the least probability mass, making the strategy approximative.

The approximation strategies act as heuristics which try to balance formula removal against the overhead of reprogressing formulas and losing precision due to leaking probability mass. The choice of parameters impacts this balance in ways that are difficult to predict beforehand, as there exists an interaction with the incomplete state stream and formula being progressed.

4.5 Progression-based monitoring

Progression graphs have been shown to be suitable for progression tasks that involve incomplete state streams. However, they can also be adapted towards monitoring tasks, where for a wff ϕ we want to know for each time-point $i \in \mathbb{N}$ whether $\rho, i \models \phi$. This means that rather than a single (probabilistic) verdict, we receive such a verdict for every time-point. In the following, we therefore consider progression-based monitoring by 'stacking' progression graphs, adding a new graph to the stack for every time-point.

Definition 4.11 (Progression graph stack)**.** *A progression graph stack is a collection of progression graphs denoted by*

$$\mathbf{G}_n(\phi) = \left\{ \mathcal{G}_n^{(1)}(\phi), \mathcal{G}_{n-1}^{(2)}(\phi), \ldots, \mathcal{G}_1^{(n)}(\phi) \right\}, \tag{4.44}$$

where each $\mathcal{G}_m^{(i)}(\phi)$ represents a progression graph at iteration m path-checking whether $\rho_{\geq i} \models \phi$ holds for stream suffix $\rho_{\geq i}$.

Each progression graph in a progression graph stack $\mathbf{G}_n(\phi)$ has a pmf describing the probability of an incomplete state stream prefix being a model of ϕ. This means that per Theorem 4.4, for each time-point, a lower bound can be determined for verdicts $\top, \bot$. However, as the time $n \in \mathbb{N}$ increases, the stack becomes too large to store, since it approximates the size of the full stream. We can utilise the structure of wff ϕ to limit the size of our stack. Specifically, we make use of the concept of *future reach* introduced by Ho et al. (2014), which describes the maximum prefix length required for determining the truth value of a formula.

Definition 4.12 (Future reach adapted from Ho et al. (2014))**.** *The future reach for an MTL formula ϕ is determined by the function $FR(\phi)$, defined incrementally as:*

$$FR(p) = 0 \text{ for all } p \in \mathcal{P} \cup \{\top, \bot\} \tag{4.45}$$

$$FR(\neg\phi_1) = FR(\phi_1) \tag{4.46}$$

$$FR(\phi_1 \vee \phi_2) = \max(FR(\phi_1), FR(\phi_2)) \tag{4.47}$$

$$FR(\phi_1 \mathsf{U}_I \phi_2) = \sup(I) + \max(FR(\phi_1), FR(\phi_2)) \tag{4.48}$$

The future reach of a formula thus describes the temporal interval covered by that formula. By limiting the prefix length for progression a formula ϕ by its future reach $FR(\phi)$, one is guaranteed to obtain a resulting progression graph $\mathcal{G}_{FR(\phi)}(\phi)$ for which $m_{FR(\phi)}(\top) + m_{FR(\phi)}(\bot) = 1$. Unfortunately, this is not

Algorithm 4.3: Progression-based monitor

```
1  function PROGRESSMON(G(φ), ρ_n, Ω, τ_max):
2  G(φ)[n (mod |G(φ)|)] ← G_0(φ)
3  foreach i ∈ [n − |G(φ)| + 1, n] do
4      if G(φ)[n − i] ≠ nil then
5          (G, Ω) ← GRAPH-PROGRESS(G(φ)[n − i], ρ_n, Ω)
6          if (Pr(ρ_{i:n} ⊨ φ) + Pr(ρ_{i:n} ⊨ φ) ≥ τ_max) ∨ (i = n − |G(φ)| + 1) then
7              Report Pr(ρ_{i:n} ⊨ φ), Pr(ρ_{i:n} ⊭ φ) and ℓ_n
8              G(φ)[n − i] ← nil
9          else
10             G(φ)[n − i] ← G
11         end
12     end
13 end
14 return (G(φ), Ω)
```

enough for wffs ϕ containing unbounded temporal operators, for which $FR(\phi)$ approaches infinity. We can therefore further limit the prefix length to a predetermined constant `MAX_WINDOW`. This means that for those formulas ϕ for which $FR(\phi) >$ `MAX_WINDOW`, we will not have finished progressing all the possible progression traces, resulting in further leakage and thus corresponding to an approximation.

Algorithm 4.3 shows the progression-based monitoring procedure `PROGRESSMON`, where $\rho_{i:n}$ denotes a substream of ρ from time-point i to time-point n inclusive. The procedure takes an initially-empty progression stack $\mathbf{G}(\phi)$ implemented as an array, a cache Ω initially initialised for formulas in $SF(\phi)$, an incomplete state ρ_n, and an early termination threshold τ_{max}. At the start of each call, the procedure updates the stack in a round-robin fashion, overwriting the oldest progression graph with a new progression graph. After every call, it returns an updated progression stack $\mathbf{G}(\phi)$ and an updated cache Ω by applying `GRAPH-PROGRESS` to each of the progression graphs in the stack. The space complexity of `PROGRESSMON` is the same as the probability mass space complexity from `GRAPH-PROGRESS` multiplied by a constant window size, which was defined to be the smaller of $FR(\phi)$ and `MAX_WINDOW`. Since the graph can be shared between all progression graphs in the stack, no additional copies are required.

4.6 Empirical evaluation

Since the impact of the proposed removal strategies is difficult to predict, we performed an empirical evaluation to measure the impact of changing the parameters on both time and space requirements, as well as the impact of the approximative

MAX_AGE	MAX_NODES	Iterations	Max Size	Median Size	Avg Density
∞	∞	226,867	15,706	15,706	0.024
5	∞	226,867	11,851	1,162	0.243
1	∞	226,867	4,074	**335**	**0.665**
∞	250	226,863	3,858	3,726	0.099
5	250	226,863	3,855	1,163	0.254
1	250	226,863	3,722	**335**	**0.665**
∞	225	226,295	3,480	3,352	0.110
5	225	226,295	3,480	1,164	0.259
1	225	226,295	3,361	**335**	**0.665**
∞	200	225,644	3,105	2,978	0.124
5	200	225,644	3,105	1,165	0.266
1	200	225,644	2,999	**335**	**0.665**
∞	175	**222,599**	2,730	2,604	0.142
5	175	**222,599**	2,730	1,164	0.277
1	175	**222,599**	**2,653**	**335**	**0.665**

Table 4.1: Empirical results illustrating the impact of removal strategies π_{ttl} and π_{max}.

strategy on probability mass leakage. To do so, the graph progression procedure was implemented in Java[7].

Removal strategies

Table 4.1 shows an empirical evaluation of the impact of changing the removal strategy parameters for π_{ttl} and π_{max}; `MAX_AGE` and `MAX_NODES` respectively. We used the formula $\phi = \mathsf{G}\left(\neg p \rightarrow \left(\mathsf{F}_{[0,100]}\left(\mathsf{G}_{[0,10]}p\right)\right)\right)$ as a benchmarking formula, based on its membership in the class of *response* formulas — described by the pattern $\mathsf{G}_I\left(\phi \rightarrow \mathsf{F}_J\psi\right)$ — which is a formula class most commonly observed in runtime verification (Dwyer et al., 1999). For our choice of stream we randomly generated incomplete streams containing $\{\{p\}\}$ for 80% of the samples, and let the remaining samples be uniformly unknown, i.e. $\{\{p\}, \varnothing\}$. Our experiments terminated whenever 99% of the total probability mass was leaked or made its way into verdict nodes. Additionally, the values for `MAX_NODES` were chosen such that the amount of leaked probability mass would not exceed 1%; the impact on mass leakage was considered separately. The best results are marked in bold-face.

The exact strategy π_{ttl}, when configured with a low `MAX_AGE` value, results in a higher average density (i.e. inverse staleness) of progression graphs, corresponding to a high ratio of mass-bearing nodes relative to the total number of nodes. As mentioned previously, this does however result in an increased workload by requiring the regeneration of previously-deleted formulas. We can also observe a decrease in the maximum and medium progression graph size, determined by the cumulative size of all of the contained-within formulas, indicating that the graph shrinks as ex-

[7]The `jprogress` implementation is available at `https://github.com/dnleng/jprogress`.

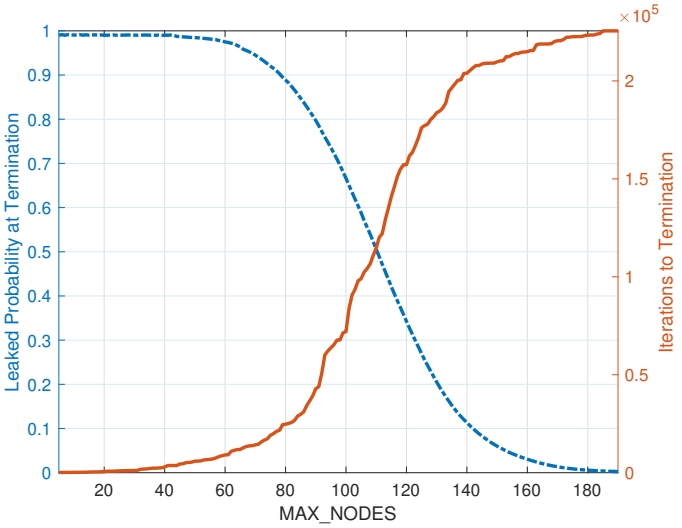

Figure 4.4: Leaked probability mass at termination (left), and number of iterations to termination (right).

pected. This is further affected by the choice of `MAX_NODES` from the π_{max} strategy. By decreasing the value of `MAX_NODES`, we can also observe that the number of iterations to termination is reduced, although this is partially due to the leakage of probability mass.

Leakage characteristics

A more detailed analysis of mass leakage is shown in Figure 4.4. On the left-hand side (corresponding to the blue dashed line), we see the leaked probability mass at termination, which occurs when 99% of probability mass has found its way into verdict nodes for 'false' and 'unknown'. As the probability of an 'unknown' verdict decreases, the probability of a $\perp$ verdict — here its inverse; not shown explicitly — increases. The switch-over, denoted by the 0.5 leaked probability mark, occurs between `MAX_NODES` $= 110$ and `MAX_NODES` $= 111$. When we decrease the value for `MAX_NODES`, the 'unknown' verdict dominates the 'false' verdict, and vice-versa. On the right-hand side of Figure 4.4 (the red line) we see the iterations to termination. As is common with phase transitions, the 'unknown' verdicts take far less time to compute than 'false' verdicts. We also observe that the iterations to termination increases at a faster rate than the 'false' verdict probability as it approaches 1. This is supported by the observations made in Table 4.1, which showed a correlation between progression graph size and time to termination.

The average time per iteration (in microseconds) is shown in Figure 4.5 for the same values of `MAX_NODES` as before. The shown times were obtained by running `jprogress` on a fourth-generation Intel Xeon E5-1650 CPU (6 cores, 12 threads)

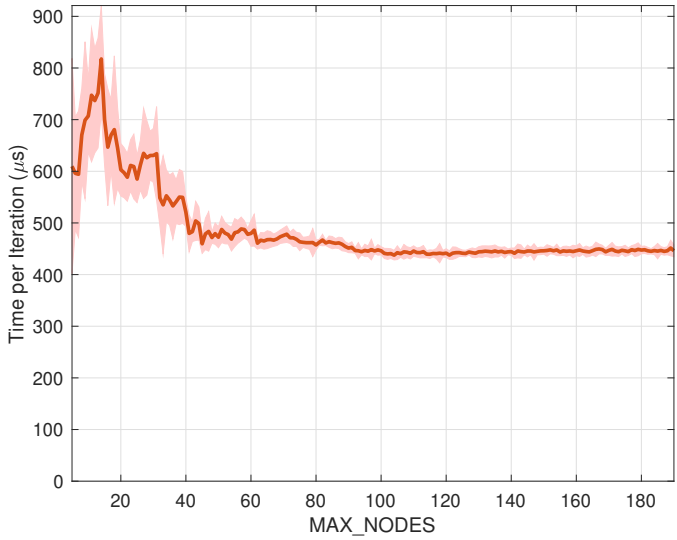

Figure 4.5: Average time per iteration $\pm 2\sigma$ (right).

with 50GiB of RAM allocated to the JVM. For lower values of `MAX_NODES`, the time per iteration is a bit unstable (i.e. yielding a high variance) due to the relatively low number of iterations involved, but this value stabilises as the value of `MAX_NODES` increases. This is because the first couple of iterations result in the construction of the progression graph up to the limit imposed by `MAX_NODES`, after which any extensions of the graph are followed by removals. Since the number of iterations needed until termination increases over time, as shown in Figure 4.4, the initial time penalty becomes increasingly less important for the average. We can also observe that the time needed to perform an iteration is in the order of microseconds. This indicates that this experimental setup can handle a high-frequency stream.

4.7 Summary

Path-checking is an important task for many safety-critical systems in which, given a path and a temporal logic formula, a procedure must determine whether the path satisfies the formula. Increasingly many safety-critical systems have to work with incomplete and uncertain information, for example due to having a physical operational environment shared with people. In this chapter, we therefore focused on developing a procedure that allows for path-checking of MITL formulas, which can be used to formally specify the desired behaviour of a system. The presented path-checking approach is novel in that it considers situations wherein state information may be incomplete, meaning that there may be a set of possible states, one of which is known to be the 'true' state. Furthermore, we support assigning probabilities to such hypothetical states. The combination of these two properties means

that we have to keep track of potentially many incrementally-available hypothetical streams. The presented approach makes use of and extends the progression procedure originally introduced by Bacchus and Kabanza (1998). We are able to limit the combinatoric explosion caused by incomplete state information by incrementally constructing progression graphs that contain probability mass. To do so, we make use of a formula simplification calculus which assists in the collapsing of potentially many state streams into the same formula. While a progression graph can in the worst case grow exponentially, the collapsing behaviour in combination with removal heuristics can be used to limit the growth. The configuration of the heuristics further allows for a trade-off between precision and speed; by choosing whether to abandon low-probability state streams in favour of keeping the progression graph small.

Chapter

5 Reasoning about space

LOGIC-BASED stream reasoning commonly makes use of temporal logics to express statements concerning the truth value of properties over time. Similar to temporal statements, many autonomous robotic systems can also benefit from or require the ability to make statements concerning spatial properties. This chapter presents MSTL, which is a spatio-temporal logic that combines MITL with qualitative spatial relations. We present and empirically evaluate techniques for determining the truth value of MSTL-statements. We call the applications of these techniques *reasoning about space*. This chapter includes and extends previously published material (Heintz and de Leng, 2014; de Leng and Heintz, 2016a) on spatio-temporal stream reasoning.

5.1 Introduction

Qualitative spatio-temporal reasoning is concerned with reasoning over time and space, in particular reasoning about spatial change (Cohn and Renz, 2008). This chapter presents a logic for spatio-temporal stream reasoning, alongside the tools required to incrementally evaluate spatio-temporal formulas in this logic. Furthermore, this chapter presents techniques that allow us to efficiently determine the truth value of such a formula. Combining spatial and temporal reasoning can be extremely useful in situations wherein one deals with for example physical objects, as it allows for the expression of spatial constraints that must hold over time. Consider the following example concerning a quad-rotor.

Example 5.1 (Containment in a virtual box)**.** *A quad-rotor is a small unmanned aerial vehicle that can be used in small spaces, for example indoors. In some cases, a quad-rotor may have to share space together with humans. Safety conditions could include restricting such a quad-rotor to a specific area of space, like a virtual box. An example*

statement combining spatial and temporal constraints is as follows: "It is always the case that if the UAV leaves the virtual box, it should be inside the virtual box within five seconds."

The constraints above are useful to detect situations where safety is compromised. A different example concerns itself with the detection of suspicious activity in order to prevent unsafe situations from occurring in the first place.

Example 5.2 (Perimeter monitoring)**.** *Consider a restricted area close to a public road. The area's perimeter is under surveillance by autonomous UAVs. A high-level task planner is responsible for detecting and tracking intrusions. An example rule could be expressed as: "If a moving object outside the perimeter stops moving for more than 60 seconds, dispatch a UAV to that object."*

In the above example, a type of spatio-temporal behaviour can be detected and responded to. Note that neither example deals with exact spatial coordinates. Rather, spatial entities are referenced by their spatial relations. We therefore focus on qualitative spatial relations when dealing with the spatial properties of objects.

5.2 Qualitative spatial reasoning

The *Region Connection Calculus* (RCC) was presented by Randell et al. (1992) as a calculus for topological reasoning over abstract regions based on their spatial relations. These regions are assumed to be composed of non-empty regions of topological space that can be characterised in terms of sets of points. The calculus defines and builds up spatial relations between regions from a primitive 'connected' relation $\mathrm{C}(x, y)$, which has the intended meaning that (non-empty) regions x and y share at least one point. Randell et al. (1992) recursively define a set of 15 RCC relations (including C) as shown in Table 5.1.

RCC-8 is a subset of RCC that is composed of eight jointly exhaustive and pairwise disjoint relations that allow us to describe the topological spatial relations between regions. Using composition-table based reasoning in RCC-8 (Cui et al., 1993), new spatial relations can be inferred from incomplete spatial knowledge. Figure 5.1 shows the eight qualitative relations that are considered by RCC-8 as well as their transitions. The transitions are interesting in situations where observations of a pair of regions yield non-adjacent spatial relations, because those intermediate and unobserved relations can then be inferred.

Example 5.3 (Busy student)**.** *Suppose that we have a spatial configuration in which we consider three regions* student, office, *and* canteen. *A robot observes that region* student *is strictly within region* office, *i.e.* $\mathrm{NTTP}(\mathrm{student}, \mathrm{office})$. *Further, the robot knows that region* canteen *is disconnected from region* office, *i.e.* $\mathrm{DC}(\mathrm{canteen}, \mathrm{office})$. *When asked whether the student is in the canteen, the robot cannot rely on direct observations. In fact, the robot might even consider it likely for a student to be in a canteen. By using the composition table for RCC-8, the robot can correctly deduce the unobserved spatial relation* $\mathrm{DC}(\mathrm{student}, \mathrm{canteen})$.

Definition		Description
$\mathsf{C}(x, y)$	$\equiv_{def} x \cap y \neq \varnothing$	Connected
$\mathsf{DC}(x, y)$	$\equiv_{def} \neg\mathsf{C}(x, y)$	Disconnected
$\mathsf{P}(x, y)$	$\equiv_{def} \forall z[\mathsf{C}(z, x) \rightarrow \mathsf{C}(z, y)]$	Part of
$\mathsf{PP}(x, y)$	$\equiv_{def} \mathsf{P}(x, y) \wedge \neg\mathsf{P}(y, x)$	Proper part
$\mathsf{EQ}(x, y)$	$\equiv_{def} \mathsf{P}(x, y) \wedge \mathsf{P}(y, x)$	Equals
$x = y$	$\equiv_{def} \mathsf{P}(x, y) \wedge \mathsf{P}(y, x)$	
$\mathsf{O}(x, y)$	$\equiv_{def} \exists z[\mathsf{P}(z, x) \wedge \mathsf{P}(z, y)]$	Overlapping
$\mathsf{PO}(x, y)$	$\equiv_{def} \mathsf{O}(x, y) \wedge \neg\mathsf{P}(x, y) \wedge \neg\mathsf{P}(y, x)$	Partially overlapping
$\mathsf{DR}(x, y)$	$\equiv_{def} \neg\mathsf{O}(x, y)$	Discrete from
$\mathsf{TPP}(x, y)$	$\equiv_{def} \mathsf{PP}(x, y) \wedge \exists z[\mathsf{EC}(z, x) \wedge \mathsf{EC}(z, y)]$	Tangential proper part
$\mathsf{EC}(x, y)$	$\equiv_{def} \mathsf{C}(x, y) \wedge \neg\mathsf{O}(x, y)$	Externally connected
$\mathsf{NTPP}(x, y)$	$\equiv_{def} \mathsf{PP}(x, y)$ $\wedge\neg\exists z[\mathsf{EC}(z, x) \wedge \mathsf{EC}(z, y)]$	Non-tangential proper part
$\mathsf{P}^{-1}(x, y)$	$\equiv_{def} \mathsf{P}(y, x)$	Inverse part of
$\mathsf{PP}^{-1}(x, y)$	$\equiv_{def} \mathsf{PP}(y, x)$	Inverse proper part
$\mathsf{TPP}^{-1}(x, y)$	$\equiv_{def} \mathsf{TPP}(y, x)$	Inverse tangential proper part
$\mathsf{NTPP}^{-1}(x, y)$	$\equiv_{def} \mathsf{NTPP}(y, x)$	Inverse non-tangential proper part

Table 5.1: Definitions for the 15 RCC relations.

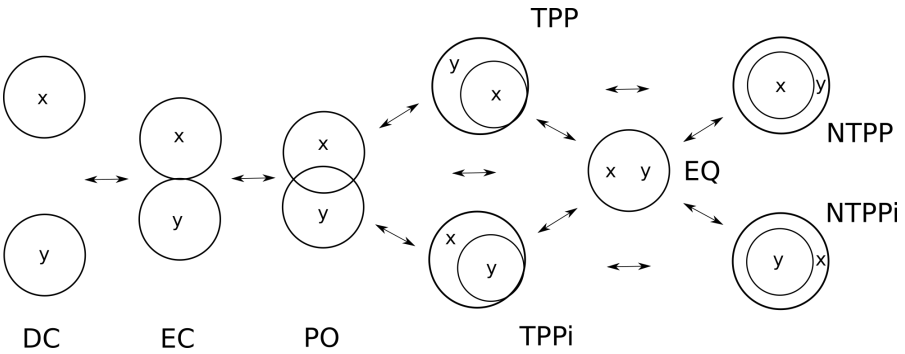

Figure 5.1: The eight qualitative spatial relations considered by RCC-8 and their transitions as illustrated by regions x and y.

In the above example, the observed spatial relations are used to infer unobserved facts about the world. This can be especially useful when there is a need for information that is not easily observable, or even unobservable.

5.3 Metric Spatio-Temporal Logic

Several qualitative spatio-temporal reasoning formalisms have been created by combining a spatial formalism with a temporal one. Examples are STCC (Gerevini and Nebel, 2002) and ARCC-8 (Bennett et al., 2002) which both combine RCC-8 with

Allen's Interval Algebra (Allen, 1983). The ST_i family[8] (Wolter and Zakharyaschev, 2000) of spatio-temporal logics represent a language for reasoning over spatio-temporal representations and offers such a temporalisation of RCC-8 using temporal operators. ST_i member language ST_0 makes use of the temporal operators 'it will always be the case' G, 'at some point in the future' F, and 'at the next time-point' X. Its extension ST_1 introduces spatio-temporal representations for spatial relations between two time-points through the 'next' operator, but does not attempt to provide reasoning techniques that handle such instantaneous observations. One problem is for example that ST_1 can refer to future states, which clearly causes difficulties when observations are assumed to be incremental over time. Furthermore, the ST_i family is a pure temporalisation of RCC-8 in the sense that it does not allow for expressing other (non-spatial) properties. This means that the domain of discourse exclusively treats its objects as spatial entities in relation to each other. A survey of other approaches that combine spatial and temporal reasoning techniques is provided by Kontchakov et al. (2007).

To make and evaluate statements about the spatial and temporal properties of objects, we introduce a logic called *Metric Spatio-Temporal Logic* (MSTL), which combines elements from MITL and RCC-8. MITL provides the ability to reason over propositions in time, but does not include a spatial formalism. We extend these languages by considering a finite domain composed of temporal objects that are spatial in nature. MSTL is thus similar to ST_1, which temporalises RCC-8 but restricts its language to spatial relations. Because MSTL is in part based on MITL, statements in MSTL can contain both spatial relations and predicates. Note that since we assume a finite, fixed domain of regions, these elements of MSTL are equivalent to propositions, with universal and existential quantifiers being short-hands for finite repetitions of conjunctions and disjunctions respectively.

Modelling intertemporal qualitative dynamics

Spatial relations are of the form $R(r_1, r_2)$ where R is any of

$$\left\{\mathsf{EC}, \mathsf{EQ}, \mathsf{DC}, \mathsf{PO}, \mathsf{NTTP}, \mathsf{TPP}, \mathsf{NTTP}^{-1}, \mathsf{TPP}^{-1}\right\} \tag{5.1}$$

and r_1, r_2 are spatial objects, also referred to as regions. We call this set $\mathcal{R}_8$ for brevity to indicate that its elements correspond to the RCC-8 relations 'externally connected', 'equals', 'disconnected', 'non-tangential proper part', 'tangential proper part', 'inverse non-tangential proper part' and 'inverse tangential proper part' respectively.

Definition 5.1 (MSTL syntax)**.** *Given an n-ary predicate P, binary spatial relation $\mathcal{R}_8$, variable or constant terms $\tau_1, \ldots, \tau_n$, and integers $i, j \in \mathbb{Z}$ the following statements are well-formed formulas (wffs) in MSTL:*

$$\mathcal{R}_8(X^i\tau_1, X^j\tau_2) \mid P(\tau_1, \ldots, \tau_n) \mid X^i\tau_1 = X^j\tau_2 \tag{5.2}$$

[8] For consistency reasons we use the same typesetting for all logics; the original literature — as well as the papers this chapter is based on — use a calligraphic version $\mathcal{ST}_i$ instead.

We will write τ for $X^0\tau$, $X\tau$ for $X^1\tau$, and $X^-\tau$ for $X^{-1}\tau$ as syntactic sugar. By recursion, for wffs ϕ and ψ and variable x the following statements are also wffs in MSTL:

$$\neg\phi \mid \phi \vee \psi \mid \phi \wedge \psi \mid \phi \rightarrow \psi \mid \forall x[\phi] \mid \exists x[\phi] \tag{5.3}$$

Finally, temporal notations are also defined by recursion for wff ϕ, natural numbers $n_1, n_2 \in \mathbb{N}$, and integers $i \in \mathbb{Z}$:

$$X^i\phi \mid G_{[n_1,n_2]}\phi \mid G\phi \mid F_{[n_1,n_2]}\phi \mid F\phi \mid \phi\ U_{[n_1,n_2]}\ \psi \tag{5.4}$$

Note that we apply the same syntactic sugar as for X over terms.

The syntax allows us to make complex spatio-temporal statements. Take for example the following statement, where informally G means 'it will always be the case', F means 'at some point in the future', and X means 'at the next time-point'. The spatial relation PO is contained in $\mathcal{R}_8$ and stands for 'partially overlapping'.

$$\begin{aligned}&\forall c_1[\forall c_2[c_1 \neq c_2 \wedge Car(c_1) \wedge Car(c_2) \rightarrow \\ &\quad (\mathsf{G}(\mathsf{PO}(\mathsf{X}c_1, c_2) \wedge Speeding(c_1) \rightarrow \mathsf{F}(\mathsf{PO}(c_1, c_2))))]]\end{aligned} \tag{5.5}$$

This wff has the intended meaning 'it is always the case that if a car is speeding and tails another car, they will eventually collide'.

Because we are interested in statements over space and time, we make use of *spatio-temporal models* for MSTL. It borrows the notion of a spatial assignment function from the *topological temporal model* (tt-model) from ST_i.

Definition 5.2 (Spatio-temporal model)**.** *A spatio-temporal model is a tuple of the form $\mathcal{M} = \langle T, <, U, \mathcal{D}, I, \alpha\rangle$, where T represents a set of time-points, $<$ represents an ordering over T, U represents the non-empty universe of the space as a set of points, and $\mathcal{D} = \langle \mathcal{P}, \mathcal{R}\rangle$ represents the domain consisting of predicates $\mathcal{P}$ and spatial objects $\mathcal{R}$. An interpretation $I^t \in I$ maps predicates and constant terms to $\mathcal{P}$ and $\mathcal{R}$ respectively for every time-point in T. For constant terms this mapping will be the same for all t, but for predicates this is not necessarily the case. A spatial assignment function α associates at every time-point in T every spatial object label in $\mathcal{R}$ to a subset of U. It is extended to interpret 'next' as $\alpha(X^i r, t) = \alpha(X^{i-j} r, t + j)$ for spatial object label $r \in \mathcal{R}$ and integers $i, j \in \mathbb{Z}$.*

From this definition it is clear that we are only considering objects that have some spatial properties associated with them, expressed in the form of spatial relations. Spatial objects therefore are also commonly called *regions* when we only focus on temporal and spatial properties.

Definition 5.3 (MSTL semantics)**.** *The MSTL statement that a spatio-temporal formula ϕ holds in $\mathcal{M} = \langle T, <, U, \mathcal{D}, I, \alpha\rangle$ at time-point $t \in T$ is defined recursively*

for integers $i, j \in \mathbb{Z}$.

$$\mathcal{M}, t \models P(\tau_1, \ldots, \tau_n) \text{ iff } \left\langle I^t(\tau_1), \ldots, I^t(\tau_n) \right\rangle \in I^t(P) \tag{5.6}$$

$$\mathcal{M}, t \models \forall x[\phi] \text{ iff } \forall r \in \mathcal{R} : \mathcal{M}, t \models \phi[x/r] \tag{5.7}$$

$$\mathcal{M}, t \models \exists x[\phi] \text{ iff } \exists r \in \mathcal{R} : \mathcal{M}, t \models \phi[x/r] \tag{5.8}$$

$$\mathcal{M}, t \models \neg\phi \text{ iff } \mathcal{M}, t \not\models \phi \tag{5.9}$$

$$\mathcal{M}, t \models \phi \vee \psi \text{ iff } \mathcal{M}, t \models \phi \text{ or } \mathcal{M}, t \models \psi \tag{5.10}$$

$$\mathcal{M}, t \models \phi \, \mathsf{U}_{[t_1,t_2]} \, \psi \text{ iff } \exists t' \in [t + t_1, t + t_2] : \mathcal{M}, t' \models \psi \text{ and } \forall t'' \in [t, t') : \mathcal{M}, t'' \models \phi \tag{5.11}$$

$$\mathcal{M}, t \models \mathsf{X}^i\phi \text{ iff } \mathcal{M}, t + i \models \phi \tag{5.12}$$

$$\mathcal{M}, t \models \mathsf{C}(\mathsf{X}^i r_1, \mathsf{X}^j r_2) \text{ iff } \alpha(r_1, t + i) \cap \alpha(r_2, t + j) \neq \emptyset \tag{5.13}$$

From the RCC 'connected' spatial relation C, the usual semantics of all RCC-8 relations can be recursively defined, but here they are left out for the sake of brevity.

Allowing for the 'next' operator to be invoked over region variables is a powerful extension that makes it possible to refer to a particular region at the next time-point, or by recursive application any past or future time-point.

5.4 Spatio-temporal inference with RCC-8

RCC-8 allows for both representation of observed spatial relations as well as the inference of unobserved spatial relations. However, these observations are usually assumed to be restricted to a single time-point rather than across different time-points. To represent spatial relations across time-points, we can add a temporal element. The addition of a 'next' operator X as initially proposed by ST_1 can lead to situations wherein regions at different time-points are considered. In what follows, we explore the consequences to spatio-temporal inference when the 'next' operator is used to describe relations across time-points, starting with the representation of these relations.

Temporal constraint networks

While the 'next' operator allows for powerful representations, it complicates evaluation of those statements when we consider observations of the world to occur within rather than across time-points. Spatial relations for regions can be partially observed at time-point t and at time-point $t+1$ independently, but no observations can be made with regards to the spatial relations between regions at time-point t and regions at time-point $t + 1$. To better illustrate how these concepts relate, we introduce the spatial relation matrix as a representation of constraint networks.

Definition 5.4 (Spatial relation matrix)**.** *Given a spatio-temporal model $\mathcal{M}$, a spatial relation matrix is an $n \times n$ matrix M^t for time-point $t \in T$ where n denotes the total*

number of region variables $|\mathcal{R}|$. For every matrix element $M^t_{i,j}$ and region variables $r_i, r_j \in \mathcal{R}$ we have $M^t_{i,j} = (r_i R r_j)$ such that $R \subseteq \mathcal{R}_8$ and $R \neq \emptyset$. The semantics of M^t are then as follows.

$$M^t_{i,j} = (r_i R r_j) \text{ iff } \mathcal{M}, t \models \bigvee_{R_k \in R} R_k(r_i, r_j) \tag{5.14}$$

The spatial relation matrix allows us to intuitively represent spatial facts about regions and corresponds to a complete RCC-8 network. The main diagonal always consists of the singleton $\{\mathsf{EQ}\}$. Further, the matrix is semi-symmetric; symmetry holds for all relations except for NTTP and TPP, which have inverses NTTP^{-1} and TPP^{-1} respectively. Existing general solvers for qualitative CSPs can be used to determine the algebraic closure of spatial relation matrices, i.e. given spatial relation matrix M^t, the algebraic closure $AC(M^t)$ yields a spatial relation matrix N^t such that for every corresponding set of spatial relations $N^t_{i,j} \subseteq M^t_{i,j} \subseteq \mathcal{R}_8$. A small example of a spatial relation matrix for regions r_1, r_2, r_3 at time-point t with partial knowledge is shown below.

$$M^t = \begin{bmatrix} \{\mathsf{EQ}\} & \{\mathsf{NTTP}^{-1}\} & \{\mathsf{PO}, \mathsf{EC}\} \\ \{\mathsf{NTTP}\} & \{\mathsf{EQ}\} & \{\mathsf{DC}\} \\ \{\mathsf{PO}, \mathsf{EC}\} & \{\mathsf{DC}\} & \{\mathsf{EQ}\} \end{bmatrix} \tag{5.15}$$

Region r_2 is inside of region r_1 but disconnected from region r_3, and region r_1 is partially overlapping or externally connected with region r_3.

A spatial relation matrix can be extended to describe relations between multiple time-points. This is a useful property because it allows us to describe relations between regions at different time-points that are not necessarily consecutive.

Definition 5.5 (Intertemporal spatial relation matrix)**.** *An intertemporal spatial relation matrix M^{t_1,t_2} is a spatial relation matrix describing the spatial relations between regions $r_i, r_j \in \mathcal{R}$ such that we relate r_i at time-point t_1 to r_j at time-point t_2, i.e. relating $\alpha(r_i, t_1)$ to $\alpha(r_j, t_2)$.*

A spatial relation matrix M^t from Definition 5.4 is then equivalent to an intertemporal spatial relation matrix $M^{t,t}$. Intertemporal spatial relations can thus be represented by an intertemporal spatial relation matrix. For the 'next' operator, this would for example be $M^{t,t+1}$. However, we assume that these relations are unobservable and must somehow be inferred from our observations at time-points t and $t+1$, represented by M^t and M^{t+1}.

By combining the four different combinations for intertemporal spatial relation matrices over two time-points t_1 and t_2, we can concisely describe in one matrix the relations between regions at single time-points as well as the relations between those regions at different time-points. This corresponds to an RCC-8 network in which every region is contained twice, i.e. once for every time-point.

Definition 5.6 (Extended spatial relation matrix)**.** *An extended spatial relation matrix* $M^{t_1 \cup t_2}$ *for* $t_1 < t_2$ *combines four intertemporal spatial relation matrices as follows:*

$$M^{t_1 \cup t_2} = \begin{bmatrix} M^{t_1,t_1} & M^{t_1,t_2} \\ M^{t_2,t_1} & M^{t_2,t_2} \end{bmatrix} \tag{5.16}$$

In general, spatial relation matrices can be used to represent uncertainty for spatial relations between regions by using non-singleton sets. This is important because often we can not deduce that a single relation must hold. We can use extended spatial relation matrices to talk about the spatial relations both within individual time-points and between time-points. This makes them a suitable representation tool for intertemporal RCC-8 networks when considering the problem of deducing unobservable intertemporal relations.

Intratemporal inference

Intratemporal inference with RCC-8 assumes that all spatial relations are observed within the same time-point, i.e. $M^{t,t}$ for some time-point t. In this case, $M^{t,t}$ represents a constraint network for a single time-point, for which it may be possible to reduce the uncertainty of spatial relations between regions based on the observed spatial relations between other regions. It is possible to apply composition table based reasoning for RCC-8 to this effect. A composition table presumes regions i, j, and k such that the spatial relations for (i, j) and (j, k) are knowns, and presents the possible spatial relations that may exist between regions (i, k).

Gantner et al. (2008) present the *Generic Qualitative Solver* (GQR) which can be used to perform qualitative reasoning on a number of calculi, including RCC-8. They make use of the path consistency algorithm shown in Algorithm 5.1, based on the path consistency algorithm by Mackworth (1977). The algorithm takes a constraint network and produces a refined constraint network in $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space. Path consistency continuously updates spatial relation C_{ik} by computing $C_{ik} \cap (C_{ij} \circ C_{jk})$, utilising a third variable j. These updates can be performed based on a composition table.

Intertemporal inference

Sometimes we want to talk about spatial relations between regions at different time-points. By following the example of ST_1, we can extend our definition of region symbols accordingly. If 'region' is a region symbol, then '$\mathsf{X}(\text{region})$' is also a region symbol, such that $\alpha(\mathsf{X}(\text{region}, t)) = \alpha(\text{region}, t + 1)$. This allows us to refer to regions at different time-points using the same region symbol 'region'. However, this also complicates the semantics of the mapping α. From its definition, it is clear that we are referring to the same universe of points, but it is not clear whether $\alpha(x, t) = \alpha(x, t + 1)$ for all time-point t, or whether it is possible that $\alpha(x, t) \neq \alpha(x, t + 1)$ for some time-point t. In this dissertation we will assume space itself to be rigid.

Algorithm 5.1: Path consistency (Gantner et al., 2008)

function PATH-CONSISTENCY((V, C)):
  $Q \leftarrow \{(i, j) \mid 1 \leq i < j \leq n\}$
  while *Q is not empty* **do**
    select and delete an (i, j) from Q
    for $k \leftarrow 2$ **to** $n, k \neq i \wedge k \neq j$ **do**
      $t \leftarrow C_{ik} \cap (C_{ij} \circ C_{jk})$
      if $t \neq C_{ik}$ **then**
        $C_{ik} \leftarrow t$
        $C_{ki} \leftarrow t^{\smile}$
        $Q \leftarrow Q \cup \{(i, j)\}$
      end
      $t \leftarrow C_{kj} \cap (C_{ki} \circ C_{ij})$
      if $t \neq C_{kj}$ **then**
        $C_{kj} \leftarrow t$
        $C_{jk} \leftarrow t^{\smile}$
        $Q \leftarrow Q \cup \{(k, j)\}$
      end
    end
  end
  return (V, C)

Definition 5.7 (Rigid space assumption)**.** *The rigid space assumption assumes that space itself is fixed across time, i.e.*

$$\exists t \in T[x \in \mathcal{R} \rightarrow \alpha(x, t) \neq \alpha(x, t + 1) \tag{5.17}$$

for time T, regions $\mathcal{R}$, and spatial assignment function α.

Reasoning alone thus does not allow us to say anything about intertemporal relations, represented by M^{t_1,t_2} and M^{t_2,t_1} in extended spatial relation matrices. These relations cannot be observed, nor can they be inferred from individual time-points. Concretely, observations are limited to M^{t_1,t_1} and M^{t_2,t_2}. This may seem counter-intuitive, but this is because humans often assume a *frame of reference* when observing spatial changes over time. One way around this problem is therefore to make assumptions about some or all intertemporal relations represented by M^{t_1,t_2} and M^{t_2,t_1} in order to establish such a frame of reference. Effectively this corresponds to 'pegging' only these landmark regions to the space they occupy, allowing outside space to warp relative to the landmarks and fixing the frame of reference. The set of landmarks is indicated by $\mathcal{LM} \subseteq \mathcal{R}$. For all landmarks $x \in \mathcal{LM}$, the α-mapping is fixed such that $\alpha(x, t) = \alpha(x, t + 1)$ for all $t \in T$. By using a consistent set of landmarks, it is possible to infer intertemporal relations based on the spatial relations between non-landmark and landmark regions. Additionally, since the landmark regions are rigid, the spatial relations between landmark regions do not change.

