FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Declarative Adaptive Interface Monitoring

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Stefan Brocanelli
Matrikelnummer 11728331

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Thomas Eiter
Mitwirkung: Patrik Schneider

Wien, 7. Mai 2020

Stefan Brocanelli	Thomas Eiter

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Declarative Adaptive Interface Monitoring

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Stefan Brocanelli
Registration Number 11728331

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Thomas Eiter
Assistance: Patrik Schneider

Vienna, 7th May, 2020

Stefan Brocanelli	Thomas Eiter

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Stefan Brocanelli

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 7. Mai 2020

Stefan Brocanelli

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Danksagung

Ich möchte mich herzlich bei meinem Supervisor Professor Thomas Eiter und meinem Tutor Patrik Schneider bedanken, die mir in den letzten Monaten immer erfreut dabei geholfen haben meine Abschlussarbeit optimal zu gestalten.

Ein großes Dankeschön ergeht auch meiner Familie, die mich immer bei all meinen Entscheidungen unterstützt hat.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

I would like to greatly thank my supervisor Professor Thomas Eiter and my Tutor Patrik Schneider who were of great help during this endeavor and showed no hesitation when I needed feedback or advice.

A big thank you also to my family that has always supported me in any and all decisions allowing me to choose my own path.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Mit Industry 4.0 werden enorme Datenströme generiert, welche den Weg ebnen für dynamische Optimierung. Wir bauen auf das Konzept von Dynacon, eine dynamisch konfigurierbare Architektur, auf und führen eine Schnittstellenbeschreibung ein, welche mit deklarativen Aktualisierungsbefehlen umgeändert werden kann. Dadurch, bringen wir die oft starren und schwer adaptierbaren Konfigurierungssysteme des jetzigen Stands der Technik auf ein neues Niveau, welches Änderungen in Echtzeit, basierend auf Informationen des Stream Reasoners, ermöglicht. Derzeitige Versuche, Stream Reasoner neu zu konfigurieren, haben oft keine Komponente, welche sich darum kümmert, einkommende Datenströme zu analysieren und dann anhand dieser zu entscheiden, ob eine Neukonfiguration überhaupt notwendig ist. Zusätzlich nutzen viele dynamische Systeme keine Schnittstelle um die einzelnen Komponenten zu konfigurieren und sind auch nicht kompatibel mit Regelbasierten Stream Reasoner. Daher führen wir zwei neue Komponenten ein: Der Communication Manger sorgt sich um Daten, die vom Stream Reasoner übergeben werden und passt die Kommunikation anhand der Schnittstelle an, bevor die Nachrichten zum Operator weitergesendet werden. Der Update Manager hingegen empfängt Befehle vom Operator und führte diese auf der Schnittstelle und dem Stream Reasoner aus. In der von uns entwickelten Timed-LUPS Sprache erstellen wir Regeln mit Bedingungen und Zeitdauer welche es ermöglichen anhand des Datenstroms Befehle zu aktivieren. Die Einführung von einer Zeitdauer sorgt dafür, dass der Update Manager jegliche Befehle nach einer gewissen Zeit wieder rückgängig machen kann. Zuletzt haben wir ein Java Prototyp gebaut, der Hand in Hand mit einem HexLite Stream Reasoner arbeitet um unsere Konzepte zu implementieren. Mit mehreren Experimenten beweisen wir dann die Realisierbarkeit einiger unserer Funktionalitäten, wie das wechseln von Kommunikationseinstellung ohne Neustart des Stream Reasoners und das Umändern von Regeln mit Neustarts in Bruchteilen einer Sekunde.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

With Industry 4.0 enormous streams of sensor data are generated and open the doors to dynamic optimization. We build on the concept of DynaCon, a dynamic configuration system, and introduce an interface description that can be altered with declarative update commands. By doing so, we elevated the often static and slow to adapt state of the art configuration systems to a new level that allows performing changes in real-time based on information extracted by a stream reasoner. State of the art efforts to reconfigure stream reasoners always lacked a layer of reasoning that would analyze incoming data to decide when a reconfiguration is necessary. Besides, many dynamic systems do not make use of an interface to configure its components and they are not compatible with rule-based reasoners that play a key role in unlocking reconfiguration. We introduce two new modules: the Communication Manager deals with incoming stream reasoner messages and relays them to the Operator based on settings stored in the interface and the Update Manager executes update commands sent by the Operator. Through Timed-LUPS we create a reasoning layer through policies that regulate conditions and duration for the update commands. By defining the duration of a command, the Update Manager can retract the changes and revert the interface or stream reasoner to its previous state after the duration has expired. Lastly, we built a Java prototype working hand in hand with a HexLite Stream Reasoner to implement our concepts. With several experiments, we then prove the feasibility of many of the outlined features like changing the communication while the stream reasoner is running and performing rule changes with a fraction of a second restarts.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

CHAPTER 1

Introduction

With the continuous advancement of modern industrial production and manufacturing techniques, approaches like Industry 4.0[1] gain more and more traction [Roj17]. Industry 4.0 describes the combined usage of Cyber-Physical Systems (CPS), Internet of Things (IoT) and Big Data. These type of systems allow for optimization and adaptive behavior due to the sheer amount of data that is gathered through sensors and monitors. This continuous flow of sensor data creates a stream that is fed into so called reasoners. The reasoners then attempt to evaluate the current state through temporal logic and belief revision.

However, nowadays still static configuration systems are widespread and the data streams are characterized by rapidly changing information. Static configurations work in a way where an initial configuration is established through calculations and then deployed. Such a configuration can for example be a set of rules that triggers certain events depending on sensor information (e.g., sensor measuring temperature below 4 °C triggers a snow warning). These configurations will remain fixed for the most part and will only be changed in case of architectural changes, often requiring lengthy maintenance phases or reboots. This type of static model does not allow for making dynamic changes aimed at improving performance or optimizing use case specific metrics. In addition, simple reasoners struggle to deal with rapidly changing data streams and thus create a bottleneck for the rest of the system that relies on real time information to make changes aimed at optimizing the system.