Definition 5.8 (Landmark)**.** *A landmark given a set of region variables $\mathcal{R}$ over any two time-points $t, t+1$ is a region variable $r \in \mathcal{R}$ that is rigid between t and $t+1$, i.e.* $\mathsf{EQ}(r, Xr)$*. The set of landmarks is indicated by $\mathcal{LM} \subseteq \mathcal{R}$ such that $r \in \mathcal{LM}$ implies that landmark r is rigid.*

Example real-world landmark candidates are e.g. buildings, lakes, monuments, trees, and roads. These physical entities are unlikely to change during the run-time of a system, and therefore provide a reasonable frame of reference. An immediate effect of landmarks being rigid is that their relations to other landmark regions remain unchanged. Effectively, the set of landmarks $\mathcal{LM}$ provides a possible frame of reference with respect to which relations may change over time. Since this affects the truth semantics of statements in MSTL, we introduce a landmark extension to the spatio-temporal model to capture this.

Definition 5.9 (Landmark-based spatio-temporal model)**.** *A landmark-based spatio-temporal model is a spatio-temporal model*

$$\mathcal{M}_{\mathcal{LM}} = \langle T, <, U, \mathcal{D}, I, \alpha_{\mathcal{LM}} \rangle \tag{5.18}$$

and $\mathcal{LM} \subseteq \mathcal{R}$ represents the landmark set. $\mathcal{LM}$ then restricts α such that for all time-points $t \in T$ and all landmark regions $r \in \mathcal{LM}$ it is the case that $\alpha(r,t) = \alpha(r,t+1)$.

Landmarks may introduce inconsistencies if we make observations that conflict with the landmark-imposed restriction of α. To illustrate how this might happen, consider an example where at time-point t we make the observation $\mathsf{PO}(r_1, r_2)$, and at time-point $t+1$ we make the observation $\mathsf{DC}(r_1, r_2)$. If we only consider the individual time-points, there is no problem. The following extended spatial relation matrix illustrates our ignorance of the intertemporal spatial relations M^{t_1,t_2} and M^{t_2,t_1}.

$$M^{t_1 \cup t_2} = \begin{bmatrix} \{\mathsf{EQ}\} & \{\mathsf{PO}\} & \mathcal{R}_8 & \mathcal{R}_8 \\ \{\mathsf{PO}\} & \{\mathsf{EQ}\} & \mathcal{R}_8 & \mathcal{R}_8 \\ \mathcal{R}_8 & \mathcal{R}_8 & \{\mathsf{EQ}\} & \{\mathsf{DC}\} \\ \mathcal{R}_8 & \mathcal{R}_8 & \{\mathsf{DC}\} & \{\mathsf{EQ}\} \end{bmatrix} \tag{5.19}$$

However, if we use landmarks, the choice of $\mathcal{LM}$ results in an assumption about some intertemporal relations. Choosing $\mathcal{LM} = \{r_1, r_2\}$ is inconsistent, because it implies that regions r_1 and r_2 need to be partially overlapping and disconnected at the same time, which is a contradiction. Instead picking $\mathcal{LM} = \{r_1\}$ is consistent, and one could imagine region r_2 'moving away from' region r_1. Naturally, the converse holds as well if we pick region r_2 as our frame of reference.

We can show that consistency is guaranteed if only one landmark is chosen, and the above example shows that this does not always hold for the case of $|\mathcal{LM}| \geq 2$. Picking a single landmark corresponds to the case of adding a single connection between two disconnected RCC-8 networks for different time-points. To further illustrate the impact of the choice of $\mathcal{LM}$, consider again the scenario above and

suppose we wish to evaluate the formula $\mathsf{G}(\mathsf{EQ}(r_1, \mathsf{X}r_1))$ at time-point t. Choosing $\mathcal{LM} = \{r_1\}$ means this formula will evaluate to True, i.e.

$$\mathcal{M}_{\{r_1\}}, t \models \mathsf{G}(\mathsf{EQ}(r_1, \mathsf{X}r_1)). \tag{5.20}$$

Choosing $\mathcal{LM} = \{r_2\}$ means this formula will evaluate to False, i.e.

$$\mathcal{M}_{\{r_2\}}, t \not\models \mathsf{G}(\mathsf{EQ}(r_1, \mathsf{X}r_1)). \tag{5.21}$$

Choosing any other consistent $\mathcal{LM}$ we can only conclude

$$\mathcal{M}_{\mathcal{LM}}, t \models \mathsf{G}(\mathsf{EQ}(r_1, \mathsf{X}r_1)) \vee \neg(\mathsf{G}(\mathsf{EQ}(r_1, \mathsf{X}r_1))); \tag{5.22}$$

we cannot say for certain which one is true. This is specifically caused by the choice of landmark in combination with the observations at the two time-points. The following two statements then hold for the same two observations described earlier:

$$\mathcal{M}_{\{r_1\}}, t \models \mathsf{G}(\mathsf{EQ}(r_1, \mathsf{X}r_1)) \wedge \neg\mathsf{G}(\mathsf{EQ}(r_2, \mathsf{X}r_2)) \tag{5.23}$$

$$\mathcal{M}_{\{r_2\}}, t \models \mathsf{G}(\mathsf{EQ}(r_2, \mathsf{X}r_2)) \wedge \neg\mathsf{G}(\mathsf{EQ}(r_1, \mathsf{X}r_1)) \tag{5.24}$$

This clearly shows how landmark choice shapes the frame of reference within which MSTL statements may hold.

5.5 MSTL progression

In stream reasoning, information is assumed to become incrementally available. Recall that progression is a technique for evaluating temporal logic formulas where we try to determine the truth value of the formula based on the information received thus far. This makes it possible to sometimes determine the truth value for an MSTL formula without having to wait for the entire stream to arrive. However, MSTL progression differs from MITL progression in that MSTL formulas can combine information from multiple time-points due to the introduction of the 'next' operator over region terms. This would first require such terms to be rewritten such that they refer to past regions, after which progression has to potentially take into account multiple states if intertemporal spatial relations are used. The former can be achieved by adding additional rewriting rules that extract the 'next' operator, whereas the latter can be achieved by using landmarks and multiple hypotheses as introduced earlier.

Rewriting rules for 'next'

In order for progression to be applicable to MSTL, some changes are needed to deal with the spatial relations. In particular, the application of temporal operators to spatial objects needs to be handled before progression can operate on the propositions in a wff.

By combining temporal with spatial reasoning, we effectively need both temporal and spatial evaluation methods. For every step in the progression, spatial reasoning is performed within that step. This however does not include spatial reasoning

between different time-points. Therefore, progression needs to be extended to handle intertemporal relations that are the result of the 'next' operator in MSTL. This gives rise to additional rewriting rules based on occurrences of the 'next' operator.

Progressing the 'next' operator when it occurs in front of wffs in MSTL corresponds to rewriting that formula by removing the operator, i.e. during progression $\mathsf{X}\phi$ is rewritten to ϕ for wff ϕ. The following proofs show equivalences for occurrence of 'next' excluding intertemporal relations, and make use of the semantics presented in Definition 5.3.

Proposition 5.1 (Next and negation)**.** *'Not at the next time-point' is equivalent to 'it is not the case at the next time-point', i.e.*

$$\models \neg \mathsf{X} R(x, y) \leftrightarrow \mathsf{X} \neg R(x, y). \tag{5.25}$$

Proof. Decomposing bi-implication into cases:

($\Rightarrow$) Assume $\mathcal{M}, t \models \neg \mathsf{X} R(x, y)$ holds for some arbitrary $\mathcal{M}$ and t. From the semantics of negation this means $\mathcal{M}, t \not\models \mathsf{X} R(x, y)$. According to the semantics of X, this is equivalent to $\mathcal{M}, t + 1 \not\models R(x, y)$, thus
$\mathcal{M}, t + 1 \models \neg R(x, y)$. Reintroducing X then yields $\mathcal{M}, t \models \mathsf{X} \neg R(x, y)$.

($\Leftarrow$) Analogous to the above in reverse order. ■

Proposition 5.2 (Next and always)**.** *The 'next' operator can be integrated into the interval of an 'always' operator, i.e.*

$$\models \mathsf{G}_{[t_1, t_2]} \mathsf{X} R(x, y) \leftrightarrow \mathsf{G}_{[t_1+1, t_2+1]} R(x, y). \tag{5.26}$$

Proof. Decomposing bi-implication into cases:

($\Rightarrow$) Assume $\mathcal{M}, t \models \mathsf{G}_{[t_1, t_2]} \mathsf{X} R(x, y)$ holds for some arbitrary $\mathcal{M}$ and t. From the semantics of G, this means $\forall t_1 \leq t' \leq t_2 : \mathcal{M}, t' \models \mathsf{X} R(x, y)$ holds. By definition of X, for every t' we get $\mathcal{M}, t' + 1 \models R(x, y)$. Reintroducing the universal quantifier, we get $\forall t_1 + 1 \leq t' \leq t_2 + 1 : \mathcal{M}, t' \models R(x, y)$. Reintroducing G, this yields $\mathcal{M}, t' \models \mathsf{G}_{[t_1+1, t_2+1]} R(x, y)$.

($\Leftarrow$) Analogous to the above in reverse order. ■

Proposition 5.3 (Next and eventually)**.** *The 'next' operator can be integrated into the interval of an 'eventually' operator, i.e.*

$$\models \mathsf{F}_{[t_1, t_2]} \mathsf{X} R(x, y) \leftrightarrow \mathsf{F}_{[t_1+1, t_2+1]} R(x, y). \tag{5.27}$$

Proof. Analogous to the proof of Proposition 5.2, replacing quantifiers $\forall$ and temporal operators G by $\exists$ and F respectively. ■

The 'next' operator can also occur inside intertemporal relations $R(x, \mathsf{X}y)$. In this case, it is not possible to evaluate $R(x, \mathsf{X}y)$ at the current time-point, because the relation depends on a future state of y. To work around this problem, we make use of the 'previous' operator X^{-}, which is the inverse of the 'next' operator. The following proofs show equivalences for 'next' involving intertemporal relations, and make use of the 'previous' operator.

Proposition 5.4 (Extract next)**.** *The 'next' operator can be extracted from terms, i.e.*

$$\models \mathsf{X}R(x, y) \leftrightarrow R(\mathsf{X}x, \mathsf{X}y). \tag{5.28}$$

Proof. Decomposing bi-implication into cases:

($\Rightarrow$) Assume $\mathcal{M}, t \models \mathsf{X}R(x, y)$ holds for some arbitrary $\mathcal{M}$ and t. From the semantics of X, this means $\mathcal{M}, t + 1 \models R(x, y)$. Further, we have $\alpha(z, t + 1) = \alpha(\mathsf{X}z, t)$ for any region z, so we get $\mathcal{M}, t \models R(\mathsf{X}x, \mathsf{X}y)$.

($\Leftarrow$) Analogous to the above in reverse order. ■

Proposition 5.5 (Partially extract next)**.** *The 'next' operator can be subtracted from terms, i.e.*

$$\models R(x, \mathsf{X}y) \leftrightarrow \mathsf{X}R(\mathsf{X}^{-}x, y). \tag{5.29}$$

Proof. Decomposing bi-implication into cases:

($\Rightarrow$) Assume $\mathcal{M}, t \models R(x, \mathsf{X}y)$ holds for some arbitrary $\mathcal{M}$ and t. From the semantics of X over regions, we have $\alpha(z, t) = \alpha(\mathsf{X}^{-}z, t + 1)$ and $\alpha(\mathsf{X}z, t) = \alpha(z, t + 1)$ for any region z. Therefore this is equivalent to

$$\mathcal{M}, t + 1 \models R(\mathsf{X}^{-}x, y) \tag{5.30}$$

when applied to regions x and y respectively. Introducing X then yields

$$\mathcal{M}, t \models \mathsf{X}R(\mathsf{X}^{-}x, y). \tag{5.31}$$

($\Leftarrow$) Analogous to the above in reverse order. ■

The ability to rewrite MSTL formulas such that occurrences of 'next' over regions are either removed or replaced by 'previous' is vital for stream reasoning, because it allows for the delayed evaluation of formulas so that, at the time of evaluation, they only refer to the current and previous state(s) of the world. This makes the earlier-presented landmark approach applicable in a stream reasoning context.

Algorithm 5.2: Progression adapted for MSTL

1 **function** PROGRESS-MSTL(ϕ, I^t, $\{M_0, \ldots, M_{N-1}\}$, $\mathcal{LM}$, $\mathcal{D}$, $\mathcal{G}$, Ω):
2 expand quantifiers in ϕ using domain $\mathcal{D}$
3 simplify ϕ using SIMPLIFY and Propositions 5.1–5.5 to extract 'next'
4 compute AC for combinations of M_i and $\mathcal{LM}$ used by ϕ
5 apply the AC to I^t to obtain a set of m hypotheses $\rho = \left\{I^t_0, \ldots, I^t_{m-1}\right\}$
6 **return** GRAPH-PROGRESS($\mathcal{G}$, ρ, Ω)

MSTL-adapted progression with incomplete information

Recall that the simplified progression procedure, as presented in Algorithm 3.1, does not handle multiple hypotheses, while we know that composition table-based RCC-8 reasoning can yield multiple such hypotheses corresponding to different consistent spatial configurations. This is due to the topological space, as modelled through the α mapping, not being directly available in practice. In Chapter 6, we consider the state stream synthesis needed to generate sets of states containing RCC-8 relations. This provides us with a realisation of the partial knowledge about the topological space. Given a stream containing sets of states pertaining to the RCC-8 relations, we can then apply progression under uncertainty as presented in Chapter 4. This does however require changes to accommodate the usage of relations and predicates with a finite domain of spatial objects.

The MSTL-adapted progression procedure PROGRESS-MSTL is shown in Algorithm 5.2. It takes a wff ϕ, a state I^t, spatial relation matrices M_i, a set of landmarks $\mathcal{LM}$, a domain $\mathcal{D}$ containing spatial objects, a progression graph $\mathcal{G}$, and a cache Ω. Note that the value of N depends on the range of the intertemporal spatial relations occurring in ϕ. It then expands the universal quantifiers in ϕ into a conjunction over all spatial objects in $\mathcal{D}$, and the existential quantifiers similarly into a disjunction. It then simplifies the formula by using the simplification procedure in Algorithm 3.2 together with the extraction rules of 'next'. Once this is done, it computes the algebraic closure for the combinations of M_i used in ϕ, using a CSP solver. This produces a set of possible spatial configurations, which are joined with I^t to produce a set of m hypothetical states ρ. These hypotheses are then fed to the progression graph as usual, using GRAPH-PROGRESS listed in Algorithm 4.2.

5.6 Empirical evaluation

Thus far we introduced a logic for spatio-temporal stream reasoning and a number of methods for the evaluation of formulas in that logic. In the following, we provide experimental results measuring the impact of separating static and dynamic components in RCC-8 scenarios, and the impact of landmarks on the disjunction size after applying an algebraic closure to intertemporal RCC-8 scenarios.

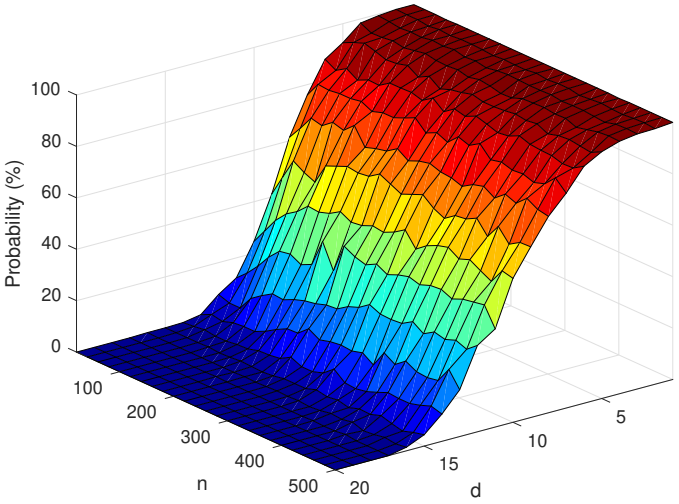

Figure 5.2: The probability of satisfiability of CSPs drawn from $A(n, d, 4.0) = A'(n, d, 4.0, 1.0)$ for varying numbers of regions n and varying degrees d. A phase transition can be observed to occur for $d \in [5, 15]$.

Caching spatial relations between rigid objects

The spatial reasoning is mainly dependent on the number of variables, the number of constraints (degree) and the label size (Renz and Nebel, 2001). In the experiments we try to estimate the function $A'(n, d, l, r)$ by measuring the execution time on instances with the number of variables n, degree d, label size l and ratio of dynamic variables r. The number of variables can be divided in a dynamic part $v_d = r \times v$ and a static part $v_s = v - v_d$. The expected degree is the expected number of relations from a given dynamic variable to other variables. The expected label size is the expected size of the disjunction of RCC-8 relations for a given relation between a dynamic variable and some other variable.

Using basically the same method as Renz and Nebel (2001) we evaluate the effect of precomputing the algebraic closure of the static variables, compared to computing the whole algebraic closure for each time-step. In accordance with the methodology proposed by Renz and Nebel (2001), problems are randomly generated for different values for n and d, with the label size fixed to $l = 4.0$. First, $nd/2$ edges are selected out of the $n(n-1)/2$ possible edges, using a uniform distribution. For these edges, one RCC-8 relation is assigned at random, and the remaining relations are added with a probability of $(l-1)/7$ for each such relation. For the remaining edges, the universal relation is assigned. Generating problems with the help of this methodology results in the satisfiability graph shown in Figure 5.2, which also shows the phase transition identified by Renz and Nebel (2001) by averaging over 500 runs. The phase-transition occurs where the majority of problem instances flip from being satisfiable to being unsatisfiable.

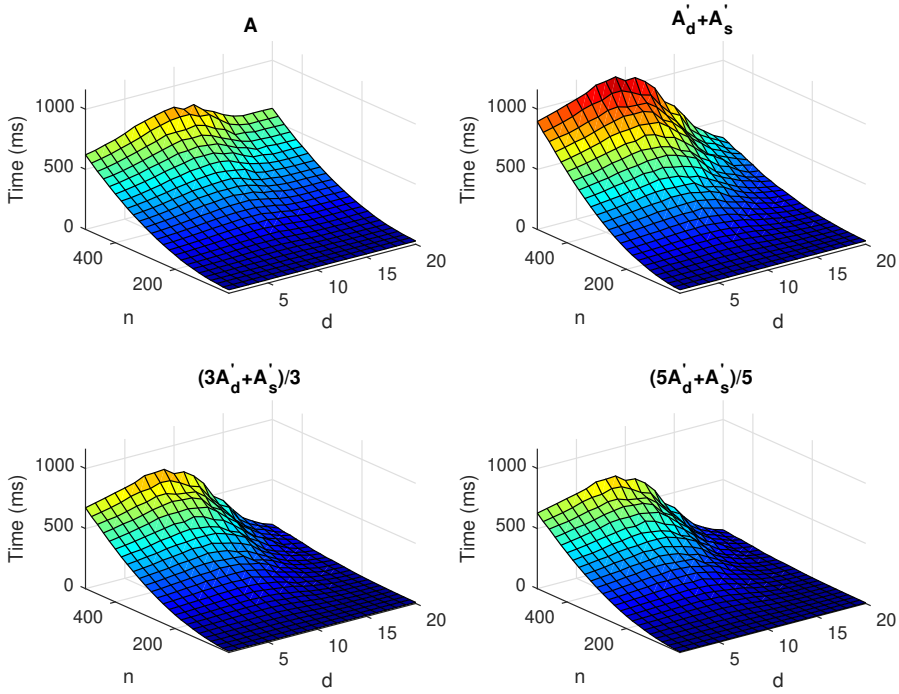

Figure 5.3: Average time per iteration in milliseconds for four different cases. The top left shows the average time in milliseconds for $A(n, d, 4.0)$. The top right shows an increased cost after one iteration when separating the dynamic component $A'_d(n, d, 4.0, 0.25)$ from the static component $A'_s(n, d, 4.0, 0.25)$. The bottom row shows how the one-time overhead imposed by computing the static and dynamic components separately decreases, for three (bottom left) and five (bottom right) iterations respectively.

Using this methodology, we can separate static regions from dynamic regions and precompute the algebraic closure for the static component before considering dynamic regions. The mean performance results of the former are denoted by $A(v, E(deg), 4.0)$. For the mean performance results of the dynamic component of the latter, the notation $A'_d(n, d, 4.0, r)$ is used. The performance experiments used values of d ranging from 1 to 20 with step size 1, and values of n ranging from 20 to 500 with step size 20. The value of r was chosen to be constant, $r = 0.25$; similar results are obtained for different values of r, which are characterised by the moving of the phase transition from Figure 5.2. For each combination we took the average over 100 runs. The average metric was chosen to account for the difference in distribution between the satisfiable and unsatisfiable problem instances.

Figure 5.3 shows the impact on runtime for computing the algebraic closure of the generated CSPs by averaging over 500 runs. The top left graph shows the runtime in milliseconds for estimating $A(n, d, 4.0) = A'(n, d, 4.0, 1.0)$. This is the base

case without separating out a static component, hence it is equivalent to $r = 1.0$. The top right graph shows what happens if we separate out the static component $A'_s(n, d, 4.0, 0.25)$ from the dynamic component $A'_d(n, d, 4.0, 0.25)$. There is clear additional overhead for the low values of d corresponding to problems with a high probability of satisfiability, and the converse for high values of d. However, since the static component is only computed once, we are also interested in the impact over time. The bottom left and bottom right graphs therefore show the average per-run time performance after three and five runs respectively, which show the impact of the initial overhead diminishing, After five runs, even problems around the phase transition for $n \leq 500$ see gains as the result of separating out the static component from the dynamic component. The choice of whether or not to separate the two thus depends on the expected number of re-uses of the static component $A'_s(n, d, 4.0, 0.25)$, which is likely to be high in the context of stream reasoning applications.

The exact times of course depend on the system on which the computations are performed. The results listed here are the product of a system containing a fourth-generation Intel Xeon E5-1650 CPU (6 cores, 12 threads) with access to 64GiB of RAM, utilising the General Qualitative Reasoner (GQR)[9] by Gantner et al. (2008), configured for RCC-8. The results show that processing streams at over 1Hz is still possible even for problems near the phase transition involving 500 regions.

Effectiveness and scalability of landmarks

In order to empirically evaluate MSTL with landmarks we ran experiments to test the effectiveness and the scalability of the landmark based approach compared to the case where no landmarks were used. In these experiments, we were only interested in consistent scenarios, to capture the operational real-world domain. In particular, we are interested in the effects of landmarks on the resulting intertemporal disjunction size for non-landmark to non-landmark relations.

When considering two time-points t_1 and t_2, the problem of generating scenarios is given a consistent scenario with landmarks for time-point t_1 generate a consistent scenario with those same landmarks for time-point t_2. To achieve this, we make use of a variation of the scenario generation method presented earlier. Scenarios for a single time-point are generated based on the number of (non-landmark) regions n and the average disjunction size l. We extend this by also considering the number of landmarks m such that $n + m = |\mathcal{R}|$, and again fixing parameter $l = 4$. Our parameter combinations consist of varying numbers of regions between 20 and 200 with step size 20, and varying landmark ratios relative to the number of regions (i.e. m/n) between 0 and 0.9 with step size 0.1.

The initial 'seed' for a scenario covers the landmark regions and their relations to each other. In our experiments we generated 30 such seeds per parameter combination. Here we are only interested in a consistent scenario with complete knowl-

[9] The GQR implementation is available at `https://github.com/m-westphal/gqr`.

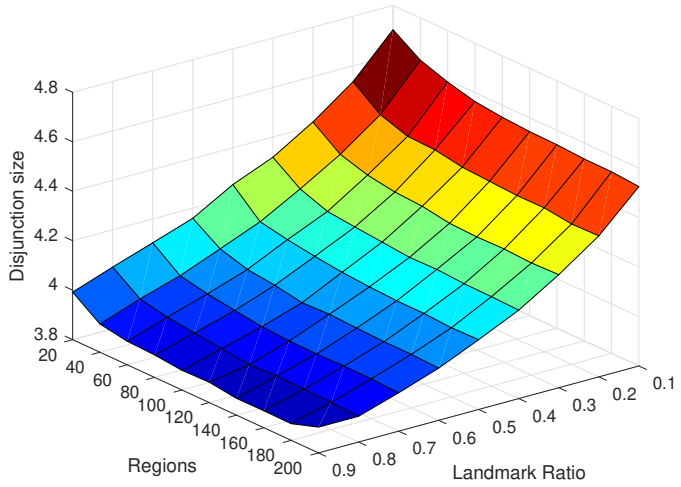

Figure 5.4: Absolute disjunction size for varying number of regions and landmark ratio; smaller is better.

edge, so GQR is used to generate consistent interpretations of scenarios. These fully known seeds can then be used as the basis for a larger spatial relation matrix by adding further regions until we obtain the desired $|\mathcal{R}|$ regions. The number of CSPs generated from a seed was kept constant at 20. Note that these CSPs then all share a seed as a common component. We can therefore combine two CSPs that share a common seed. Excluding combinations that involve the same CSP twice, given 30 seeds and 20 CSPs per seed we get 30 $\times$ (20 $\times$ (20-1))/2 = 5,700 instances for each parameter set.

The results of our experiments are illustrated in Figures 5.4 and 5.5, where every point represents the average over 5,700 instances. In Figure 5.4, the number of regions and the landmark ratio are changed to see how they affect the disjunction size of non-landmark to non-landmark spatial relations. Here we limit ourselves to the average over the spatial relations that are not fully unknown. The results show that the more landmarks are added, the less uncertainty in terms of disjunction size is measured for these relations, reaching a disjunction size of about 4 for a landmark ratio of 0.9. The landmark approach is also scalable in terms of the number of regions. This is also shown in Figure 5.5, which illustrates the percentage of non-landmark to non-landmark intertemporal relations that remain fully unknown. Previously, we could not say anything about these relations, as illustrated by the percentage of fully unknown relations being 100%. Using landmarks, this is reduced to 30% for landmark ratio 0.9, but having a landmark ratio as low as 0.1 results in an improvement of roughly 20%.

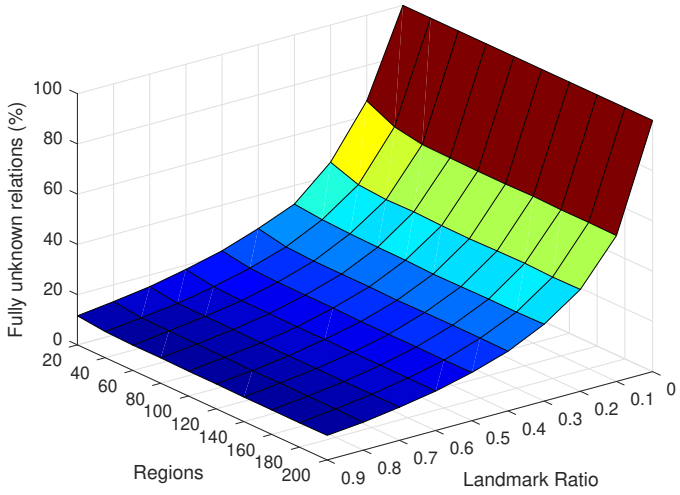

Figure 5.5: Percentage of relations fully unknown for varying number of regions and landmark ratio.

5.7 Summary

While spatial extension to temporal reasoning have been investigated in the past, these works have not specifically focused on the application of these resulting spatio-temporal logics in a stream reasoning context. We presented MSTL, a metric spatio-temporal logic that combines MITL and RCC-8 qualitative spatial calculus. Similar to ST_1, MSTL allows for the application of the 'next' operator to region terms, which makes it possible to express intertemporal spatial relations between regions. Since qualitative intertemporal spatial relations cannot be observed directly, a frame of reference formed by landmark regions is used to reduce the uncertainty of the intertemporal spatial relations. To facilitate incremental reasoning over streams, the generation of state streams is discussed. These state streams are used to progress MSTL formulas using an extension of the classical progression procedure for MITL. This makes it possible to apply path checking to MSTL formulas, which is useful in applications such as execution monitoring.

Part III

ADAPTIVE STREAM PROCESSING

Chapter

6 State stream synthesis

TEMPORAL models have thus far been presented as ω-words, which can be used to represent streams. That way a prefix of an ω-word could be used to model the observed stream, with the suffix representing the part of the stream that is yet to be observed. The problem focused on here is how to synchronise a stream that could be represented using prefixes. As this is not a primary focus for this dissertation, this chapter primarily serves as a chronological overview based on observations made during the research, utilising parts of earlier publications (Heintz and de Leng, 2013; de Leng, 2013; de Leng and Heintz, 2014, 2015a,b).

6.1 Introduction

The goal of *state stream synthesis* is to generate a stream containing state information on demand. This requires utilising stream processing to capture and refine observations such that they can be used for the grounding of logical propositional or predicate symbols, and to synchronise potentially many such streams such that state information can be delivered at a regular time interval.

The type of stream processing considered here is therefore different from the type of stream processing research commonly found in the context of Big Data and distributed systems. In those areas, the problem focus is on the speed at which streaming data is processed. Indeed, as has also been pointed out more recently by Hirzel et al. (2018), many stream processing tools and languages have been developed with differing emphases on for example performance (e.g. the use of windows and parallelisation), generality (e.g. language extendibility), and productivity (e.g. ease of adoption). One example of a language for stream processing is the CAL Actor Language (Eker and Janneck, 2003), which is a language for describing low-level operations at the stream transformation level, whereas stream processing languages like the Continuous Query Language (CQL) by Arasu et al. (2006) in-

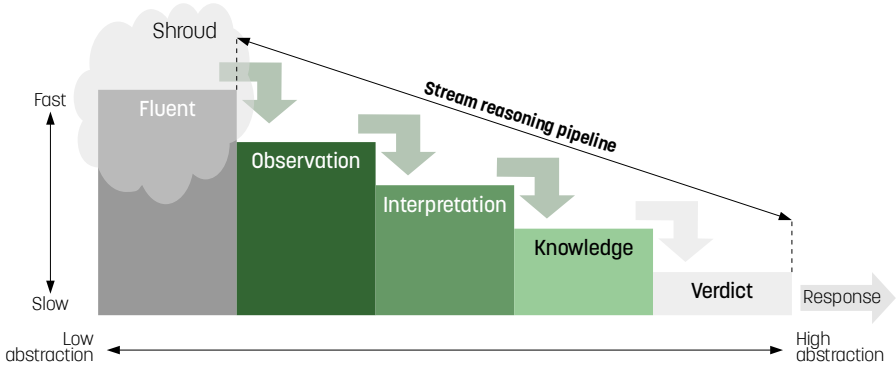

Figure 6.1: The stream reasoning waterfall model with the transformation of observations into knowledge via interpretations highlighted.

stead take a SQL-like declarative approach, with the added benefit of being easy to adopt by those familiar with SQL. An example for a larger stream processing architecture is Apache Flink (Carbone et al., 2015), which emphasises high-velocity distributed stream processing at a meta-level. Instead of focusing on throughput, the work presented here focuses on the qualitative aspects of stream processing. Of particular interest is the issue of generating a synchronised stream with state information based on some user-requested grounding, which is characterised by a mapping from logical propositional or predicate symbols to stream objects. This first requires a choice of data model for streams, combined with a way of accessing parts of individual states. Since different stream processing languages assume different data models (e.g. RDF stream processing assumes the RDF data model for streams, as will be discussed later in Chapter 11), it may not be possible to use pre-existing stream processing languages. Recall the stream reasoning waterfall shown again in Figure 6.1 with the transformation from observation to knowledge via interpretation highlighted. The data model concerns are captured by the interpretation step, which in part deals with the problem of representation. Finally, the knowledge step incorporates background theories and allows for implicit information to be made explicit.

In this chapter, we consider these transitions in relation to previous work and the lessons learned. In particular, we give an overview of different kinds of stream subscriptions, and consider how these could be used to synthesise state streams for the purpose of stream reasoning by applying a synchronisation procedure. We also briefly consider the problem of incorporating background knowledge given such a state stream. Chapters 7 and 8 then focus on the robust maintenance of stream subscriptions.

6.2 Timed data streams

Timed data streams were originally informally discussed in Chapter 2. They are a specific instance of the *stream* concept, which we consider to be a sequence of time-stamped values.

Definition 6.1 (Timed data stream)**.** *A timed data stream is an unbounded sequence of time-stamped values*

$$((l_0, v_0, t_0), (l_1, v_1, t_1), \dots) \tag{6.1}$$

where $v_i \in \mathcal{V}$ *represents a (structured) value,* $l_i \in \mathbb{V}\text{ar}$ *represents a variable name, and* $t_i \in \mathcal{T}$ *represents a time-point.*

The individual triplets that make up a stream are referred to as *data samples*. Streams are assumed to flow from a source that generates the stream's samples to a receiver that consumes those samples. This connection is referred to as a *channel*, which can be realised by a transportation mechanism and annotated with meta-information such as a label. These channels are closely related to *subscriptions*. A subscription is a statement of interest in a sequence of data samples that conform to a particular description provided by an interested client. Such a statement of interest usually leads to the establishment of a channel between a stream provider and the aforementioned subscribing client. Subscriptions are therefore a useful starting point for synthesising a state stream that can be used for stream reasoning. Recall that streams can be regarded through an internal view, in which a stream is seen as a sequence of states, and an external view, in which a stream is seen as an object with associated properties. Consequently, there are different interpretations of what subscription entails. In general, a subscription can be regarded as an expression of interest with the goal of receiving said thing of interest. One might therefore identify a stream by name subject to certain constraints as an expression of interest, which can be regarded as a *syntactic subscription*. Alternatively, one might instead identify desired states based on semantic characteristics, which can be regarded as a *semantic subscription*.

6.3 Syntactic subscriptions

When setting up a syntactic subscription, one describes a desired stream by identifying one or more source streams and constraints. For example, it is common to see multiple streams get combined and filtered according to a set of logical conditions. Query languages are used to describe the specifications of such a desired stream, and it is the task of a stream processing engine to apply the necessary operations. There is therefore a need for a query language that is designed to be compatible with the stream model described informally in Chapter 2, in which a stream is characterised as a sequence of samples containing potentially multiple named fields containing values. It is to this end that the languages SPL and FSL were developed.

Listing 6.1: Formal grammar for SPL.

```
decl                : source_decl | sink_decl | compunit_decl |
                      stream_decl ;
decls               : decl | decl SEMICOLON decls ;
source_decl         : 'source' type_decl STRING ;
sink_decl           : 'sink' stream ;
compunit_decl       : 'compunit' type_decl STRING LP
                      type_decl (COMMA type_decl)* RP ;
stream_decl         : 'stream' NAME EQ stream ;
type_decl           : basic_type | complex_type ;
basic_type          : NAME COLON type ;
complex_type        : LP basic_type (COMMA basic_type)* RP ;
type                : 'int' | 'float' | 'string' | 'boolean' ;

stream              : stream_term 'with' stream_constraints ;
streams             : stream | stream COMMA streams ;
stream_term         : STRING
                      | STRING LP streams RP
                      | 'sync' LP streams RP
                      | 'merge' LP streams RP
                      | LP 'select' select_exprs 'from' stream
                        ('where' where_exprs)? RP ;
select_exprs        : select_expr | select_expr COMMA select_exprs
    ;
select_expr         : field_id ('as' pstring)? ;
where_exprs         : where_expr | where_expr 'and' where_exprs ;
where_expr          : field_id EQ value ;
field_id            : STRING | STRING DOT field_id ;
pstring             : STRING | STRING? PERCENT field_id PERCENT
                      pstring? ;
value               : STRING | NUMBER
stream_constraints  : stream_constraint | stream_constraint COMMA
                      stream_constraints ;
stream_constraint   : 'start_time' EQ NUMBER | 'end_time' EQ
                      NUMBER | 'max_delay' EQ NUMBER |
                      'sample_period' EQ NUMBER |
                      'sample_period_deviation' EQ NUMBER ;
```

Stream Processing Language

The Stream Processing Language (SPL) was originally designed by Hongslo (2012) and Heintz (2013), and was inspired by the Structured Query Language (SQL) used in many relational database management systems (RDBMS). SPL is typically used to filter existing streams through selection and to combine streams through merging or synchronisation, and contains the option to set policy constraints. Additionally, aliases can be used to resolve conflicting field names and to improve readability.

However, SPL initially had a number of issues, the most critical being the lack of support for transformations stemming from its design. SPL was modified and

Listing 6.2: Example SPL statements.

```
1 s1 = select output as value from somestream where id = uav1
2 s2 = select output as value from cu(somestream, anotherstream)
      where id = uav1
3 s3 = merge(s1, s2)
4 s4 = sync(s1, s2) with sample_period = 100
5 s5 = select * from (select * from s1) where value = 0
```

extended to address these shortcomings, and its formal grammar is shown in Listing 6.1. The language provides two key features: stream manipulation support and knowledge process declaration support. The stream manipulation support allows for the selection, synchronisation and merging of streams. Knowledge process declaration support allows for the declarative specification of a stream transformation instance by describing sources, sinks and computational units.

Example 6.1 (Example SPL statements)**.** *Consider the statements shown in Listing 6.2. The first two statements are select statements, where the first is called a simple select and the second a complex select. For the first statement, the stream processing engine is asked to select the field 'output' from stream somestream for all samples in which the field 'id' has a value equal to 'uav1'. The resulting stream is then called stream s1. For the second statement, the stream processing engine is requested to use a computational unit cu, which is parameterised by two streams. A complex select differentiates itself from simple select by the invocation of computational units. The stream processing engine in this situation first creates an internal stream produced by cu, and then uses this stream to apply a simple select in order to apply the filtering. This resulting stream is then called stream s2. The third statement shows a merge statement, with the intended meaning that stream s3 is constructed by combining all of the samples arriving from two streams s1 and s2, which are of the same type. The fourth statement shows a synchronisation statement. During synchronisation, a stream processing engine is requested to generate a new stream at a certain frequency so that the values are all valid at the same time. The example statement tells the stream processing engine to synchronise streams s1 and s2 at every 100ms, producing a new stream s4. For each synchronised state to be generated, the procedure decides whether to wait for a data sample to arrive on the input streams, or whether to generate such a data sample based on the previously-received data sample from that stream. In the simplest case, such a previously-received sample is simply repeated. The resulting synchronised stream adheres to constraints that include the requested frequency and a maximum delay for the individual data samples. Finally, the fifth statement shows the importance of the parentheses around select statement when 'where' parts are involved. In this example, the filtering is done over the outermost select statement. However, had the parentheses been absent, the filtering would have been done over the inner-most select statement instead. The resulting stream is called s5.*

Listing 6.3: Formal grammar for FSL.

```
decl          : source_decl | sink_decl | compunit_decl ;
decls         : decl | decl SEMICOLON decls ;
source_decl   : 'source' NAME EQ path arguments? ;
sink_decl     : 'sink' NAME EQ path arguments? ;
compunit_decl : 'compunit' NAME EQ path arguments? ;

path          : system_path | ros_path ;
system_path   : ('a'..'z'|'A'..'Z'|'0'..'9'| SLASH | DASH |
                UNDERSCORE | DOT )+ ;
ros_path      : ('a'..'z'|'A'..'Z'|'0'..'9'| SLASH | DASH |
                UNDERSCORE )+ ;
arguments     : NAME (COMMA NAME)* ;
NAME          : ('a'..'z'|'A'..'Z'|'0'..'9')+ ;
```

Factory Specification Language

The *Factory Specification Language* (FSL) was developed (de Leng, 2013) to connect transformation symbols to programs (e.g. shared objects) that perform the desired computations, and acted as a companion language to SPL. The FSL grammar is shown in Listing 6.3, and can easily be extended to include more path types. An FSL statement is converted into a 'factory specification'. Conceptually, the factory specification serves as a transformation, i.e. it serves as a factory for the generation of a computation unit (with special cases being sources and sinks), which is an instance of a transformation.

6.4 Semantic subscriptions

Semantic subscriptions are subscriptions to a certain kind of information rather than streaming resources. For example, if a user wants to obtain temperature measurements for a particular room, this interest is decoupled from any specific information source; any resource providing the desired information suffices. The idea behind semantic subscriptions is closely related to topics such as *semantic web services* utilising the OWL-S service ontology (Martin et al., 2007) for annotating semantic web services, or *content-centric networking* (CCN) (Jacobson et al., 2009) where documents are stored at various points in the network based on demand and supplied to users based on a specification of interest in a particular document rather than a particular address.

Semantic Specification Language

To support semantic subscriptions in a stream reasoning setting, the *Semantic Specification Language* (SSL) was developed. SSL was intended to enable the declaration of semantic specifications for streams and transformations as characterised

Listing 6.4: Formal grammar for SSL.

```
decl           : stream_decl | source_decl | compunit_decl ;
stream_decl    : 'stream' NAME 'contains' feature_list
                 for_part? ;
source_decl    : 'source' NAME 'provides' field_feature ;
compunit_decl  : 'compunit' NAME 'transforms' field_features
                 'to' field_feature ;
field_features : field_feature (COMMA field_feature)* ;
field_feature  : NAME COLON NAME unit_list? ;
feature_list   : feature (COMMA feature)* ;
feature        : NAME LP feature_args RP EQ NAME unit_list? ;
feature_args   : feature_arg (COMMA feature_arg)* ;
feature_arg    : NAME alias? ;
for_part       : 'for' entity (COMMA entity)* ;
entity         : sort | object ;
unit_list      : ( OPEN unit (COMMA unit)* CLOSE ) | 'no_unit' ;
unit           : NAME power? ;
power          : ('+' | '-')? NUMBER ;
alias          : 'as' NAME ;
object         : entity_full ;
sort           : sort_type entity_full ;
entity_full    : NAME EQ NAME ;
sort_type      : 'some' | 'every' ;
NAME           : ( 'a'..'z'|'A'..'Z'|'0'..'9')+ ;
NUMBER         : ('0'..'9')+ ;
```

Listing 6.5: Example SSL statements for streams.

```
stream s1 contains Altitude(uav1) = alt
stream s2 contains Altitude(uav1) = alt for uav1 = id
stream s3 contains Speed(UAV) = spd for every UAV = id
stream s4 contains XYDist(UAV as arg1, UAV as arg2) = dist for
    every arg1 = id1, arg2 = id2
```

in this dissertation. The initial version of SSL was the *Semantic Specification Language for Topics* (SSL_T), and was used to semantically annotate middleware-specific named transportation channels (called 'topics' in this case) by the ontological concepts they contained (Dragisic, 2011; Heintz and Dragisic, 2012). Subsequent work (Heintz and de Leng, 2013; de Leng, 2013) extended SSL_T with units of measurement and transformations, called the *Semantic Specification Language for Transformations* (SSL_{TF}). SSL combines the two languages, and its full grammar is shown in Listing 6.4.

Example 6.2 (Example SSL statements)**.** *Consider the SSL statements in Listing 6.5. The first statement states that stream s1 contains information on the Altitude of object uav1 in the field named 'alt'. This is different from the second statement, which states that stream s2 contains the same information as stream s1 with the*

Listing 6.6: Example SSL statements for transformations.

```
compunit cu1 transforms from Distance [m] to Speed [m.s-1]
compunit cu2 transforms from Distance [km] to Speed [mi.h-1]
compunit cu3 transforms from NumVehicles no_unit to NumUAVs
    no_unit
source src1 provides Distance [m]
source src2 provides NumVehicles no_unit
```

difference that this is only the case when the field named 'id' has the value 'uav1'. The last two statements make use of sorts that are specified in the object ontology. Stream s3 contains information on the Speed for all objects in sort UAV, where the speed information is presented in the field named 'spd' for the UAV object referred to in the field named 'id'. We can see a similar construct in the semantic stream specification for stream s4. However, here we encounter some ambiguity as the sort UAV occurs twice. This is resolved by using an alias, in this case 'arg1' and 'arg2'.

SSL allows for the semantic annotations of transformations by semantically annotating a transformation with its input features and output feature. Alongside every feature the assumed unit of measurement is also included. Listing 6.6 shows a number of example SSL declarations used to describe transformation in this fashion. Here, three computational units and two sources are described by a semantic specification. The first computational unit, called cu1, transforms from the feature Distance to the feature Speed, where the temporal information in samples is utilised. It assumes distances in metres and speeds in m/s. Computational unit cu2 performs a similar transformation, but expects kilometres and mph respectively for its units of measurement. Not all features have to have units of measurements, and this is clearly shown in the case of computational unit cu3. The first source, called src1, provides the feature Distance in metres. Just like computational units, sources can handle features that do not assume units of measurement, as is shown for source src2.

Automatic query construction

A combination of SPL, FSL, and SSL was used in previous work (Heintz and de Leng, 2013; de Leng, 2013) for state stream synthesis through a process called *semantic matching*. In this approach, a user could describe a desired information stream by specifying ontological concepts of interest. This vocabulary matched the one used by SSL to annotate streaming resources, such that a stream reasoning framework could identify matching resources by comparing the requested concepts to the resource annotations. Such a framework could then construct an SPL query based on a select-merge-synchronise pattern; relevant fields were selected from source streams, matching fields from different streams were merged, and the resulting streams were synchronised into a single stream containing a field for every concept of interest. This is referred to as an *automated query construction* process.

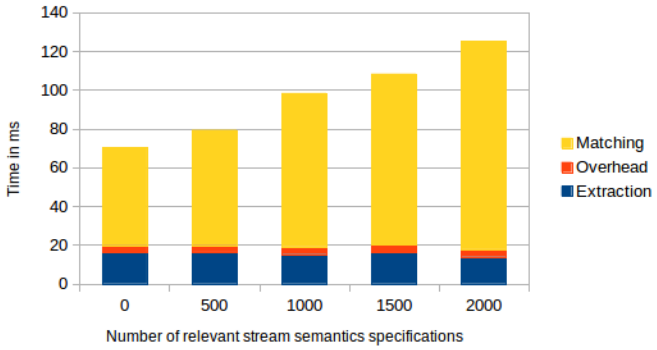

Figure 6.2: Breakdown of automated query construction performance.

One problem with this approach, however, is that the resulting query is static. If anything needs to be changed, another query needs to be constructed to effect that change. Another observed problem with such a process, is the cost involved in constructing a query. SPL was not designed to be efficient in terms of the required query length, resulting in a disproportionate amount of time being spent on first constructing and then executing a query. Figure 6.2 (de Leng, 2013; Heintz, 2013) shows a breakdown of the performance of automated query construction as the number of relevant streaming resources (i.e. resources that satisfy a semantic query) increases. The extraction time refers to the time needed to parse a semantic query, followed by a communication overhead, and finally the actual time spent on constructing and executing an SPL query. These results led to a change in our methodology for state stream synthesis; instead of constructing a query, the focus shifted towards managing stream processing directly.

6.5 Synchronisation

The title of this dissertation refers to 'robust stream reasoning'. Robustness in the context of this dissertation means a resilience to changing conditions when performing stream reasoning. In particular, this requires a steady supply of states, which is characterised as a state stream. Such a state stream has to be synthesised through the combination of potentially multiple source streams.

For stream reasoning applications such as the runtime verification discussed previously, this state stream represents an ω-word for which it is checked whether it satisfies a logical formula. This means that the fields that make up the samples in the state stream can be used as interpretations for the propositional symbols in that formula. This can be considered as a type of *symbol grounding*. Each sample of the state stream corresponds to a particular time-point, and the information in the fields of that sample is valid for that time-point. While in a formal setting it is beneficial to just consider boolean-typed fields, in a practical setting the fields' information does

Algorithm 6.1: Synchronisation (Heintz, 2009)

```
1 function SYNCHRONISE(f_1, ..., f_n, t_start, t_end, Δ, d_approx, d_max):
2 foreach f_i do
3 |   create a buffer and set up a subscription to f_i
4 end
5 t_sync ← t_start
6 while t_sync ≤ t_end do
7 |   SYNCHRONISE_AT(t_sync, d_approx, d_max)
8 |   t_sync ← t_sync + Δ
9 end
```

not necessarily have to be boolean. It can be useful to make statements about expected observations, for example $\mathsf{G}_{[0,10]}(altitude > 100)$. Here, the boolean condition can be regarded as a greater-than built-in binary predicate taking two numbers. When a predicate is built-in, its interpretation given any combination of terms is well-defined and can be regarded as part of background knowledge that does not have to be provided via a stream. The 'altitude' term in this example can therefore be grounded in a state stream that has a numeric 'altitude' field. This mapping can either be established *implicitly*, based on string matching, or *explicitly*, based on a user-provided mapping. The mappings connect symbols in a formula to fields in a state stream. This state stream needs to first be synthesised. The procedure taken here is to start with a subscription (of either the syntactic or semantic kind) for each symbol mapping. The subscribed-to streams then need to be combined into one state stream. One such synchronisation procedure is that by Heintz (2009), called `SYNCHRONISE`, which makes use of the two helper procedures `SYNCHRONISE_AT` and `IS_SYNCHRONISED`.

The main idea behind the synchronisation procedure in Algorithm 6.1 is to synchronise samples from different streams by their valid times for predetermined time-points. Recall that the valid time of a sample is the time at which the information contained within the sample is valid, and that this time can differ from the time at which the sample becomes available to the synchronisation procedure. The `SYNCHRONISE` procedure listed in Algorithm 6.1 first sets up subscriptions to the streams that need to be synchronised, denoted by $f_1, \ldots, f_n$. The synchronisation procedure is then supposed to only perform its task between times t_{start} and t_{end}, with a period of Δ time-units. It then simply calculates the next time at which synchronisation is supposed to take place — called *synchronisation time* and denoted by t_{sync} — and calls the `SYNCHRONISE_AT` procedure from Algorithm 6.2 to compute a state for this time.

Algorithm 6.2 assumes that the valid times in states grow monotonically such that receiving a sample for a specific valid time comes with the guarantee that no further samples will arrive with valid times that precede that of the current sample. Furthermore, approximation may be necessary because samples may not necessarily have valid times for the desired synchronisation time. Heintz (2009) therefore

Algorithm 6.2: Synchronisation at a specific time (Heintz, 2009)

```
1  function SYNCHRONISE_AT(t, d_approx, d_max):
2  t_sync ← t
3  add a time-out for time t_sync + d_approx
4  add a time-out for time t_sync + d_max
5  do
6      wait for input or timeout
7      if received sample s from input stream i at time t then
8          add sample s to buffer i
9          remove obsolete samples from buffer i
10         update the category for buffer i
11     else if received timeout t then
12         foreach buffer i do
13             update the category for buffer i
14         end
15     end
16 while ¬IS_SYNCHRONISED(t, d_approx, d_max)
17 compute state at t_sync
```

Category	Description
Exact	An exact value is available.
AprxFinal	Approximation available; no further information expected.
AprxMore	Approximation available; additional information possible.
NoAprxFinal	No approximation available; no further information expected.
NoAprxMore	No approximation available; additional information possible.

Table 6.1: The five categories for streams when performing synchronisation using the SYNCHRONISE procedure.

considered five possible classifications for each stream — one based on perfect timing and another four based on combinations of approximation and delayed state information — shown in Table 6.1. At the start, each stream is classified as NoAprx-More, as no information has yet been received to make an approximation and more information may still arrive.

Given these categories, it is possible to consider at least two types of delay thresholds. The first deals with the maximum delay before approximation, denoted by d_{approx}, for which a time-out is set (line 3) by setting the delay relative to the synchronisation time t_{sync}. If this time-out is reached, the buffered information is used to attempt to approximate the state at t_{sync}. Depending on the approximation method used, this may require more or less state information. For example, given quantitative state information, a linear approximation would need two samples whereas a most-recent value approximation only requires a single such sample. Likewise, for boolean values one could assume that a truth value holds unless replaced, which is equivalent to a most-recent value approximation. These approximations are computed as part of the IS_SYNCHRONISED procedure shown in Algo-

Algorithm 6.3: Synchronisation check (Heintz, 2009)

```
1  function IS_SYNCHRONISED(t, d_approx, d_max):
2  foreach input stream i in category NoAprxFinal do
3  |   approximate i with no_value
4  |   set the category of i to AprxFinal
5  end
6  if all input streams are in categories Exact, AprxFinal, or AprxMore and
     t_sync + d_approx ≤ t_now then
7  |   return True
8  else if t_sync + d_max ≤ t_now then
9  |   foreach input stream i in category NoAprxMore do
10 |   |   approximate i with no_value
11 |   |   set the category of i to AprxFinal
12 |   end
13 |   return True
14 else
15 |   return False
16 end
```

rithm 6.3, which determines whether we can proceed to generating a synchronised state. Similarly, the second delay type is the maximum delay, denoted by d_{max}, for which a time-out is also set (line 4) by setting the delay relative to the synchronisation time t_{sync}. This time-out is then guaranteed to lead to the computation of a state based on the information received thus far.

Algorithm 6.3 consider four cases. First, `IS_SYNCHRONISED` 'approximates' a value to `no_value` (line 3) if the corresponding stream reached the maximum delay without being able to approximate a meaningful value, leading to a reclassification to `AprxFinal` (line 4). Second, if exact or approximated values exist for each stream, and the maximum delay before approximation has been reached or exceeded, synchronisation is deemed completed. Third, if the maximum delay has been reached or exceeded, streams that lack approximation are 'approximated' to `no_value` and reclassified to `AprxFinal`, after which synchronisation is deemed completed. Finally, by default, synchronisation is not yet finished. In the first three cases, however, the `SYNCHRONISE_AT` procedure from Algorithm 6.2 computes a state for t_{sync} (line 17) based on the exact or approximated values for each stream. Of course, this can be a problem if no exact value or usable approximation (i.e. a `no_value` that cannot be resolved by using an incomplete state as per Definition 4.2) is produced. There is no 'correct' solution for producing a synchronised state in this case — Heintz (2009) suggests notifying the system of an error or approximating the state anyway using whatever information is available, for example based on the previous synchronised state if one exists.

This finally brings us to the issue of obtaining suitable subscriptions, which is needed for the `SYNCHRONISE` procedure (line 3) in Algorithm 6.1. As explained previously, there are a number of problems when relying on query construction. We

therefore seeks to instead dynamically reconfigure stream processing based on a user's needs. Chapter 7 formalises these desired semantic subscriptions as objects called *targets*, which can be satisfied by a configuration. This additionally makes it possible to change stream processing on the fly if the need arises, which is discussed in more detail in Chapter 8. The combination makes it possible to robustly perform state stream synthesis using the synchronisation and grounding procedures described here. The investigation of more flexible ways of grounding logical symbols in state streams provided in this fashion is left for future work.

6.6 Incorporating background knowledge

Part of stream reasoning involves reasoning about the individual states that make up a stream. This type of a-temporal reasoning is common, and works by combining state information (without necessarily considering the time for which it is valid) with a *background theory*. In all of these cases, the reasoning can be regarded as a function taking a state and a theory that, when combined, yield a set of states with their individual probabilities.

An example of this is qualitative reasoning, as shown in Chapter 5, which focuses on reasoning about abstract relations rather than quantities. Commonly a set of relations forms a state which can then be closed through the use of composition-table based reasoning, where a reasoner determines whether the state is consistent and, if so, what the possible consistent configurations are. Augmenting state streams with spatial information can then be done in a number of ways. A straightforward and naive method would be to collect the complete set of spatial information for a given time-point, run it through a qualitative reasoner to infer more information on the spatial relations, and then augment the state stream with these resulting spatial relations.

A slightly better way would be to only augment the state stream with those spatial relations that are relevant. To efficiently infer implicit spatial relations we use the facts that relations between (rigid) variables that have not changed are the same and the algebraic closure for the same set of variables must be computed many times (every time some of the variables have changed). As an example, the spatial relations between static buildings do not change, so it is not necessary to compute their spatial relations at every time-point even if they are not explicitly given. If the set of variables is partitioned into those that are static and those that are dynamic, it is enough to compute the algebraic closure of the constraints involving only static variables once and then add the constraints involving at least one changing variable when they have changed and compute the new algebraic closure. The effect is that there is an initial cost of computing the static part while the cost for each update is reduced (Heintz and de Leng, 2014).

Since qualitative reasoning may yield multiple consistent hypotheses, we can generate a set of states containing each of these hypotheses together with the prob-

ability of that hypothesis. In the absence of additional information, the probability of each hypothesis is equal to that of the others, i.e. they are uniformly distributed.

6.7 Summary

In this chapter, we considered the problem of state stream synthesis for the purpose of generating a state stream that can be used for stream reasoning tasks as introduced earlier. This particular problem is not a primary focus of this dissertation, but nevertheless plays an important role in connecting the two strands for reasoning over and about streams. Some earlier efforts towards automated query construction were discussed, resulting in the introduction of SPL and FSL for setting up stream processing, and SSL for allowing streams and transformations to be semantically annotated. Semantic matching describes the process of finding suitable transformations or streams given a semantic specification. It was used to ultimately construct an SPL query that would result in a state stream containing the desired information. This approach however had a few down-sides. The automated construction of queries before their execution was expensive. Additionally, if further changes were needed, new queries would have to be constructed with the present configuration of the ongoing stream processing in mind. Nevertheless, if the relevant streams are subscribed to, they can then be synchronised using the synchronisation procedure by Heintz (2009). In the next chapters, we consider a different approach to setting up semantic subscriptions, which are compatible with the state stream synthesis methodology presented here.

Chapter

7 Reasoning about composition

LOGIC-BASED stream reasoning commonly makes use of temporal logics to express statements concerning the truth value of properties over time. Stream reasoning techniques usually do not consider where their data originates from, and assume it to be given. However, the generation of streaming data for the purpose of stream reasoning is an important stream processing task. We call this ability *reasoning about composition*, which treats streams as objects. This chapter borrows from and extends previous work on configuration modelling and planning (de Leng and Heintz, 2015a,b, 2017).

7.1 Introduction

Robotic systems are getting increasingly complex, with more and more components usually connected by some form of publish-subscribe messaging pattern. Support for this type of integration is often provided by middleware such as the Common Object Request Broker Architecture (CORBA) and the Robot Operating System (ROS). The configuration of what channels a component publishes and subscribes to is often done manually or through scripts. This is both error-prone and assumes that the set of available components does not change at run-time. However, IoT development towards for example swarmlets (Latronico et al., 2015) points to a future in which systems are increasingly heterogeneous, decentralised and geographically spread-out. The assumption of an unchanging or slowly changing set of available components is therefore rapidly becoming unreasonable.

The challenge of dealing with this volatility also affects the task of transforming streams with the goal of producing state streams. After all, if component sets cannot be assumed to be constant, the task of generating a stream needs to be complemented with the task of maintaining one. In this chapter, the problem of generating a state stream is therefore translated into the problem of satisfying a semantic

subscription in a stream reasoning framework. We first consider a formalisation of a stream reasoning framework and its dynamics, called the *DyKnow model*, which allows us to frame the problem as an optimisation problem. The purpose of the formal model is to be general enough such that implementation details are abstracted away, allowing for potentially many different realisations. Lastly, with the DyKnow model formalised, we consider a common representation of configurations relative to an ontology.

7.2 Service composition

The problem of finding a suitable composition of transformations through reasoning about those transformations shares a lot of similarities with the work on automatic service composition. For example, an approach to 'semantically-enabled sensor plug & play' was proposed by Bröring et al. (2009), who identified challenges to achieving sensor plug-and-play based on semantic knowledge of sensor observations. They subsequently proposed a method for automatic plug-and-play functionality by making use of a Sensor Bus (Bröring et al., 2011) that matches services to sensors. The approach to semantic subscriptions taken in this dissertation is more advanced than the Sensor Bus approach in that we periodically recombine and reconnect components whereas the Sensor Bus directly connects with information sources. Another example is research towards Semantic Sensor Networks, which led to the development of the Semantic Sensor Network ontology (SSN) (Compton et al., 2012). SSN focuses on well-structured semantic descriptions of sensors. The work presented here makes use of semantic descriptions of streaming components rather than sensors by using functional descriptions of the inputs and outputs of these components. These functional descriptions are extensions of the OWL-S service ontology (Martin et al., 2004) applied to a streaming context.

The ability to reconfigure a system on demand is also closely related to configuration planning. Automatic (re)configuration techniques have been studied in detail (Rao and Su, 2005; Dustdar and Schreiner, 2005; Pejman et al., 2012). The work by Tang and Parker (2005) on ASyMTRe is an example of a system geared towards the automatic self-configuration of robot resources in order to execute a certain task. Similar work was performed by Lundh et al. (2008) related to the Ecology of Physically Embedded Intelligent Systems, also called the PEIS-ecology (Saffiotti et al., 2008). Given a high-level goal describing a task, a configuration planner is used to configure a collection of robots towards the execution of the task rather than logic-based stream reasoning. Their solution is however designed for use within the PEIS middleware and does not easily transfer to other environments such as the ROS middleware. Lundh (2009) further points out that their approach uses static cost measures and could benefit from incorporating semantic knowledge. Our approach focuses on a more advanced representation of cost, and makes use of semantic descriptions for components. The SAMSON Wireless Sensor Networks (WSNs) middleware by Portocarrero et al. (2016) is similar to run-time reconfigurable systems in

Symbol	Description
$l_i \in \mathsf{Var}$	Set of variables
$tag, itag_i, otag \in \mathsf{Tag}$	Set of tags
$v_i \in \mathcal{V}$	Set of (structured) values
$t_i \in \mathcal{T}$	Set of time-points
$tid, cid, qid \in \mathbb{N}$	Set of identifiers
$in_i, out, chan \in \mathbb{N}$	Set of channels
$\left\langle cid, tid, [in_1, in_2, \ldots, in_n]^T, out, \mathcal{S} \right\rangle \in CU$	Computation units
$\left\langle tid, f(x_1, \ldots, x_n, \mathcal{S}), [itag_1, \ldots, itag_n]^T, otag \right\rangle \in F$	Transformations
$\langle qid, tag, chan \rangle \in T$	Targets
$\mathcal{S} \subseteq \mathsf{Var} \times \mathcal{V}$	States
$\sim \subseteq \mathsf{Tag} \times \mathsf{Tag}$	Similarity relation
$f : \mathcal{V}^n \times \mathcal{S} \hookrightarrow \mathcal{V} \times \mathcal{S}$	Transformation function
$\varepsilon = \langle CU, F, T, \sim \rangle$	Environment
$\delta = (CU^+, CU^-, F^+, F^-, T^+, T^-)$	Change set
$\varepsilon' = \varepsilon \otimes \delta$	Update
$\varepsilon \Rightarrow_\delta \varepsilon'$	
$\varepsilon \in \mathsf{Valid}$	Set of valid environments

Table 7.1: Notation for the DyKnow model.

their consideration of a dynamic environment in which a network can be reconfigured to deal with changes, albeit at a lower level. In the case of SAMSON, these changes include faults, but also disconnection and power concerns. A survey of other recent work towards WSN middlewares is presented by Kerasiotis et al. (2015).

None of these approaches are specifically suitable for stream reasoning frameworks, however. Furthermore, the choice of cost measure for services is difficult. Previous work by Lundh (2009) for example notes the same difficulties and instead simplifies the problem by assigning constant utility values. It seems more likely, however, that the cost of services would change based on the context of the operations, which is one angle we will therefore consider here.

7.3 DyKnow model

The *DyKnow model* is a formalisation of stream reasoning frameworks and extends earlier work (Heintz, 2009; Heintz et al., 2010) that considered such frameworks to be composed of possibly many interconnected components. The formal model is general and serves as a specification from which potentially many different realisations can be created. Table 7.1 provides a complete summary for the notation used in describing the model.

Computational environment

A *computational environment* is composed of a *computation graph*, *transformations* and *targets*. The computation graph consists of *computation units* connected by *channels*.

Streams are the product of transformations, which can either refine existing streams into new streams, or act as sources by generating streams without requiring any input streams. In practice, sources often use information external to the computational environment to generate streams, for example through sensor observations. A transformation is considered to be an annotated function that can be instantiated as a computation unit for application within a specific configuration.

Definition 7.1 (Transformation)**.** *A transformation (TF) is an annotated stream-generating function that takes streams as inputs. It is described by a tuple*

$$\left\langle tid, f(x_1, \ldots, x_n, \mathcal{S}), [itag_1, \ldots, itag_n]^T, otag \right\rangle, \tag{7.1}$$

where $tid \in \mathbb{N}$ *represents a unique transformation identifier,* $f : \mathcal{V}^n \times \mathcal{S} \hookrightarrow \mathcal{V} \times \mathcal{S}$ *represents a partial function from input values and an initial state to an output value and a resulting state,* $itag_i \in \mathsf{Tag}$ *represent tags for inputs, and* $otag \in \mathsf{Tag}$ *represents the output tag.*

Definition 7.2 (Computation unit)**.** *A computation unit (CU) is a component that is described by a tuple*

$$\left\langle cid, tid, [in_1, in_2, \ldots, in_n]^T, out, \mathcal{S} \right\rangle, \tag{7.2}$$

where $cid \in \mathbb{N}$ *represents a unique identifier for CUs,* $tid \in \mathbb{N}$ *represents the unique identifier of the transformation which this CU is an instance of,* $in_i \in \mathbb{N} \cup \{\mathsf{none}\}$ *represent incoming channels,* $out \in \mathbb{N} \cup \{\mathsf{none}\}$ *represents the outgoing channel, and* $\mathcal{S} \subseteq \mathsf{Var} \times \mathcal{V}$ *represents the state as a relation between variables and values.*

Note that there is a close relation between CUs and TFs — a CU is called an *instance* of a TF iff their tid identifiers match.

Example 7.1 (TFs and CUs)**.** *Robots commonly use visual sensing methods to detect and track objects of interest. Consider a ball detector that is able to detect footballs by their round white shape with black spots. The ball detector can be represented in terms of a transformation and a computation unit. The ball detector transformation refers to the mathematical function describing the detection method, together with meta-information for this function. It is annotated with tags describing its input as camera images, and its output as bounding boxes. We can apply the transformation by connecting it to an input stream of camera images, yielding a stream of bounding boxes. This application of the transformation is called a computation unit. Every CU has an identity, a reference to its corresponding TF, connections to input and outputs channels, and state information. The state information allows the transformations to be stateful, meaning they can retain information that makes it easier to for example perform tracking after an initial detection.*

Lastly, the computational environment contains *targets*, which describe semantic subscriptions for outside modules such as the stream reasoning engine. Note that subscriptions also occur *within* the computational environment, but that these are not referred to as targets because they do not reflect the global configuration goals of the computational environment. Subscriptions of the latter kind are described by the connections between CUs and channels as shown earlier.

Definition 7.3 (Target)**.** *A target describes a desired semantic subscription and is denoted by a tuple*

$$\langle qid, tag, chan \rangle , \tag{7.3}$$

where $qid \in \mathbb{N}$ *is a unique (query) identifier,* $tag \in$ Tag *is a description of the desired information, and* $chan$ *is the channel the described stream is expected on.*

Targets thus indirectly represent configuration goals for the computational environment[10] by indirectly referencing desired streams by their semantic descriptions. These streams are generated by instantiated transformations, which in turn have input requirements. For a given set of targets, there may be many different computation graphs which satisfy all of the input requirements and similarity relations at different costs.

By combining these elements, we can formally describe the computational *environment*.

Definition 7.4 (Environment)**.** *An environment is denoted by a tuple*

$$\varepsilon = \langle CU, F, T, \sim \rangle , \tag{7.4}$$

where CU *denotes a set of computation units called a computation graph,* F *denotes a collection of transformations called a library,* T *denotes a set of targets called a goal, and* $\sim \subseteq$ Tag $\times$ Tag *denotes a similarity relation between tags. Elements of environment* ε *have short-hand representations* CU_ε, F_ε, T_ε, *and* $\sim_\varepsilon$ *respectively.*

An environment thus encodes the configuration of the system as well as the state of its individual components. It is connected to streams through the collection of channels that connect the various CUs, because they are a product of those CUs. There is therefore a total mapping from streams to channels. Since CUs define outgoing and incoming channels, there is a clear connection between streams and their source and destination CUs as well.

Dynamics

An environment is a representation of the state of the configuration of the computational environment. This environment may be subjected to changes over time. These changes are represented by a *change set*.

[10] Alternatively, one can consider targets to represent constraints on channels. These constraints are then described in terms of desired semantic descriptions.

Definition 7.5 (Change set)**.** *A change set is a tuple*

$$\delta = (CU^+, CU^-, F^+, F^-, T^+, T^-) \tag{7.5}$$

consisting of set additions and set removals denoted by superscript '$+$' and '$-$' respectively. The notation $\delta_\varnothing$ is used as a short-hand to describe the absence of change, i.e. $\delta_\varnothing = (\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing)$.

Change sets can thus add and remove elements to and from the environment. These additions and removals can also be used to for example represent tag changes in transformations or connection changes of CUs to channels. Whenever an environment changes in a way that can be represented using a change set, we call this change an *update*. More formally, an update is the application of a change set to an environment, yielding a new environment.

Definition 7.6 (Update)**.** *An update applying a change set δ to an environment ε is denoted by $\varepsilon' = \varepsilon \otimes \delta$ (alternatively: $\varepsilon \Rightarrow_\delta \varepsilon'$), where $\otimes$ maps environments ε and change set δ to resulting environments ε' such that*

$$CU_{\varepsilon'} = (CU_\varepsilon \cup CU_\delta^+) \setminus CU_\delta^-, \tag{7.6}$$
$$F_{\varepsilon'} = (F_\varepsilon \cup F_\delta^+) \setminus F_\delta^-, \tag{7.7}$$
$$T_{\varepsilon'} = (T_\varepsilon \cup T_\delta^+) \setminus T_\delta^-. \tag{7.8}$$

Change sets can be used to express operations of interest on environments. We call these operations *actions*. In particular, we are interested in the addition and removal actions for environment elements, as well as actions for changing connections between CUs and channels.

TFs are identified by a unique tid and describe a function $f(x_1, \ldots, x_n, \mathcal{S})$ from inputs and current state to an output and resulting state. They are further annotated with tags in Tag for the inputs and the output. Common actions affecting TFs in a computational environment are *register* and *deregister*.

Definition 7.7 (Register action)**.** *The register action covers the class of change sets defined by the function*

$$register(\varepsilon, tid, f, \boldsymbol{itag}, otag) = (\varnothing, \varnothing, \{\langle tid, f, \boldsymbol{itag}, otag\rangle\}, \varnothing, \varnothing, \varnothing). \tag{7.9}$$

Definition 7.8 (Deregister action)**.** *The deregister action covers the class of change sets defined by the function*

$$deregister(\varepsilon, tid) = (\varnothing, \varnothing, \varnothing, F, \varnothing, \varnothing), \tag{7.10}$$

where $F = \{\langle tid, \_, \_, \_\rangle \in F_\varepsilon\}$ and _ represents a wildcard.

Targets are composed of a (query) identifier, tag, similarity relation, and a specified channel. Like TFs, targets can be added and removed by the *query* and *release* actions.

Definition 7.9 (Query action)**.** *The query action covers the class of change sets defined by the function*

$$query(\varepsilon, qid, tag, chan) = (\varnothing, \varnothing, \varnothing, \varnothing, \{\langle qid, tag, chan\rangle\}, \varnothing). \tag{7.11}$$

Definition 7.10 (Release action)**.** *The release action covers the class of change sets defined by the function*

$$release(\varepsilon, qid) = (\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, T), \tag{7.12}$$

where $T = \{\langle qid, \_, \_\rangle \in T_\varepsilon\}$ *and* _ *represents a wildcard.*

Like TFs and targets, CUs can also be added and removed. However, unlike with TFs and targets, existing CUs can be connected to and disconnected from channels as well. We therefore consider the addition and removal of CUs to be two actions in addition to the connecting and disconnecting of existing CUs. Adding and removing CUs is represented by the *spawn* and *destroy* actions.

Definition 7.11 (Spawn action)**.** *The spawn action covers the class of change sets defined by the function*

$$spawn(\varepsilon, cid, tid, S) = (CU, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing), \tag{7.13}$$

where $CU = \left\{\left\langle cid, tid, [\text{none}, \ldots, \text{none}]^T, \text{none}, \mathcal{S}\right\rangle\right\}$.

Definition 7.12 (Destroy action)**.** *The destroy action covers the class of change sets defined by the function*

$$destroy(\varepsilon, cid) = (\varnothing, CU, \varnothing, \varnothing, \varnothing, \varnothing), \tag{7.14}$$

where $CU = \{\langle cid, \_, \_, \_, \_\rangle \in CU_\varepsilon\}$ *and* _ *represents a wildcard.*

The spawn action thus adds a CU with a provided state to account for e.g. parameters. Since CUs encode their own connections to channels, the removal of a CU implicitly breaks any connections to channels. When the spawn action is applied, a CU is added such that all of its connections are set to none by default. This initial state can then be altered by using the *connect* and *disconnect* actions, for each of which we have to consider two variants to distinguish between inputs and output.

Definition 7.13 (Connect action)**.** *The connect action covers the class of change sets defined by the functions*

$$connect_{\downarrow}(\varepsilon, cid, i, chan) = (CU^+, CU^-, \varnothing, \varnothing, \varnothing, \varnothing), \tag{7.15}$$

where CU^+ *and* CU^- *are defined for every* $\langle cid, tid', \boldsymbol{in}', out', \mathcal{S}' \rangle \in CU_\varepsilon$ *as*

$$CU^+ = \left\{ \left\langle cid, tid', \begin{bmatrix} \vdots \\ in'_{i-1} \\ chan \\ in'_{i+1} \\ \vdots \end{bmatrix}, out', \mathcal{S}' \right\rangle \right\}, \tag{7.16}$$

$$CU^- = \left\{ \langle cid, tid', \boldsymbol{in}', out', \mathcal{S}' \rangle \in CU_\varepsilon \right\}, \tag{7.17}$$

and its outgoing variant

$$connect_\uparrow(\varepsilon, cid, chan) = (CU^+, CU^-, \varnothing, \varnothing, \varnothing, \varnothing), \tag{7.18}$$

where CU^+ *and* CU^- *are defined for every* $\langle cid, tid', \boldsymbol{in}', out', \mathcal{S}' \rangle \in CU_\varepsilon$ *as*

$$CU^+ = \left\{ \langle cid, tid', \boldsymbol{in}', chan, \mathcal{S}' \rangle \right\}, \tag{7.19}$$

$$CU^- = \left\{ \langle cid, tid', \boldsymbol{in}', out', \mathcal{S}' \rangle \in CU_\varepsilon \right\}. \tag{7.20}$$

Definition 7.14 (Disconnect action)**.** *The disconnect action covers the class of change sets defined by the functions*

$$disconnect_\downarrow(\varepsilon, cid, i) = connect_\downarrow(\varepsilon, cid, i, \text{none}), \tag{7.21}$$

$$disconnect_\uparrow(\varepsilon, cid) = connect_\uparrow(\varepsilon, cid, \text{none}). \tag{7.22}$$

Actions are useful to concisely describe common change sets, and will be used later as part of a reconfiguration algorithm.

Cost and optimality

While there may be many different environments that would satisfy a target, not all such environments are equally preferred. This is due to the costs associated with the run-time expenses of maintaining such a resulting environment, and the one-time expense of applying the change set that yields such a resulting environment. We refer to the cost of maintaining a CU as *upkeep*. Likewise, the cost of instantiating a CU is called *labour*. While labour is a one-time cost, upkeep accumulates over time.

The measured labour and upkeep are represented by functions from environments or change sets to cost. These global cost measures are obtained from the individual CUs.

Definition 7.15 (Labour)**.** *Labour is the observed non-negative cost of performing an update* $\varepsilon \otimes \delta$ *and is equal to*

$$labour(\delta) = \sum_{cu \in (CU^+ - CU^-)} labour(tid(cu)). \tag{7.23}$$

Definition 7.16 (Upkeep)**.** *The run-time cost of an environment $\varepsilon = \langle CU, F, T, \sim \rangle$ is referred to as upkeep. Upkeep represents the observed non-negative run-time cost for one time-unit and is calculated as*

$$upkeep(\varepsilon) = \sum_{cu \in CU_\varepsilon} upkeep(cid(cu)). \tag{7.24}$$

Labour and upkeep can be used to represent the cost of change sets and environments. This is useful when we wish to compare the costs of different (alternative) updates. We will make use of estimators $\widehat{labour}$ and $\widehat{upkeep}$ to represent the estimated rather than measured labour and upkeep of change sets and environments.

A computational environment may become invalid or suboptimal as the result of updates. This may for example happen due to changing operational costs associated with CUs (upkeep), CUs may crash and require replacing, transformations may become unavailable rendering their CU instances invalid, or new transformations may become available for a lower cost. In order to maintain adaptive semantic subscriptions, the problem is to find a change set such that, when applied to an environment, the resulting environment is valid and update is optimal.

Definition 7.17 (Validity)**.** *An environment ε is valid, denoted by $\varepsilon \in$ Valid, iff for every CU:*

1. *there exists an associated TF in F_ε;*
2. *for every identifier in_i there exists a CU in CU_ε for every $1 \leq i \leq n$, i.e. no subscriptions to* none*;*
3. *for every target $\langle qid, tag, chan \rangle$ in T_ε, there exists a CU with an associated TF such that $tag \sim_\varepsilon otag$; and*
4. *$itag_i \sim_\varepsilon otag$ holds for every connected pair of CUs.*

We exclude change sets that yield an invalid environment when used in an update. This reduces the number of applicable change sets to just those that yield environments that satisfy all targets. A pragmatic relaxation is to also allow for change sets that satisfy some targets, if it is not possible to satisfy all targets.

By combining validity with the estimators for labour and upkeep, we obtain a cost estimator that takes into account whether the resulting environment is valid. A value `MAX_COST` is used to represent an upper limit on the cost of an update. For updates yielding invalid environments, this is represented by a cost exceeding `MAX_COST`.

Definition 7.18 (Cost)**.** *The cost estimator $\widehat{cost}$ combining estimators $\widehat{upkeep}$ and $\widehat{labour}$ is defined as*

$$\widehat{cost}(\varepsilon, \delta, H) = \begin{cases} \widehat{labour}(\delta) + H \times \widehat{upkeep}(\varepsilon \otimes \delta), & \text{if } \varepsilon \otimes \delta \in \text{Valid}, \\ \texttt{MAX\_COST} + 1, & \text{otherwise}. \end{cases} \tag{7.25}$$

The cost estimator is used for determining the estimated cost of updates. An *optimal* update is one that minimises the estimated cost of applying a change set and the estimated upkeep over a predetermined horizon. It makes use of the cost estimator and excludes updates that exceed the maximum cost, for example due to being absent from Valid.

Definition 7.19 (Optimality)**.** *An update $\varepsilon' = \varepsilon \otimes \delta^*$ is optimal relative to a horizon of H time-units iff $\delta^* \in \Delta^*$, where*

$$\Delta^* = \arg\min_{\delta} \widehat{cost}(\varepsilon, \delta, H) \quad \text{subject to } \widehat{cost}(\varepsilon, \delta, H) \leq \texttt{MAX\_COST} \tag{7.26}$$

for cost estimator $\widehat{cost}$ and upper bound `MAX_COST`*.*

Note that there may be many optimal change sets, in which case any can be chosen. Alternatively, if no change set can make the resulting environment valid, there are no optimal change sets. The choice of horizon determines how conservative change sets are; if the horizon is large, upkeep starts to outweigh labour more than in cases where the horizon is kept short. Different estimators can be used, ranging from simplistic constant values to advanced predictive models whose accuracy is used to increase or decrease the length of the next horizon.

7.4 Ontology-based model representation

The formal model for stream reasoning frameworks allows us to precisely describe system configurations in terms of environments, and the change sets that can be applied to those environments. However, different realisations of this type of framework may use different internal representations. This can lead to situations wherein two different realisations based on the same formal model use two different representations. Such inconsistencies can lead to difficulties if the two are expected to interoperate.