These issues can be solved through the combination of Stream Reasoners [CHVF09] and declarative adaptive interface monitoring, where the interface stands at the center of the architecture allowing to dynamically change the stream reasoner and the interface itself. Such changes are made possible through update policies and update commands that can be issued in a responsive manner by the configuration component acting upon the feed

[1] http://smart-industries.ch/industry-4-0/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

of data collected by sensors and processed through the stream reasoner. This process creates a configuration cycle as illustrated in Figure 1.1.

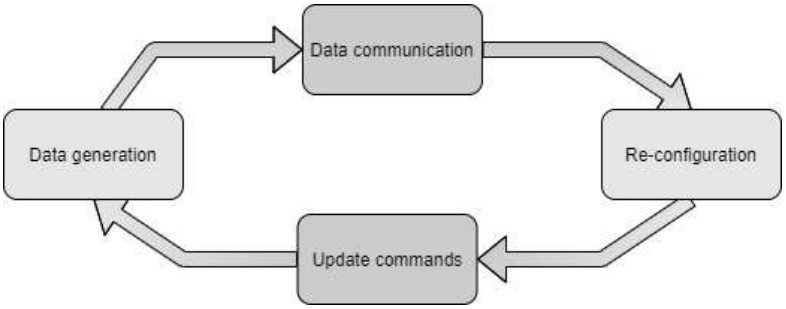


Figure 1.1: Configuration cycle

Example 1. Take traffic lights at an intersection as an example. These traffic lights will initially be configured with a fixed timing respecting all constraints given by crossing roads, pedestrians, bicycle lanes and trams. Once a schedule is determined, it will remain unchanged unless new elements are introduced to the intersection, such as additional lanes. However, this system does not take into account the amount of traffic at specific hours of the day, accidents or other unpredictable events that might profit from a different configuration. In the case of traffic congestion, the configuration could be optimized and made more efficient through the interface by giving longer green phases to certain car lanes.

We can thus summarize the problems with the current state of the art:

1. Lack of dynamic stream reasoners. Even though reconfigurable stream reasoners do exist and have been researched in other projects like the DyKnow framework [dH16], they merely issue reconfigurations but do not reason on the incoming data streams to determine whether a reconfiguration is necessary. Ideally, we would want a configuration system that can react to the data delivered by the stream reasoner, and then decide based on that data whether a reconfiguration is necessary and how to best optimize it.

2. The reconfiguration process itself presents many challenges. Many knowledge based reconfiguration approaches are not compatible with dynamic systems such as Cyber-Physical Systems (CPS), especially when faced with logic based stream reasoners [VCB+09]. Preferably, one could make use of commands that aim to perform specific changes in the configuration like adding or removing rules from the logic based rule system in the stream reasoner.

3. Lack of clear architectural structure. Even reconfiguration methods that do involve CPS often do not make use of a logic based framework and thus makes it hard to apply changes to the rule set. Additionally, the architectures are often not described

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

in detail, omitting crucial information about the connection points between stream reasoners and the other components making up the reconfiguration cycle [SWD+14]. Ideally, a dynamic architecture should be in place, defining exchangeable modules that allow the execution of crucial functions like communication and adaptation of configurations.

4. As a result of the previous problems, we also do not have well documented interface descriptions. While some of the previously quoted systems do use interfaces, they once again are badly documented and do not present an interface description detailing all the connections to the other parts of the architecture.

The intent behind this thesis is to research previous efforts done in the field of Cyber-Physical Systems, Update Languages and dynamic configurations and to combine them in a new system that can make use of all the advantages displayed by the single components. Most important is the addition of an interface that acts as a bridge between what we would conventionally call the configurator, responsible for elaborating new configurations for the stream reasoner and the data production part. This dynamically adaptable interface builds the core of the architecture and allows to eliminate many of the disadvantages of current state of the art technologies. We will thus extend the works done by Eiter et al. in the DynaCon [EFTW18] project by elaborating on many of its principles and creating a language to define the interface. Hand in hand comes the task of creating a concept for a declarative command language that will be the conduit to making dynamic changes to the interface and the rule set that defines what information is collected in the stream reasoner. To showcase the concept of declarative adaptive interface monitoring a small prototype including all the crucial parts of the architecture and most importantly the previously conceptualized command language will be created. This Java programmed prototype will operate on a simulated data output that is fed to our configuration system through a stream reasoner.

We thus address the problems and challenges given by the current state of the art and summarise the contributions of this thesis as follows:

1. We introduce the element of adaptability by creating a language for the interface description that shows how the interface will look like as a file and what components around it allow the system to be dynamically adaptable. Hand in hand comes a detailed outline of the architecture and the exchangeable modules employed by our systems, namely the Communication Manager and the Update Manager, respectively responsible for communicating the data from the stream reasoner to the configuration module and applying update commands.

2. We create *TLUPS - Timed LUPS* as an elaborate concept for a declarative command language based on *LUPS - A Language for Updating Logic Programs* [APPP02] with the addition of timed commands. The goal of the command language is to act as a tool to change the rule set of the stream reasoner and the configurations stored in the interface.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

3. Apply the developed concepts and languages by creating a prototype of the whole configuration system that makes use of an interface, a declarative command language and all components necessary to close the configuration cycle in Figure 1.1. With the prototype we will display some of the functionalities of our dynamic system on a selected traffic use case.

1.1 Stream Reasoners

Stream reasoners in their most basic form take a data stream together with a large background knowledge base as input and try to extract non-trivial knowledge to be further utilized in the system. Ideally, a stream reasoner should be able to deal with *big volumes* of data delivered on a *high frequency*. New grand scale projects often require the use of thousands of sensors that each supply information multiple times each second. For example a water diversion project in southern China[2] employs 100,000 sensors along the waterway in order to scan for structural weaknesses, water quality, flow rates and many more. In addition, they should be able to deal with *different data formats* and *incomplete* data caused by interruptions or *noise* over networks. At the same time, the stream reasoner should provide answers in a *timely fashion* in order to allow any type of listeners on the other side to act upon the feed of data quickly. These properties highlighted in cursive writing create the requirements for an ideal Stream Reasoner.

Using the summary given by [DVvHB17], encompassing the advances of stream reasoning, we take a look at the evolution of this research area between 2007 and 2017. In 2007 many efforts were made to find a solution satisfying all the requirements named above, starting with *data-stream management systems (DSMS)*, able to deal with rapidly changing data on the fly [CHVF09]. However, DSMS could not deal with differing data types and lacked the ability to perform complex reasoning tasks on big data instances. In the years following, other attempts were made through Knowledge Representation (KR) and Semantic Web (SW). These solutions could deal with different data types and allowed for fine-grained access to identify the exact source of a piece information, like the exact location of a flow sensor located on the previously mentioned example of the water diversion project. Even though KR and SW improved on data-stream management systems, they were still dealing with static data and were unable to give answers in a timely fashion.

DSMS, KR and SW created the ingredients for the *ideal* solution that would satisfy all the requirements given above: Stream Reasoning. The best results have been achieved by solutions extending SPARQL [PAG09] and extended SPARQL (eSPARQL), described as a query language for Resource Description Frameworks (RDF). We talk of ideal solutions because no implementation is yet able to satisfy all the requirements to a good level. The problems of volume, incompleteness and noise still require a lot of attention. [DVvHB17]

[2] https://internetofbusiness.com/100000-iot-sensors-line-chinas-ambitious-water-diversion-project/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

1.2 Industrial digitization

CPS have been combined with the Internet of Things (IoT), the Internet of Service (IoS) and smart factory to initiate Industry 4.0 [KWH13, Roj17], which in recent years and combined with smart manufacturing has been hyped up as the 4th industrial revolution following the computer and general automation [YZL+19].

1.2.1 IoT and IoS

The IoT is concerned with tangible objects like machines and sensors and aims at incorporating them to the internet. It is rendered possible by assigning to each unique object a virtual representation. IoT aims to construct an environment where all current smart embedded devices like smartphones, sensors, etc. can be supported by a connecting environment using radio-frequency identification (RFID), ipv6, barcodes, QR codes, NFC, GPS and more [AS13]. Even though IoT will not play a big role in this thesis, we can find it on the data generation side in the form of sensors that supply the data to the stream reasoner. Meanwhile, IoS focuses on the two concepts of Web 2.0 and Service-Oriented Architecture (SOA) [RG18]. In particular we are interested in the following two properties that partly characterize Web 2.0 and are applicable to our project:

- Interactivity: XML allows for dynamic manipulation of configuration data between the client (web browser) and the server. In this thesis, it will be used as the description format for the interface.
- Web services: Exposing services for other software to use and not only for human clients. In our project, we can find this behavior in the Update Manager. It exposes some functions (like adding/removing rules in the stream reasoner) to the Operator that can make use of these methods without implementing them himself.

1.3 Cyber-Physical Systems (CPS)

The term Cyber-Physical Systems surfaced and gained traction at the NSF Workshop on Cyber-Physical Systems in Austin, Texas in 2006. The core idea of a CPS is to combine the physical and software components in an even stronger way than with ordinary embedded systems. The goal is to deeply intertwine the components to allow them to affect each other with feedback loops and to change their behavior based on the given context [Lee08]. CPS are more reliable and efficient compared to conventional embedded systems and allow to model real world scenarios better due to the ability to handle situations with incomplete or unpredictable information [Lee08, Lee06]. The applications go from small implementations in medical monitoring (pace makers) to bigger scale projects like robotic systems, automated piloting and smart grid [KM14].

Even nowadays CPS are still a very relevant research field. In 2015 the *1st European Experts's Workshop on CPS* (CyPhERS) took place in Munich to identify short, medium

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

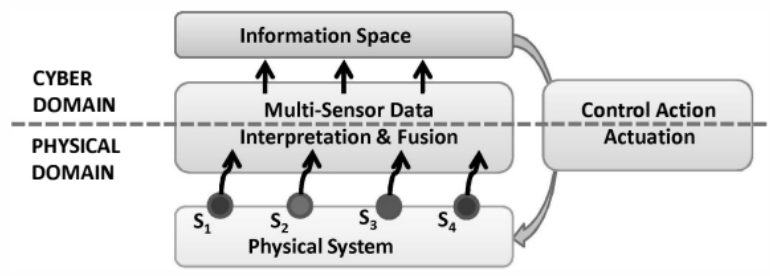


Figure 1.2: Technical model of a CPS [Ray13]

and long term evolution goals for CPS. This later resulted in the publishing of the *Cyber-Physical European Roadmap & Strategy*[3] document outlining future scenarios, challenges and recommended actions. The document also served as base to define the four generic CPS characteristic [TAB+17]:

- *Technical emphasis:* The aim of CPS is to integrate physical and embedded systems together with communication and software systems. With technical emphasis, we both decide how to design physical and embedded systems and how to use communication in order to achieve the most optimal performances.
- *Cross-cutting aspects:* Here we mean system properties (responsible for safety and security), jurisdiction (legislation) and governance (responsibility distribution). These points need to be taken into consideration when applications start to span across multiple domains and involve interconnectivity and dynamic reconfiguration.
- *Level of automation:* Defines what aspects are automated and to what degree. For example with autonomous driving there can be different levels of automation, where we only want automated parking or automated lane centering or maybe a fully automated vehicle that is independent from human input.
- *Life-cycle integration:* The degree to which a CPS is integrated in the management of existing products and services. The better the integration, the larger the benefits but also the larger the costs, creating a trade-off that needs to be considered.

The DynaCon example that we will later analyze and our improved version both strongly focus on the technical aspects of a CPS system.

1.3.1 DynaCon

Taking the CPS concept as a blueprint, we look to take the configuration cycle in Figure 1.1 and instantiate the abstract building blocks to effective components as they are described in DynaCon [ESS19]. What we defined as data generation is accomplished

[3] https://ec.europa.eu/digital-single-market/en/news/cyber-physical-european-roadmap-strategy

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

by data *Producers* like sensors. The data is then filtered by observers that can exist in the form of stream reasoners with optional communication managers. The brain behind the re-configuration aspect is a multitude of modules summarised as the Operator. The Operator can both request new data when necessary and design new configurations in form of update commands to optimize the systems. *Actuators*, or as they will be later referred to, update managers are responsible to perform the changes decided by the Operator.