DyKnow ontology

Semantic Web technologies were used to generate a *DyKnow ontology*. The Semantic Web was initially proposed by Berners-Lee et al. (2001) as an approach to making the World Wide Web machine-readable so that concepts could be formalised and exchanged, making it a good candidate to realise semantic interoperability. The Web Ontology Language (OWL) was described by the W3C in McGuinness et al. (2004), and was designed to describe such ontologies. Ontologies in the Semantic Web are based on Description Logic (DL), which makes it possible to perform inference on them to obtain indirect knowledge. The DyKnow ontology describes the concepts presented as part of the formal model, as well as the relations that exist between these concepts. A concept hierarchy is shown in Figure 7.1, and a more detailed

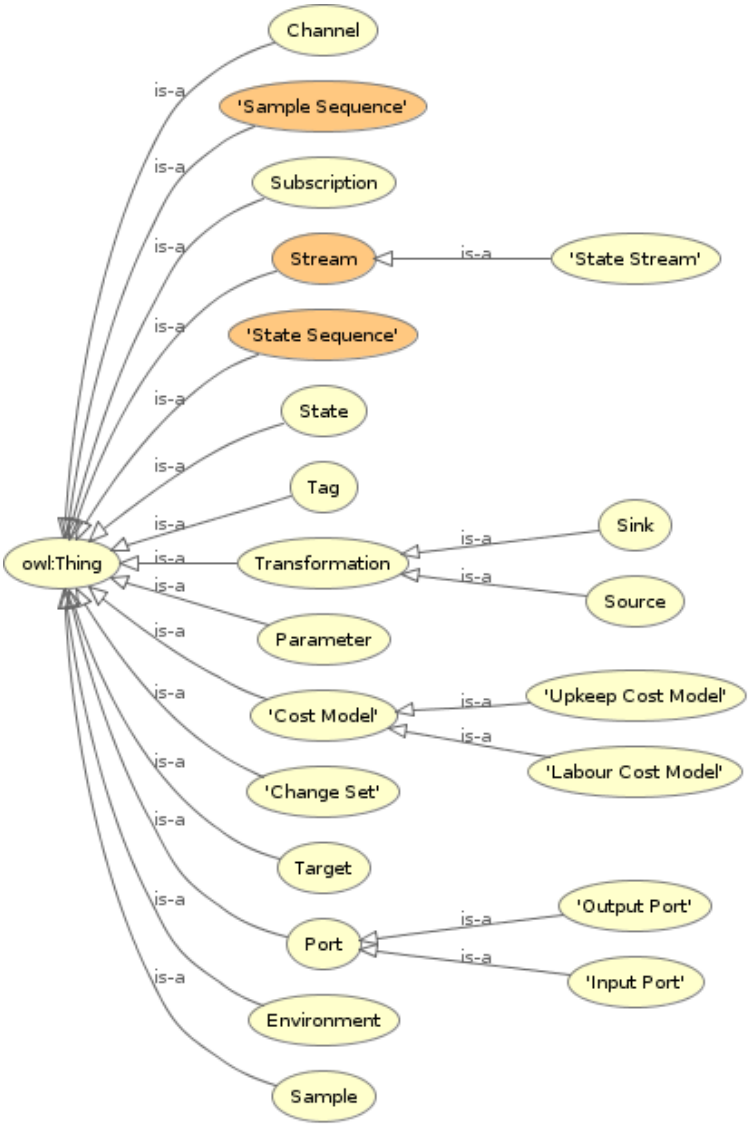

Figure 7.1: Hierarchical concept graph of the DyKnow ontology.

description of the ontology is presented in Appendix A using Manchester syntax for human readability.

The ontology formalises concepts such as CUs and the transformations they are instances of. For example, the dyknow:Transformation concept is defined in DL as

$$\begin{aligned}\text{Transformation} \sqsubseteq &\exists \text{hasName.xsd:Name} \\ &\sqcap \exists \text{hasCostModel.LabourCostModel},\end{aligned} \tag{7.27}$$

where

$$\text{LabourCostModel} \sqsubseteq \text{CostModel}. \tag{7.28}$$

dyknow:Transformation objects can further have input and output ports using the dyknow:hasInPort and dyknow:hasOutPort relations. The name of a dyknow:Transformation object then corresponds to a tid; the relations to dyknow:Port objects are used for $itag_1, \ldots, itag_n$ and $otag$; and the $cost$ is represented by a dyknow:LabourCostModel.

CUs are also encoded in the ontology with the dyknow:ComputationUnit concept;

$$\begin{aligned}\text{ComputationUnit} \sqsubseteq &\exists \text{hasName.xsd:Name} \\ &\sqcap \exists \text{hasCostModel.UpkeepCostModel}.\end{aligned} \tag{7.29}$$

CUs can be connected via a dyknow:Subscription, which is defined as

$$\begin{aligned}\text{Subscription} \sqsubseteq &\exists \text{fromCU.ComputationUnit} \\ &\sqcap \exists \text{fromPort.OutPort} \\ &\sqcap \exists \text{toCU.ComputationUnit} \\ &\sqcap \exists \text{toPort.InPort} \\ &\sqcap \exists \text{hasChannel.Channel},\end{aligned} \tag{7.30}$$

meaning that a dyknow:Subscription must have some input and output port, as well as some input and output CU. Further, it is associated with a dyknow:Channel, which is used to represent the transportation mechanism over which streams can flow from CU to CU. These channels are only required to have some name, i.e.

$$\text{Channel} \sqsubseteq \exists \text{hasChannelName.xsd:string}. \tag{7.31}$$

The semantic representation thus matches the formal definition of computation graphs, and adds additional concepts (i.e. channel) that are necessary for realisations of the formal model.

Finally, targets are represented using the dyknow:Target concept;

$$\begin{aligned}\text{Target} \sqsubseteq &\exists \text{hasName.xsd:Name} \\ &\sqcap \exists \text{hasChannel.Channel} \\ &\sqcap \exists \text{hasTag.Tag}.\end{aligned} \tag{7.32}$$

Targets are thus also extended with a channel over which the resulting stream is expected. The dyknow:hasTag connects dyknow:Tag objects to a dyknow:Target object. The dyknow:Tag objects are in turn connected to semantic descriptions with the dyknow:hasTagDescription relation.

CUs, transformations and targets can be associated with dyknow:Environment objects to clearly distinguish between different environments. This makes it possible for a knowledge base to represent not just a representation of a local environment, but also that of external environments, for example on different platforms. Configuration information can further be exchanged using a common vocabulary, allowing agents to interpret configurations of other agents and to share them in a multi-agent system. Furthermore, different realisations of the formal model for stream reasoning frameworks can use and extend the ontology while retaining interoperability. For example, the dyknow:Channel concept does not specify a specific transportation mechanism.

Because ontologies in OWL are based on DL, we can apply inference to the ontological data. This makes it possible to obtain implicit information from explicit information. One example of a potentially useful property is the transitive dyknow:dependsOn object property, which is defined by

$$\mathsf{dependsOn} \sqsubseteq \mathsf{hasSubscription} \circ \mathsf{fromCU}. \tag{7.33}$$

The dyknow:dependsOn relation for a given CU will connect it to all other CUs down the subscription pipeline. A reasoner can be used to infer these relations for every CU, such that the relations do not have to be provided explicitly, reducing the size of the populated ontology. This makes it possible to easily obtain for some CU all CUs it depends on, which can be useful for example when removing a CU to check for broken dependencies.

Ontological extensions

The DyKnow ontology thus provides a tool to support semantic interoperability between different realisations of the formal model for stream reasoning frameworks, even when these realisations make use of different internal representations of environments. A key observation is that the DyKnow ontology is designed to be extendible for purposes of realising the DyKnow model. These extensions can be performed in different ways while retaining a cross-compatible representation. One could thus see the DyKnow ontology as a *top-level ontology*. There are two sets of expected extensions for the DyKnow ontology: *system realisations* and *annotation language realisations*.

System realisations. The first category for ontological extensions deals with the realisation of the DyKnow model into a concrete system. In this case, concepts such as Channel or Transformation need an application-specific conceptualisation. These conceptualisations are more specific than the general concepts described in the DyKnow ontology. For example, while a channel is assumed to have an identifier, the

DyKnow model does not put any constraints on what this identifier may look like, whereas a specific realisation might do so. Likewise, transformations may be realised as programs, resulting in more specific properties.

Annotation language realisations. The second category deals with the realisation of languages to annotate transformations or targets. These annotations are conceptualised by the DyKnow ontology using the Tag concept. A tag could be many things. For example, a tag may simply be a simple string of text, or it might be something more specific such as logical propositions or ontological concepts.

Different realisations can thus be represented using ontological extensions of the DyKnow ontology, as demonstrated later. Different realisations however still understand the high-level conceptualisations; a channel is a channel regardless of how it is implemented. This makes it possible for different realisations of DyKnow to remain compatible. While a multi-agent approach is beyond the scope of this dissertation, the ontology serves as an important starting point for multi-agent support.

7.5 Summary

In many stream reasoning application domains, and especially in the case of robotic systems, information enters the system at a low level of abstraction, for example as raw sensor observations. Generating a high-level information stream requires the ability to reason about one's own stream refinement capabilities. This chapter formalised the stream reasoning framework's computational environment as the DyKnow model. It does so by considering targets for formula symbols, abstract transformations, concrete CUs, and channels connecting CUs. The model can be represented relative to a Semantic Web ontology, allowing other (heterogeneous) systems to reason about a system's internal configuration.

Chapter

8 Reasoning about perturbations

Sometimes the context of a stream reasoning system may change. This is especially true for systems which are expected to run for extended periods of time. In those situations, it is possible for system components, both hardware and software, to fail. Conversely, it is possible for new and improved (external) services to become available. Being able to cope with the loss (and capitalise on the becoming available) of services is an important ability. We call reasoning about such changes *reasoning about perturbations*. This chapter borrows from and extends previous work on configuration modelling and planning (de Leng and Heintz, 2015b,a, 2017).

8.1 Introduction

During the run-time of a stream reasoning system, it is possible for the environment to change outside of its own control. We call these changes *perturbations*, which can be represented in terms of change sets. Some perturbations can be relatively harmless; for example, a transformation that is currently not in use could be deregistered. Worse would be the case wherein a transformation for which CUs exist is deregistered. In such a case, the behaviour of those CUs becomes undefined, and they therefore require removal. Furthermore, the loss of these CUs can leave holes in the computation graph, leaving the environment invalid. In yet another example, a CU could crash and thereby be removed from the computation graph, resulting in similar potential problems. These last examples are clear cases wherein a perturbation results in an expensive and suboptimal environment. Less clear cases are those wherein new transformations become available. A new transformation could be cheaper to use than the transformations currently in use by an environment, but making this change is not critical.

In this chapter, we consider a formal definition of perturbations, and present procedures for correctly handling these perturbations. Specifically, whenever a perturbation occurs, the stream reasoning system needs to make changes to its configuration. Since many such changes may be correct, this becomes an optimisation problem wherein the cost of a change to correct a perturbation is minimised. The proposed procedures are any-time algorithms, allowing the stream reasoning system to additionally choose how much resources it allocates to the perturbation handling process.

8.2 Perturbation handling

A perturbation can be defined as a change set which was not expected by the stream reasoning system. Formally, the definition of a perturbation is as follows:

Definition 8.1 (Perturbation)**.** *We can consider different types of perturbations denoted by δ_p. Short-term negative perturbations*[11] *result in an immediate cost increase (compared to no change) when considering an equal horizon H:*

$$cost(\varepsilon, \delta_p, H) > cost(\varepsilon, \delta_\varnothing, H). \tag{8.1}$$

When the cost does not change as the result of δ_p, it is considered to be a short-term neutral perturbation. Similarly, long-term positive perturbations make possible an update that would result in a cost decrease, i.e.

$$\exists \delta^* [cost(\varepsilon \otimes \delta_p, \delta_\varnothing, H) > cost(\varepsilon \otimes \delta_p, \delta^*, H)], \tag{8.2}$$

with (inversely) long-term neutral perturbations lacking such an update. Different perturbations can thus have different effects in the short and long term.

To handle both the short and long term repercussions of perturbations, semantic subscriptions are periodically evaluated and updated to repair or improve the underlying environment. This recurring process is referred to as the *configuration life-cycle*. The life-cycle is composed of a number of phases which are repeated every cycle, which starts with a *review interval* followed by a *stable interval*.

The purpose of the review interval is to reflect on the preceding stable interval (if any) and to improve the environment configuration. During this interval, a stream reasoning manager searches for a change set such that its application to the current environment constitutes an optimal update. Whether an update is optimal is determined by a combination of labour and cumulative upkeep relative to a horizon. If an optimal update is found (i.e. $\Delta^* \neq \varnothing$), it is then applied; otherwise the environment remains unchanged (i.e. $\delta^* = \delta_\varnothing$). During the application of an update, the labour costs are measured and used to update the labour estimator $\widehat{labour}$. The review interval is then succeeded by a new stable interval.

[11] *Short-term positive perturbations* are generally ignored as they would require an outside force to for example remove a target together with any CUs that would no longer be necessary.

Once the update produced during the review interval has been performed, the stable interval begins. The purpose of the stable interval is to maintain uninterrupted streams that satisfy targets, while monitoring the upkeep of the environment to update the $\widehat{upkeep}$ estimator. The stable interval ends when one of two events occur: (1) if a short-term negative perturbation is detected, the review interval is started immediately in order to mitigate the increase in cost induced by such a perturbation; and (2) if the horizon is reached, the review interval is started as scheduled in order to check for possible improvements as the result of any long-term positive perturbations that occurred during the stable interval.

8.3 Update procedure

Whenever the review interval is started, we search for and apply an optimal update if one exists. We denote δ_p to represent the perturbation that started to review cycle, if one exists; otherwise $\delta_p = \delta_\varnothing$. It is applied to a previous environment ε_{-1} to yield the current environment $\varepsilon_0 = \varepsilon_{-1} \otimes \delta_p$. The challenge is to find an optimal update δ^* to mitigate any suboptimality induced by δ_p, yielding the next environment[12] $\varepsilon_1 = \varepsilon_0 \otimes \delta^*$. This is done through a three-step approach shown below.

Exploration

The procedure for reconfiguration is shown in Algorithm 8.1. Nodes represent CUs-to-be that should become part of the resulting environment. The `EXPLORE` procedure first generates a root node which is a placeholder that is used to represent the targets (line 7). For example, if there are three targets, the root node will be a ternary node such that the tags for every input correspond to the tags of the targets, and the ports correspond to the desired ports of the targets. The task of `EXPLORE` is to build a valid computation graph starting from the root node. To do so, it will need to expand nodes in the graph with children satisfying that node's inputs. The combination of a node and an input index is therefore called a *job*. Jobs are kept track of as part of the $openJobs$ stack (line 3), and updated when necessary. The choices made while building the graph are likewise stored in the $trace$ stack (line 4).

The procedure runs by sequentially considering every job in $openJobs$ and calls the `EXPAND` procedure on these nodes (lines 11–23). If the `EXPAND` procedure succeeds, any new children have their inputs added to $openJobs$. Sometimes `EXPAND` will find an existing node. In that case it has already been expanded as the result of the DFS approach, and does not need its inputs added as jobs. Whenever `EXPAND` fails, the failing job is returned to $openJobs$ and backtracking is applied (lines 24–39). `EXPAND` can fail when all candidates for expansion have been exhausted, either due to having been attempted already, or because they result in the graph's

[12] The perturbation $\varepsilon_{-1} \otimes \delta_p$ is thus similar to the game-theoretical *move by nature*.

Algorithm 8.1: Exploration procedure

```
1  function EXPLORE(Environment ε, ChangeSet δ_p):
2  registry ← new Map()
3  openJobs ← new Stack()
4  trace ← new Stack()
5  bestTrace ← new Stack()
6  bestCost ← ∞
7  Node root = new Node(createRoot(ε))
8  running ← true
9  while running do
10     expansionFailure ← false
11     while |openJobs| > 0 ∧ ¬expansionFailure do
12         Job job ← openJobs.pop()
13         Node next ← registry[job.tid]
14         if EXPAND (next, trace, registry, ε, δ_p, bestCost) then
15             if ¬next.virtual[job.port] then
16                 Add children to openJobs
17             end
18             Reset candIndex for all jobs in openJobs
19         else
20             expansionFailure ← true
21             openJobs.push(job)
22         end
23     end
24     if |trace > 0| then
25         (from ⇒_i to, cost) ← trace.pop()
26         if ¬expansionFailure ∧ bestCost > cost then
27             bestTrace ← trace ∪ (from ⇒_i to, cost)
28             bestCost ← cost
29         end
30         registry[from].children[i] ← nil
31         registry[from].virtual[i] ← false
32         if ⇒=→ then
33             registry[to] ← nil
34             Remove invalidated jobs from openJobs
35         end
36         openJobs.push(new Job(from, i))
37     else
38         running ← false
39     end
40 end
41 return COMPILE (bestTrace, ε)
```

cost exceeding the current best cost. When backtracking is performed, the last action stored in $trace$ is reverted and a corresponding job as added. This will cause EXPAND to try a different candidate. For every valid graph, we check whether it is better than the currently best solution, and if so we replace it. Once no more back-

Algorithm 8.2: Node expansion

```
1  function EXPAND(Node node, var i, Stack trace, Map registry, Environment ε,
     ChangeSet δ_p, var bestCost):
2  node.children[i] ← nil
3  node.virtual[i] ← false
4  node.expanded ← false
5  while ¬expanded ∧ candIndex[i] < numCandidates(node.tid, ε, i) do
6      candidateTID ← getCandidate(node.tid, ε, δ_p, candIndex[i])
7      (from ⇒_i to, sumCost) ← trace.peek()
8      cost ← cost(candidateTID)
9      if registry[candidateTID] = nil then
10         if sumCost + cost < bestCost then
11             Node child ← new Node(candidateTID)
12             node.children[i] ← child
13             node.virtual[i] ← false
14             registry[candidateTID] ← child
15             trace.push((tid →_i candidateTID, sumCost + cost))
16             Reset inputs succeeding i
17             node.expanded ← true
18         end
19     else
20         node.children[i] ← registry[candidateTID]
21         node.virtual[i] ← true
22         trace.push((tid ⇝_i candidateTID, sumCost))
23         node.expanded ← true
24     end
25     candIndex[i] ← candIndex[i] + 1
26 end
27 return expanded
```

tracking is possible, we use the best trace and convert it into a change set using the `COMPILE` procedure (line 41).

Expansion

The `EXPAND` procedure is described in greater detail in Algorithm 8.2. The procedure is applied to a specific node and attempts to find a valid child node for a specified input index i. The corresponding *action* taken is then added to the $trace$. Actions can represent the spawning of and connecting to new CUs ($\rightarrow$), or the reusing of nodes ($\rightsquigarrow$) that were previously added to the exploration graph as part of the current call to `EXPLORE`.

Every node keeps track of which candidates it has thus far considered for expansion for every input index. This is done by maintaining a $candIndex$ array of candidate indices, where each index corresponds to the next candidate to be attempted for that input index. The procedure attempts successive candidates until it

Algorithm 8.3: Compilation procedure

```
1  function COMPILE(Stack trace, Environment ε):
2  δ ← (∅, ∅, ∅, ∅, ∅, ∅)
3  removalSet ← CU_ε
4  channelMap ← new Map()
5  cidMap ← new Map()
6  foreach (from ⇒_i to, cost) ∈ trace do
7      if to.cid ∈ removalSet then
8          removalSet \ {to.cid}
9      end
10     if ⇒ = → then
11         chan ← getUniqueID()
12         channelMap[to] ← chan
13         cid ← getUniqueID()
14         cidMap[to.tid] ← cid
15         δ ← δ ∪ spawn(ε, cid, to.tid, ∅) ∪
             connect↓(ε, cidMap[from.tid], i, chan) ∪ connect↑(ε, cid, chan)
16     else
17         chan ← channelMap[to]
18         δ ← δ ∪ connect↓(ε, cidMap[from.tid], i, chan)
19     end
20 end
21 foreach cid ∈ removalSet do
22     δ ← δ ∪ destroy(ε, cid)
23 end
24 return δ
```

either finds one that works, or runs out of candidates for input index i (lines 5–26). Specifically, the procedure considers transformations $candidateTID$ and checks if they occur in the $registry$ map, which maps TIDs to nodes. If a node already exists for $candidateTID$, it is reused (lines 20–23), i.e. a virtual connection. Otherwise, a new node is created iff this does not result in the cost exceeding the best cost (lines 10–18), i.e. a new connection. Finally, the procedure returns whether expansion was successful or not (line 27).

Change set compilation

The `COMPILE` procedure described in Algorithm 8.3 constructs a change set δ^* from the trace produced by `EXPLORE` in conjunction with `EXPAND`. The change set is first composed of a number of spawn and connect actions. Spawn actions are called for nodes representing new CUs (line 15). A newly spawned CU then needs to be connected to a channel for its output port. The same channel is used to connect the input port of the receiving CU to. For connections to existing CUs, only the receiving CU needs to be connected with its input port. In these cases, the pre-existing channel is used to connect to. Finally, any CUs existing in the original environment that

do not occur in the trace are scheduled for destruction. This ensures that CUs that are not in use do not linger and therefore do not accumulate upkeep.

Finding an optimal change set

To better illustrate how the three procedures interact, the following example illustrates two key scenarios. In the first, the environment is completely empty as it would be when the system is first started, and a perturbation populates the environment for the first time. The second case deals with perturbations that negatively impact the environment and which must be resolved to guarantee semantic subscriptions are maintained. As an example, consider for a horizon $H = 10$ an environment $\varepsilon = \langle \varnothing, F, T, \sim \rangle$ such that

$$\begin{aligned} F = \{ & \langle tid_1, f_1(x), [A], B \rangle, \\ & \langle tid_2, f_2(x), [C], D \rangle, \\ & \langle tid_3, f_3(), [\,], E \rangle, \\ & \langle tid_4, f_4(), [\,], F \rangle, \\ & \langle tid_5, f_5(), [\,], G \rangle \}; \end{aligned} \tag{8.3}$$

$$\begin{aligned} T = \{ & \langle qid_1, B, 101 \rangle, \\ & \langle qid_2, D, 102 \rangle \}, \end{aligned} \tag{8.4}$$

and the similarity relation is reflexive and further includes $A \sim E$, $A \sim F$, $B \sim F$, and $B \sim G$. We thus have an environment in which no CUs are active, five transformations are registered, and two targets are registered. We will assume that the cost estimators yield 1.0 labour and 1.0 upkeep for each of the five transformations. Additionally, the perturbation is described by

$$\delta_p = (\varnothing, \varnothing, \varnothing, \varnothing, T, \varnothing), \tag{8.5}$$

meaning that the disturbance is the registration of the targets T for example by a human operator. Since δ_p is a short-term negative perturbation ($\infty > 0$), the `EXPLORE`(ε, δ_p) procedure described in Algorithm 8.1 is called to mitigate the perturbation-induced cost increase.

After initialising the stacks and map, the root node is created. This root node is based on the set of targets T and is represented by a placeholder transformation $\langle root, \varnothing, [D, E], \text{none} \rangle$. The first call to `EXPAND` is done on this root node with an empty trace and a bestCost value corresponding to ∞. The expansion is performed in a depth-first manner, starting with the first input of the root node. The candidates are determined by the similarity relation $\sim$. Therefore, any transformations with an output tag equal to an input tag under consideration qualify as candidates for expansion. The first input tag of the root node is B, which is only equal to the output tag of transformation tid_1. Since tid_1 does not exist in the registry, it cannot be reused, so a new CU would have to be spawned from it. The total cost of such a CU would be 11.0; 1.0 from the labour and 10.0 from the upkeep over the length of the

horizon. Since our current cost is 0.0, adding 11.0 would not exceed the bestCost value of ∞, so the candidate is used. The trace now consists of one entry;

$$[(root \rightarrow_0 tid_1, 11.0)]. \tag{8.6}$$

The EXPAND procedure is subsequently called again for the first input index of the node for tid_1. Its input tag corresponds to A, which is similar to the output tag of tid_3 and tid_4. Maintaining the order of transformations, tid_3 is chosen first, resulting in the trace

$$\begin{aligned}[&(tid_1 \rightarrow_0 tid_3, 22.0)\,, \\ &(root \rightarrow_0 tid_1, 11.0)].\end{aligned} \tag{8.7}$$

Since tid_3 has no dependencies, the search continues with the second input of the root node, yielding tid_2 as a candidate, followed by tid_4. The first solution thus has a trace

$$\begin{aligned}[&(tid_2 \rightarrow_0 tid_4, 44.0)\,, \\ &(root \rightarrow_1 tid_2, 33.0)\,, \\ &(tid_1 \rightarrow_0 tid_3, 22.0)\,, \\ &(root \rightarrow_0 tid_1, 11.0)]\end{aligned} \tag{8.8}$$

and a cost of 44.0. The EXPLORE procedure then starts backtracking. The trace head

$$(tid_2 \rightarrow_0 tid_4, 44.0) \tag{8.9}$$

is first removed, and EXPAND is called on its head node tid_2 with a best cost of 44.0. While tid_5 is a valid candidate, its cost would be equal or greater than the best cost of 44.0, so EXPAND returns failure and backtracking continues. The next trace head is

$$(root \rightarrow_1 tid_2, 33.0)\,, \tag{8.10}$$

for which there are no alternatives, so backtracking continues further. Next is trace head

$$(tid_1 \rightarrow_0 tid_3, 22.0)\,, \tag{8.11}$$

where EXPAND is called on the tid_1 node, which does have an alternative candidate tid_4. Since picking tid_4 would not exceed the best cost, it is picked, resulting in a trace

$$\begin{aligned}[&(tid_1 \rightarrow_0 tid_4, 22.0)\,, \\ &(root \rightarrow_0 tid_1, 11.0)].\end{aligned} \tag{8.12}$$

The EXPAND procedure returns success, but EXPLORE still has the root node as an open job to reflect the backtracking which removed its subgraph at its second input, so EXPAND is called on the root node. Due to the change from tid_3 to tid_4 earlier

in the trace, the root node is allowed to pick tid_2 as its candidate again. Expansion of the tid_2 node subsequently yields tid_4 and tid_5 as candidates. Adhering to the ordering, tid_4 is chosen first. This time, tid_4 already exists in the registry, so it can be reused for free, resulting in the trace

$$\begin{aligned}[&(tid_2 \rightsquigarrow_0 tid_4, 33.0)\,, \\ &(root \rightarrow_1 tid_2, 33.0)\,, \\ &(tid_1 \rightarrow_0 tid_4, 22.0)\,, \\ &(root \rightarrow_0 tid_1, 11.0)]\end{aligned} \tag{8.13}$$

and a cost of 33.0. After this point, no better solutions are found, and backtracking exhausts the trace.

The `EXPLORE` procedure then returns the result of applying the `COMPILE` procedure to the trace given the environment ε. This procedure runs through the trace, starting at the bottom of the stack, choosing unique channels and CU identifiers. The first item is $root \rightarrow_0 tid_1$, which requires the spawning of a CU of type tid_1 and its subsequent connection to the desired target channel 101. A unique channel is randomly chosen for its input; we will assume it is channel 1. This is followed by the spawning of a CU of type tid_4, the output for which is connected to channel 1, and which has no inputs. Then follows the spawning of a CU of type tid_2 whose output is connected to channel 102 as determined by the second target, and whose input channel us chosen to be channel 1 as it shares the source CU of type tid_4. The resulting change set then becomes $\delta^* = (CU^+, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing)$, where

$$\begin{aligned}CU^+ = \{\, &\langle cid_1, tid_1, [1], 101, \varnothing\rangle\,, \\ &\langle cid_2, tid_4, [\,], 1, \varnothing\rangle\,, \\ &\langle cid_3, tid_2, [1], 102, \varnothing\rangle\}.\end{aligned} \tag{8.14}$$

The update $\varepsilon' = \varepsilon \otimes \texttt{EXPLORE}(\varepsilon, \delta_p)$ thus yields a resulting environment

$$\varepsilon' = \left\langle CU^+, F_\varepsilon, T_\varepsilon, \sim_\varepsilon \right\rangle. \tag{8.15}$$

When this update is performed, the estimators for $\widehat{labour}$ are updated based on the observed resource usage associated with instantiating transformations.

The update above marks the end of the review interval and the start of the stable interval. The stable interval normally has a duration equal to the horizon length, during which the estimators for $\widehat{upkeep}$ are updates based on the observed resource usage of CUs. Short-term negative perturbations could however cut this duration short. To better illustrate the adaptivity of semantic subscriptions, we will assume that such a perturbation indeed occurs.

For the duration from the review cycle's completion to the perturbation, the targets qid_1 and qid_2 ensured that there would be a stream sent over channels 101 and 102, for which the semantics are described by the tags B and D respectively. The occurrence of the perturbation jeopardises these streams. In the worst case,

no more samples are sent out on the channels, effectively freezing the streams. The premature termination of the stable interval and start of the review interval is meant to quickly mitigate this problem. We will assume that the perturbation corresponds to the crash of a CU, illustrated by

$$\delta_p = (\varnothing, \{\langle cid_3, tid_2, [1], 102, S\rangle\}, \varnothing, \varnothing, \varnothing, \varnothing). \tag{8.16}$$

The perturbation δ_p encodes the fact that the CU of type tid_2 and with identity cid_3 has been removed from the environment. This puts a hole in the computation graph, as streaming data from CU cid_2 sent over channel 1 is no longer processed, nor is the stream that would have resulted from that processing sent over channel 102 to satisfy target qid_2. Furthermore, the estimators for labour and upkeep have been updated since our previous optimal update, and the upkeep cost of CU cid_2 is determined to be 3.0 per time-unit instead of the estimated 1.0 per time-unit.

The `EXPLORE`$(\varepsilon' \otimes \delta_p, \delta_p)$ procedure is run to obtain an optimal update that will cost-efficiently resume the data stream on channel 102. The process is the same as before, except that now we can use CUs from the environment $\varepsilon'' = \varepsilon' \otimes \delta_p$, which are prioritised over spawning new CUs from transformations. One advantage of reusing CUs is that no labour cost is acrued. This leads to an initial trace

$$\begin{aligned}[\,&(tid_2 \rightsquigarrow_0 cid_2, 51.0)\,,\\ &(root \rightarrow_1 tid_2, 51.0)\,,\\ &(cid_1 \rightarrow_0 cid_2, 40.0)\,,\\ &(root \rightarrow_0 cid_1, 10.0)]\end{aligned} \tag{8.17}$$

and a cost of 51.0; reusing cid_1 requires an upkeep of 10.0, reusing cid_2 requires an upkeep of 30.0, spawning a CU of type tid_2 requires an upkeep of 10.0 and labour equal to 1.0, and connecting this new CU to a CU we already paid for is cost-free. Unfortunately, while this solution is a quick-fix of the problem, it is not the best solution. The upkeep cost of cid_2 has increased sharply since the last review interval. Therefore, backtracking yields another solution which is optimal;

$$\begin{aligned}[\,&(tid_2 \rightarrow_0 tid_5, 43.0)\,,\\ &(root \rightarrow_1 tid_2, 32.0)\,,\\ &(cid_1 \rightarrow_0 tid_3, 21.0)\,,\\ &(root \rightarrow_0 cid_1, 10.0)].\end{aligned} \tag{8.18}$$

This way, we no longer expend resources on the upkeep of cid_2, and the one-off labour cost is insignificant with a horizon length of 10 time-units. The `EXPLORE` algorithm next calls on `COMPILE`. As before, a change set is generated which spawns and connects CUs. This time we however also destroy CU cid_2 to remove its drain on the upkeep, as it is never removed from the $removalSet$ due to not occurring in the trace. The resulting change set is therefore $\delta^* = (CU^+, CU^-, \varnothing, \varnothing, \varnothing, \varnothing)$,

where

$$CU^{+} = \{\, \langle cid_4, tid_3, [\,], 1, \varnothing \rangle\,, \quad \langle cid_5, tid_2, [2], 102, \varnothing \rangle\,, \quad \langle cid_6, tid_5, [\,], 2, \varnothing \rangle \}; \tag{8.19}$$

$$CU^{-} = \{\, \langle cid_2, tid_4, [\,], 1, S \rangle \}. \tag{8.20}$$

The application of δ^* to environment ε'' then yields a new environment

$$\varepsilon'' \otimes \delta^* = \langle CU, F_\varepsilon, T_\varepsilon, \sim_\varepsilon \rangle \tag{8.21}$$

such that the new computation graph CU is described by

$$CU = \{\, \langle cid_1, tid_1, [1], 101, S \rangle\,, \quad \langle cid_4, tid_3, [\,], 1, \varnothing \rangle\,, \quad \langle cid_5, tid_2, [2], 102, \varnothing \rangle\,, \quad \langle cid_6, tid_5, [\,], 2, \varnothing \rangle \}. \tag{8.22}$$

As can be seen from the environment $\varepsilon'' \otimes \delta^*$, we now have four CUs satisfying the two targets. This leads to the resumption of the previously-frozen stream over channel 102, fixing the problem caused by the most recent perturbation in a cost-efficient manner.

8.4 Correctness

The `EXPLORE` procedure is designed to find an optimal update if one exists, even if the original environment is invalid and therefore has a cost exceeding `MAX_COST`. Note that while there may exist different optimal updates with the same cost, only the first one found is selected; the others are pruned. In order to show the *correctness* of the `EXPLORE` procedure, it must be shown to return an optimal update.

Theorem 8.1: Correctness

The EXPLORE procedure is *correct*, meaning that for any environment ε resulting from a perturbation δ_p, and any horizon of length H, for the set of optimal change sets Δ^* defined as

$$\Delta^* = \arg\min_{\delta} \widehat{cost}(\varepsilon, \delta, H) \quad \text{subject to } \widehat{cost}(\varepsilon, \delta, H) \leq \texttt{MAX\_COST}, \tag{8.23}$$

the following implication holds:

$$\Delta^* \neq \varnothing \rightarrow \texttt{EXPLORE}(\varepsilon, \delta_p) \in \Delta^*, \tag{8.24}$$

i.e. if some optimal change sets exist, the EXPLORE procedure will return one of them.

Proof. The proof is based on Algorithms 8.1, 8.2, and 8.3. In particular, it is first shown that the exploration procedure exhaustively finds all change sets δ so that $\varepsilon \otimes \delta \in$ Valid if the guard $sumCost + cost < bestCost$ on line 10 in Algorithm 8.2 is omitted. It is then shown that the inclusion of this guard excludes suboptimal change sets, thereby returning an optimal change set if one exists.

The EXPLORE procedure performs a depth-first expansion of the root node when run for the first time. This sequence of operations is enforced by the stack of open jobs; whenever more expansions are available, they are pushed to the top of the stack. This means that when the stack is empty, no more expansions can be performed, and a complete computation graph has been found. The sequence of actions resulting in this graph is maintained as a trace stack. This allows us to backtrack by undoing actions and considering alternative candidates.

The candidates for expansion are kept track of within the nodes of the graph. Whenever we backtrack to a node, we increment the candidate index ($candIndex$) for the input index of interest. When a suitable alternative candidate is found, the trace is updated accordingly. Since a change has been made to the graph, this means we can reset all of the candidate indices of future jobs. This way we ensure that we find all valid change sets.

Now consider what happens if we reinstate the cost guard. For EXPLORE to return a suboptimal solution, either there exists no solution in Δ^* or the optimal solution is not considered in lines 9–40. The former is a contradiction with our assumption that $\Delta \neq \varnothing$. The latter can only occur if the cost guard on line 10 in EXPAND prunes away the optimal solution. Since the guard only excludes expansions that would lead to costs greater than the best cost, that would mean negative costs are necessary for this to occur. But negative costs are not allowed, so EXPAND cannot prune away the optimal solution. Therefore EXPLORE cannot return a suboptimal solution whenever Δ^* is non-empty. ■

8.5 Any-time extension

The EXPLORE procedure quickly finds a first solution (or finds that none exist), which it subsequently improves on through an exhaustive consideration of alternatives. Alternatively, the procedure could stop considering alternatives prematurely and return the best solution found thus far. Such an any-time extension of EXPLORE is useful in cases where an exhaustive search would take too long and we are willing to sacrifice the optimality of the produced change set in favour of getting a change set faster. We therefore consider a variant of EXPLORE which in addition to its usual arguments takes a value $timeout$ corresponding to the time-point after which EXPLORE stops backtracking on its trace (line 24), effectively extending the guard to $|trace| > 0 \wedge runtime \geq timeout$ where $runtime$ represents the number of time-units that passed since the procedure was started. The original correctness criterion can then be generalised to

$$\lim_{timeout \to \infty} (\texttt{EXPLORE}(\varepsilon, \delta_p, timeout)) \in \arg\min_{\delta} \widehat{cost}(\varepsilon, \delta, H) \quad \text{(8.25)}$$

$$\text{subject to } \widehat{cost}(\varepsilon, \delta, H) \leq \texttt{MAX\_COST}.$$

Given a finite value for $timeout$, it cannot be guaranteed that EXPLORE will return an optimal change set. In such a case the direction of exploration becomes a determining factor for the quality of the result. Heuristics can be used to improve the result quality by guiding the direction of exploration based on background knowledge of the search space. Specifically, the `getCandidate` procedure in EXPAND imposes a total ordering $\prec$ on the available candidates at every node. When the procedure plans to perform a spawn action, both labour and upkeep costs are acrued. When instead an existing CU is used, labour costs are eliminated. If a previously-expanded node can be used, all costs are eliminated. Therefore, the total order

$$\text{Spawn from TFs} \prec \text{Reuse CUs} \prec \text{Reuse nodes} \quad \text{(8.26)}$$

would prioritise cheap options before considering more expensive ones. A perturbation δ_p can provide further guidance by encoding cases in which CUs are destroyed. If the associated transformation was not removed, the ordering of candidate transformations can consider this transformation first, as it will likely provide an initial solution fast. Finally, additional heuristics taking into account properties of transformations and CUs can be considered, for example their cost, tags, or identifiers which imply freshness.

The any-time extension of EXPLORE also allows for the inclusion of its own runtime into the configuration cycle. The length of the horizon H is used to determine accumulated upkeep during the stable interval. In the any-time version, we can consider a *configuration cycle length*

$$H^+ = H_{review} + H_{stable} \quad \text{(8.27)}$$

$$= timeout + H \quad \text{(8.28)}$$

instead, which fixes the configuration cycle to a regular pattern.

8.6 Summary

In this chapter, we considered the case wherein a stream reasoning system is subjected to unexpected change sets, called perturbations. The ability to respond to a perturbation with a correcting change set is important in order for such a system to be robust. Likewise, sometimes transformations that are better than the ones instantiated at that time become available. In those cases, the system needs to be able to leverage the newly available transformations to reduce its computational resource usage. Therefore, an algorithm for finding optimal updates was presented in the context of a configuration life-cycle consisting of recurring review and stable intervals. The algorithm can be extended to an any-time algorithm, allowing the underlying system to additionally plan computation resource usage towards perturbation handling. This allows for a trade-off between finding an optimal solution and saving computational resources that would be expended finding such an optimal solution.

Part IV

APPLIED STREAM REASONING

Chapter

9 DyKnow-ROS

THROUGH the integration of the previously-presented techniques, an adaptive stream reasoning framework can be constructed. In this chapter an instance of such a stream reasoning framework called *DyKnow-ROS* is presented, which provides a realisation of the DyKnow model integrated with ROS. The choice of implementation is however general and could be applied to other supporting software. The chapter presents an overview of the software architecture and services by providing concrete realisations of the abstract components presented previously. Finally, the framework is empirically tested to measure overhead cost resulting from the indirection induced by DyKnow-ROS. This chapter uses and extends materials that primarily focus on extending ROS with reconfigurable subscriptions (de Leng and Heintz, 2016b), and materials that focus on semantic subscriptions for ROS (de Leng and Heintz, 2017).

9.1 Introduction

The DyKnow-ROS stream reasoning framework is an extension to ROS (Quigley et al., 2009), which is a popular robot middleware used frequently in both industry and academia. DyKnow-ROS is capable of reasoning about which streams to subscribe to and can reconfigure the system during run-time to for example generate streams required for spatio-temporal reasoning tasks.