In Chapter 2 we will pick these concepts up and take a better look into the nuances of reconfiguration and stream reasoners in addition to giving a detailed description of the DynaCon architecture and the issues we are attempting to address with this thesis.

1.4 Thesis structure

The remainder of this thesis is structured as follows. In Chapter 2 we dive deeper into cyber-physical systems by focusing particularly on Dynacon [EDTF+19] and later introduce the concept of logic programming and our second big stepping stone: LUPS [APPP02]. In Chapter 3 we describe the functionality of our interface and how it interacts with the other components of the dynamic system. Chapter 4 shows a redesign of the LUPS* [Lei01] command language into the TLUPS policy language and also shows our own interface and stream reasoner update commands. In Chapter 5 we introduce our Java prototype that includes core parts such as the Interface, Communication and Update Manager and the Operator. Chapter 6 gives some examples focused on the C-ITS use case to showcase the functionalities of our prototype. Lastly, in Chapter 7, we give a summary of the main results and give some insight to future work related to this thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

State of the Art

2.1 DynaCon

As discussed, one of the core functions of a CPS is its ability to adapt to a changing environment through the information that it gathers. This process of reconfiguration has been explored for other systems: *Plastik* [BJC05] for example is a meta-framework that aims to facilitate reconfigurations by ensuring integrity in component-framework-based software systems. *DyKnow* [dH16] looks to extend the robot operating system (ROS) by allowing run-time reconfigurations to its components. These knowledge based reconfiguration methods have been successful in reducing development and maintenance costs, while remaining flexible and maintaining soundness and completeness properties. However, these types of systems are not compatible with dynamic systems such as CPS due to their adaptive nature and large problem instances.

Reconfiguration methods involving CPS have also been researched [SWD+14]. However, these systems often do not make use of a logic based framework to define the system configuration, which also renders it harder to apply changes to the rule sets. In addition, the architectures are often not described in detail and omit the crucial connection points between monitors, stream reasoners and the appropriate interfaces to interact with the decision and reconfiguration modules.

As a prime example for a dynamic configuration system we will take a closer look at the DynaCon architecture proposed by Eiter et al. [EDTF+19]. At its base lie the concepts of Cyber-physical sytems (CPS), stream reasoners (SR) and (re)-configuration. The idea is to collect sensor information through a stream of data from the CPS. The data is then filtered by the stream reasoner, which sends the condensed information to the (re)-configuration module responsible for issuing new configurations through configuration commands.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

2.1.1 The architecture

We will briefly go over the architecture that inspired our design visible in Chapter 3. Let

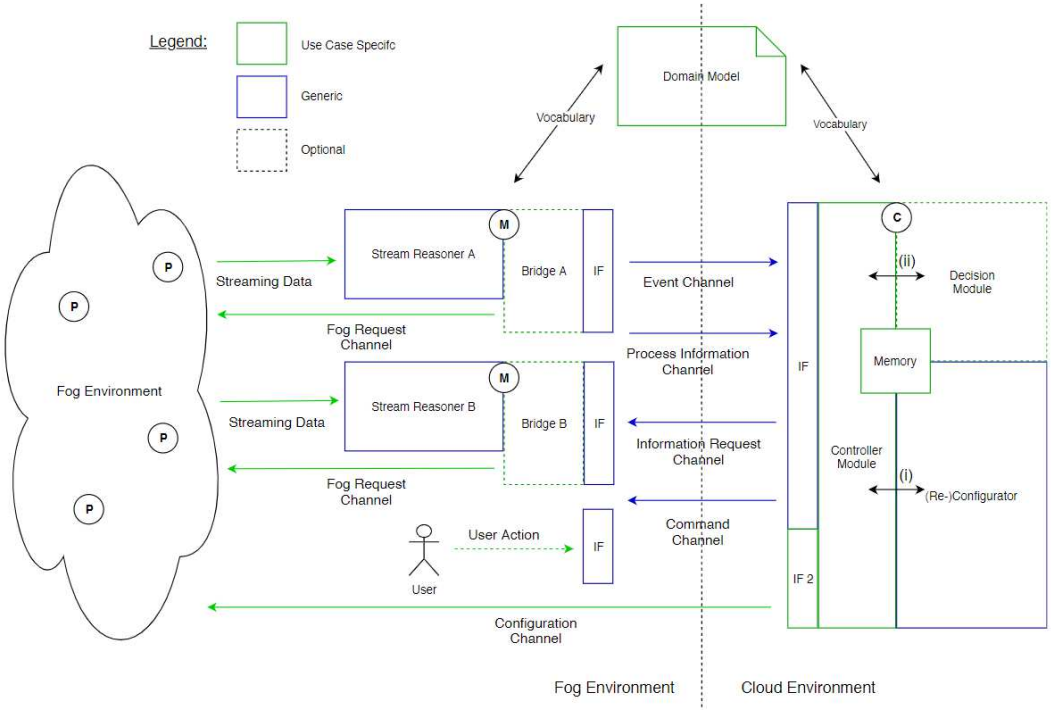


Figure 2.1: Original DynaCon architecture [EDTF+19, ESF+19]

Eiter et al., 2019, Stream Reasoning and Multi-Context Systems. Stream Reasoning Workshop 2019, Linköping, Sweden

us look at the architecture shown in Figure 2.1 in the scope of a use case. Imagine a traffic control situation where the goal is to change the traffic light signal plan in order to deal with the changing traffic situation. We will now separately analyze the two sides of the architecture divided by the interface.

The Fog Environment

On the left side of the figure lies the Fog Environment, which takes care of the collection, filtering and transmission of data. In the traffic control use case, the producers (P) are either vehicles or traffic lights that communicate through V2X communication messages and thus producing a data stream. We would then have the so called roadside units (RSU) located at intersections acting as monitors (M) by observing and collecting the V2X messages in order to detect events (accidents, traffic jams ecc.) or to aggregate data such as the number of cars in a lane. The collected data would then be transmitted through the communication channels to the configurator (C), which in the traffic control case is a traffic control centre (TCS). Here a new reconfiguration can be evaluated through the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

decision module. The RSU also assumes the role of an actuator (A)that lastly updates the signal plan for the traffic lights based on the new configuration.