ROS allows developers to write implementations as ROS *nodes*, which can communicate with each other by using *services* and *topics*. These nodes are combined into *packages*, of which many have been made publicly available. Topics can be used to connect nodes to establish a flow of information, which makes them the implementation counterpart to the concept of channels capable of transporting information streams. Topics are advertised by *publishers* and can be subscribed to by other nodes using *subscribers*, such that a single topic can have multiple publish-

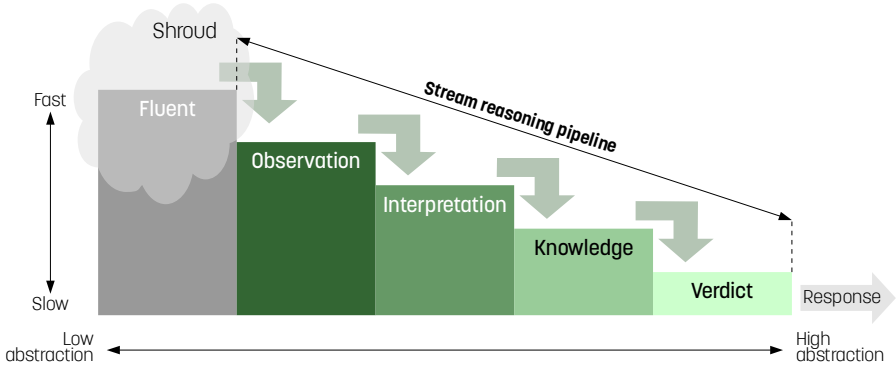

Figure 9.1: The stream reasoning waterfall model with the components within the stream reasoning pipeline range highlighted.

ers and subscribers. Services allow nodes to advertise functionality to other nodes, which can then be requested by these nodes. Services (optionally) take a number of arguments and can (optionally) return a result to the service caller. ROS uses *Node Handles* to expose its API to developers of nodes, which packages such as *Image Transport* augment to support efficient image transportation. Where standard nodes correspond to individual processes when run, *nodelets* are run on threads within a *Nodelet Manager* node. Communication between nodelets is consequently generally more efficient than communication between nodes. Further, nodes are instantiated either manually or through a launch file, whereas nodelets can also be instantiated using the Nodelet Manager's services. This makes it possible for programs to instantiate other programs at will.

In practice, most ROS-based systems rely on sometimes large collections of repeatedly nested launch files to operate. It may also be necessary for a user to run a number of launch files in a particular sequence in order for a system to function properly. It can be quite challenging to make changes to such files to accommodate new components, or to change configurations as part of a set-up phase. Clearly a lot of manual configurations may be necessary to operate a ROS-based system. This may work for small systems, but quickly becomes infeasible as systems grow to for example hundreds of robots. Depending on the application, operator errors can be very expensive. By applying the DyKnow model to ROS, DyKnow can benefit from the underlying architecture provided by ROS, while providing ROS with adaptive reconfigurability. We consider this to be an extension of core ROS features. In the performance evaluation we show that the induced overhead cost is minimal.

In this chapter, we use ROS to realise the DyKnow model and thereby realise an adaptive spatio-temporal stream reasoning application. The choice of ROS is based on the fact that it closely follows the DyKnow model, but other middleware could be used as well.

9.2 DyKnow-ROS

DyKnow-ROS is a concrete realisation of the DyKnow model based on the ROS middleware. Figure 9.1 shows the stream reasoning pipeline adopted by DyKnow-ROS highlighted. In this chapter, we focus on the integration of the various steps into a single stream reasoning framework. In particular, given a formula and a semantic interpretation of its symbols, the full system should be able to automatically generate a state stream over which it then evaluates the provided formula. Once the formula has been evaluated, the computational environment should automatically be cleaned up. This combined functionality requires implementations for a *stream reasoning manager*, a *stream reasoning engine*, a *computational environment*, and their connecting interfaces.

While ROS provides most of the transportation functionality needed to support this type of stream reasoning, its service-based interface lacks control over the way nodelets in ROS are connected. The first step towards DyKnow-ROS is therefore to extend this interface with additional services. By default, the nodelet manager provides the following services:

- **NodeletLoad:** Given a nodelet name and type, the Nodelet Manager instantiates a nodelet of that type, where the type is a reference to the nodelet's source.
- **NodeletUnload:** Given a nodelet name, the Nodelet Manager destroys that nodelet. A nodelet cannot unload itself.
- **NodeletList:** Returns an array of nodelet names.

Nodelets can thus be added, removed, and enumerated using the Nodelet Manager's services. These services are prerequisites to the $spawn$ and $destroy$ actions formally defined as part of the DyKnow model. Neither the Nodelet Manager nor nodelets however provide services that allow for subscribers and publishers to be changed at run-time. Developers are expected to specify configurations manually using launch files instead — ROS was not designed for the purpose of automatic (re)configuration.

To extend ROS with run-time reconfiguration services for subscribers and publishers in nodelets, different approaches could be taken. The architecture of DyKnow-ROS was chosen based on three factors: ease of adoption, ease of use, and minimal computational overhead. DyKnow-ROS does not require a custom version of ROS, but instead provides an optional extension through the use of add-on components that build on top of nodelets. This extension is collectively referred to as a *nodelet proxy*. This allows developers to use DyKnow-ROS in some parts of their system but not others, if they so choose. Where DyKnow-ROS replaces standard ROS components, it sticks as closely to the original interface as possible and tries to limit required changes to namespace changes. This was done to make it easy to switch from standard ROS nodelets to DyKnow-ROS CUs while retaining a familiar

interface. Lastly, DyKnow-ROS inevitably induces overhead computational costs by virtue of being an add-on layer on top of ROS. It seeks to keep this overhead minimal by keeping it relative to the degree of control granted to DyKnow-ROS. The more extra features from DyKnow-ROS are used, the larger the overhead.

9.3 The nodelet proxy

The flexibility offered by the Nodelet Manager makes it an excellent tool for dynamically reconfiguring a ROS system. As mentioned earlier, the services offered by a Nodelet Manager are limited to the loading and unloading of nodelets. DyKnow-ROS therefore complements these services with the help of persistent *nodelet proxies* that augment the ROS Node Handle. The persistent nodelet proxy is the key component that allows DyKnow-ROS to exert a greater control over the augmented nodelets, which are realisations of CUs. A developer establishes a nodelet proxy by creating a DyKnow variant of the nodelet handle instead of the usual ROS nodelet handle. Recall that the ROS nodelet handle serves as an API that can be used to call ROS functionality, such as creating publishers and subscribers. The DyKnow-ROS node handle instead delegates these calls to the nodelet proxy, which either delegates to the ROS node handle or to custom DyKnow variants depending on the functionality requested. Specifically, DyKnow-ROS provides its own publishers and subscribers that can be used in the same way as ordinary ROS publishers and subscribers. The key difference between the two lies in the indirection imposed by DyKnow-ROS. ROS publishers and subscribers connect directly to topics; a subscriber can name a topic and a callback method, whereas a publisher can name a topic and a message to be sent. The DyKnow-ROS variants instead use ports, which are in turn connected to a topic. The nodelet proxy maintains a mapping between ports and topics, and allows for this mapping to change as the result of services that are offered by the proxy. This way, ports can be associated with different topics over time, which allows for run-time reconfiguration to occur.

To illustrate the extension, a schematic of the nodelet proxy and its relation to a host nodelet is shown in Figure 9.2. The nodelet implementation by a developer is indicated by NodeletImpl, which extends ros::Nodelet. The developer is able to create a dyknow::NodeHandle, which takes a ros::NodeHandle as an argument. The dyknow::NodeHandle extends the interface provided by ros::NodeHandle, overriding some of its functionality. When a developer creates subscriptions or publishers, DyKnow-ROS provides dyknow::Subscriber and dyknow::Publisher handles. These are run-time reconfigurable version of the ros::Subscriber and ros::Publisher. A CU is also able to set a callback for whenever it is reconfigured. This can be useful when a reconfiguration requires actions to be taken by a nodelet, for example to notify some part of the system of its new subscriptions and publishers. Further, statistics such as the number of reconfigurations are maintained and made available.

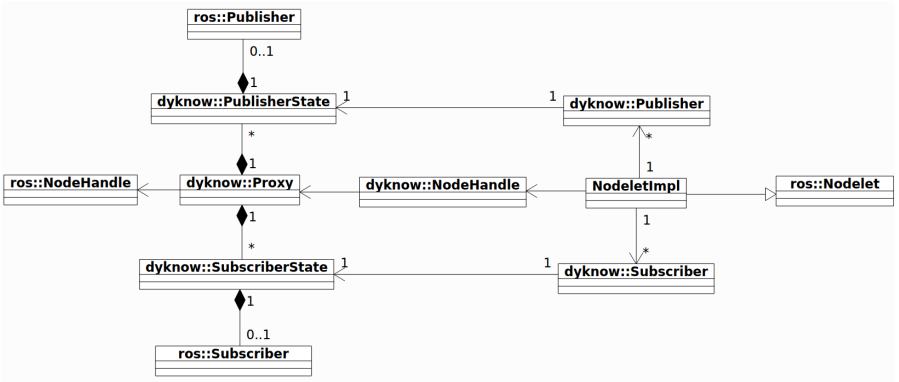

Figure 9.2: UML diagram showing the DyKnow nodelet implementation and its relation to standard ROS components.

Listing 9.1: ROS echo example

```
1 void Echo::onInit() {
2     ros::NodeHandle nh = getMTPrivateNodeHandle();
3     sub = nh.subscribe("in", 1000, &Echo::callback, this);
4     pub = nh.advertise<MessageType>("out", 1000);
5 }
6
7 void Echo::callback(const MessageType::ConstPtr& msg) {
8     pub.publish(msg);
9 }
```

The proxy adds additional services to control the mappings between topics and ports. It can also list for a given nodelet what topics are connected to which ports at the time of the service call.

- **GetConfig:** Returns a list of ports and associated topics for the nodelet the proxy is associated with.
- **SetConfig:** Takes a list of ports and topics to be connected for the nodelet the proxy is associated with.
- **GetStatistics:** Returns nodelet statistics in terms of uptime, the number of reconfigurations performed, and the number of messages sent or received for each port.

These additional services are tied to individual CUs and allow external components to keep track of and modify how CUs are connected to other CUs. With the addition of these services, the lack of configuration control is resolved.

Example 9.1 (A simple echo nodelet)**.** *To illustrate the subtle differences between standard ROS nodelets and DyKnow-ROS nodelets, we consider a simple echo unit.*

Listing 9.2: DyKnow-ROS echo example

```
void Echo::onInit() {
    nh = dyknow::NodeHandle(getMTPrivateNodeHandle());
    sub = nh.subscribe("in", 1000, &Echo::callback, this);
    pub = nh.advertise<MessageType>("out", 1000);
}

void Echo::callback(const MessageType::ConstPtr& msg) {
    pub.publish(msg);
}
```

Echo units can receive messages, which they then immediately forward, without performing any kind of processing on them. As such, they are one of the smallest example nodelets. Listing 9.1 shows a ROS implementation of an echo unit. We can use a local ros::NodeHandle to create a ros::Subscriber and ros::Publisher. The subscriber is connected to the 'in' topic, with a callback to the 'callback' method. Anytime a message arrives, this method is called. Since we are using an echo unit, the message is immediately published on the 'out' topic using the publisher.

Switching to DyKnow-ROS requires some changes as shown in Listing 9.2. Creating a dyknow::NodeHandle results in the creation of a proxy behind the scenes. When all node handles go out of scope, so does the proxy, so we store the node handle as a member variable. The reason for requiring the proxy to be persistent is because it hosts the reconfiguration services — if it goes out of scope, the services become unavailable. The remainder of the code is the same, although instead of ROS subscribers and publishers, we get a dyknow::Subscriber and a dyknow::Publisher. The subscriber uses the 'in' port; we do not control what topic it is connected to. The same holds for the producer, which is connected to the 'out' port.

The difference between the two code snippets is thus minimal from the perspective of the developer. However, while the syntax is largely the same, the semantics have slightly changed. As always, a developer should be aware of these underlying mechanisms.

9.4 Management of stream processing

The stream reasoning manager is responsible for setting up and maintaining configurations in support of stream reasoning. It interacts with the stream reasoning engine and the computational environment, and is implemented in DyKnow-ROS as a node — as is the stream reasoning engine. The manager can interact with the computational environment with the help of the proxy services. Likewise, the stream reasoning engine provides services which are presented later. Both sets of services are used by the manager, which in turn provides its own set of services acting as a client-facing interface. The services provided by the stream reasoning manager are:

- **AddTarget:** Given a target specification, store the specification under the associated label. Specifications can be overridden.
- **RemoveTarget:** Given a label, remove the target specification with that label, if any.
- **AddTransformation:** Given a transformation specification, store the specification under the associated label. Specifications can be overridden.
- **RemoveTransformation:** Given a label, remove the transformation specification with that label, if any.
- **Spawn:** Given a transformation label and name, instantiate a nodelet of that transformation type with the supplied name. Nodelets can be protected from unloading. Uses NodeletLoad.
- **Destroy:** Given a name, destroy the nodelet with that name if it exists and if it is not protected. An unprotected nodelet can destroy itself this way. Uses NodeletUnload.
- **GetModel:** Returns a listing of all running DyKnow nodelets and their port-topic connections. Also returns all stored transformation specifications.

The manager provides supporting services for changing configurations as well as acquiring a representation of the current environment. The latter service is useful for taking configuration snapshots, for example for the purpose of representing the environment in a client.

We can subdivide the tasks of the stream reasoning manager into two parts. The first is to keep track of the environment, i.e. what its current state is, what TFs exist, what CUs exist, etc. This basically boils down to a storage task. The second is to enforce the configuration life cycle, by regularly updating the configuration. This is the daemon component of the manager. We consider both tasks in more detail.

Representation of configurations

The stream reasoning manager keeps track of the state of the computational environment and provides services that can be used to change this environment. The DyKnow model specifies an ontology for representing an environment with a well-structured grammar. DyKnow-ROS makes use of this ontology to not just represent the environment, but also as a grammar for specifications of transformations and targets. Since DyKnow-ROS is a concrete realisation of the DyKnow model, it extends the DyKnow ontology to capture ROS-specifics. For example, whereas the DyKnow ontology uses the Channel concept, DyKnow-ROS refers to the Topic concept. The latter is a specialisation of the former, and enforces a well-defined grammar for topics as defined by the ROS specifications. Figure 7.1 in the previous chapter illustrated the concept hierarchy of the DyKnow ontology, which is listed in Appendix A. We briefly consider its extension to DyKnow-ROS here.

Service calls to the AddTransformation service provided by the stream reasoning manager require a uniquely-labeled transformation specification. Recall that the dyknow:Transformation concept is defined in DL as

$$\begin{aligned}\text{Transformation} \sqsubseteq &\exists \text{hasName.xsd:Name} \\ &\sqcap \exists \text{hasCostModel.LabourCostModel},\end{aligned} \tag{9.1}$$

where

$$\text{LabourCostModel} \sqsubseteq \text{CostModel}. \tag{9.2}$$

DyKnow-ROS uses the specialised concept dyknowros:ROSTransformation such that ROSTransformation $\sqsubseteq$ Transformation. Concretely, the ROSTransformation is defined in DL as

$$\begin{aligned}\text{ROSTransformation} \sqsubseteq &\text{Transformation} \\ &\sqcap =1\ \text{hasSource.xsd:anyURI} \\ &\sqcap \exists \text{hasPort.Port},\end{aligned} \tag{9.3}$$

where

$$\text{hasInPort} \sqsubseteq \text{hasPort}, \tag{9.4}$$

$$\text{hasOutPort} \sqsubseteq \text{hasPort}. \tag{9.5}$$

This means that a ROSTransformation has at least one port, either an input or an output port. Furthermore, it has exactly one source, which is represented by a URI to a nodelet binary. This makes it possible for the manager to dynamically load specific nodelet implementations. Lastly, ports can be annotated with tags describing the semantics of the data flowing through those ports and the channels they connect to. Listing 9.3 shows an example of a transformation specification in DyKnow-ROS.

Listing 9.3: Example transformation specification in Turtle syntax

```
:undistort a :ROSTransformation ;
    :hasType "nodelet" ;
    :hasSource "package/Undistort" ;
    :hasParam [
        a :Parameter ;
        :hasName "configPath" ;
        :hasType "string" ;
        :hasValue "/path/to/configuration/cam1/" .
    ] ;
    :hasPort [
        a :InPort ;
        :hasName "rawCamera" ;
        :hasTag [
            a :Tag ;
            :hasValue "RawRGB(cam1)" .
        ] .
    ] ;
    :hasPort [
        a :OutPort ;
```

```
        :hasName "undist" ;
        :hasTag [
            a :Tag ;
            :hasValue "Undistorted(cam1)" .
        ] .
    ] ;
    rdfs:label "Undistort(cam1)" .
```

Service calls to the AddTarget service require a target specification. Similar to transformation specifications, target specifications make use of the dyknow:Target concept extended to dyknowros:ROSTarget for ROS. Targets are composed of a label, channel, and tag. In the case of ROS, the channel corresponds to a topic, i.e.

$$\begin{aligned} \text{ROSTarget} \sqsubseteq \text{Target} \\ \sqcap \exists \text{hasTopic}.\text{Topic}. \end{aligned} \tag{9.6}$$

where dyknowros:Topic is a specialisation of dyknow:Channel. Topics use a standardised naming convention enforced by ROS which is similar to the way paths are represented in Unix-based systems. Listing 9.4 shows an example target specification in DyKnow-ROS.

Listing 9.4: Example target specification in Turtle syntax

```
:undistortSub a :ROSTarget ;
    :hasTopicName "/result"^^rosTopic ;
    :hasTag [
        a :Tag ;
        :hasValue "Undistorted(cam1)" .
    ] .
    rdfs:label "undistortSub" .
```

This leaves us with CUs. While the service calls do not require CUs as arguments, some do return them as part of the service response. Therefore, they too have a DyKnow-ROS specification. Since nodelets act as CUs in DyKnow-ROS, we simply get

$$\text{Nodelet} \sqsubseteq \text{ComputationUnit} \tag{9.7}$$

for the sake of completeness.

Example 9.2 (Transformation and target specifications in DyKnow-ROS)**.** *Consider a smart lab equipped with four similar ceiling cameras. In this example, the cameras are using fish-eye lenses, and their positions allow them to cover most of the lab's ground surface area with some overlap. A transformation could be applied to the image streams from the cameras if they are first undistorted.*

A subscription to an undistorted stream from a camera called 'cam1' is illustrated in Listing 9.4 as a target specification. The target is labelled undistortSub and represents the desire for a stream to be produced on the /result topic with the semantic description Undistorted(cam1). The semantic description is part of the tagging language and intended to represent an undistorted image stream originating from the 'cam1' camera.

A transformation that might produce a suitable stream is illustrated in Listing 9.3. It represents a nodelet for which the binary is referred to in package/Undistort. Note that this binary makes no reference to a particular camera. This is because the transformation combined the binary with a configuration for the 'cam1' camera. It does so by providing the binary with a 'configPath' parameter, which the binary understands to be the location of a lens model file specific to the 'cam1' camera. Consequently, the transformation is labelled Undistort(cam1) to illustrate it is specific to the 'cam1' camera eventhough the binary it uses is not. The transformation has two ports; one input port expecting raw RGB images from the 'cam1' camera, and one output port producing undistorted versions of those images. The transformation is therefore a suitable candidate for satisfying the target, assuming that its dependency on raw RGB images can be resolved.

Both transformations and targets make use of tags, which are used for semantic annotations. The focus of this work was however not on the development of an annotation language, but rather how one could be used in the context of the DyKnow model. Despite this, a tagging language is necessary for the DyKnow-ROS system to work. Therefore a simple tagging language is used when tags are explicitly specified, with the understanding that a better tagging language can replace this placeholder language in the future.

Configuration life-cycle daemon

The second task of the stream reasoning manager is to act as a daemon by reconfiguring the DyKnow-ROS configuration in accordance with the configuration life-cycle. This requires the realisation of Algorithm 8.1 from the previous chapter in a ROS environment. Additionally, functionality is needed for the observation of computational resource usage and its effect on the cost estimators.

The `EXPLORE` procedure and its dependencies take a computational environment ε and perturbation δ_p, and subsequently construct an optimal change set δ^* with the help of the $spawn$, $connect_{\downarrow}$, $connect_{\uparrow}$ and $destroy$ actions. In DyKnow-ROS, the environment is available and monitored by the stream reasoning manager. Perturbations can be detected as well through the service calls that would qualify as perturbations. For example, if a target is added, this is achieved to a service call to the manager, thereby implicitly notifying the manager of a perturbation. As such, ε and δ^* have counterparts in DyKnow-ROS. The same holds for the aforementioned actions which make up δ^*, as each of them can be performed using the services available to the stream reasoning manager. The application of an optimal change set is then equivalent to a sequence of service calls. The `EXPLORE` procedure is deviated from in two ways. Firstly, since ROS allows nodelets to have multiple ros::Producer instances, a CU can have multiple outputs. In such a case, the CU is said to be an instance of multiple transformations which take the same inputs but produce different outputs. Secondly, CUs can be designated 'protected', in which case they are

never destroyed by a resulting change set. This will cause the procedure to find a best change set given that the protected CUs are kept around.

Computational resource usage is measured in terms of CPU time, specifically by combining `utime` and `stime`. Both values are obtained by reading from `/proc/$tid/statm` on Linux for the corresponding thread identifier. For labour, we measure the CPU time associated with creating new CUs. For upkeep, the CPU time per wall time minute is accumulated by measuring the CPU times of individual callbacks from dyknow::Subscriber and dyknow::Timer objects. The former is useful for CUs that react to new inputs, whereas the other is useful for CUs that run at specific time intervals. Upkeep is measured relative to a transformations, so if multiple CUs exist for the same transformation, the upkeep would be the average CPU time measured over all of those CUs.

Finally, the estimators need to be updated using these observations. Since the focus of here was not on precise estimations, a simple placeholder was chosen to fulfil the requirement of having estimators. Future work could produce better models of CPU usage. In the current state, the predicted CPU usage is simply an average over the observed CPU usage. The cost models for labour and upkeep are referred to as

$$\mathrm{LabourCostModel}(\mathrm{historicalAverageLabour}), \tag{9.8}$$

$$\mathrm{UpkeepCostModel}(\mathrm{historicalAverageUpkeep}), \tag{9.9}$$

in the DyKnow-ROS ontology extension.

9.5 Stream reasoning support

Reasoning over streams is performed by the *stream reasoning engine* in DyKnow. This component takes as its input formulas, state streams, and grounding information for the purpose of progression of the formulas over the provided data. Formulas can be part of a *formula group*, which represents a collection of formulas that are to be evaluated simultaneously over a single state stream. The connecting information grounds the symbols in logical formulas to specific values in the state stream.

The stream reasoning engine provides a number of services that control the evaluation of formulas, as shown below.

- **CreateGroup:** Creates a formula group with an optionally provided label and result topic. If no label or result topic are provided, DyKnow generates them instead.
- **DestroyGroup:** Destroys a formula group by its label. This stops the progression of formulas in the group.
- **StartGroup:** Activates progression for a formula group identified by name.
- **StopGroup:** Stops the progression for a formula group by name. Cannot be resumed.

- **AddFormula:** Adds a provided formula to a formula group identified by its label. Yields an identifier (index) for the formula.
- **RemoveFormula:** Removes a formula from a formula group identified by the group's label and formula's index. This can be done while a formula is being progressed.

Each group in the stream reasoning engine is configured with parameters such as `MAX_NODES` as well as policy information such as the stream frequency, which determines how far apart time-points are in real time. Whenever a formula (including its syntactic or semantic grounding) is added to a group, a new formula graph consisting of a single node for the added formula is created. Starting a group will result in the generation of a stream containing the combined propositional information required for evaluating the formulas in the group, as well as the generation of a group-shared formula cache. This will also lock the group, meaning it can no longer be altered without resetting the group in its entirety to its initial state. A group thus makes it possible to start evaluating a logical specification composed of potentially many formulas using the same starting point. Once started, these graphs will grow depending on the choice of parameters and the uncertainty in the stream used to evaluate the formula. After each progressed state, the progressor will emit a verdict status for each formula graph in the group.

9.6 Empirical evaluation

The proxy introduced by DyKnow-ROS potentially introduces an overhead in throughput. Measuring the overhead gives insights into the cost of adopting DyKnow-ROS.

Topic-based communication between nodelets is assumed to be faster than between nodes because nodelets are part of the same process and nodes are not. In this experiment, we use both as benchmarks for comparison. The computation graph is a linear sequence of connected node(let)s such that each intermediate node(let) receives from a predecessor node(let) and immediately publishes to a successor node(let). The source produces messages containing current time-stamps at a fixed frequency f. Every (intermediate) receiver checks that time-stamp against the arrival time and reports the time difference. The number of node(let)s n then corresponds to the number of message hops.

The performance results are shown in Figure 9.3, where the performance graph contrasts the number of hops to the average time-to-arrival for messages sent along the node(let) chain. The source produced 1,000 time-stamped messages at a frequency of $f = 5$Hz, which every receiver compared to the local time upon arrival prior to forwarding the message. The graph illustrates the time results for DyKnow-ROS nodelets and ROS nodelets, as well as ROS nodes. As expected, nodes are much slower than nodelets because they have to communicate between processes. The results for nodes put into perspective the overhead we can see for DyKnow-ROS

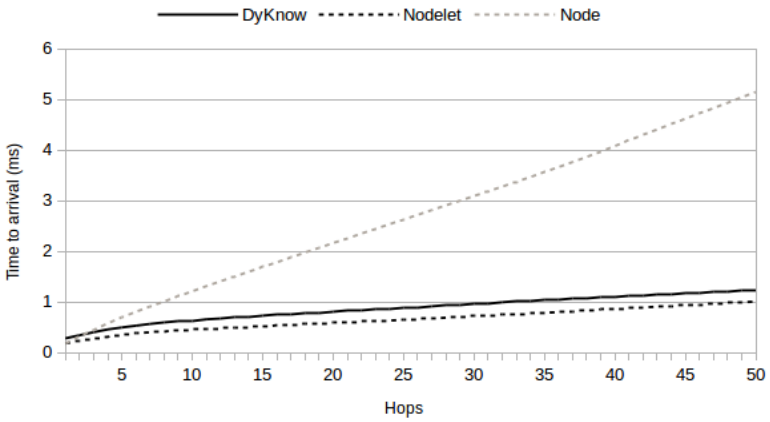

Figure 9.3: Performance graph showing the different time-to-arrivals for messages relative to the number of hops for a linear chain.

nodelets when compared against standard ROS nodelets, which grows slowly to about 0.2ms after $n = 50$ hops. We may therefore conclude that the overhead induced by DyKnow-ROS is negligible.

9.7 Summary

This chapter presented a realisation of the DyKnow model with the Robot Operating System (ROS), resulting in the DyKnow-ROS system. An extension of the services provided by ROS is presented to support the DyKnow model. This was followed by a concrete representation of entities in the DyKnow model with the help of the DyKnow ontology extended for ROS. The configuration life cycle was realised by a daemon that uses CPU time as the computational resource of choice in DyKnow-ROS, and some simple estimators for labour and upkeep were presented. Finally, the overhead induced by the extensions to ROS was measured and shown to be negligible.

Chapter

10 Case-studies

CASE-STUDIES have been performed to show the application of the DyKnow-ROS stream reasoning framework as a proof of concept. These case-studies focus on particular aspects of the DyKnow-ROS framework, which in turn translate to the stream reasoning waterfall model.

10.1 Introduction

Case-studies are useful to show the functioning of an information system in a practical setting. In this chapter, we focus primarily on the visualisation and adaptive reconfiguration functionality of the DyKnow-ROS stream reasoning framework. The idea is to show examples of reaching verdicts which lead to a system response, as shown in Figure 10.1. Such a response can be internal, for example by changing the computational environment, or external, for example by causing an agent to act. The latter case — external responses — have been left outside the scope of this dissertation due to the added complexity of aspects such as control.

10.2 Interactive visualisation

ROS provides a wide array of visualisation tools using a Qt-based framework. For the visualisation of nodes and topics, rqt_graph provides a graphical user interface that communicates with the ROS master and produces a DOT graph. While this approach works great for nodes, it fails to detect nodelets as they are threads within the Nodelet Manager node. We therefore forked rqt_graph and replaced the communication with the ROS master to instead query the stream reasoning manager in DyKnow-ROS for its configuration model. Since ROS does not take run-time reconfiguration into consideration, we also had to switch from the manual refresh in

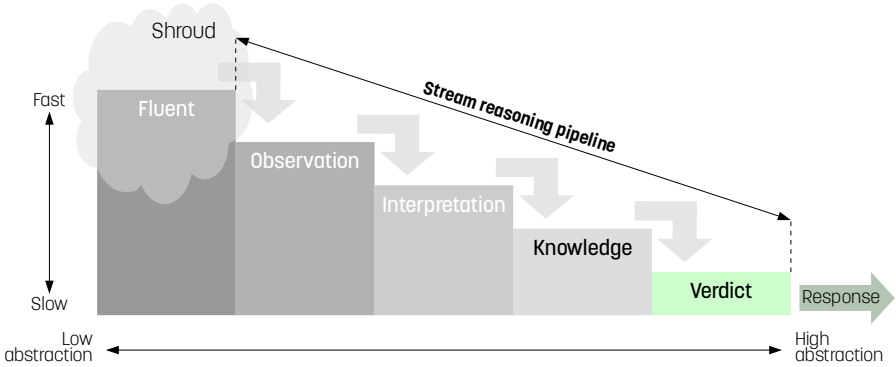

Figure 10.1: The stream reasoning waterfall model with the agent response to verdicts highlighted.

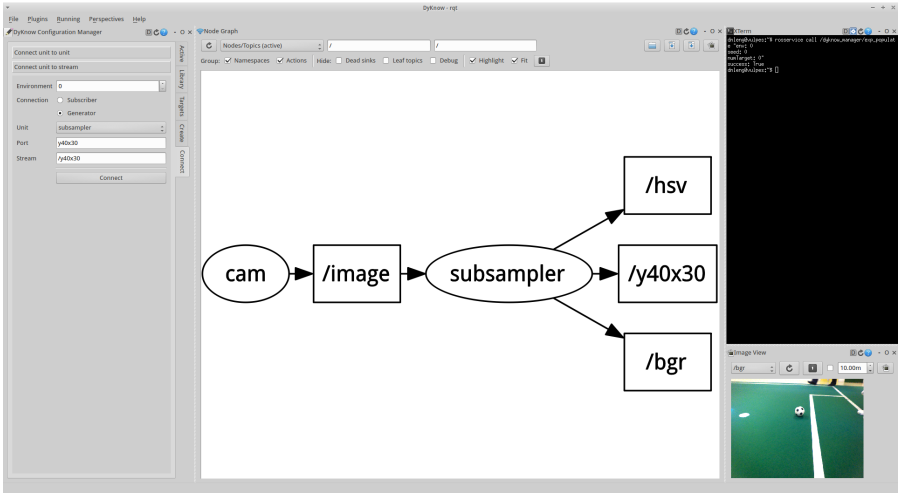

Figure 10.2: Screenshot of the interactive visualisation tool.

rqt_graph to a frequency-based refresh. This was combined with a control widget to allow a user to interact with the stream reasoning manager.

A screenshot of the tool at work is shown in Figure 10.2, where the bottom left camera view was produced by the rqt_image_view widget. The graph shown in the centre panel was created using the control panel on the left. Ovals in the centre graph correspond to nodelets, and rectangles correspond to topics. The bottom-right image view shows the colour video stream. Since no changes were made to the rqt_graph interface itself, this representation is natural to ROS developers.

The control panel on the left supports a number of features based on the services provided by DyKnow-ROS. The *active* tab lists the currently active CUs together with their associated transformations, the number of input ports, and the number of out-

Figure 10.3: Humanoid lab (left) equipped with four ceiling cameras (right).

put ports. A *library* tab offers a listing of transformation specifications by label, and allows a user to import or delete transformations. CUs can be instantiated through the panel as well with the *create* panel; the user provides a name for the nodelet to be created and a type in terms of transformation specifications. The panel shown is the *connect* panel, where either a combination of two nodelets and ports are selected to be connected with a topic decided by the tool, or a single port and topic can be connected where the user gets to specify the topic name manually.

The visualisation tool gives access to all of the stream reasoning manager's services, offering an interface that can be managed by human operators. As a result, it offers the functionality expected by ROS developers as well as some extra control over the configuration during run-time.

10.3 Collaborative tracking of a ball

This case study focuses on two NAO robots, called *Piff* and *Puff* (Swedish for *Chip 'n Dale*). Both Piff and Puff are capable of running a processing pipeline that takes in sensor information and produces ball coordinates relative to the soccer field. For the case study, we were interested in situations where semantic subscriptions could provide added value to Piff in performing its task of tracking the ball. We consider two cases; 1) Piff is tracking the ball but something goes wrong; and 2) Piff is tracking the ball and Puff offers to help for a while. Piff and Puff are assumed to be part of the same computational environment; a multi-agent system approach is beyond the scope of this dissertation. The operational environment is provided by a humanoid lab at Linköping University, which is organised to support software development for NAO robots.

The humanoid lab is equipped with a green felt RoboCup soccer field as shown in Figure 10.3 on the left. As shown on the right, there are four cameras attached to the ceiling over the field. The ceiling cameras are AXIS M3005-V network cameras with a 118° angle of view, producing 1920×1080 images. These images can be used for accurate positioning of objects on the field. The coordinate system uses one of

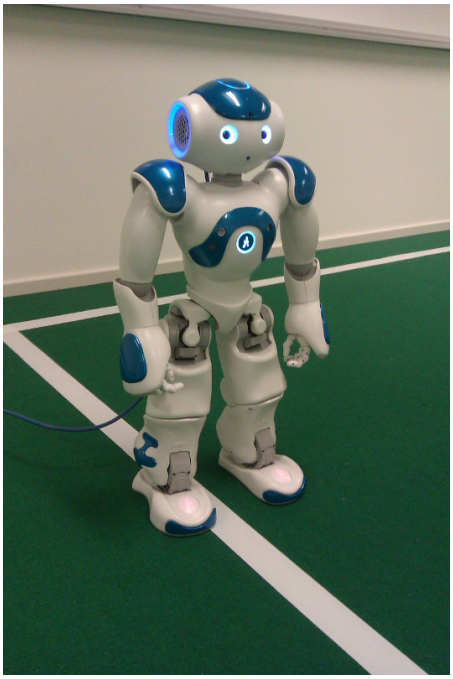

Figure 10.4: A SoftBank Robotics NAO V4 robot.

the field corners as its origin, relative to which the coordinates of other objects such as balls or robots are determined.

Piff and Puff are Softbank Robotics NAO humanoid robot platforms, of which an example is shown in Figure 10.4. Standing upright, they are 58cm tall and weigh 5.4kg. With normal use, the battery provides 90 minutes of autonomy. The head houses two HD cameras producing 1280×960 images at 30 FPS in YUV422 colour space. One is located in the forehead and faces forward; the other is located in the 'mouth' area and faces downwards. The various joints provide pose information to the system through joint position sensors. The NAO comes equipped with an Intel Atom Z530 processor running at 1.6GHz, with 1GB of RAM, 2GB of Flash memory, and an 8GB Micro SDHC. The system runs the Ubuntu 14.04 LTS operating system with the NAOqi programming framework. Our NAO platforms use a publicly-available ROS driver[13] exposing the NAOqi API through ROS. More technical details can be found in the NAO technical guide[14].

[13]The NAO packages for ROS are documented at `http://wiki.ros.org/nao` (Last accessed: September 10th, 2019)

[14]The NAO technical documentation is available at `http://doc.aldebaran.com` (Last accessed: September 10th, 2019)

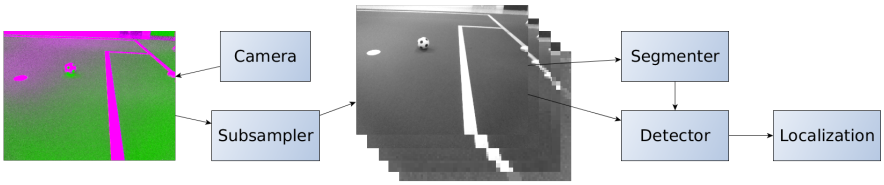

Figure 10.5: Piff and Puff's transformation pipeline conceptually showing the transformations from camera images to ball positions.

Recovery from failures

We start with a scenario in which a user wants to perform perimeter monitoring. That is, check whether a ball 'breaches' the perimeter indicated by the centre circle on the football field and, if so, notify the user. While a toy scenario, it allows us to consider the full DyKnow-ROS system's operations. The user makes no assumptions about what equipment exists in the system; only that there is a DyKnow-ROS instance which he or she is able to interact with. Neither does the user make any assumptions about the availability of equipment over time. In a sense this is a natural behaviour for a non-expert user. The user queries a system's services and the system tries to meet the user's expectations to the best of its abilities. The system further seeks to minimise the cost it incurs while satisfying the user's needs. The first step for a user is then to describe those needs in some language expression. DyKnow-ROS uses temporal-logic formulas for this purpose, so the user's inquiry is described by a wff

$$\mathsf{G}_{[0,1440]}\left[\mathsf{InsideCircle}(\mathsf{ball})\right], \tag{10.1}$$

meaning that for the next 24 hours (measured in minutes), the ball will remain within the circle[15]. The statement is not intended to enforce a particular situation, but rather specifies what is expected to happen. It might be the case that the ball leaves the circle for whatever reason, which should then result in the statement being evaluated to false. However, in order to determine whether the statement is true or false (or even unknown), a state stream is needed over which it can be evaluated.

The task of generating a state stream starts with the specification of targets. Recall that targets are composed of a channel identifier and a tag describing the semantics of the sought-after streaming data. The target should thus reflect that we require information on the truth value of $\mathsf{InsideCircle}(\mathsf{ball})$, which is a predicate. The target is illustrated in Listing 10.1.

Listing 10.1: Target specification for $\mathsf{InsideCircle}(\mathsf{ball})$ information

```
:target1 a :ROSTarget ;
    :hasTopicName "/target1"^^rosTopic ;
    :hasTag [
        a :Tag ;
```

[15] Alternatively, qualitative spatial relations NTPP and TPP could be used.

TID	TF label		Tags
tid_1	pose(piff)		$\varnothing$
		$\Rightarrow$	pose(piff)
tid_2	bottom_cam(piff)		$\varnothing$
		$\Rightarrow$	yuvImage(piff)
tid_3	subsampler(piff)		yuvImage(piff)
		$\Rightarrow$	imageScalePyramid(piff)
tid_4	segmenter(piff)		imageScalePyramid(piff)
		$\Rightarrow$	convHull(piff, field)
tid_5	ball_detector(piff)		convHull(piff, field), imageScalePyramid(piff)
		$\Rightarrow$	pixelPos(piff, ball)
tid_6	ball_localization(piff)		pixelPos(piff, ball), pose(piff)
		$\Rightarrow$	position(ball)
tid_7	circle_monitor(ball)		position(ball)
		$\Rightarrow$	InsideCircle(ball)

Table 10.1: Piff's TFs and their tags denoted by $itag_1, \ldots, itag_n \Rightarrow otag$.

```
5         :hasValue "InsideCircle(ball)" .
6     ] .
7     rdfs:label "InsideCircle(ball)" .
```

After adding this target to the environment ε, it looks like

$$\langle \varnothing, \varnothing, \{\langle \text{target1}, \text{InsideCircle(ball)}, \texttt{/target\_1}\rangle\}, =\rangle . \tag{10.2}$$

This represents a perturbation, so the environment tries to reconfigure itself. However, since no transformations exist, no solutions are found yet, and no state stream is generated. We can solve this by considering the situation wherein Piff registers its transformations to the environment. Table 10.1 shows the semantics of the set of transformations provided by the NAO robot using a short-hand notation.