The Cloud Environment

On the right side of the architecture lies the Cloud Environment containing all modules pertaining to configuration.

- **The Controller Module:** The specific tasks of this module depends on the use case. Mainly, it is responsible for capturing the information transmitted through the interface in order to manage, set-up and trigger the reconfigurator. It is also in control of preparing different ASP programs needed for the problem encoding necessary for the reconfiguration. In those use cases where a decision module is necessary, the controller module is also in charge of keeping an up-to-date state of the fog-side objects. This can either happen through the information that is received automatically from the fog-side, or through pull requests made by the controller to obtain any missing information. This information is stored in the memory module.

- **Reconfigurator Module:** When the controller module requests a reconfiguration, this module directly communicates with a reasoning engine (an ASP solver) by preparing the input for the solver. The re-configuration problem can be approached in a divide and conquer manner, where the main problem instance is divided up in smaller configuration problems. Once all of them are solved they can be combined to obtain the full reconfiguration.
  It is important to note that the reconfiguration is based on previous configurations and the differences between them, possibly trying to make use of previous conflicts and search decisions. One could also decide to use a normal configurator that would treat every reconfiguration attempt as a fresh one, thus being independent from previous configurations.

- **Decision Module:** In the DynaCon architecture the decision module is the brain to the muscle (the reconfigurator). It is the decision module's job to start off new reconfigurations, communicate information of the new configuration to the other modules in the architecture through the controller module and finally reconfigure specific stream reasoners. Its responsibility is basically to provide the specifications for the reconfiguration extracted from the information received from the controller, while the reconfiguration module then actually does the computations. These two tasks are separated in different components to encourage modularization, which facilitates component replacement.
  The module is marked as optional since in some cases it might be implemented as a virtual component or omitted completely. For example in the traffic light configuration scenario above, deciding whether one single traffic light should be reconfigured might be decided by using simple thresholds.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Stream reasoner in the DynaCon framework

As previously stated in Section 1.1 describing the basic concept of stream reasoners, its role is to act as a mediator between the data source and the target. Now that we see the whole architecture we can go into detail regarding the stream reasoner's role in the context of this dynamically configurable system. It has four main tasks:

- First and foremost comes the collection of data from multiple sensors that send information at irregular intervals and possibly in different formats. It is also important that the capture of this information takes place in real time. This allows for quicker reactions to the changing environment.
- Then, through logical rules, where intuitively the head of the rule is the event and the body of the rule are the single or multiple conditions required for the event to be triggered, the stream reasoner can filter out events from the data stream.
- Again using logical rules, information can be aggregated. For example if the sensors in our hypothetical scenario where to check the number of cars in each lane of an intersection, then the body of a rule could be composed of the measurements for each lane and the head can then build the average.
- Finally, the information is transmitted to the reconfiguration module through the communication channels.

2.1.2 Issues

While the DynaCon projects lays good foundations for a dynamic configuration system that makes use of an interface, it does not go into detail on many parts of the architecture and specifically does not specify details and a syntactic structure for an interface definition language. The descriptions are also vague regarding the transfer of information between the stream reasoner and the reconfiguration module through the bridge module, lacking critical parts like communication mode (push/pull/buffered) and eventual delays.

In Chapter 3 we will show an alternate version of this architecture with detailed explanations of the components. In that chapter we will later transition into a syntactical definition of the interface definition language.

2.2 Logic programming

The execution of Java, Python, C# and other general-purpose programming languages generally follows the pattern of imperative programming languages, where memory locations are updated based on instructions following a sequential style of execution. They precisely describe how a result is to be achieved by a series of statements that need to be executed by the computer. The idea behind logic programming is to focus on the relationship between components instead of step to step instructions. [Lee17b]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

The first steps in logic programming were made in early 1970s, building on previous works done in the fields of artificial intelligence and automated theorem proving. The basic building blocks laid out by Herbrand in the 1930s proved useful for Putnam and others in the 1960s who focused their research on theorem proving. Finally, in 1965, a landmark paper introducing the resolution rule, authored by Robinson [Rob65], was published. In short, resolution is an inference rule that, given two clauses, can derive a new clause implied by the two. A simple example is two clauses containing complementary literals that can be merged into one by omitting those literals and combining the clauses in a disjunction. Resolution is core to logic programming because it is very fitting for automated computation [Lee17a]. In 1972, Kowalski and Colmerauer [Kow74] introduced a programming language that was later called *Prolog*. It was born from the idea of using logic as a programming language and its execution makes use of the resolution method.

2.2.1 Answer set programming (ASP)

Answer set programming is a prime example of logic programming. It falls under the paradigm of declarative programming and is mostly used to solve NP-hard search problems. ASP is based on the *Stable Model Semantics*, a concept introduced by Gelfond and Lifschnitz in 1988 [GL88]. In contrast to the previously mentioned Prolog, where the order of rules influences the outcome of execution, ASP is purely declarative and as a consequence the order of rules is unimportant. In addition, as opposed to possibly running into infinite loops like Prolog , many answer set solvers use *DPLL* style algorithms [OC99] which, in principle, always terminates. Modern solvers also employ conflict driven clause learning (CDCL), which in terms of efficiency greatly outmatches previous efforts [MSLM09].

Syntax

We define the Alphabet for ASP following the syntactic definitions given in [Bal09] as follows:

Definition 1. (Alphabet)
The alphabet $\mathcal{A}$ consists of:

- variables $V = \{X, Y, Z...\}$
- function symbols $F = \{f, g, h, ...\}$ with arity >0 (function symbols with arity 0 are constants)
- predicate symbols $P = \{p, q, r, ...\}$ with arity >0 (predicate symbols with arity 0 are propositional variables)
- logical connectives of arity 0 ($\bot$, $\top$), arity 1 ($\sim$), arity 2 ($\wedge$, $\vee$, $\leftarrow$)
- quantifiers $\{\forall, \exists\}$
- punctuation symbols {"(", ")", ","}

We can thus define the language $\mathcal{L}$ as:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Definition 2. (Language)
A language $\mathcal{L}$ is a triple (F, P, *arity*) where F is a set of function symbols or constants, P is a set of predicate symbols or propositional variables and *arity* is an arity function $F \cup P \mapsto N$ (for example f(X) is of arity 1, p(X,Y) is of arity 2 etc.)

To reach the concept of a rule we need to first introduce the definitions for terms, atoms, formulae and literals, which are the basic building blocks for rules and by extension, logic programs.

Definition 3. (Term)
A term can be inductively defined in the following way:

- A variable is a term
- A constant is a term
- If f is a function symbol of arity n, and t_1, ... t_n are terms, then $f(t_1, ..., t_n)$ is a term as well.

Definition 4. (Atom)
An atom can be defined as follows:

- A propositional variable is an atom
- If p is a predicate symbol of arity n, and t_1, ... t_n are terms, then $p(t_1, ..., t_n)$ is an atom

Definition 5. (Formulae)
A Formula $\mathcal{F}$ can be inductively defined as follows:

- An atom is a formula
- $\bot$ and $\top$ are formulae (arity 0)
- If $\mathcal{F}$ is a formula, so is $\sim\mathcal{F}$ (arity 1)
- If $\mathcal{F}$ and $\mathcal{G}$ are formulae, so are $\mathcal{F} \wedge \mathcal{G}$, $\mathcal{F} \vee \mathcal{G}$ and $\mathcal{F} \leftarrow \mathcal{G}$ (arity 2)
- If X is a variable and $\mathcal{F}$ is a formula, then $\forall X\mathcal{F}$ and $\exists X\mathcal{F}$ are formulae (quantifiers)

Regarding literals, we define a Default Literal as an atom preceded by the $\sim$ symbol. A Literal is then either an atom or a default literal. Keeping these definitions in mind we define a Rule as:

Definition 6. (Rule)
A Rule is a formula of the form:

$$L_1 \vee ... \vee L_m \leftarrow L_{m+1} \wedge ... \wedge L_n$$

where L_i, $1 \leq i \leq n$ are literals. The formula $L_1 \vee ... \vee L_m$ is called the head of the rule and the formula $L_{m+1} \wedge ... \wedge L_n$ is called the body of the rule. A rule having $m = n$ (no rule body) and $m = 1$ (single disjunct in head) is called a fact. Instead, a rule with an empty head is called a constraint.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

We finally define logic programs as follows:

Definition 7. (Logic program)
A logic program is a set of rules. A positive logic program does not posses default negation. A normal logic program is allowed to contain default negation, but only in the body of the rule. Instead, a generalized logic program is allowed to contain negation also in the rule head. If a logic program has a disjunction in its head, we speak of a disjunctive logic program.

For example we could have the following:

$$\begin{aligned} \text{variable} &: X \\ \text{constant} &: 0 \\ \text{function symbol} &: s \text{ (arity 1)} \\ \text{predicate symbol} &: p \text{ (arity 1)} \end{aligned}$$

$$\begin{aligned} \text{terms} &: X, 0, s(X), s(0), s(s(X)), s(s(0)), ... \\ \text{atoms} &: p(X), p(0), p(s(X)), p(s(0)), ... \\ \text{literals} &: p(X), {\sim}p(X), p(0), {\sim}p(0), ... \\ \text{fact} &: p(0). \\ \text{rule} &: p(s(s(X))) \leftarrow p(X). \end{aligned}$$

Semantics

For the semantics we will refer to [Eit16] for the following definitions and look at normal logic programs P in the context of *Herbrand interpretations*, which are sets of ground atoms that are true. Most logic programming languages like Prolog and ASP use the following syntax for rules r:

$$a \leftarrow b_1, ..., b_m, not\ c_1, ..., not\ c_n$$

where the comma substitutes the logical conjunction symbol and *not* substitutes the unary default negation $\sim$.
In Herbrand interpretations we define the following:

- *Models*: are such interpretations where a is true whenever $b_1, ..., b_m$ are true and $c_1, ..., c_n$ cannot be proven as true.
- *grounding* of P: is defined as $grnd(P) = \bigcup_{r \in P} grnd(r)$ and $grnd(r)$ resembles the set of all ground instances of r (using ground substitution).
- *Herbrand universe (HU):* given a logic program P, $HU(P)$ is the set of all terms that can be formed from constants and function symbols. In our previous example, the HU is $0, s(0), s(s(0)), ...$
- *Herbrand base (HB):* given a logic program P, $HB(P)$ is the set of all ground atoms that can be formed from predicate symbols and terms $t \in HU(P)$. In our previously example, the HB is $p(0), p(s(0)), p(s(s(0))), ...$

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

- *Herbrand interpretation (I):* Is a first order interpretation $I = (D, \cdot^I)$ which has its domain $D = HU(P)$ and where each term $t \in HU(P)$ is interpreted by itself. A Herbrand interpretation can thus be seen as a set denoting what ground atoms are true in a given scenario.

Example 2. Take the logic program P:

$$p(X) \leftarrow not\ r(X),\ q(X).$$
$$q(2) \leftarrow.$$
$$q(1) \leftarrow.$$

where p, q, r are predicate symbols with arity 1. We identify:

- Constant symbols: 1, 2
- HU(P): {1,2} (since we have no function symbols)
- HB(P): $\{q(1), q(2),$
  $p(1), p(2),$
  $r(1), r(2)\}$
- Possible interpretations can be any subset of $HB(P)$, including the empty set.

We are interested in finding out, which interpretations are a Model of P. Using the Closed World Assumption (CWA) we can find the minimal model of a logic program P. A minimal model I distinguishes itself from other models J because no $J \supset I$. In general, we want the truth of an atom in I to be "founded" by a clause, which means it should either be stated as a fact or be the conclusion to a clause. Through the CWA, all other atoms are to be regarded as not true. For the program P the minimal model is:

$$M_1 = \{q(1), p(1), q(2), p(2)\}$$

In normal logic programs, negation can introduce multiple valid minimal models. For example we can add the clause

$$r(X) \leftarrow not\ p(X),\ q(X).$$

to the logic program P. In this case we would be presented with four models, namely:

$$M_1 = \{q(1), q(2), p(1), p(2)\}$$
$$M_2 = \{q(1), q(2), p(1), r(2)\}$$
$$M_3 = \{q(1), q(2), r(1), p(2)\}$$
$$M_4 = \{q(1), q(2), r(1), r(2)\}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Which one should we choose from? To deal with multiple valid Models we introduce the notion of Answer Sets. They are based on Herbrand interpretations and are also referred to as stable models using the CWA.