The bottom_cam TF provides a YUV image stream, which can be subscribed to by the subsampler TF. This transformation down-samples the resolution of the three channels into 640x480, 320x240, 160x120, 80x60, and 40x30. The segmenter TF instances may subscribe to low-resolution Y and V channels to determine the convex hull of the green field, ignoring the space in the image which captures things outside of the field. This convex hull is combined with the Y channel by the ball_detector TF to produce pixel coordinates of balls, which is then combined with pose information by the ball_localization TF to produce ball position data, which matches the query. Since the matrix is updated when transformations are added or removed, the result is a 11×19 matrix for the 11 inputs and 19 outputs.

As the result of the perturbation, the stream reasoning manager searches for an optimal configuration and finds one as shown conceptually in Figure 10.5. The associated change set δ is the instantiation of all transformations, and the connection

of the resulting CUs in accordance with their annotations. The new environment ε is described by

$$\langle CU, F, \{\langle \text{target1}, \text{InsideCircle(ball)}, \texttt{/target\_1}\rangle\}, =\rangle, \tag{10.3}$$

where the set of transformations F remains unchanged, and the set of CUs is described by

$$\begin{aligned} CU = \{ & \langle cid_1, tid_7, [\texttt{/topic\_1}], \texttt{/target\_1}\rangle, \\ & \langle cid_2, tid_6, [\texttt{/topic\_2}, \texttt{/topic\_6}], \texttt{/topic\_1}\rangle, \\ & \langle cid_3, tid_5, [\texttt{/topic\_3}, \texttt{/topic\_4}], \texttt{/topic\_2}\rangle, \\ & \langle cid_4, tid_4, [\texttt{/topic\_4}], \texttt{/topic\_3}\rangle, \\ & \langle cid_5, tid_3, [\texttt{/topic\_5}], \texttt{/topic\_4}\rangle, \\ & \langle cid_6, tid_2, [\,], \texttt{/topic\_5}\rangle, \\ & \langle cid_7, tid_1, [\,], \texttt{/topic\_6}\rangle\}. \end{aligned} \tag{10.4}$$

Piff now produces a ball position stream on the `/topic_1` topic, which can be used by the circle monitor TF to determine whether the ball is inside the circle. This Boolean information is then transmitted on topic `/target_1` as specified by the target target1. The new environment results in a stream containing the information needed for interpreting the symbols of the formula, which is synchronised, flattened, and connected to the stream reasoning engine. The progression procedure now uses the resulting state stream to incrementally evaluate the formula through rewritings.

Unfortunately, something goes wrong and the image segmenter is unloaded, leaving a hole in the computation graph and interrupting the flow of position information. This perturbation is detected as

$$\delta_p = (\varnothing, \{\langle cid_4, tid_4, [\texttt{/topic\_4}], \texttt{/topic\_3}\rangle\}, \varnothing, \varnothing, \varnothing, \varnothing). \tag{10.5}$$

The subsampler is still producing a stream of low-resolution images, but the segmenter no longer exists to do anything with them. The environment is now as in Figure 10.5 but without a segmenter. This perturbation results in the update procedure generating a change set by re-using the part of CU that still exists, but instantiating a new computation unit cid_8 of type tid_4 and reconfiguring it to subscribe to the streams that were already being produced by the subsampler, i.e.

$$\delta^* = (\{\langle cid_8, tid_4, [\texttt{/topic\_4}], \texttt{/topic\_3}\rangle\}, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing). \tag{10.6}$$

The detector's subscription to the defunct segmenter is thus replaced by one to the new segmenter, and the information flow is restored.

Some time later, Puff joins Piff on the field and registers its own transformations in accordance with Table 10.1, where piff is replaced by puff for new TIDs 8–14. The computational environment looks like before, but now $|F_\varepsilon| = 14$. Given the

possibility to generate a second pipeline for ball positions, the life-cycle daemon nevertheless does not use the second pipeline as-is. The reason for this is that the cost for re-using Piff's part of the computation graph is assumed to be free, whereas a lot of effort would have to be spent in order to instantiate Puff's pipeline to switch away from Piff's stream. Only if Puff's alternatives are significantly cheaper to make up for the extra labour cost will the daemon switch pipelines. Since both Piff and Puff are NAO robots with similar equipment and the same transformations, we assume this is not the case.

At some point, Piff needs to recharge its batteries. It therefore first deregisters its transformations from the environment, i.e.

$$\delta_p = (\varnothing, \varnothing, \varnothing, F^-, \varnothing, \varnothing), \tag{10.7}$$

where $|F^-| = 7$ consists of the seven transformations from Table 10.1. The life-cycle daemon correctly identifies the perturbation and starts a review interval, in which it determines that all active CUs are now defunct. This means that there is no longer any guarantee that the CUs provide streaming data. Thankfully, Puff's transformations provide a suitable replacement for the defunct CUs, leading to a change set

$$\delta^* = (CU^+, CU^-, \varnothing, \varnothing, \varnothing, \varnothing), \tag{10.8}$$

where the CU additions are based on Puff's transformations, and the CU removals are Piff's defunct CUs;

$$\begin{aligned} CU^- = \{ & \langle cid_1, tid_7, [\texttt{/topic\_1}], \texttt{/target\_1} \rangle , \\ & \langle cid_2, tid_6, [\texttt{/topic\_2}, \texttt{/topic\_6}], \texttt{/topic\_1} \rangle , \\ & \langle cid_3, tid_5, [\texttt{/topic\_3}, \texttt{/topic\_4}], \texttt{/topic\_2} \rangle , \\ & \langle cid_8, tid_4, [\texttt{/topic\_4}], \texttt{/topic\_3} \rangle , \\ & \langle cid_5, tid_3, [\texttt{/topic\_5}], \texttt{/topic\_4} \rangle , \\ & \langle cid_6, tid_2, [\,], \texttt{/topic\_5} \rangle , \\ & \langle cid_7, tid_1, [\,], \texttt{/topic\_6} \rangle \}, \end{aligned} \tag{10.9}$$

$$\begin{aligned} CU^+ = \{ & \langle cid_9, tid_{14}, [\texttt{/topic\_7}], \texttt{/target\_1} \rangle , \\ & \langle cid_{10}, tid_{13}, [\texttt{/topic\_8}, \texttt{/topic\_12}], \texttt{/topic\_1} \rangle , \\ & \langle cid_{11}, tid_{12}, [\texttt{/topic\_9}, \texttt{/topic\_10}], \texttt{/topic\_2} \rangle , \\ & \langle cid_{12}, tid_{11}, [\texttt{/topic\_10}], \texttt{/topic\_9} \rangle , \\ & \langle cid_{13}, tid_{10}, [\texttt{/topic\_11}], \texttt{/topic\_10} \rangle , \\ & \langle cid_{14}, tid_9, [\,], \texttt{/topic\_11} \rangle , \\ & \langle cid_{15}, tid_8, [\,], \texttt{/topic\_12} \rangle \}. \end{aligned} \tag{10.10}$$

Note also the removal of the cid_8 CU which was previously used to patch a gap in the computation graph.

After the occurrence of the perturbation δ_p, the stream being produced on `/target_1` is temporarily interrupted as the review interval is performed. At the

end of the review interval, the change set δ^* will have been applied to the environment, cancelling out the suboptimality imposed by δ_p by repairing the computation graph with an alternative pipeline satisfying the target. Consequently, the formula can be evaluated further. The system operates on a best-effort basis by quickly finding ways to repair broken computation graphs, thereby minimising the interruption in the streams used to construct a state stream for formula evaluation.

Exploitation of new optima

Continuing the scenario, Puff is currently observing the ball which has not yet left the circle on the field, and Piff is in the process of recharging. The only transformations available to the computational environment are therefore Puff's. However, the lab itself is also equipped with four cameras. These cameras can be used in unison to generate a top-down image of the field. More importantly, these cameras could be used to locate objects on the field, in particular NAO robots and balls. A system can obtain the camera video feeds, stitch the images together, and perform localisation, without the computational limitations imposed by NAO hardware.

We can thus register the ceiling camera system with the computational environment, resulting in transformations with TIDs 15–24. The pipeline consists of the transformations shown in Table 10.2. The registration of these transformations constitutes a perturbation

$$\delta_p = (\varnothing, \varnothing, F^+, \varnothing, \varnothing, \varnothing), \tag{10.11}$$

where the set of added transformations $|F^+| = 10$ consists of the ten transformations listed in Table 10.2. Even though the perturbation did not break anything — there is still a stream on topic $/\mathsf{target\_1}$ — it is nevertheless a potential long-term positive perturbation, because the added transformations might yield cheaper-cost solutions. If this is not the case, then $\delta^* = \delta_\varnothing$; otherwise, the stream reasoning manager can replace part of the active pipeline with the transformations from the ceiling cameras. Since the perturbation is not a short-term negative perturbation, no immediate response is required, so the life-cycle daemon waits with the review cycle until the horizon is reached or a short-term negative perturbation occurs.

Note that the ceiling camera pipeline provided by the lab lacks a transformation that can provide $\mathsf{InsideCircle}(\mathsf{ball})$ information. This is because that pipeline does not have the background knowledge to understand what InsideCircle means; it does not care about the lines on the field, but only about NAO robots and balls and their positions. However, Puff *does* care about the circle on the field, and therefore knows given a position whether that position is within the circle. Assuming that the reduction in upkeep outweighs the labour cost of switching pipelines, DyKnow-ROS finds a solution during the next review interval. It uses the entire ceiling camera pipeline plus the circle monitor from Puff, while unloading the remaining CUs. This leads to a change set

$$\delta^* = (CU^+, CU^-, \varnothing, \varnothing, \varnothing, \varnothing), \tag{10.12}$$

TID	TF label		Tags
tid_{15}	ceiling_cam(cam_1)		$\varnothing$
		$\Rightarrow$	rgbImageDistorted(cam_1)
tid_{16}	undistort(cam_1)		rgbImageDistorted(cam_1)
		$\Rightarrow$	rgbImage(cam_1)
tid_{17}	ceiling_cam(cam_2)		$\varnothing$
		$\Rightarrow$	rgbImageDistorted(cam_2)
tid_{18}	undistort(cam_2)		rgbImageDistorted(cam_2)
		$\Rightarrow$	rgbImage(cam_2)
tid_{19}	ceiling_cam(cam_3)		$\varnothing$
		$\Rightarrow$	rgbImageDistorted(cam_3)
tid_{20}	undistort(cam_3)		rgbImageDistorted(cam_3)
		$\Rightarrow$	rgbImage(cam_3)
tid_{21}	ceiling_cam(cam_4)		$\varnothing$
		$\Rightarrow$	rgbImageDistorted(cam_4)
tid_{22}	undistort(cam_4)		rgbImageDistorted(cam_4)
		$\Rightarrow$	rgbImage(cam_4)
tid_{23}	stitch(field)		rgbImage(cam_1), rgbImage(cam_2), rgbImage(cam_3), rgbImage(cam_4)
		$\Rightarrow$	rgbImage(field)
tid_{24}	ball_detector(field)		rgbImage(field)
		$\Rightarrow$	position(ball)

Table 10.2: The Humanoid lab's ceiling camera transformations and their tags denoted by $itag_1, \ldots, itag_n \Rightarrow otag$.

where the sets of CU additions and removals are described by

$$
\begin{aligned}
CU^{-} = \{ & \langle cid_{10}, tid_{13}, [\texttt{/topic\_8}, \texttt{/topic\_12}], \texttt{/topic\_1} \rangle , \\
& \langle cid_{11}, tid_{12}, [\texttt{/topic\_9}, \texttt{/topic\_10}], \texttt{/topic\_2} \rangle , \\
& \langle cid_{12}, tid_{11}, [\texttt{/topic\_10}], \texttt{/topic\_9} \rangle , \\
& \langle cid_{13}, tid_{10}, [\texttt{/topic\_11}], \texttt{/topic\_10} \rangle , \\
& \langle cid_{14}, tid_{9}, [\,], \texttt{/topic\_11} \rangle , \\
& \langle cid_{15}, tid_{8}, [\,], \texttt{/topic\_12} \rangle \},
\end{aligned}
\tag{10.13}
$$

$$
\begin{aligned}
CU^{+} = \{\, & \langle cid_{16}, tid_{24}, [\texttt{/topic\_13}], \texttt{/topic\_7} \rangle\,, \\
& \langle cid_{17}, tid_{23}, [\texttt{/topic\_14}, \texttt{/topic\_15}, \texttt{/topic\_16}, \\
& \quad \texttt{/topic\_17}], \texttt{/topic\_13} \rangle, \\
& \langle cid_{18}, tid_{22}, [\texttt{/topic\_18}], \texttt{/topic\_14} \rangle\,, \\
& \langle cid_{19}, tid_{20}, [\texttt{/topic\_19}], \texttt{/topic\_15} \rangle\,, \\
& \langle cid_{20}, tid_{18}, [\texttt{/topic\_20}], \texttt{/topic\_16} \rangle\,, \\
& \langle cid_{21}, tid_{16}, [\texttt{/topic\_21}], \texttt{/topic\_17} \rangle\,, \\
& \langle cid_{22}, tid_{21}, [\,], \texttt{/topic\_18} \rangle\,, \\
& \langle cid_{23}, tid_{19}, [\,], \texttt{/topic\_19} \rangle\,, \\
& \langle cid_{24}, tid_{17}, [\,], \texttt{/topic\_20} \rangle\,, \\
& \langle cid_{25}, tid_{15}, [\,], \texttt{/topic\_21} \rangle\}.
\end{aligned}
\tag{10.14}
$$

Note that CU cid_9 is not among the CUs being unloaded as it is being reused in the new environment. The topic /topic_7 is therefore being reused as the output channel for the CU cid_{16}. The resulting environment has a cheaper long-term cost than would have been acrued by not performing the update, while still generating a stream on `/target_1` with minimal interruption. When at some point Puff also requires recharging, it is possible for Puff to leave while keeping the circle monitor available to the system. From that point on, the ceiling camera generate position data which is processed by Puff off-site. The system has successfully exploited the potential improvement to the configuration when it became available.

Cleaning up

Finally, with the field devoid of NAO robots and perhaps at the start of a new day, an unsuspecting student enters the lab and, not realising the experimental setup, takes the ball. The InsideCircle(ball) predicate evaluates to false, thereby violating the formula, yielding 'false' as its final answer. The stream reasoning manager releases the targets, corresponding to a perturbation

$$\delta_p = (\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \{\langle \text{target1}, \text{InsideCircle}(\text{ball}), \texttt{/target\_1} \rangle\}). \tag{10.15}$$

With the target removed, the DyKnow-ROS life-cycle daemon immediately starts a review interval. Since no targets exist, any active CUs are needlessly expending upkeep cost. Therefore, all active CUs are removed while retaining the set of transformations;

$$\delta^{*} = (\varnothing, CU_{\varepsilon}, \varnothing, \varnothing, \varnothing, \varnothing). \tag{10.16}$$

Without any targets, the environment remains idle until new targets are registered, either as the result of a formula needing evaluation or because of another purpose for needing a semantic subscription.

10.4 Summary

In this chapter, we looked at some use-cases for the DyKnow-ROS stream reasoning framework which was presented in Chapter 9. In particular, we first looked at the visualisation component for the DyKnow-ROS daemon, which manages the configuration of the computational environment. The visualisation is an extension of the standard ROS rqt_graph package, which has been improved by adding the ability to visualise DyKnow-ROS nodelets. The interface also allows a user to interact with the daemon. We next looked at a use-case involving two NAO robots, illustrating the impact of the dynamic reconfigurability in cases where new transformations become available or in which existing transformations become unavailable. The streams generated in this process can then be used with state stream synthesis in order to support the evaluation of temporal formulas. The application of DyKnow-ROS to these use-cases serves as a proof of concept of the proposed system.

Chapter

11 Related work

RESEARCH towards stream reasoning has resulted in many different perspectives. This chapter serves as a survey of recent and/or ongoing research projects towards stream reasoning. The survey is not meant to be exhaustive, and does not include many of the well-known stream processing tools and libraries. The listed works are compared to the contributions presented in this dissertation. Some of the listed work refers to specific projects; others represent areas of research of high relevance to this dissertation.

11.1 Introduction

The DyKnow-ROS stream reasoning framework presented in Chapter 9 shares some similarities with other stream reasoning systems. Although we covered specific related work for the various contributions throughout this dissertation, in this chapter, we present some related work coming from the general area of stream reasoning. In the following, a mostly chronological overview is given for stream reasoning research conducted in parallel to the work presented in this dissertation, as well as research upon which those related works are supported. In the presentation of the related works, we also consider the similarities and differences between said works and the DyKnow model and associated implementation on a case-by-case basis. After covering the various related works, a high-level summary is provided, in which we also related some of the presented works to each other.

11.2 STREAM

The Stanford Stream Data Manager (STREAM) was first introduced by Motwani et al. (2003) and Arasu et al. (2004) as a DSMS. The goal of the STREAM project was stated

to be "to build and evaluate a general-purpose DSMS that supports a declarative language and can cope with high data rates and thousands of continuous queries." (Motwani et al., 2003) They therefore consider both data and queries to be dynamic entities. One particular consideration was how such a system would utilise system resources, and how approximations could be utilised to reduce resource requirements to fit the resources such a system has access to. This led to the development of a language for DSMS, as well as the development of query plan sharing and approximation techniques.

The Structured Query Language (SQL) was chosen as the basis for the development of the Continuous Query Language (CQL), and was first introduced by Arasu et al. (2003). Arasu et al. (2006) later covered CQL in more detail. The basic structure of CQL queries follows that of SQL, but allows for the handling of streams in addition to relations. This means that, whereas in SQL one can make use of operators to obtain relations from relations (R2R), CQL supports operators for converting streams to relations (S2R) and relations to streams (R2S). Successive applications of S2R, R2R, and R2S operators makes it possible to perform stream transformations. For S2R operators, CQL makes use of time-based, tuple-based, and partitioned (i.e. splitting into substreams based on a logical condition) window operators. For R2S operators, CQL makes use of insert, delete and relation streams, which generate streams of new elements, old elements, and current elements respectively. Finally, R2R operators are inherited from SQL.

Registered CQL queries are compiled into *query plans*, which execute the query. These query plans are composed of operators connected by queues, and synopses for the storage of operator state. This is similar to the configuration environments used in DyKnow, where queues are represented by channels over which streams are transmitted and received. Synopses can share their state to avoid duplicate computations. The STREAM system additionally contains the *StreaMon* (Babu and Widom, 2004) monitoring and adaptive query processing infrastructure. It is tasked with monitoring the computational load of the components that make up the STREAM system, and to re-optimise if necessary. The ability to re-optimise is especially important when circumstances change during the run-time of a STREAM system. StreaMon thus shares similarities with the DyKnow daemon in terms of their responsibilities.

11.3 Aurora and Borealis

Aurora is a model and architecture for data stream management introduced by Abadi et al. (2003), with application to monitoring. It is the predecessor to *Borealis*, discussed later. Aurora considers data sources to generate streams, which in turn are processed by Aurora (operator) boxes. These boxes correspond roughly to CUs in DyKnow terminology. The stream processing can be changed during run-time for purposes of optimisation. Aurora specifically considers connection points, which are storage containers residing between Aurora boxes that can cache a finite stream

history. More importantly, subgraphs can be attached to or detached from these connection points during run-time.

Aurora's query algebra is referred to as the Aurora ***S**tream **Qu**ery **Al**gebra* (SQuAl). It supports seven operators corresponding roughly to DyKnow transformation. The filter, map, and union operators are straightforward in their meaning, and all three are order-agnostic in Aurora. The remaining operators — BSort, aggregate, join, and resample — are order-sensitive. BSort performs an approximate sort utilising a buffer; aggregate applies window functions together with relational aggregation operations; join allows streams to be combined; and resample can be used to align streams. Queries written in SQuAl are compiled into Aurora networks, which can then be optimised during run-time.

To assist in optimising the network, run-time statistics are gathered during execution. Specifically, each Aurora box (a CU in DyKnow terminology) has its time to process a tuple measured. Further, the selectivity — meaning the expected number of output tuples per input — is measured to give an estimate of the cost of running a network. The processing time is used as one of three possible ways to measure the quality of service (QoS) for the system. Aurora relies on the user (application administrator) to provide the QoS as a function of at least the delay. Other QoS functions pertain to the percentage of tuples delivered, and the received output values — both of which are optional but require a user to define them. Based on QoS metrics, Aurora tries to reorder the Aurora boxes where possible based on the predicted quality of the resulting graph. This is similar to DyKnow's configuration daemon, which can however replace entire pipelines based on semantic annotations. Just like Aurora, this can be done during run-time, and efforts are made to minimise the impact of changing the network structure on the resulting streams. Unlike DyKnow, however, Aurora has support for load shedding based on QoS information.

The work towards Aurora resulted in the exploration of distributed stream processing. *Medusa* (Cherniack et al., 2003) allows for service delivery in a federated multi-agent setting, in which nodes are administered by a single entity. Medusa seeks to solve load management by having nodes compensate each other using economical incentives. *Borealis* (Abadi et al., 2005) combines lessons from Aurora and Medusa and provided what the authors call a 'second generation stream processing engine'. In addition to Aurora's features, among other things, it considers revision records (i.e. corrections for prior tuples) and utilises distribution features from Medusa. Borealis also continues efforts towards run-time query optimisation and methods towards fault-tolerance.

11.4 TelegraphCQ

The *Telegraph project* at UC Berkeley focused on developing an 'Adaptive Dataflow Architecture' starting in 2000. *TelegraphCQ* (Chandrasekaran et al., 2003; Reiss and Hellerstein, 2005) is a realisation of the Telegraph architecture with a focus towards continuous query processing, and is based on the open-source *PostgreSQL* rela-

tional database management system. One of the key features pursued by the Telegraph project was adaptivity to the addition and removal of queries, by adjusting the processing during run-time. Specifically, Chandrasekaran et al. (2003) note that "[S]hared processing must be made robust to the addition of new queries and the removal of old ones over time, so on-the-fly adaptivity must be an essential component of any solution for shared processing of continuous queries." Importantly, Telegraph does so in conjunction with the re-optimisation of query plans.

TelegraphCQ is a modular system in which communication between modules is handled by the *Fjords* API (Madden and Franklin, 2002). It utilises components called *Eddies* to execute a routing policy between modules by sending tuples back and forth between Eddies and other modules. This makes it possible for an Eddy to enforce a particular processing order. TelegraphCQ also makes use of *State Modules* (SteMs) which can temporarily store tuples and which can be accessed by an Eddy. Since TelegraphCQ processes potentially infinite-length streams of data which arrive outside of the system's control, it makes use of finite-length windows. Queries on streams in TelegraphCQ can make use of such finite-length windows by supporting for-loops iterating over time-points. A variable `ST` is used to represent the time-point at which a query was started. The query language used extends standard SQL with the specification of the window in a for-loop to generate a relational table over which aggregation is applied.

TelegraphCQ shares similarities with DyKnow-ROS in multiple areas. The philosophy of supporting adaptive concurrent queries is shared by both frameworks; both DyKnow-ROS and TelegraphCQ handle the addition and removal of queries during run-time and apply shared processing where possible. Whereas TelegraphCQ makes use of Eddies to pass around tuples to modules, a similar function is performed by the DyKnow-ROS daemon, which is tasked with reconfiguring the computational environment in order to satisfy DyKnow targets. The responsibility of SteMs is fulfilled by computation units, which have their own storage capabilities. Whereas TelegraphCQ uses the Fjords API, DyKnow-ROS makes use of the topic and service communication supported by ROS. Unlike TelegraphCQ, DyKnow does not directly support windows in SPL, although transformations performing windowing operations could be defined.

11.5 ETALIS

The ***E**vent **T**r**a**nsaction **L**ogic **I**nference **S**ystem* (ETALIS) was originally introduced by Anicic et al. (2009) — as a footnote referring to a software package — for the purpose of logic-based event processing. A comprehensive overview of the full ETALIS system, its Prolog-based implementation, and its applications is given by Anicic (2012).

One of the main contributions put forth by ETALIS is the *ETALIS Language for Events* (ELE), which is a rule-based language for event processing and reasoning that is grounded in a formal semantics. Note however that, as with many other query languages, the name ETALIS can thus refer to both the query language and the query

engine. ELE makes it possible to specify an ETALIS rule base, denoted by $\mathcal{R}$, which is composed of *static rules* $\mathcal{R}^{\text{s}}$ and *event rules* $\mathcal{R}^{\text{e}}$. The set $\mathcal{R}^{\text{s}}$ is composed of Horn clauses describing a background knowledge base, whereas the set $\mathcal{R}^{\text{e}}$ is composed of temporal event patterns. These patterns describe the temporal relationship between events using a set of temporal relations that extends Allen's interval algebra (Allen, 1983) with quantitative time intervals describing windows. The composition of events then makes it possible to describe complex events utilising these temporal patterns. To illustrate ELE's expressivity, the following example patterns and their natural-language interpretations have been copied from Anicic (2012) (pp 71–72):

- $(P_1).3$ detects an occurrence of P_1 if it happens within an interval of length 3, i.e., 3 represents the (maximum) time window.
- P_1 `SEQ` P_3 represents a sequence of two events, i.e., an occurrence of P_1 is followed by an occurrence of P_3 ; here P_1 must end before P_3 starts.
- P_2 `AND` P_3 is a pattern that is detected when instances of both P_2 and P_3 occur no matter in which order.
- P_1 `PAR` P_2 occurs when instances of both P_1 and P_2 happen, provided that their intervals have a non-zero overlap.
- P_2 `OR` P_3 is triggered for every instance of P_2 or P_3 .
- P_1 `DURING` $(0$ `SEQ` $6)$ happens when an instance of P_1 occurs during an interval; in this case, the interval is built using a sequence of two atomic time-point events. In general, the interval may consist of other (derived) events too.
- P_3 `STARTS` P_1 is detected when an instance of P_3 starts at the same time as an instance of P_1 but ends earlier.
- P_1 `EQUALS` P_3 is triggered when the two events occur exactly at the same time interval.
- `NOT` $(P_3).[P_1, P_1]$ represents a negated pattern. It is defined by a sequence of events (delimiting events) in the square brackets where there is no occurrence of P_3 in the interval. In order to invalidate an occurrence of the pattern, an instance of P_3 must happen in the interval formed by the end time of the first delimiting event and the start time of the second delimiting event. In this example delimiting events are just two instances of the same event, i.e. P_1.
- P_3 `FINISHES` P_2 is detected when an instance of P_3 ends at the same time as an instance of P_2 but starts later.
- P_2 `MEETS` P_3 happens when the interval of an occurrence of P_2 ends exactly when the interval of an occurrence of P_3 starts.

The task for the ETALIS engine is to generate complex events matching user-provided event patterns. These event patterns are compared against potentially many input event streams, and the engine may utilise high-level domain knowledge (e.g. static rules) in conjunction with these input streams to facilitate this process. The end product is then a stream of detected complex events.

There are some similarities between DyKnow and ETALIS with regards to ELE. Whereas DyKnow's stream processing support as presented in this work can be categorised as falling under DSMS rather than CEP, the prior work on object linkage structures for DyKnow was based on chronicle recognition, which can be regarded as an early form of CEP. ELE additionally introduces temporal ranges or windows for otherwise qualitative temporal relations, which created some overlap with MTL as used by DyKnow in conjunction with progression. Finally, without going into detail, ETALIS also considers the problems of event retraction and out-of-order events, which are important problems, but neither of which are currently handled by DyKnow.

11.6 Retalis

The *Retalis* (ETALIS for Robotics) framework (Ziafati et al., 2015; Ziafati, 2015) focuses on stream reasoning within robotics applications using ROS, and is therefore closely related to DyKnow. Retalis combines ELE with the *Synchronized Logical Reasoning* (SLR) language, originally proposed by Ziafati et al. (2013). SLR is a formal logical language for knowledge management in robotics applications, where events are used to represent observations which may be reasoned with. SLR programs are composed of rules represented as Horn clauses, which allows Retalis to draw conclusions from robot observations. Retalis uses Prolog to parse and execute programs.

ELE is used to generate complex event streams, which are fed to SLR as inputs. By adding these events as facts, the knowledge base described by SLR changes, resulting in the inference of new facts that can be represented as events. These events, in turn, can be used by ELE in its generation of complex event streams. Similarly, an ELE pattern can include SLR queries. The combination of ELE and SLR takes place in an autonomous component called an *information-engineering component* (IEC) in Retalis. Each IEC can receive and produce event streams, process queries, and maintains a knowledge base that gets updated incrementally upon receipt of events.

Because streams produced based on ELE patterns may be infinitely long, Retalis is able to perform memory management (Ziafati et al., 2014) on its stored event history by using buffers to limit the size of the knowledge base. Additionally, an IEC must be able to specify streams of interest and be able to subscribe to those streams, potentially during run-time. Retalis therefore supports *run-time subscriptions*, which specify a query pattern describing an event atom and conditions on its arguments, and a query window restricting the time interval for matching events. Any events matching the query patterns are sent to the respective topics. Subscriptions are entities with identifiers, such that one can unsubscribe by specifying the

identifier of the subscription that needs to be terminated. These run-time subscriptions are set up through a ROS service hosted by a Retalis-ROS interface.

Retalis shares a lot of similarities with DyKnow due to its focus on stream reasoning in the domain of robotics applications. Both approaches use ROS due to its prevalence in this area, and so both approaches could technically be used side by side within the same system. The two approaches are complementary; whereas Retalis focuses on maintaining a knowledge base through incremental updates based on event patterns, DyKnow focuses on maintaining streams of interest that can be used by a system that builds upon the DyKnow framework. DyKnow does not explicitly store histories in a knowledge base, although its transformations can keep such histories for the purpose of stream refinement. Retalis event streams are therefore also at a higher level of abstraction compared to DyKnow's streams, which do not contain events but rather represent fluents. Both approaches also provide functionality to set up subscriptions during run-time, but whereas Retalis pulls in event streams for filtering in accordance with an event pattern, a DyKnow target results in a reconfiguration based on semantic annotations. If one regards a complex event specification as a semantic annotation, Retalis can be argued to set up and manage its own cyclic stream processing environment. It however exists only within an IEC, whereas DyKnow's computational environments are composed of structurally-dynamic networks of stream processing nodes connected by streams.

Implementation-wise, Retalis and DyKnow both have to consider the interface with ROS. Retalis makes use of the Python execution environment for running Retalis programs, and employs a Retalis-ROS interface module to import ROS topic data into an IEC represented by a ROS node. DyKnow instead makes use of a DyKnow daemon, also represented by a ROS node, which keeps track of DyKnow proxies. These proxies are in control of ROS subscribers and publishers, and provide ROS services to adjust the topics they are connected to. In this sense, DyKnow is more of an extension of the ROS framework, whereas Retalis is integrated in ROS.

11.7 T-Rex

T-Rex[16] is a CEP middleware introduced by Cugola and Margara (2012b), which makes use of the ***T**RIO-based **E**vent **S**pecification **La**nguage* (TESLA) (Cugola and Margara, 2010) for describing complex event patterns. *TRIO* (Ghezzi et al., 1990) refers to the temporal first-order logic in which the TESLA semantics is described.

A TESLA rule are composed of (at most) four clauses, in accordance with the following structure:

$$\begin{aligned} &\textsf{define } CE(Att_1 \ : \ Type_1, \ldots, Att_n \ : \ Type_n) \\ &\quad \textsf{from } Pattern \\ &\quad \textsf{where } Att_1 = f_1, \ldots, Att_n = f_n \\ &\textsf{consuming } e_1, \ldots, e_n \end{aligned}$$

[16] Also sometimes written as 'T-REX', but does not appear to be an acronym.

The *define* clause allows for a user to define the name (i.e. CE) and payload (i.e. attributes and their types) of a complex event. The *from* clause describes a pattern in terms of simple events, which can include event composition through the use of windows and aggregation. This is one of TESLA's differentiating features, since these types of window aggregations are more common in DSMS than in CEP systems. The (optional) *where* clause can be used to apply filtering on the payloads of those simple events through the use of comparators. Finally, the (optional) *consuming* clause is used to select a consumption policy, which is another feature TESLA supports that differentiates it from its contemporaries. A consumption policy determines whether an observed event is removed (consumed) after having been used to generate a complex event, as well as when to stop consuming events like it. To illustrate the application of the consumption policy, Cugola and Margara (2010) often make use of a 'fire' event, which is triggered when a sequence of high temperatures is followed by smoke. If the high temperature event is selected for consumption, all high temperature events in the sequence are removed, thus requiring the detection of new high temperature events followed by smoke before the complex 'fire' event is triggered for a second time.

The T-Rex engine translates TESLA rules by compiling them into event detection automata. It then uses these automata for efficient event notification. Cugola and Margara (2012b) provide an extensive empirical evaluation of the T-Rex engine's performance, focussing primarily on throughput. The engine itself was written in C++, but provides adapters for remote clients written in C++ or Java.

11.8 LARS

LARS is a ***L**ogic-based framework for **A**nalysing **R**easoning over **S**treams* by Beck et al. (2014, 2015) and provides a logical formalisation of stream reasoning. LARS considers stream reasoning to be logical reasoning on streaming data, and therefore takes an approach wherein streaming data is modelled logically, i.e. as predicates. This approach shares similarities with DyKnow's state streams, which carry the truth values of predicates over time as well. Unlike DyKnow, however, LARS does not consider the production of state streams.

Key contributions presented as part of the LARS framework are reported (Beck et al., 2015) to include

1. a rule-based formalism for reasoning over streams;
2. different means to refer to or abstract from time; and
3. a window operator to this effect.

The window operator $\boxplus^{\mathbf{x}}_{\iota,ch}$ is applied to a stream S in order to produce a resulting stream S', where ι indicates a window type, ch a stream choice function, and $\mathbf{x}$ a vector of window parameters. The window type ι is used to identify a window function w_ι. It maps from an input stream S, a reference (starting) time point t,

and parameters **x** to a *substream* $S' \subseteq S$. LARS has successfully modelled time-based, tuple-based and partition-based windows, making it expressive enough to capture languages such as CQL (Arasu et al., 2003, 2006). Implementations of LARS reasoners for example include *Laser* by Bazoobandi et al. (2017), and *Ticker* by Beck et al. (2017).

LARS' window operator can be used to filter elements from a stream and apply logical reasoning to the resulting substream, thereby providing different potential views. In the DyKnow model, a window operator would instead exist as a transformation that filters a stream based on windowing conditions, rather than be part of the logical representation. DyKnow's computational environment can also make a distinction between a filtering operation akin to the LARS substream-producing windowing operation on the one hand, and the case wherein every sample contains a window on the other hand. It is presently unclear how this distinction could be leveraged in the LARS framework. In conclusion, LARS shares similarities with DyKnow in terms of reasoning with the help of transformations on streams, which allow LARS to switch views and make logical statements on those views.

11.9 SECRET

Similar to LARS, *SECRET* is a model for analysing the execution semantics of stream processing systems proposed by Botan et al. (2010). The motivation behind SECRET is rooted in the existence of multiple stream processing engines, each with their own capabilities and semantics, and the desire to compare the execution behaviour of these heterogeneous stream processing engines. In particular, the heterogeneity manifests itself in terms of syntax, capability, and the execution model. SECRET is (arguably loosely) named after the four dimensions it considers; ***s**cop**e**, **c**ontent*, ***re**port and **t**ick*. Dindar et al. (2013) consider these four dimensions with SECRET in their coverage of the heterogeneity of the Coral8, STREAM, StreamBase, and Oracle CEP stream processing engines.

SECRET considers streams to be countably infinite sets of elements $s \in \mathbb{S}$, such that a stream element (or a sample in DyKnow's terminology) is described by $\langle v, t^{app}, t^{sys}, tid, bid \rangle$. Here v denotes a relational tuple conforming to a schema S (i.e. a table), $t^{app}, t^{sys} \in \mathbb{T}$ denote the application time and system time, and tid, bid denote tuple ID and batch ID values. This type formalisation of a stream is similar to DyKnow, which considers named structured values that could be represented as done in SECRET. A batch $\mathbb{B}$ is described as a set of stream elements such that each element making up a batch has the same t^{app} as all other elements of that same batch. State streams in DyKnow could thus be described in terms of batches. Finally, as in LARS, SECRET describes a variety of window semantics using the definition of a stream, where a window over a stream produces a substream. In particular, SECRET describes time-based windows and tuple-based windows with varying window sizes and slides.

A key motivation for SECRET was the heterogeneity in the window operations supported by various stream processing engines. SECRET thus captures the window-based query execution semantics along the aforementioned four dimensions. **Scope** deals with the scope of a window, meaning the window intervals, given a set of parameters. Scope can be interpreted differently by different stream processing systems. **Content** deals with how the scope of these windows translates into the content of the produced substreams given an input stream. The content is then commonly sent on for processing, such as for example aggregation. *When* the content becomes visible to the query processor can vary by system. **Report** states the conditions on when content becomes visible. Lastly, **tick** deals with the control loop of a stream processing engine, and in particular *when* it acts on a given input stream. Given these four dimensions, Dindar et al. (2013) consider both time-based and tuple-based windows for the aforementioned stream processing engines.

SECRET is primarily a tool for analysing different stream processing engines. As with LARS, SECRET has some overlap with the formal specifications of DyKnow. The main difference between LARS and SECRET appears to be the level of detail; LARS provides high-level semantics relative to a logical model, whereas SECRET is closer to the operational semantics of a set of pre-existing stream processing engines. In both approaches, the semantic of the window operator were a primary point of attention. DyKnow currently does not support window operations directly, although windowing does take place in the form of interval-bounded temporal operators. Nevertheless, SECRET's formal specification of window operations can be of use when considering similar operations such as merging and synchronisation as part of for example state stream generation in DyKnow.

11.10 RSP

RDF Stream Processing (RSP) refers to stream processing techniques that assume streaming data to be formatted in the RDF data format. This data format is usually represented as RDF triples, consisting of subject, predicate, and object resource identifiers. RSP is distinct from continuous query languages due to its connection to Semantic Web ontologies represented as knowledge graphs. The identifiers occurring in triples are commonly associated with such ontologies, and an RDF stream can then be regarded as a dynamic subgraph. Queries posed in an RSP setting may thus pertain to both the dynamic and static parts of an ontology. A change in the dynamic subgraph has as an important consequence that the implicit facts in the complete graph may change as well, affecting the result of a continuous query. There are different ways for handling the changes described by RDF streams, impacting the performance of continuous query engines in different ways, the details of which are outside of the scope of this dissertation.

RSP holds an interesting position within the area of stream reasoning not only due to its large system contributions to stream reasoning in the form of query engines and tools, but also due to the way it is positioned relative to stream process-

ing. While RSP engines — by their definition — perform stream processing tasks, each RSP triple processed from a stream has the potential to trigger a Description Logic-based reasoning process, albeit atemporal, followed by unification and window-based aggregation.[17] This clearly moved beyond relatively simple filtering as provided by traditional database systems. RSP could thus be regarded as straddling the Interpretation–Verdict range in the stream reasoning pipeline, with little focus on the issue of RDF triple provenance, i.e. the issue of generating RDF triples from real-world data while handling issues like uncertainty. This is one area where the work presented here could potentially be adapted towards RSP. Systems like DyKnow-ROS do not assume a specific data-type, and previous work towards this dissertation (de Leng and Heintz, 2014) provided an initial discussion of the suitability of RSP engines as CUs within the scope of the DyKnow model from Chapter 7.

RSP engines

Several querying engines and languages have been designed for RSP, usually based on a continuous version of the SPARQL query language for RDF graphs. These engines are responsible for transforming RDF streams, taking into account background knowledge in the form of an ontology. In the following, we look at some of the more common instances.

C-SPARQL. The *Continuous SPARQL* (C-SPARQL) language is a pure extension of the SPARQL query language, originally introduced syntactically in Barbieri et al. (2009, 2010c). The semantics of C-SPARQL were subsequently presented in Barbieri et al. (2010b,a), together with an execution environment by the same name. C-SPARQL introduces keywords allowing a user to specify a stream resource to query using a tumbling or sliding window. The resulting tables can be aggregated using aggregation functions such as sum, count, average, maximum, and minimum.