Definition 8. (Satisfaction)
Consider an interpretation $M \subseteq HB(P)$: M satisfies:

- a ground atom a (resp. $not\ a$) if $a \in M$ ($a \notin M$)
- a ground variable free rule r,

$$a_1 \vee ... \vee a_k \leftarrow b_1, ..., b_m, not\ c_1, ..., not\ c_n$$

  if either

  (i) M does not satisfy some literal b_i or $not\ c_j$ in $Body(r)$
  (ii) M satisfies some $a_i \in Head(r)$

- a ground program P, if M satisfies each $r \in P$
- a rule r (resp. program P), if M satisfies each $r' \in grnd(r)$ (resp. $grnd(P)$)

We can thus raise two conditions for a minimal model. An interpretation $M \subseteq HB(P)$ is a minimal model of P if:

(i) M satisfies P
(ii) no $N \subset M$ satisfies P

We can finally define Answer Sets in the following way:

Definition 9. (Answer Set)
M is an answer set of a program P, if M is a minimal model of P^M. $AS(P)$ denotes the set of all answer sets of program P.

Going back to Example 2 with the rule we added we can take the four models M_1, M_2, M_3, M_4 which then creates the set of all answer sets $AS(P)$.

2.2.2 HEX and ActHEX

Hex programs, introduced by Eiter et al. in [EIST05], are an extension to answer set programming allowing for higher-order atoms and external atoms. External atoms, as the name suggests, allow to exchange knowledge with outside sources and thus fostering software interoperability. For example a task can be delegated to an external computational source as visible in the rule

$$reached(X) \leftarrow \&reach[edge, a](X)$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

where $\&reach[edge, a]$[1] computes all nodes in a graph *edge* that are reachable from node *a*. If as a result we derive $reached(n1)$, it means the node *n1* is reachable by node *a* in the graph *edge*. In 2013, Fink et al. extended Hex programs with the ActHex formalism allowing to use action atoms in the rule head to change the state of an environment outside of the logic program. The rule

$$\#robot[goto, charger]\{1\} \leftarrow \&sensor[bat](low)$$

employs the action atom $\#robot$ to control the robot and the external atom $\&sensor$ to access external sensor data. In this example, the parameter appended to the action [*goto,charger*]{**1**} represents the priority given to the action.

2.2.3 HexLite

HexLite [Sch19] is a lightweight solver developed by Schüller for the aforementioned ActHex programs. It is written in the Python programming language, which is also the language in which the external atoms will be evaluated in. We mention it here because it will be our solver of choice for the prototype implementation presented in Chapters 5 and 6. For instance, it will allow us to read the contents of a database where traffic sensor data is stored in real time. The rule

```
@wsConnect(X) :- not &wsConnected[X], server_adr(X).
```
[2]

for example creates a connection to the server address `X` if no web socket connection is already present.

2.2.4 LARS

LARS [BDTE15] presented by Beck et al. is a logic-based framework for analyzing reasoning over streams. This will be the main example for a stream reasoner that we will come back to later when expanding on the architecture of an ideal dynamically configurable system that makes use of an adaptable interface. LARS follows the trend of data *pushing* and while a dynamically reconfigurable stream reasoner has been researched in the DyKnow framework [dH16], it merely issues reconfigurations and does not reason on the incoming data streams to determine whether a reconfiguration is necessary. LARS instead offers a flexible mechanism to alter the view on streaming data through the usage of windows. It can prepare the streamed data in a way that the Reconfiguration Module can make decision based on the extracted events and discretized information. By applying these window operators, LARS attempts to deal with the possibly infinite input streams. We now give some definitions that will later be used to describe the stream reasoning module:

[1] in [EFI+15] the ampersand symbol substitutes the # symbol used in earlier versions
[2] our implementation uses @ as a symbol for action atoms instead of #

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Definition 10. (Stream)
A stream of data S denoted by $S = (T, v)$ where T, the timeline, is a finite interval in $\mathbb{N}$ consisting of time points $t \in T$ and v is an evaluation function $v : \mathbb{N} \mapsto 2^{\mathcal{A}}$, where $\mathcal{A} = \mathcal{A}^{\mathcal{E}} \cup \mathcal{A}^{\mathcal{I}}$ denotes the full set of atoms, $\mathcal{A}^{\mathcal{E}}$ the extensional atoms and $\mathcal{A}^{\mathcal{I}}$ the intensional atoms.

Intuitively, v maps to each time point the atoms pertaining to that exact moment. For example, let $T = [0, 20]$ and let there be a power outage at time point 13. Then $v(13) \mapsto \{outage\}$ and $v(t) \mapsto \emptyset$ for all other $t \in T$. The window of observation can be reduced in an effort to deal with big amounts of data. Assume that in our previous example, any measurements taken more than x time points ago are to be regarded as obsolete. We can then define a stream $S' = (T', v')$ where $S' \subseteq S$, i.e., $T' \subseteq T$ and $v'(t') \subseteq v(t')$ for all $t' \in T'$. We call S' a substream of S.

Definition 11. (Window function)
Any (computable) function w that returns, given a stream $S = (T, v)$ and a time point $t \in T$, a substream S' of S is called a window function.

In Plain LARS [BDTE15] we distinguish *time-based* and *tuple-based* window functions. These can be expressed through the window operator $\boxplus$. An expression $\boxplus\alpha$ is to be evaluated only in the interval delivered by the respective window function $w_{\boxplus}$. The two window functions are represented by the following operators:

- Time based $\boxplus^x$ restricts the snapshot to the last x time points.
- Tuple based $\boxplus^{\#x}$ restricts the snapshot to the last x tuples, where a tuple consists of a time point and an atom.

Additionally, we can have the following set of extended atoms defined by the grammar

$$a \mid @_t a \mid \boxplus^w @_t a \mid \boxplus^w \Diamond a \mid \boxplus^w \Box a$$

where $a \in \mathcal{A}$ and $t \in \mathbb{N}$ a time point.