SPARQL$_{\text{Stream}}$. *Streaming SPARQL* (SPARQL$_{\text{Stream}}$) is a query language by Calbimonte et al. (2010) which is based on SPARQL. It takes an ontology-based data access (OBDA) approach to streams, where queries are written using ontological concepts. These queries are then automatically translated to access specific streaming data resources. The target language for the query rewriting is the ***S**ensor **Ne**twork **E**ngine **q**uery **l**anguage* (SNEEql) by Galpin et al. (2009).

EP-SPARQL. Recall that CEP systems focus on the detection of complex events from sequences of events. *Event Processing SPARQL* (EP-SPARQL), introduced by Anicic et al. (2011, 2012), focuses specifically on events and allows for the querying of event patterns by their temporal relationship. As is usual for CEP systems, this means EP-SPARQL does not use windowing and aggregation operations. Instead, sequences of

[17] This type of materialisation process is not necessarily performed by all RSP engines, just like not all SPARQL query engines consider implicit facts.

events can be detected, and the temporal distance between events can be used in a filter. EP-SPARQL, like C-SPARQL, is a pure extension of SPARQL that adds additional keywords for describing event sequences.

EP-SPARQL is an application and extension of ETALIS; queries are compiled into ELE rules and RDF streams and ontologies are converted into the ETALIS ELE format.

CQELS. The above RSP engines are layered 'on top' of pre-existing engines, i.e. they rewrite queries into SPARQL queries or ELE facts and then utilise a SPARQL engine or ETALIS implementation to perform the reasoning. Le-Phuoc et al. (2011) point out that this amounts to what they call a 'black box' approach, where control of the way queries are executed is instead delegated to another engine. They therefore proposed the *Continuous Query Evaluation over Linked Streams* (CQELS) engine, which instead handles these tasks natively. This gives CQELS control over aspects such as data encoding and caching, yielding an overall good performance. The CQELS language itself is again an extension of the SPARQL grammar.

RSP-QL. RSP originally continued the same pattern forming the basis for efforts such as LARS or SECRET; different RSP implementations used different semantics for windowing operations, resulting in different answers depending on the system used. While the representation of RDF graphs is well-defined, the content of RDF streams is not. Furthermore, since operations on RDF graphs were time-invariant (incorporating time into ontologies is a difficult open problem), combining streams with ontologies resulted in different approaches. The *RSP Query Language* (RSP-QL) was therefore proposed by Dell'Aglio et al. (2014) as a unifying query model to explain the heterogeneity of these various RSP languages. To this effect, it extends the SPARQL model and bases off the CQL and SECRET models.

RSP orchestration

The orchestration of RSP beyond single engines appears to have only started recently within the Semantic Web community. It shared some similarities with older work towards semantic web services, but with a specific focus on the generation of RDF streams and the transportation mechanism for such streams within the domain of existing Web-based communication technologies. In the following, we look at approaches towards the generation of RDF streams, the orchestration of RSP tools, and the possibility of annotating RDF streams by treating them as first-class citizens.

TripleWave. As mentioned earlier, raw data streams usually do not follow the RDF structure. This makes the distribution of such streams more complex than simply taking existing RDF data and streaming this data. TripleWave is a framework that allows users to distribute RDF streams on the web, and was originally proposed by Mauri et al. (2016). It considers both RDF and non-RDF resources and provides the means to stream these resources as RDF streams. For RDF resources, this includes the streaming of time-annotated datasets and the replaying of recorded RDF

streams, also allowing these replays to be looped to generate an infinite-length stream. For non-RDF resources, plugins exist that convert to JSON raw data from for example social media or open-source encyclopedia. The resulting JSON data can then be converted into an RDF stream. In doing so, TripleWave offers a solution to the problem of generating RDF streams from data providers that may not necessarily support the RDF format natively.

WeSP. Previously, we looked at several different RSP engines for querying RDF streams. Similarly, systems such as TripleWave act as sources of such data. The WeSP framework by Dell'Aglio et al. (2017b) is tasked with connecting these sources to graphs of potentially many RSP engines, using existing Web-based technologies (i.e. HTTP, Websockets) for realising communication. WeSP therefore defines communication protocols that can be used by different RSP engines to establish RSF stream-based communication in a network of engines. They additionally describe RDF documents called *stream descriptors*, which describe a stream at the metalevel. This follows a similar approach taken in the development of SSL, discussed in Chapter 6, and the DyKnow model, presented in Chapter 7.

VoCaLS. One important part of orchastration is the availability of a vocabulary to describe streams and transformations. The ***Vo**cabulary for **Ca**taloging and **L**inking **S**treams and streaming services on the web* (VoCaLS) was introduced by Tommasini et al. (2018, 2019) for this purpose. It provides a vocabulary for annotating streaming services and transformation, and makes it possible to annotate streams with provenance information describing the process through which they are generated. VoCaLS can thus provide a realisation of WeSP's stream descriptors. This follows the same line of work as presented here in Chapter 7, where the DyKnow model can be expressed using the DyKnow ontology.

11.11 PEIS

Research towards analysis of stream reasoning such as proposed as part of LARS, SECRET and to some extent RSP generally ignores questions of integration into a larger (eco)system. Saffiotti et al. (2008) presented the *PEIS ecology*[18] for ***P**hysically **E**mbedded **I**ntelligent **S**ystems*. The cornerstone of the PEIS ecology is its conceptualisation of physically embedded intelligent systems (PEIS) as agents that operate in a physical environment and are themselves physical entities. Every PEIS is assumed to at least have

1. some computational resources;
2. some communication resources; and

[18] Pronounced '*pace* ecology'

3. sensors and/or actuators allowing the system to interact with the physical environment.

Consequently, PEIS are assumed to be heterogeneous entities with different capabilities. A PEIS ecology consists of potentially many PEIS, each with their own functionalities and communication capabilities. While the PEIS ecology considers communication problems, DyKnow instead chooses to use ROS as a commonly-used platform that provides communication support. The PEIS ecology as a whole is intended to solve problems in a multi-agent organisation setting by interacting with the physical environment.

Lundh et al. (2008) focused on the problem of self-configuration and proposed techniques for configuration planning. The underlying motivation is that in the PEIS ecology robots can and should help other robots to collectively achieve goals common to the ecology they are part of. Functionalities are formalised in a logical representation that can be used by general planners. Given a goal, the planner is able to find a set of functionalities that, when activated, fulfill the goal. This approach shares similarities with DyKnow's semantic subscriptions. Both consider a computational environment in which functionalities can be activated or transformations can be instantiated for a cost. However, in DyKnow this cost is estimated and may change over time, whereas the PEIS ecology uses simple constant values. Furthermore, DyKnow's similarity relation is based on the semantic tags of transformations, whereas the PEIS ecology matches propositional statements. Both the lack of meaningful cost measures and the potential value in using semantic descriptions were later identified (Lundh, 2009) as future work. On the other hand, the PEIS ecology is able to model actions taken by PEIS at the level of configuration planning, whereas DyKnow can only consider stream processing without taking into account the actions of agents. The preconditions for transformations are not explicitly modeled in DyKnow either; transformations are expected to only be available when preconditions are met, as exemplified in the synergy scenarios. DyKnow focuses to a large degree on maintaining semantic subscriptions and therefore emphasises the need for efficient and fast reconfiguration in light of failures. The PEIS ecology instead focuses on achieving a goal in a physical environment, where the configuration of functionalities of PEIS plays one role. DyKnow and the PEIS ecology are thus complementary in their results, where the difference in motivations means there is a different focus.

Moving from the configuration-centric abstraction level down to the data-centric abstraction level, Alirezaie (2015) more recently focused on the problem of streaming data semantics. In particular, the focus was on bridging the semantic gap between sensor data and ontological knowledge, which is reminiscent of the sense-reasoning gap that was the motivation (Heintz et al., 2010) behind earlier DyKnow efforts. The semantic gap between sensor data and ontological knowledge is described as the disconnect between quantitative sensor values and crisp high-level knowledge encoded into ontologies. Alirezaie (2015) focuses on two aspects. First, correspondences between sensor data and conceptual knowledge needs to be automatically determined. Second, the two types of information are combined in an in-

ferencing process. In particular, the focus is on enriching the sensor data, meaning it is 'lifted up' to the conceptual level. This is different from DyKnow's approach of describing the low-level sensor information using high-level concepts, as this is purely descriptive rather than formative. The use of CEP on semantic events obtained from sensor information is an interesting approach currently not used by DyKnow.

Overall, the PEIS ecology shares many similarities with the DyKnow project. Both efforts consider a larger integration problem in which stream reasoning combining sensor data with high-level knowledge is essential for decision-making, albeit from different angles.

11.12 Summary

The research presented in this dissertation focuses on robust stream reasoning under uncertainty. In doing so, it also considers the application area of intelligent robotics. In this overview of related work, we covered a wide area of work pertaining to stream reasoning for various application domains.

This includes early work on DSMS for stream processing, such as STREAM with its CQL, Aurora with its SQuAl, and TelegraphCQ with its iterable time windows; each supporting some form of windowing to handle the potentially infinite-length streams they process. We also discussed early CEP systems, such as ETALIS with its ELE language generalising Allen's interval algebra, or T-Rex with its TESLA language that also supports window-based aggregation, further blurring the boundaries between DSMS and CEP. We also discussed the various ways RSP has pushed the boundaries of stream reasoning, and where the term was coined originally. Some of the RSP engines mentioned make use of some of the languages mentioned earlier; EP-SPARQL combines ELE with SPARQL. But RSP also considers a background knowledge base in the form of an ontology, which must be taken together with a stream to perform stream reasoning. Retalis takes a similar approach, extending ELE with knowledge base management using rules written in SLR, combining a stream with an incrementally-updated knowledge base.

One of the lessons learned in RSP research was the difficulty in formalising the semantics of RSP languages. SECRET was one formalisation of stream reasoning, considering a formal definition of streams and windows on streams. SECRET was used in combination with CQL and SPARQL to develop RSP-QL. At around the same time, LARS was developed to also formally describe streams and windows on streams, and was used to describe the semantics of CQL as an illustration of its expressiveness. The LARS framework was also realised; several implementations of reasoners for LARS fragments exist, including Laser and Ticker.

Yet none of the above systems, with the exception of Retalis, specifically focused on intelligent robotics. This application domain has its own difficulties, including the problem of having to cope with low-level sensor information, whereas the above systems commonly expect crisp relational data or RDF triples. Another issue is that this information may originate from different streaming resources. Some

of the early work on stream reasoning did consider a changing stream processing environment. STREAM used the StreaMon monitoring and adaptive query processing infrastructure, which tried to re-optimise the query processing whenever necessary. Aurora specifically considers user-defined quality of service, and tries to optimise that during run-time. Borealis was a continuation of Aurora that incorporated Medusa, which considered a multi-agent setting in which nodes compensated each other based on economical incentives. TelegraphCQ made use of Eddies for routing streams, and Retalis extended ETALIS with support for run-time subscriptions that may change dynamically.

More advanced orchestration of stream processing is less common. PEIS specifically considers the sharing of information between separate physical agents to achieve common goals. It does so by formalising the agents' functionalities and applying configuration planning to align these functionalities when needed. On the RSP side, recent developments as part of WeSP considered graphs of interconnected RSP engines communicating using standard Web-based technologies. Systems like TripleWave focused on the generation of RDF streams from both RDF and non-RDF data resources. This was further complemented with support for the semantic annotation of streaming services using VoCaLS. The combination of the two can be part of semantically-aware RSP orchestration.

The work towards DyKnow thus covers a fairly wide range of related works. On the one hand, there is the reasoning over streams, ranging from simple processing to logical reasoning tasks with background theories. On the other hand, there is the support for reasoning about streams, dealing with the smart orchestration of stream processing to achieve goals. Overall, one can regards the DyKnow system as being similar to a hypothetical combination of Retalis with PEIS; sharing some similarities in their features, while complementing both.

Part V

CONCLUSIONS

Chapter

12 Conclusions and future work

THIS dissertation presents a logic, algorithms, formal models, semantic representations, integration, a concrete implementation, and a case study for robust spatio-temporal stream reasoning under uncertainty. The logic MSTL was used to make spatio-temporal statements, and of which the truth value can be robustly determined even in the face of incomplete information and unexpected changes in the availability of (latent) streams. The presented work is multidisciplinary in nature, resulting in the focus on the development and integration of two related strands. This chapter first provides a high-level summary of the contributions, revisits the research questions and considers open problems, before considering potential future work.

12.1 Overview

The results presented in this work represent the latest achievements within the DyKnow project, divided into two integrated strands. The first strand focused on stream reasoning under uncertainty, where we specifically looked at path checking over sets of states representing different consistent hypotheses. This can be used for performing spatio-temporal stream reasoning with MSTL. MSTL was presented as an extension of MITL by incorporating RCC-8 for qualitative spatial reasoning, allowing for spatio-temporal statements to be made. The truth value of these statements can be determined incrementally using an extended version of progression. These statements can further contain intertemporal spatial relations similar to ST_1. Importantly, we assume that these intertemporal spatial relations cannot be observed directly, and thus need to be inferred. Without any additional information about intertemporal relations, nothing is known about them. Our solution therefore makes use of landmark regions which can reduce the uncertainty over intertemporal spatial relations.

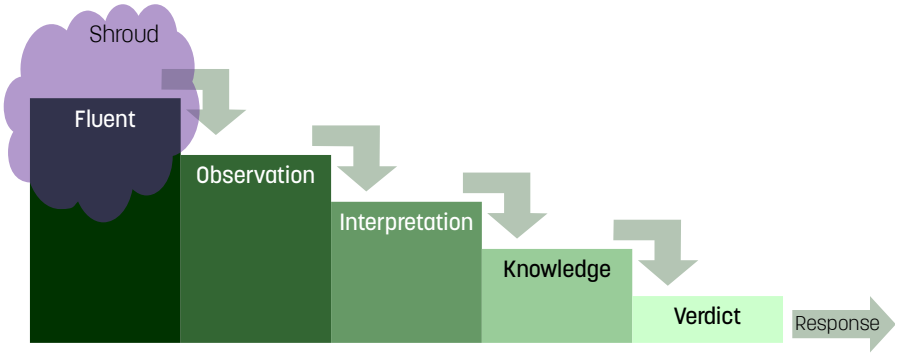

Figure 12.1: A simplified version of the stream reasoning waterfall model.

The second strand focuses on the problem of generating a state stream over which a formula can be evaluated. The symbols in a formula are therefore grounded in a computational environment through syntactic or semantic subscriptions, such that the truth value of these symbols depends on the data that is produced by this underlying environment. Semantic annotations of the logical symbols (through the use of targets) as well as the available stream transformations allow us to find suitable configurations of the computational environment that produce a state stream containing the information necessary to evaluate a formula. By reconsidering the configuration periodically, the computation graphs can be repaired or improved in case where the underlying system changes unexpectedly. This ensures that the progression of a formula is not necessarily interrupted or fails as the result of such changes, making the system more robust. Additionally, the configurations can be expressed relative to a Semantic Web ontology, allowing for the exchange of configuration information.

The two strands were integrated into a single stream reasoning framework in which reasoning about streams synergises with reasoning over streams. The stream reasoning waterfall is shown once more in Figure 12.1, and shows the various steps from fluents down to verdicts, which may elicit a response. The resulting DyKnow model was integrated with ROS and allows existing ROS nodelet implementations to be used in DyKnow with minimal overhead in terms of delays and developer burden. This concrete implementation was then deployed on NAO platforms, adapting software produced by the Linköping RoboCup SPL team to be usable by DyKnow for a case-study that highlights the added value of adaptive reconfigurability during stream reasoning tasks.

While the focus of the work was primarily on robotic applications, the solutions are general and do not rely on specific supporting software such as ROS. For example, experimental CUs have been written for non-robotic domains such as Twitter, or to interact with DigitalOcean's cloud computing API by instantiating, managing, and destroying virtual machines in off-platform data centres. This highlights potential applicability of the presented solutions to much broader application areas that

involve many diverse computational resources, for example smart cities or sensor networks, making them potentially interesting to industrial applications of this kind. The computation resources also do not necessarily have to be physical. One can imagine virtual services that deal with areas such as advertisement, travel agencies, or stock markets wherein financial information and their sources may change continually. In fact, many CEP languages have query examples that deal precisely with stock market events.

12.2 Conclusions

In the introduction covered by Chapter 1, the following research questions were posed:

- **[RQ1]**: How can uncertainty be formally modelled for the purpose of logical stream reasoning?
- **[RQ2]**: How can a spatio-temporal logic be constructed by combining spatial and temporal formalisms, and how can statements in such a logic be tested for satisfaction given a stream?
- **[RQ3]**: How can a stream be generated for the purpose of symbol grounding?
- **[RQ4]**: How can the procedure for generating a stream for the purpose of runtime verification be made robust to changes that affect its ability to keep generating such a stream?
- **[RQ5]**: How can the techniques developed towards answering the aforementioned research questions be leveraged in a concrete middleware framework such as the Robot Operating System?

We can now revisit the contributions in this dissertation that seek to answer these questions.

Modelling uncertainty for the purpose of logical stream reasoning

The need to model uncertainty when performing logical stream reasoning is based on the introduction of uncertainty when making observations of an environment, and the need to represent this uncertainty at higher levels of abstraction as well. We focused primarily on representing uncertainty by considering multiple hypotheses, and keeping track of these hypotheses. Chapters 3 and 4 formalised the concept of an incomplete stream as a sequence of incomplete states, each of which represents multiple hypotheses with potentially different probabilities. The progression procedure by Bacchus and Kabanza (1998) was enhanced with rewriting rules, shown in Table 3.1, allowing for formulas to be simplified such that their length is reduced. Since the time and space complexity of progression are based on formula size, formula simplification can make progression more efficient.

Satisfaction-checking spatio-temporal statements

We made use of progression to determine whether a stream satisfies a formula, because of the incremental nature of progression. Chapters 4 and 5 considered path checking to determine whether a stream satisfies a logical formula, with the latter extending this to the spatio-temporal logic MSTL. The semantics of MSTL was provided in Definition 5.3 and combines MITL with RCC-8. Uncertainty in terms of incomplete streams is propagated into the task of path checking because an incomplete stream represents a potentially large collection of possible complete streams. The uncertainty is efficiently kept track of by utilising progression graphs — shown to be correct in Theorem 4.4 — which keep track of a probability mass distribution representing the probability of progression having ended up in a particular formula given an incomplete stream prefix.

Generating a stream for symbol grounding

Symbol grounding is used to give meaning to the symbols used to represent propositions in logical formulas. Chapter 6 shows how subscriptions can be used to obtain the necessary state information, and how background knowledge can be used to enhance such states. The chapter also showed three languages for stream processing; SPL, SSL, and FSL. SPL and FSL allow a user to filter, combine and otherwise transform streams using descriptive SQL-like queries, whereas SSL allows a user to semantically annotate streams and transformations. Chapter 7 introduces a formalisation of the concepts of transformations, computation units, and targets. The semantic description of transformations allows for the automatic configuration of a system to generate a stream described by its semantics. This makes it possible for a user to not have to care about how the stream is generated. We also looked at an ontology, shown in Figure 7.1, to represent a snapshot of a computational environment.

Robust stream generation under change

Adaptive semantic subscriptions are robust to changes affecting the computational environment's abilities to transform streams. Chapter 8 formalises the concept of a perturbation and introduced the problem of finding the optimal change set to recover from a perturbation. To also utilise possible improvements, Algorithms 8.1, 8.2 and 8.3 use periods of exploration and exploitation as part of an update procedure.

Application in a concrete middleware framework

DyKnow-ROS is an implementation of the DyKnow model in the Robot Operating System (ROS). Chapter 9 shows how the model can be realised by describing the required ROS-based services. To perform reconfigurations, additional control is needed in the form of proxies. Chapter 10 finally covers case-studies involving

DyKnow-ROS as a proof of concept by focusing on the robust generation of a stream needed to evaluate a formula.

12.3 Limitations and open problems

While the work presented in this dissertation is interdisciplinary, this also invariably means that there are limitations to aspects of the presented work. We therefore focus on the limitations of the results presented, and consider some problems which have not yet been resolved. We do so by considering the relevant parts this work is composed of (i.e. Parts II, III, IV) in isolation, as they represent different — albeit related — strands.

Stream reasoning under uncertainty

Part II focuses on contributions towards stream reasoning under uncertainty. In this work, the emphasis was on a specific kind of stream reasoning, i.e. path checking. The approach foresees the use of background theories when performing state stream synthesis, which is required for spatio-temporal stream reasoning using MSTL. A closer integration of state stream synthesis with progression remains an open problem. One idea here is to encode the background knowledge into the progression graph by removing edges labelled with states which are inconsistent when combined with the background knowledge. This could be used to further limit the size of progression graphs. An investigation into the potential interaction between graph-based progression and reasoning with background knowledge has also been left for future work. Of particular interest is ASP-based reasoning, which has previously been shown (Brenton et al., 2016) to also be able to perform qualitative spatial reasoning tasks. Another open issue concerns the potential to use verdict streams to generate new incomplete state streams. This would allow for the reasoning about satisfaction probabilities within the logic itself. Finally, we considered a specific type of uncertainty, and an investigation of additional alternatives is an open problem. One potentially interesting approach is to consider a variant of the probability thresholding operator $\mathbb{P}_{>p}(\phi)$ recently proposed by Koopmann (2019) in the context of OBDA, or to further develop a probabilistic extension of STL as proposed by Tiger and Heintz (2016).

Adaptive stream processing

Part III focuses on contributions towards robust stream reasoning through adaptive stream processing. The DyKnow model seeks to reconfigure the computational environment by attempting to reach a goal configuration represented by a set of targets, while at the same time keeping the configuration's cost low. The choice of cost measures for CUs is however notoriously difficult. Previous work, for example Lundh (2009), notes the same difficulties and instead simplifies the problem by assigning constant utility values. It seems more likely, however, that the cost of CUs

would change based on the context of the operations. It would be interesting to see how well a predictive cost model could be learned in terms of computational resource usage, and which features would be the most informative for these predictions. While the model presented in this chapter provides a framework for using such predictions, learning good estimators is beyond the scope of this work. The DyKnow model does consider the cost of environments, but it does not consider the utility of the produced streams. In some implementations, a higher upkeep is associated with a higher-quality data stream. The representation of utility and the trade-off between cost and utility are interesting open problems. Lastly, the presented approach allows for the configuration model to be represented relative to a Semantic Web ontology. This is done because we foresee future configurations spanning multiple agents in a multi-agent organisation, but additional work towards this type of support is necessary.

Applied stream reasoning

Part IV focuses on applied stream reasoning and presents a realisation of the DyKnow model in ROS, called DyKnow-ROS. DyKnow-ROS relies on nodelets for dynamic instantiation of CUs. This presents some practical problems. First, this excludes ROS nodes, since these can only be started by command-line or via `roslaunch`. Currently, node-based implementations have to be converted to nodelets, although many support both types. The second issue is that a crash of a nodelet brings down the nodelet manager, and thereby all CUs that are running as part of that nodelet manager. This means that many if not all CUs crash if one does, and recovery then requires a new nodelet manager process to be started. Some additional engineering efforts are needed to resolve these practical issues. ROS has some known shortcomings in terms of communication guarantees, making it less useful for real-time applications. A new version of ROS, going by the name ROS2, is under development. It would be interesting to see how ROS2 could be combined with the DyKnow model for a potential DyKnow-ROS2 realisation with real-time guarantees. Another issue is the realisation of an optimisation problem for the computational environment. Targets currently only consider cost, without considering *quality*. This prevents certain solutions from being chosen if they are more expensive, regardless of their quality being greater than that of cheaper solutions. As an example, sometimes redundant information can be useful. One situation wherein this is the case is sensor fusion. Given multiple sources of position information for an object, combining these sources may lead to a better position estimation. However, since this requires multiple pipelines and thus more upkeep costs, these solutions will never be chosen. It would be interesting to see how one could extend the approach presented here to a multi-target optimisation problem in which the cost is minimised and the quality is maximised. The synergy effect is demonstrated in terms of reasoning about streams supporting robust reasoning over streams in situations wherein the set of available computational resources changes. We have not yet explored in detail the opposite synergy direction, wherein reasoning over streams may

affect the reasoning about streams. This too is a topic left for future work. Finally, the lack of multi-agent support at this stage means that the two NAO platforms used in this case study were part of a single DyKnow instance. Effectively it was the lab that acted as an agent. Separating the two platforms over two different DyKnow instances brings new challenges.

12.4 Future work

There remains a lot of potential future work in the adaptive state stream generation strand, in addition to the limitations mentioned earlier. In particular, determining appropriate utility measures with meaningful properties is an issue. For example, if we can provide a higher-quality stream by fusing two probabilistic streams, there is still a trade-off to be made in terms of the labour and upkeep such a reconfiguration would cost. Finding a suitable trade-off between cost and utility is an important problem especially for robot applications.

For the work pertaining to reasoning over streams under uncertainty we have thus far focused on specific types of uncertainty. Specifically, we considered multiple hypothetical states at each time-point, resulting in multiple hypothetical complete streams. Probabilities were also assigned to the individual hypothetical states. Further efforts should be made to further develop the ability to handle uncertain information. One potentially interesting approach is to consider a variant of the probability thresholding operator $\mathbb{P}_{>p}(\phi)$ recently proposed by Koopmann (2019) in the context of OBDA. The support of probabilistic reasoning would be extremely useful in robotic scenarios, as in many cases the information we want to use in the crisp logical formulas is actually represented in terms of probability distributions. While it is trivial to provide mean values, this does not handle Boolean comparisons nicely, as a distribution might overlap with a threshold, thus making the truth value of the comparison inherently probabilistic. This also impacts the way state streams are synthesised, as more meta-information is required to properly combine probabilistic information of this kind. One interesting use-case would be that of automated fusion, wherein the underlying configuration manager takes into account the possibility of fusing probabilistic data streams in certain contexts. Another is to further investigate the integration of reasoning with a background theory into progression graphs, where such background theories could be used to eliminate edges corresponding to inconsistent states.

The current stream reasoning solution is designed with a single agent in mind. By expanding reasoning over and about streams to a multi-agent system setting, we can consider many interesting problems in addition to the ones described above. While there exists ongoing work into configuration of for example cloud computing systems, these approaches commonly have data centres in mind. Extending these techniques and others to heterogeneous autonomous robot applications would likely be interesting.

Finally, further investigation of the synergy effect resulting from reasoning about and over streams may be of interest to many problems not limited to situation awareness. Being able to reason about one's own percepts allows one to potentially resolve inconsistencies. By reasoning about streams, an agent is able to reason about perception itself and could thus find alternate modes of perception to either corroborate the contradiction or contradict the inconsistent observation. This dissertation presents but a few initial steps towards such an agent from the starting point of stream reasoning.

Bibliography

D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.

F.-M. Adolf, P. Faymonville, B. Finkbeiner, S. Schirmer, and C. Torens. Stream runtime monitoring on UAS. In *Proceedings of the 17th International Conference on Runtime Verification (RV)*, volume 10548, page 33, 2017.

M. Alirezaie. *Bridging the Semantic Gap between Sensor Data and Ontological Knowledge*. PhD thesis, Örebro university, 2015.

J. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM (JACM)*, 43(1):116–146, 1996.

D. Anicic. *Event Processing and Stream Reasoning with ETALIS*. PhD thesis, Karlsruhe Institute of Technology, 2012.

D. Anicic, P. Fodor, N. Stojanovic, and R. Stühmer. An approach for data-driven and logic-based complex event processing. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pages 26–27, 2009.

D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proceedings of the 20th International World Wide Web Conference (WWW)*, 2011.

D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic. Stream reasoning and complex event processing in ETALIS. *Semantic Web*, 3(4):397–407, 2012.

A. Arasu, S. Babu, and J. Widom. CQL: A language for continuous queries over streams and relations. In *Proceedings of the 9th International Workshop on Database Programming Languages (DBPL)*, pages 1–19. Springer, 2003.

A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. *STREAM: The Stanford data stream management system*, pages 317–336. Stanford InfoLab, 2004.

A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The International Journal on Very Large Data Bases*, 15(2):121–142, 2006.

S. Babu and J. Widom. StreaMon: an adaptive engine for stream query processing. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 931–932, 2004.

F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22(1-2):5–27, 1998.

K. Baldor and J. Niu. Monitoring dense-time, continuous-semantics, Metric Temporal Logic. In *Proceedings of the International Conference on Runtime Verification*, pages 245–259, 2012.

D. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-SPARQL: SPARQL for continuous querying. In *Proceedings of the 18th International World Wide Web Conference (WWW)*, 2009.

D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-SPARQL: a continuous query language for RDF data streams. *International Journal of Semantic Computing*, 4(1):3–25, 2010a.

D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus. An execution environment for C-SPARQL queries. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT)*, pages 441–452, 2010b.

D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. Querying RDF streams with C-SPARQL. *ACM SIGMOD Record*, 39(1):20–26, 2010c.

D. Basin, B. N. Bhatt, and D. Traytel. Almost event-rate independent monitoring of Metric Temporal Logic. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 94–112, 2017.

H. R. Bazoobandi, H. Beck, and J. Urbani. Expressive stream reasoning with Laser. In *Proceedings of the 16th International Semantic Web Conference (ISWC)*, pages 87–103, 2017.

H. Beck, M. Dao-Tran, T. Eiter, and M. Fink. Towards a logic-based framework for analyzing stream reasoning. In *Proceedings of the 3rd International Workshop on Ordering and Reasoning (OrdRing)*, 2014.

H. Beck, M. Dao-Tran, T. Eiter, and M. Fink. LARS: A logic-based framework for analyzing reasoning over streams. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI)*, 2015.

H. Beck, T. Eiter, and C. Folie. Ticker: A system for incremental ASP-based stream reasoning. *Theory and Practice of Logic Programming*, 17(5-6):744–763, 2017.

B. Bennett, A. Cohn, F. Wolter, and M. Zakharyaschev. Multi-dimensional modal logic as a framework for spatio-temporal reasoning. *Applied Intelligence*, 17(3): 239–251, 2002.

T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. SECRET: a model for analysis of the execution semantics of stream processing systems. *Proceedings of the VLDB Endowment*, 3(1-2):232–243, 2010.

C. Brenton, W. Faber, and S. Batsakis. Answer set programming for qualitative spatio-temporal reasoning: Methods and experiments. In *Technical Communications of the 32nd International Conference on Logic Programming*, volume 52, pages 4:1–4:15, 2016.

A. Bröring, K. Janowicz, C. Stasch, and W. Kuhn. Semantic challenges for sensor plug and play. In *Proceedings of the 9th International Symposium on Web and Wireless Geographical Information Systems (W2GIS)*, pages 72–86, 2009.

A. Bröring, P. Maué, K. Janowicz, D. Nüst, and C. Malewski. Semantically-enabled sensor plug & play for the sensor web. *Sensors*, 11(8):7568–7605, 2011.

J. R. Büchi. On a decision method in restricted second order arithmetic. In *The Collected Works of J. Richard Büchi*, pages 425–435. 1990.

J.-P. Calbimonte, O. Corcho, and A. J. Gray. Enabling ontology-based access to streaming data sources. In *Proceedings of the 9th International Semantic Web Conference (ISWC)*, pages 96–111, 2010.

P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. volume 36, 2015.

S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, volume 2, page 4, 2003.

M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, volume 3, pages 257–268, 2003.

C. Cini and A. Francalanza. An LTL proof system for runtime verification. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 581–595, 2015.

E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

A. Cohn and J. Renz. Qualitative spatial representation and reasoning. In *Handbook of Knowledge Representation*, pages 869–886. Elsevier, 2008.

M. Compton et al. The SSN ontology of the W3C semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17: 25–32, 2012.

G. Cugola and A. Margara. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems (DEBS)*, pages 50–61, 2010.

G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012a.

G. Cugola and A. Margara. Complex event processing with T-REX. *Journal of Systems and Software*, 85(8):1709–1728, 2012b.

Z. Cui, A. G. Cohn, and D. A. Randell. Qualitative and topological relationships in spatial databases. In *Proceedings of the Third International Symposium on Advances in Spatial Databases (SSD)*, pages 296–315, 1993.

E. Della Valle, S. Ceri, F. Van Harmelen, and D. Fensel. It's a streaming world! Reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6), 2009.

D. Dell'Aglio, E. Della Valle, J.-P. Calbimonte, and O. Corcho. RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems. *International Journal on Semantic Web and Information Systems*, 10(4):17–44, 2014.

D. Dell'Aglio, E. Della Valle, F. van Harmelen, and A. Bernstein. Stream reasoning: A survey and outlook. *Data Science*, 1(1-2):59–83, 2017a.

D. Dell'Aglio, D. Le Phuoc, A. Lê Tuán, M. I. Ali, and J.-P. Calbimonte. On a web of data streams. In *Proceedings of the ISWC2017 Workshop on Decentralizing the Semantic Web*, 2017b.

D. Dell'Aglio, T. Eiter, F. Heintz, and D. Le Phuoc. Special issue on stream reasoning. *Semantic Web*, 10(3):453–455, 2019. Editorial.

A. Desai, T. Dreossi, and S. A. Seshia. Combining model checking and runtime verification for safe robotics. In *Proceedings of the 17th International Conference on Runtime Verification (RV)*, pages 172–189, 2017.

N. Dindar, N. Tatbul, R. J. Miller, L. M. Haas, and I. Botan. Modeling the execution semantics of stream processing engines with SECRET. *The VLDB Journal*, 22(4): 421–446, 2013.

P. Doherty, G. Granlund, K. Kuchcinski, E. Sandewall, K. Nordberg, E. Skarman, and J. Wiklund. The WITAS unmanned aerial vehicle project. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI)*, pages 747–755, 2000.

P. Doherty, J. Kvarnström, and F. Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Journal of Autonomous Agents and Multi-Agent Systems*, 19(3):332–377, 2009.

P. Doherty, F. Heintz, and J. Kvarnström. Robotics, temporal logic and stream reasoning. In *Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 26, pages 42–51, 2013.

P. Doherty, J. Kvarnström, M. Wzorek, P. Rudol, F. Heintz, and G. Conte. *HDRC3: A Distributed Hybrid Deliberative/Reactive Architecture for Unmanned Aircraft Systems*, pages 849–952. 2014.

C. Dousson and P. Le Maigat. Chronicle recognition improvement using temporal focusing and hierarchization. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI)*, volume 7, pages 324–329, 2007.

Z. Dragisic. Semantic matching for stream reasoning. Master's thesis, Linköping University, 2011.

S. Dustdar and W. Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, pages 411–420, 1999.

M. Eckert and F. Bry. Complex event processing (CEP). *Informatik-Spektrum*, 32(2): 163–167, 2009. Written in German.

J. Eker and J. Janneck. CAL language report. Technical report, 2003.

S. Feng, M. Lohrey, and K. Quaas. Path checking for MTL and TPTL over data words. In *International Conference on Developments in Language Theory*, pages 326–339, 2015.

S. Feng, M. Lohrey, and K. Quaas. Path Checking for MTL and TPTL over Data Words. *Logical Methods in Computer Science*, 13, 2017.

I. Galpin, C. Y. Brenninkmeijer, F. Jabeen, A. A. Fernandes, and N. W. Paton. Comprehensive optimization of declarative sensor network queries. In *Proceedings of the 21st International Conference on Scientific and Statistical Database Management*, pages 339–360, 2009.

Z. Gantner, M. Westphal, and S. Wölfl. GQR – a fast reasoner for binary qualitative constraint calculi. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI)*, pages 24–29, 2008.

A. Gerevini and B. Nebel. Qualitative spatio-temporal reasoning with RCC-8 and Allen's interval calculus: Computational complexity. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI 2002)*, volume 2, pages 312–316, 2002.

M. Ghallab. On chronicles: Representation, on-line recognition and learning. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 597–606, 1996.

C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO: A logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, 1990.

K. Havelund and G. Roşu. Monitoring programs using rewriting. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE)*, pages 135–143, 2001.

F. Heintz. *DyKnow : A Stream-Based Knowledge Processing Middleware Framework*. PhD thesis, Linköping University, 2009.

F. Heintz. Semantically grounded stream reasoning integrated with ROS. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5935–5942, 2013.

F. Heintz and P. Doherty. Chronicle recognition in the WITAS UAV project: A preliminary report. In *Proceedings of the Swedish AI Society Workshop*, pages 1–4, 2001.

F. Heintz and P. Doherty. A distributed architecture for autonomous unmanned aerial vehicle experimentation. In *Proceedings of the 7th International Symposium on Distributed Autonomous Robotic Systems (DARS)*, pages 1–10, 2004a.

F. Heintz and P. Doherty. DyKnow: An approach to middleware for knowledge processing. *Journal of Intelligent and Fuzzy Systems*, 15(1):3–13, 2004b.

F. Heintz and P. Doherty. DyKnow: A framework for processing dynamic knowledge and object structures in autonomous systems. In *Proceedings of the International Workshop on Monitoring, Security, and Rescue Techniques in Multi-Agent Systems (MSRAS)*, pages 479–492, 2004c.

F. Heintz and P. Doherty. DyKnow: A framework for processing dynamic knowledge and object structures in autonomous systems. In *Proceedings of the Second Joint SAIS/SSLS Workshop*, pages 1–8, 2004d.

F. Heintz and P. Doherty. Managing dynamic object structures using hypothesis generation and validation. In *Proceedings of the AAAI Workshop on Anchoring Symbols to Sensor Data*, pages 54–62, 2004e.

F. Heintz and P. Doherty. A knowledge processing middleware framework and its relation to the JDL data fusion model. In *Proceedings of the 8th International Conference on Information Fusion (FUSION)*, pages 1–8, 2005a.

F. Heintz and P. Doherty. A knowledge processing middleware framework and its relation to the JDL data fusion model. In *Proceedings of the Third Joint SAIS/SSLS Workshop*, pages 1–10, 2005b.

F. Heintz and P. Doherty. A knowledge processing middleware framework and its relation to the JDL data fusion model. In *Proceedings of the Third Swedish Workshop on Autonomous Robotics (SWAR)*, pages 54–55, 2005c.

F. Heintz and P. Doherty. A knowledge processing middleware framework and its relation to the JDL data fusion model. *Journal of Intelligent and Fuzzy Systems*, 17 (4):335–351, 2006.

F. Heintz and P. Doherty. DyKnow federations: Distributing and merging information among UAVs. In *Proceedings of the 11th International Conference on Information Fusion (FUSION)*, pages 1–7, 2008.

F. Heintz and P. Doherty. Federated DyKnow, a distributed information fusion system for collaborative UAVs. In *Proceedings of the 11th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 1063–1069, 2010.

F. Heintz and Z. Dragisic. Semantic information integration for stream reasoning. In *Proceedings of the 15th International Conference on Information Fusion (FUSION)*, 2012.

F. Heintz and D. de Leng. Semantic information integration with transformations for stream reasoning. In *Proceedings of the 16th International Conference on Information Fusion (FUSION)*, pages 445–452, 2013.

F. Heintz and D. de Leng. Spatio-temporal stream reasoning with incomplete spatial information. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI)*, pages 429–434, 2014.

F. Heintz, P. Rudol, and P. Doherty. From images to traffic behavior - a UAV tracking and monitoring application. In *Proceedings of the 10th International Conference on Information Fusion (FUSION)*, pages 1–8, 2007a.

F. Heintz, P. Rudol, and P. Doherty. Bridging the sense-reasoning gap using the knowledge processing middleware DyKnow. In *Proceedings of the 30th Annual German Conference on Artificial Intelligence (KI)*, pages 460–463, 2007b.

F. Heintz, M. Krysander, J. Roll, and E. Frisk. FlexDx: A reconfigurable diagnosis framework. In *Proceedings of the 19th International Workshop on Principles of Diagnosis (DX)*, pages 1–8, 2008a.

F. Heintz, J. Kvarnström, and P. Doherty. Knowledge processing middleware. In *Proceedings of the First International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 147–158, 2008b.

F. Heintz, J. Kvarnström, and P. Doherty. A stream-based hierarchical anchoring framework. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5254–5260, 2009.

F. Heintz, J. Kvarnström, and P. Doherty. Bridging the sense-reasoning gap: DyKnow - stream-based middleware for knowledge processing. *Journal of Advanced Engineering Informatics*, 24(1):14–26, 2010.

F. Heintz, J. Kvarnström, and P. Doherty. Stream-based hierarchical anchoring. *Künstliche Intelligenz*, 27(2):119–128, 2013.

M. Hirzel, G. Baudart, A. Bonifati, E. Della Valle, S. Sakr, and A. Akrivi Vlachou. Stream processing languages in the big data era. *ACM SIGMOD Record*, 47(2):29–40, 2018.

H.-M. Ho, J. Ouaknine, and J. Worrell. Online monitoring of Metric Temporal Logic. In *Proceedings of the 5th International Conference on Runtime Verification*, pages 178–192, 2014.

A. Hongslo. Stream processing in the Robot Operating System framework. Master's thesis, Linköping University, 2012.

V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proceedings of the 5th International Conference*

on Emerging Networking Experiments and Technologies (CoNEXT), pages 1–12, 2009.

F. Kerasiotis, C. Koulamas, C. Antonopoulos, and G. Papadopoulos. Middleware approaches for wireless sensor networks based on current trends. In *Proceedings of the 4th Mediterranean Conference on Embedded Computing (MECO)*, pages 244–249, 2015.

R. Kontchakov, A. Kurucz, F. Wolter, and M. Zakharyaschev. Spatial logic + temporal logic = ? In *Handbook of Spatial Logics*, pages 497–564. 2007.

P. Koopmann. Ontology-based query answering for probabilistic temporal data. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*, 2019.

A. Kovtunova and R. Peñaloza. Cutting diamonds: A temporal logic with probabilistic distributions. In *Sixteenth International Conference on Principles of Knowledge Representation and Reasoning*, pages 561–570, 2018.

R. Koymans. Specifying real-time properties with Metric Temporal Logic. *Real-Time Systems*, 2(4):255–299, 1990.

M. Krysander, F. Heintz, J. Roll, and E. Frisk. Dynamic test selection for reconfigurable diagnosis. In *Proceedings of the 47th IEEE Conference on Decision and Control (CDC)*, pages 1066–1072, 2008.

M. Krysander, F. Heintz, J. Roll, and E. Frisk. FlexDx: A reconfigurable diagnosis framework. *Journal of Engineering Applications of Artificial Intelligence*, 23(8): 1303–1313, 2010.

O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Journal of Formal Methods in System Design*, 19(3):291–314, 2001.

J. Kvarnström, F. Heintz, and P. Doherty. A temporal logic-based planning and execution monitoring system. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 1–8, 2008.

D. Laney. 3d data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6:70, 2001.

E. Latronico, E. A. Lee, M. Lohstroh, C. Shaver, A. Wasicek, and M. Weber. A vision of swarmlets. *IEEE Internet Computing*, 19(2):20–28, 2015.

D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *Proceedings of the 10th International Conference on The Semantic Web (ISWC)*, pages 370–388, 2011.

D. de Leng. Extending semantic matching in DyKnow to handle indirectly-available streams. Master's thesis, Utrecht University, 2013.

D. de Leng. *Spatio-Temporal Stream Reasoning with Adaptive State Stream Generation*, volume 1783. Linköping University Electronic Press, 2017.

D. de Leng and F. Heintz. Towards on-demand semantic event processing for stream reasoning. In *Proceedings of the 17th International Conference on Information Fusion (FUSION)*, pages 1–8, 2014.

D. de Leng and F. Heintz. Ontology-based introspection in support of stream reasoning. In *Proceedings of the 1st Joint Ontology Workshops (JOWO) co-located with the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 2015a.

D. de Leng and F. Heintz. Ontology-based introspection in support of stream reasoning. In *Proceedings of the 13th Scandinavian Conference on Artificial Intelligence (SCAI)*, pages 78–87, 2015b.

D. de Leng and F. Heintz. Qualitative spatio-temporal stream reasoning with unobservable intertemporal spatial relations using landmarks. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI)*, pages 957–963, 2016a.

D. de Leng and F. Heintz. DyKnow: A dynamically reconfigurable stream reasoning framework as an extension to the Robot Operating System. In *Proceedings of the 5th IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 957–963, 2016b.

D. de Leng and F. Heintz. Towards adaptive semantic subscriptions for stream reasoning in the Robot Operating System. In *Proceedings of the 30th IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5445–5452, 2017.

D. de Leng and F. Heintz. Partial-state progression for stream reasoning with Metric Temporal Logic. In *Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning*, pages 633–634, 2018.

D. de Leng and F. Heintz. Approximate stream reasoning with Metric Temporal Logic under uncertainty. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*, 2019.

M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher. Formal specification and verification of autonomous robotic systems: A survey. *ACM Computing Surveys (CSUR)*, 52(5):1–41, 2019.

R. Lundh. *Robots that Help Each Other: Self-Configuration of Distributed Robot Systems*. PhD thesis, Örebro University, 2009.

R. Lundh, L. Karlsson, and A. Saffiotti. Autonomous functional configuration of a network robot system. *Robotics and Autonomous Systems*, 56(10):819–830, 2008.

A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1): 99–118, 1977.

S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, volume 2, pages 555–566, 2002.

N. Markey and P. Schnoebelen. Model checking a path. In *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR)*, pages 251–265, 2003.

D. Martin, M. Burstein, D. McDermott, S. McIlraith, M. Paolucci, K. Sycara, D. L. McGuinness, E. Sirin, and N. Srinivasan. Bringing semantics to web services with OWL-S. *World Wide Web*, 10(3):243–277, 2007.

D. Martin et al. OWL-S: Semantic markup for web services. *W3C member submission*, 2004.

A. Mauri, J.-P. Calbimonte, D. Dell'Aglio, M. Balduini, M. Brambilla, E. Della Valle, and K. Aberer. Triplewave: Spreading RDF streams on the web. In *Proceedings of the 15th International Semantic Web Conference (ISWC)*, pages 140–149, 2016.

D. L. McGuinness, F. Van Harmelen, et al. OWL web ontology language overview. *W3C recommendation*, 2004.

R. Medhat, B. Bonakdarpour, S. Fischmeister, and Y. Joshi. Accelerated runtime verification of LTL specifications with counting semantics. In *Proceedings of the 16th International Conference on Runtime Verification (RV)*, pages 251–267, 2016.

R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

D. E. Muller. Infinite sequences and finite machines. In *Proceedings of the Fourth Annual Symposium on Switching Circuit Theory and Logical Design*, pages 3–16, 1963.

E. Pejman, Y. Rastegari, P. M. Esfahani, and A. Salajegheh. Web service composition methods: A survey. In *Proceedings of the International MultiConference of Engineers and Computer Scientists (IMECS)*, volume 1, pages 560–564, 2012.

A. Pnueli. The temporal logic of programs. In *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science (SFCS)*, pages 46–57, 1977.

J. M. T. Portocarrero, F. C. Delicato, P. F. Pires, T. C. Rodrigues, and T. V. Batista. SAMSON: Self-adaptive middleware for wireless sensor networks. In *Proceedings of the 31st ACM/SIGAPP Symposium on Applied Computing (SAC)*, pages 1315–1322, 2016.

M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source robot operating system. In *Proceedings of the 2009 IEEE International Conference on Robotics and Automation (ICRA)*, 2009.

D. Randell, Z. Cui, and A. Cohn. A spatial logic based on regions and connection. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 165–176, 1992.

J. Rao and X. Su. A survey of automated web service composition methods. In *Proceedings of the International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, volume 3387, pages 43–54, 2005.

F. Reiss and J. M. Hellerstein. Data triage: An adaptive architecture for load shedding in TelegraphCQ. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 155–156, 2005.

J. Renz and B. Nebel. Efficient methods for qualitative spatial reasoning. *Journal of Artificial Intelligence Research*, 15:289–318, 2001.

A. Saffiotti, M. Broxvall, M. Gritti, K. LeBlanc, R. Lundh, J. Rashid, B. Seo, and Y.-J. Cho. The PEIS-ecology project: vision and results. In *Proceedings of the IEEE/RSJ 2008 International Conference on Intelligent Robots and Systems (IROS)*, 2008.

T. Sato. A statistical learning method for logic programs with distribution semantics. In *Proceedings of the 12th International Conference on Logic Programming (ICLP*, 1995.

T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.

Y. Shen, J. Li, Z. Wang, T. Su, B. Fang, G. Pu, W. Liu, and M. Chen. Runtime verification by convergent formula progression. In *Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC)*, pages 255–262, 2014.

A. N. Steinberg and C. L. Bowman. Revisions to the JDL data fusion model. In *Handbook of multisensor data fusion*, pages 65–88. CRC press, 2008.

F. Tang and L. Parker. ASyMTRe: Automated synthesis of multi-robot task solutions through software reconfiguration. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 1501–1508, 2005.

P. Thati and G. Roşu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005.

M. Tiger and F. Heintz. Stream reasoning using temporal logic and predictive probabilistic state models. In *Proceedings of the 23rd International Symposium on Temporal Representation and Reasoning (TIME)*, pages 196–205, 2016.

R. Tommasini, Y. A. Sedira, D. Dell'Aglio, M. Balduini, M. I. Ali, D. Le Phuoc, E. Della Valle, and J.-P. Calbimonte. VoCaLS: Vocabulary and catalog of linked streams. In *Proceedings of the 17th International Semantic Web Conference (ISWC)*, pages 256–272, 2018.

R. Tommasini, D. Calvaresi, and J.-P. Calbimonte. Stream reasoning agents: Blue sky ideas track. In *Proceedings of the 18th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1664–1680, 2019.

M. Y. Vardi. Automata-theoretic model checking revisited. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 137–150, 2007.

M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and computation*, 115(1):1–37, 1994.

P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, pages 185–194, 1983.

F. Wolter and M. Zakharyaschev. Spatio-temporal representation and reasoning based on RCC-8. In *Proceedings of the Seventh Conference on Principles of Knowledge Representation and Reasoning (KR 2000)*, pages 3–14, 2000.

P. Ziafati. *Information Engineering in Autonomous Robot Software*. PhD thesis, Utrecht University, 2015.

P. Ziafati, M. Dastani, J.-J. Meyer, and L. van der Torre. Event-processing in autonomous robot programming. In *Proceedings of the 12th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 95–102, 2013.

P. Ziafati, Y. Elrakaiby, M. van Zee, M. Dastani, J.-J. Meyer, L. van der Torre, and H. Voos. Reasoning on robot knowledge from discrete and asynchronous observations. In *Proceedings of the 2014 AAAI Spring Symposium Series*, 2014.

P. Ziafati, M. Dastani, J.-J. Meyer, L. van der Torre, and H. Voos. Retalis language for information engineering in autonomous robot software. *IfCoLog Journal of Logics and their Applications*, 2(2):65–126, 2015.

Appendix

A DyKnow ontology in Manchester syntax

THE following is a listing of the DyKnow ontology used for semantic interoperability. It makes use of Manchester syntax to improve human readability. The full up-to-date ontology in OWL/RDF syntax utilises the namespace `http://www.dyknow.eu/ontology/`.

```
Prefix: : <http://www.dyknow.eu/ontology/dyknow#>
Prefix: dc: <http://purl.org/dc/elements/1.1/>
Prefix: owl: <http://www.w3.org/2002/07/owl#>
Prefix: rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
Prefix: rdfs: <http://www.w3.org/2000/01/rdf-schema#>
Prefix: skos: <http://www.w3.org/2004/02/skos/core#>
Prefix: terms: <http://purl.org/dc/terms/>
Prefix: xml: <http://www.w3.org/XML/1998/namespace>
Prefix: xsd: <http://www.w3.org/2001/XMLSchema#>



Ontology: <http://www.dyknow.eu/ontology/dyknow>
<http://www.dyknow.eu/ontology/dyknow/201707>

Annotations:
    terms:creator "Daniel de Leng"^^xsd:string,
    terms:modified "2017-07-27",
    rdfs:comment "The DyKnow ontology can be used as a common representation
          of stream reasoning framework configurations."@en,
    rdfs:label "DyKnow Ontology"@en

AnnotationProperty: rdfs:comment


AnnotationProperty: rdfs:label


AnnotationProperty: terms:creator


AnnotationProperty: terms:modified
```

```
32
33
34  Datatype: rdf:PlainLiteral
35
36
37  Datatype: xsd:Name
38
39
40  Datatype: xsd:anyURI
41
42
43  Datatype: xsd:date
44
45
46  Datatype: xsd:dateTimeStamp
47
48
49  Datatype: xsd:string
50
51
52  ObjectProperty: dependsOn
53
54      SubPropertyChain:
55          hasSubscription o fromCU
56
57      Characteristics:
58          Transitive
59
60
61  ObjectProperty: fromCU
62
63      DisjointWith:
64          toCU
65
66      Characteristics:
67          Functional
68
69      Domain:
70          Subscription
71
72
73  ObjectProperty: fromPort
74
75      Characteristics:
76          Functional
77
78      Domain:
79          Subscription
80
81      Range:
82          OutPort
83
84
85  ObjectProperty: hasChannel
86
87      Characteristics:
88          Functional
89
90      Domain:
91          Subscription or Target
92
```

```
    Range:
        Channel


ObjectProperty: hasCostModel

    Characteristics:
        Functional


ObjectProperty: hasEnvironment

    Range:
        Environment


ObjectProperty: hasInPort

    Domain:
        Transformation

    Range:
        InPort

    InverseOf:
        isInPort


ObjectProperty: hasInstance

    Domain:
        Transformation

    InverseOf:
        instanceOf


ObjectProperty: hasOutPort

    Domain:
        Transformation

    Range:
        OutPort

    InverseOf:
        isOutPort


ObjectProperty: hasSample

    Characteristics:
        Functional

    Domain:
        SampleSequence

    Range:
        Sample


```

```
ObjectProperty: hasSampleSequence

    Characteristics:
        Functional

    Domain:
        Stream

    Range:
        SampleSequence


ObjectProperty: hasState

    Characteristics:
        Functional

    Domain:
        StateSequence

    Range:
        State


ObjectProperty: hasStateSequence

    Characteristics:
        Functional

    Range:
        StateSequence


ObjectProperty: hasSubscription

    Range:
        Subscription

    InverseOf:
        toCU


ObjectProperty: hasTag

    Range:
        Tag


ObjectProperty: hasTagDescription

    Characteristics:
        Functional

    Range:
        Tag


ObjectProperty: instanceOf

    Range:
        Transformation
```

```
215
216     InverseOf:
217         hasInstance
218
219
220 ObjectProperty: isInPort
221
222     Domain:
223         InPort
224
225     Range:
226         Transformation
227
228     InverseOf:
229         hasInPort
230
231
232 ObjectProperty: isOutPort
233
234     Domain:
235         OutPort
236
237     Range:
238         Transformation
239
240     InverseOf:
241         hasOutPort
242
243
244 ObjectProperty: nextSample
245
246     Characteristics:
247         Functional,
248         Irreflexive
249
250     Domain:
251         Sample
252
253     Range:
254         Sample
255
256
257 ObjectProperty: nextState
258
259     Characteristics:
260         Functional,
261         Irreflexive
262
263     Domain:
264         State
265
266     Range:
267         State
268
269
270 ObjectProperty: toCU
271
272     DisjointWith:
273         fromCU
274
275     Characteristics:
```

```
276         Functional
277
278     Domain:
279         Subscription
280
281     InverseOf:
282         hasSubscription
283
284
285 ObjectProperty: toPort
286
287     Characteristics:
288         Functional
289
290     Domain:
291         Subscription
292
293     Range:
294         InPort
295
296
297 DataProperty: hasChannelName
298
299     Characteristics:
300         Functional
301
302     Domain:
303         Channel
304
305     SubPropertyOf:
306         hasName
307
308
309 DataProperty: hasLabel
310
311     Characteristics:
312         Functional
313
314     Range:
315         xsd:Name
316
317
318 DataProperty: hasName
319
320     Characteristics:
321         Functional
322
323
324 DataProperty: hasPortName
325
326     SubPropertyOf:
327         hasName
328
329
330 DataProperty: hasTimeStamp
331
332     Characteristics:
333         Functional
334
335     Range:
336         xsd:dateTimeStamp
```

```
337
338
339 DataProperty: hasValue
340
341     Characteristics:
342         Functional
343
344
345 Class: ChangeSet
346
347     Annotations:
348         rdfs:comment "A change set describes changes made to an environment.
                Formally the change set at least describes the additions and
              removals of computation units, transformations, and targets."
              @en,
349         rdfs:label "Change Set"@en
350
351
352 Class: Channel
353
354     Annotations:
355         rdfs:label "Channel"@en,
356         rdfs:comment "Channels are named transportation mechanisms for data.
              "@en
357
358     SubClassOf:
359         hasChannelName some xsd:string
360
361
362 Class: CostModel
363
364     Annotations:
365         rdfs:label "Cost Model"@en,
366         rdfs:comment "A model describing how to calculate the cost of an
              update."@en
367
368
369 Class: Environment
370
371     Annotations:
372         rdfs:label "Environment"@en,
373         rdfs:comment "An environment is composed of a set of computation
              units (sometimes called a computation graph), a set of
              transformations, a set of targets, and a similarity relation
              between tags. The environment can be changed by applying a
              change set to it. This application is called an update.
              Environments describe the state of a stream reasoning framework
              ."@en
374
375     SubClassOf:
376         hasName some xsd:Name
377
378
379 Class: InPort
380
381     Annotations:
382         rdfs:label "Input Port"@en,
383         rdfs:comment "A port for receiving streaming data over a channel."
              @en
384
385     SubClassOf:
```

```
        Port

    DisjointWith:
        OutPort


Class: LabourCostModel

    Annotations:
        rdfs:label "Labour Cost Model"@en,
        rdfs:comment "A cost model for calculating the labour cost."@en,
        rdfs:label "Labor Cost Model"@en,
        rdfs:comment "A cost model for calculating the labor cost."@en

    SubClassOf:
        CostModel


Class: OutPort

    Annotations:
        rdfs:label "Output Port"@en,
        rdfs:comment "A port for transmitting streaming data over a channel.
            "@en

    SubClassOf:
        Port

    DisjointWith:
        InPort


Class: Parameter

    Annotations:
        rdfs:label "Parameter"@en

    SubClassOf:
        hasLabel some xsd:Name,
        hasValue some xsd:anyURI


Class: Port

    Annotations:
        rdfs:comment "The connection between a channel and a computation
            unit is realised in terms of ports. Ports are named entities."
            @en,
        rdfs:label "Port"@en

    SubClassOf:
        hasPortName some xsd:Name


Class: Sample

    Annotations:
        rdfs:label "Sample"@en,
        rdfs:comment "An atomic, time-stamped data point."@en

    SubClassOf:
```

```
444         hasLabel some xsd:Name,
445         hasTimeStamp some xsd:dateTimeStamp,
446         hasValue some xsd:anyURI
447
448
449 Class: SampleSequence
450
451     Annotations:
452         rdfs:label "Sample Sequence"@en
453
454     EquivalentTo:
455         hasSample some Sample
456
457
458 Class: Sink
459
460     Annotations:
461         rdfs:comment "A transformation that does not produce any resulting
                stream is called a sink."@en,
462         rdfs:label "Sink"@en
463
464     SubClassOf:
465         Transformation,
466         hasOutPort exactly 0 OutPort
467
468
469 Class: Source
470
471     Annotations:
472         rdfs:comment "A transformation that does not take any incoming
                stream is called a source."@en,
473         rdfs:label "Source"@en
474
475     SubClassOf:
476         Transformation,
477         hasInPort exactly 0 InPort
478
479
480 Class: State
481
482     Annotations:
483         rdfs:comment "A state is a mapping from a variable to a value."@en,
484         rdfs:label "State"@en
485
486     SubClassOf:
487         hasLabel some xsd:Name,
488         hasValue some xsd:anyURI
489
490
491 Class: StateSequence
492
493     Annotations:
494         rdfs:label "State Sequence"@en
495
496     EquivalentTo:
497         hasState some State
498
499
500 Class: StateStream
501
502     Annotations:
```

```
503         rdfs:label "State Stream"@en,
504         rdfs:comment "A stream composed of states is called a state stream.
                State streams thus describe mappings from sets of variables to
                sets of values for specific time-points. State streams can be
                used to for example evaluate logical formulas."@en
505
506     SubClassOf:
507         Stream
508
509
510 Class: Stream
511
512     Annotations:
513         rdfs:comment "A sequence of samples representing a flow of data is
                called a stream."@en,
514         rdfs:label "Stream"@en
515
516     EquivalentTo:
517         hasSampleSequence some SampleSequence
518
519
520 Class: Subscription
521
522     Annotations:
523         rdfs:comment "A subscription is a connection from a transmitting
                port to a receiving port over a channel."@en,
524         rdfs:label "Subscription"@en
525
526     SubClassOf:
527         fromPort some Port,
528         hasChannel some Channel,
529         toPort some Port
530
531
532 Class: Tag
533
534     Annotations:
535         rdfs:label "Tag"@en,
536         rdfs:comment "A tag is a descriptor with which concepts can be
                annotated. A concrete application can extend the Tag concept to
                 describe an annotation language."@en
537
538     SubClassOf:
539         hasTagDescription some owl:Thing
540
541
542 Class: Target
543
544     Annotations:
545         rdfs:comment "Targets describe the semantics of a desired
                information stream by using tags. Every target specifies a
                channel over which this desired information should be sent.
                Targets can be used to obtain adaptive semantic subscriptions
                which can be maintained by a DyKnow stream reasoning manager."
                @en,
546         rdfs:label "Target"@en
547
548     SubClassOf:
549         hasChannel some Channel,
550         hasTag some Tag,
551         hasName some xsd:Name
```

```
552
553
554 Class: Transformation
555
556     Annotations:
557         rdfs:comment "Transformations describe stream-generating functions
                over streams that can be instantiated as computation unit. The
                act of instantiating a transformation results in cost being
                acrued. Transformations are identifiable by a unique name."@en,
558         rdfs:label "Transformation"@en
559
560     SubClassOf:
561         hasCostModel some LabourCostModel,
562         hasName some xsd:Name
563
564
565 Class: UpkeepCostModel
566
567     Annotations:
568         rdfs:comment "A cost model for calculating the upkeep cost."@en,
569         rdfs:label "Upkeep Cost Model"@en
570
571     SubClassOf:
572         CostModel
573
574
575 Class: owl:Thing
```

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology
Linköping Studies in Arts and Science

Linköping Studies in Statistics
Linköping Studies in Information Science

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall**: Semantisk analys av process-beskrivningar i naturligt språk, 1977, ISBN 91- 7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System, 1978, ISBN 91- 7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91- 7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372- 805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies, 1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani:** Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms, 1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund:** Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld:** Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén:** Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-X.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis, 1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi:** Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN 91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91-7373-207-9.

No 725 **Tim Heyer:** Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91-7373-208-7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91-7373-212-5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91-7373-258-3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-supported Interorganisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X.

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationssystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing – An Empirical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5.

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004, ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linguistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modelling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.

No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.

No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005, ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

No 1005 **Aleksandra Tešanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.

No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.

No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.

No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.

No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.

No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8

No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.

No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.

No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.

No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.

No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.

No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.

No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.

No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.

No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.

No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.

No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.

No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.

No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 978-91-85715-65-7.

No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for Satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.

No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.

No 1086 **Ulf Johansson:** Obtaining Accurate and Comprehensible Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.

No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.

No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.

No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.

No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.

No 1112 **Jessica Lindblom:** Minding the body - Interacting socially through embodied action, 2007, ISBN 978-91-85831-48-7.

No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, 2007, ISBN 978-91-85831-47-0.

No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, 2007, ISBN 978-91-85831-33-3.

No 1127 **Alexandru Andrei:** Energy Efficient and Predictable Design of Real-time Embedded Systems, 2007, ISBN 978-91-85831-06-7.

No 1139 **Per Wikberg:** Eliciting Knowledge from Experts in Modeling of Complex Systems: Managing Variation and Interactions, 2007, ISBN 978-91-85895-66-3.

No 1143 **Mehdi Amirijoo:** QoS Control of Real-Time Data Services under Uncertain Workload, 2007, ISBN 978-91-85895-49-6.

No 1150 **Sanny Syberfeldt:** Optimistic Replication with Forward Conflict Resolution in Distributed Real-Time Databases, 2007, ISBN 978-91-85895-27-4.

No 1155 **Beatrice Alenljung:** Envisioning a Future Decision Support System for Requirements Engineering - A Holistic and Human-centred Perspective, 2008, ISBN 978-91-85895-11-3.

No 1156 **Artur Wilk:** Types for XML with Application to Xcerpt, 2008, ISBN 978-91-85895-08-3.

No 1183 **Adrian Pop:** Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages, 2008, ISBN 978-91-7393-895-2.

No 1185 **Jörgen Skågeby:** Gifting Technologies - Ethnographic Studies of End-users and Social Media Sharing, 2008, ISBN 978-91-7393-892-1.

No 1187 **Imad-Eldin Ali Abugessaisa:** Analytical tools and information-sharing methods supporting road safety organizations, 2008, ISBN 978-91-7393-887-7.

No 1204 **H. Joe Steinhauer:** A Representation Scheme for Description and Reconstruction of Object Configurations Based on Qualitative Relations, 2008, ISBN 978-91-7393-823-5.

No 1222 **Anders Larsson:** Test Optimization for Core-based System-on-Chip, 2008, ISBN 978-91-7393-768-9.

No 1238 **Andreas Borg:** Processes and Models for Capacity Requirements in Telecommunication Systems, 2009, ISBN 978-91-7393-700-9.

No 1240 **Fredrik Heintz:** DyKnow: A Stream-Based Knowledge Processing Middleware Framework, 2009, ISBN 978-91-7393-696-5.

No 1241 **Birgitta Lindström:** Testability of Dynamic Real-Time Systems, 2009, ISBN 978-91-7393-695-8.

No 1244 **Eva Blomqvist:** Semi-automatic Ontology Construction based on Patterns, 2009, ISBN 978-91-7393-683-5.

No 1249 **Rogier Woltjer:** Functional Modeling of Constraint Management in Aviation Safety and Command and Control, 2009, ISBN 978-91-7393-659-0.

No 1260 **Gianpaolo Conte:** Vision-Based Localization and Guidance for Unmanned Aerial Vehicles, 2009, ISBN 978-91-7393-603-3.

No 1262 **AnnMarie Ericsson:** Enabling Tool Support for Formal Analysis of ECA Rules, 2009, ISBN 978-91-7393-598-2.

No 1266 **Jiri Trnka:** Exploring Tactical Command and Control: A Role-Playing Simulation Approach, 2009, ISBN 978-91-7393-571-5.

No 1268 **Bahlol Rahimi:** Supporting Collaborative Work through ICT - How End-users Think of and Adopt Integrated Health Information Systems, 2009, ISBN 978-91-7393-550-0.

No 1274 **Fredrik Kuivinen:** Algorithms and Hardness Results for Some Valued CSPs, 2009, ISBN 978-91-7393-525-8.

No 1281 **Gunnar Mathiason:** Virtual Full Replication for Scalable Distributed Real-Time Databases, 2009, ISBN 978-91-7393-503-6.

No 1290 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems, 2009, ISBN 978-91-7393-482-4.

No 1294 **Johan Thapper:** Aspects of a Constraint Optimisation Problem, 2010, ISBN 978-91-7393-464-0.

No 1306 **Susanna Nilsson:** Augmentation in the Wild: User Centered Development and Evaluation of Augmented Reality Applications, 2010, ISBN 978-91-7393-416-9.

No 1313 **Christer Thörn:** On the Quality of Feature Models, 2010, ISBN 978-91-7393-394-0.

No 1321 **Zhiyuan He:** Temperature Aware and Defect-Probability Driven Test Scheduling for System-on-Chip, 2010, ISBN 978-91-7393-378-0.

No 1333 **David Broman:** Meta-Languages and Semantics for Equation-Based Modeling and Simulation, 2010, ISBN 978-91-7393-335-3.

No 1337 **Alexander Siemers:** Contributions to Modelling and Visualisation of Multibody Systems Simulations with Detailed Contact Analysis, 2010, ISBN 978-91-7393-317-9.

No 1354 **Mikael Asplund:** Disconnected Discoveries: Availability Studies in Partitioned Networks, 2010, ISBN 978-91-7393-278-3.

No 1359 **Jana Rambusch**: Mind Games Extended: Understanding Gameplay as Situated Activity, 2010, ISBN 978-91-7393-252-3.

No 1373 **Sonia Sangari**: Head Movement Correlates to Focus Assignment in Swedish, 2011, ISBN 978-91-7393-154-0.

No 1374 **Jan-Erik Källhammer**: Using False Alarms when Developing Automotive Active Safety Systems, 2011, ISBN 978-91-7393-153-3.

No 1375 **Mattias Eriksson**: Integrated Code Generation, 2011, ISBN 978-91-7393-147-2.

No 1381 **Ola Leifler**: Affordances and Constraints of Intelligent Decision Support for Military Command and Control – Three Case Studies of Support Systems, 2011, ISBN 978-91-7393-133-5.

No 1386 **Soheil Samii**: Quality-Driven Synthesis and Optimization of Embedded Control Systems, 2011, ISBN 978-91-7393-102-1.

No 1419 **Erik Kuiper**: Geographic Routing in Intermittently-connected Mobile Ad Hoc Networks: Algorithms and Performance Models, 2012, ISBN 978-91-7519-981-8.

No 1451 **Sara Stymne**: Text Harmonization Strategies for Phrase-Based Statistical Machine Translation, 2012, ISBN 978-91-7519-887-3.

No 1455 **Alberto Montebelli**: Modeling the Role of Energy Management in Embodied Cognition, 2012, ISBN 978-91-7519-882-8.

No 1465 **Mohammad Saifullah**: Biologically-Based Interactive Neural Network Models for Visual Attention and Object Recognition, 2012, ISBN 978-91-7519-838-5.

No 1490 **Tomas Bengtsson**: Testing and Logic Optimization Techniques for Systems on Chip, 2012, ISBN 978-91-7519-742-5.

No 1481 **David Byers**: Improving Software Security by Preventing Known Vulnerabilities, 2012, ISBN 978-91-7519-784-5.

No 1496 **Tommy Färnqvist**: Exploiting Structure in CSP-related Problems, 2013, ISBN 978-91-7519-711-1.

No 1503 **John Wilander**: Contributions to Specification, Implementation, and Execution of Secure Software, 2013, ISBN 978-91-7519-681-7.

No 1506 **Magnus Ingmarsson**: Creating and Enabling the Useful Service Discovery Experience, 2013, ISBN 978-91-7519-662-6.

No 1547 **Wladimir Schamai**: Model-Based Verification of Dynamic System Behavior against Requirements: Method, Language, and Tool, 2013, ISBN 978-91-7519-505-6.

No 1551 **Henrik Svensson**: Simulations, 2013, ISBN 978-91-7519-491-2.

No 1559 **Sergiu Rafiliu**: Stability of Adaptive Distributed Real-Time Systems with Dynamic Resource Management, 2013, ISBN 978-91-7519-471-4.

No 1581 **Usman Dastgeer**: Performance-aware Component Composition for GPU-based Systems, 2014, ISBN 978-91-7519-383-0.

No 1602 **Cai Li**: Reinforcement Learning of Locomotion based on Central Pattern Generators, 2014, ISBN 978-91-7519-313-7.

No 1652 **Roland Samlaus**: An Integrated Development Environment with Enhanced Domain-Specific Interactive Model Validation, 2015, ISBN 978-91-7519-090-7.

No 1663 **Hannes Uppman**: On Some Combinatorial Optimization Problems: Algorithms and Complexity, 2015, ISBN 978-91-7519-072-3.

No 1664 **Martin Sjölund**: Tools and Methods for Analysis, Debugging, and Performance Improvement of Equation-Based Models, 2015, ISBN 978-91-7519-071-6.

No 1666 **Kristian Stavåker**: Contributions to Simulation of Modelica Models on Data-Parallel Multi-Core Architectures, 2015, ISBN 978-91-7519-068-6.

No 1680 **Adrian Lifa**: Hardware/Software Codesign of Embedded Systems with Reconfigurable and Heterogeneous Platforms, 2015, ISBN 978-91-7519-040-2.

No 1685 **Bogdan Tanasa**: Timing Analysis of Distributed Embedded Systems with Stochastic Workload and Reliability Constraints, 2015, ISBN 978-91-7519-022-8.

No 1691 **Håkan Warnquist**: Troubleshooting Trucks – Automated Planning and Diagnosis, 2015, ISBN 978-91-7685-993-3.

No 1702 **Nima Aghaee**: Thermal Issues in Testing of Advanced Systems on Chip, 2015, ISBN 978-91-7685-949-0.

No 1715 **Maria Vasilevskaya**: Security in Embedded Systems: A Model-Based Approach with Risk Metrics, 2015, ISBN 978-91-7685-917-9.

No 1729 **Ke Jiang**: Security-Driven Design of Real-Time Embedded System, 2016, ISBN 978-91-7685-884-4.

No 1733 **Victor Lagerkvist**: Strong Partial Clones and the Complexity of Constraint Satisfaction Problems: Limitations and Applications, 2016, ISBN 978-91-7685-856-1.

No 1734 **Chandan Roy**: An Informed System Development Approach to Tropical Cyclone Track and Intensity Forecasting, 2016, ISBN 978-91-7685-854-7.

No 1746 **Amir Aminifar**: Analysis, Design, and Optimization of Embedded Control Systems, 2016, ISBN 978-91-7685-826-4.

No 1747 **Ekhiotz Vergara**: Energy Modelling and Fairness for Efficient Mobile Communication, 2016, ISBN 978-91-7685-822-6.

No 1748 **Dag Sonntag**: Chain Graphs – Interpretations, Expressiveness and Learning Algorithms, 2016, ISBN 978-91-7685-818-9.

No 1768 **Anna Vapen**: Web Authentication using Third-Parties in Untrusted Environments, 2016, ISBN 978-91-7685-753-3.

No 1778 **Magnus Jandinger**: On a Need to Know Basis: A Conceptual and Methodological Framework for Modelling and Analysis of Information Demand in an Enterprise Context, 2016, ISBN 978-91-7685-713-7.

No 1798 **Rahul Hiran**: Collaborative Network Security: Targeting Wide-area Routing and Edge-network Attacks, 2016, ISBN 978-91-7685-662-8.

No 1813 **Nicolas Melot**: Algorithms and Framework for Energy Efficient Parallel Stream Computing on Many-Core Architectures, 2016, ISBN 978-91-7685-623-9.

No 1823 **Amy Rankin**: Making Sense of Adaptations: Resilience in High-Risk Work, 2017, ISBN 978-91-7685-596-6.

No 1831 **Lisa Malmberg**: Building Design Capability in the Public Sector: Expanding the Horizons of Development, 2017, ISBN 978-91-7685-585-0.

No 1851 **Marcus Bendtsen**: Gated Bayesian Networks, 2017, ISBN 978-91-7685-525-6.

No 1852 **Zlatan Dragisic**: Completion of Ontologies and Ontology Networks, 2017, ISBN 978-91-7685-522-5.

No 1854 **Meysam Aghighi**: Computational Complexity of some Optimization Problems in Planning, 2017, ISBN 978-91-7685-519-5.

No 1863 **Simon Ståhlberg**: Methods for Detecting Unsolvable Planning Instances using Variable Projection, 2017, ISBN 978-91-7685-498-3.

No 1879 **Karl Hammar**: Content Ontology Design Patterns: Qualities, Methods, and Tools, 2017, ISBN 978-91-7685-454-9.

No 1887 **Ivan Ukhov**: System-Level Analysis and Design under Uncertainty, 2017, ISBN 978-91-7685-426-6.

No 1891 **Valentina Ivanova**: Fostering User Involvement in Ontology Alignment and Alignment Evaluation, 2017, ISBN 978-91-7685-403-7.

No 1902 **Vengatanathan Krishnamoorthi**: Efficient HTTP-based Adaptive Streaming of Linear and Interactive Videos, 2018, ISBN 978-91-7685-371-9.

No 1903 **Lu Li**: Programming Abstractions and Optimization Techniques for GPU-based Heterogeneous Systems, 2018, ISBN 978-91-7685-370-2.

No 1913 **Jonas Rybing**: Studying Simulations with Distributed Cognition, 2018, ISBN 978-91-7685-348-1.

No 1936 **Leif Jonsson**: Machine Learning-Based Bug Handling in Large-Scale Software Development, 2018, ISBN 978-91-7685-306-1.

No 1964 **Arian Maghazeh**: System-Level Design of GPU-Based Embedded Systems, 2018, ISBN 978-91-7685-175-3.

No 1967 **Mahder Gebremedhin**: Automatic and Explicit Parallelization Approaches for Equation Based Mathematical Modeling and Simulation, 2019, ISBN 978-91-7685-163-0.

No 1984 **Anders Andersson**: Distributed Moving Base Driving Simulators – Technology, Performance, and Requirements, 2019, ISBN 978-91-7685-090-9.

No 1993 **Ulf Kargén**: Scalable Dynamic Analysis of Binary Code, 2019, ISBN 978-91-7685-049-7.

No 2001 **Tim Overkamp**: How Service Ideas Are Implemented: Ways of Framing and Addressing Service Transformation, 2019, ISBN 978-91-7685-025-1.

No 2006 **Daniel de Leng**: Robust Stream Reasoning Under Uncertainty, 2019, ISBN 978-91-7685-013-8.

Linköping Studies in Arts and Science

No 504 **Ing-Marie Jonsson:** Social and Emotional Characteristics of Speech-based In-Vehicle Information Systems: Impact on Attitude and Driving Behaviour, 2009, ISBN 978-91-7393-478-7.

No 586 **Fabian Segelström:** Stakeholder Engagement for Service Design: How service designers identify and communicate insights, 2013, ISBN 978-91-7519-554-4.

No 618 **Johan Blomkvist:** Representing Future Situations of Service: Prototyping in Service Design, 2014, ISBN 978-91-7519-343-4.

No 620 **Marcus Mast:** Human-Robot Interaction for Semi-Autonomous Assistive Robots, 2014, ISBN 978-91-7519-319-9.

No 677 **Peter Berggren:** Assessing Shared Strategic Understanding, 2016, ISBN 978-91-7685-786-1.

No 695 **Mattias Forsblad:** Distributed cognition in home environments: The prospective memory and cognitive practices of older adults, 2016, ISBN 978-91-7685-686-4.

Linköping Studies in Statistics

No 9 **Davood Shahsavani:** Computer Experiments Designed to Explore and Approximate Complex Deterministic Models, 2008, ISBN 978-91-7393-976-8.

No 10 **Karl Wahlin:** Roadmap for Trend Detection and Assessment of Data Quality, 2008, ISBN 978-91-7393-792-4.

No 11 **Oleg Sysoev:** Monotonic regression for large multivariate datasets, 2010, ISBN 978-91-7393-412-1.

No 13 **Agné Burauskaite-Harju:** Characterizing Temporal Change and Inter-Site Correlations in Daily and Subdaily Precipitation Extremes, 2011, ISBN 978-91-7393-110-6.

No 14 **Måns Magnusson:** Scalable and Efficient Probabilistic Topic Model Inference for Textual Data, 2018, ISBN 978-91-7685-288-0.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering- att skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998. ISBN 9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998, ISBN 9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN 91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000, ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X.

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN 91-7373-736-4.

No 9 **Karin Hedström:** Spår av datoriseringens värden – Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson:** Method Configuration method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.

No 14 **Benneth Christiansson, Marie-Therese Christiansson:** Mötet mellan process och komponent - mot ett ramverk för en verksamhetsnära kravspecifikation vid anskaffning av komponentbaserade informationssystem, 2006, ISBN 91-85643-22-X.