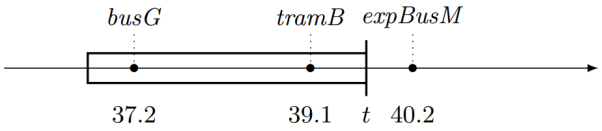
The symbols $\{@_t, \Diamond, \Box\}$ give a time reference to the atom a. Take a stream of data $S = (T, v)$ then at time point $t \in T$ and given a power outage at $t = 13$ we have the following:

- $@_{t'}$ targets a certain point in time and is used to check whether an atom holds at a specified time point. $@_{t'}a$ thus holds, if $t' \in T$ and a holds at time point t'.
  Example: $@_{13}outage$ holds at any time point since $v(13) \mapsto \{outage\}$
- $\Diamond$ is a substitute to the **existential** quantifier $\exists$ used in first order logic. $\Diamond a$ thus holds if a holds at any time point $t' \in T$.
  Example: $\boxplus^5 \Diamond outage$ only holds in time points $t \in [13, 18]$ since the window operator only looks back 5 time points.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- $\Box$ instead is a substitute to the **for all** quantifier $\forall$ used in first order logic. $\Box a$ thus holds if a holds at all time points $t' \in T$
  Example: $\boxplus^{x}\Box outage$ holds only if we restrict the timeline T to only include time points t' such that $v(t') \mapsto \{outage\}$. So if in addition we also have $v(14) \mapsto \{outage\}$, then $\boxplus^{1}@_{14} outage$ holds at all time points.

Multiple rules can then be collected to form a program P. We relay Figure 2.2 displaying a LARS program that aims to recommend to a user waiting at stop M, to either take the bus or the tram based on the current traffic situation and transport schedule.



$$
\begin{aligned}
(r_1)\quad & @_{T+3m}\, expBusM \leftarrow \boxplus^{3m} @_T\, busG, on.\\
(r_2)\quad & @_{T+5m}\, expTrM \leftarrow \boxplus^{5m} @_T\, tramB, on.\\
(r_3)\quad & on \leftarrow \boxplus^{1m} \Diamond request.\\
(r_4)\quad & takeBusM \leftarrow \boxplus^{+5m} \Diamond expBusM, \text{not } takeTrM,\\
& \qquad\qquad \text{not } \boxplus^{3m} \Diamond jam.\\
(r_5)\quad & takeTrM \leftarrow \boxplus^{+5m} \Diamond expTrM, \text{not } takeBusM.
\end{aligned}
$$

Figure 2.2: The timeline of events with a request coming in at $t = 39.7$ and a program P to recommend public transport [BDTE15]

Beck et al., 2015, Answer Update for Rule-Based Stream Reasoning. Proceedings of the 24th International Conference on Artificial Intelligence 2015, Buenos Aires, Argentina

The program makes use of the extended atoms introduced above and consists of five rules: (r_3) is triggered if a request for recommendation has been made in the last minute. If a request was made, then *on* is derived and the recommendation system is unlocked. (r_1) says that, if in the last 3 minutes a bus arrived at stop G and *on* has been derived through r_3, then (following the public transport schedule and map) we know that the bus is expected to arrive at stop M 3 minutes after it arrived at stop G. Similarly, (r_2) says that if a tram arrived at stop B in the last 5 minutes and *on* has been derived through r_3, then we can expect the tram to arrive at stop M 5 minutes afterwards. (r_4) suggests to take the bus from stop M if in the next 5 minutes we expect a bus to arrive at stop M and taking the tram has not been suggested yet. In addition, there is also a condition that checks for any traffic jams that might have occurred in the last 3 minutes as an ulterior requirement to taking the bus. Finally, (r_5) follows the same logic as r_4.

In [BDTE15], Beck et al. define the semantics as follows:

> For a data stream $S = (T_S, v_S)$, any stream $I = (T, v) \supseteq S$ that coincides with S on $\mathcal{A}^{\mathcal{E}}$ is an *interpretation stream* for S. A tuple $M = \langle T, v, W, \mathcal{B}\rangle$, where

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

W is a set of window functions and $\mathcal{B}$ is the background knowledge, is then an *interpretation* for S. Throughout, we assume W and $\mathcal{B}$ are fixed and thus also omit them.
Satisfaction by M at $t \in T$ is as follows: $M, t \vDash \alpha$ for $\alpha \in \mathcal{A}^+$, if α holds in (T, v) at time t; $M, t \vDash r$ for rule r, if $M, t \vDash \beta(r)$ implies $M, t \vDash H(r)$, where $M, t \vDash \beta(r)$, if (i) $M, t \vDash \beta_i$ for all $i \in \{1, \ldots, j\}$ and (ii) $M, t \nvDash \beta_i$ for all $i \in \{j+1, \ldots, n\}$; and $M, t \vDash P$ for program P, i.e., M is a *model* of P (for S) at t, if $M, t \vDash r$ for all $r \in P$. Moreover, M is *minimal* if in addition no model $M' = \langle T, v', W, \mathcal{B}\rangle \neq M$ of P exists such that $v' \subseteq v$.

Definition 12. (Answer Stream) An interpretation stream i is an answer stream of program P for the data stream $S \subseteq I$ at time t, if $M = \langle T, v, W, \mathcal{B}\rangle$ is a minimal model of the reduct $P^{M,t} = \{r \in P | M, t \vDash \beta(r)\}$. By $\mathcal{AS}(P, S, t)$ we denote the set of all such answer streams I.

To tie together the example in Figure 2.2 and the Answer Stream Definition, Beck et al. give the following example.

Example 3. Let $S' = (T, v')$ be the data stream which adds to the stream S from Figure 2.2 the input $39.7 \mapsto \{request\}$. We get two answer streams $I_1 = (T, v_1)$ and $I_2 = (T, v_2)$ of P for S' at $t = 39.7m$ which both contain, in addition to the mappings in v', $40.2m \mapsto \{expBusM\}$ and $44.1 \mapsto \{expTrM\}$. Answer stream I_1 additionally contains the recommendation $t \mapsto \{takeTrM\}$ and I_2 suggests $t \mapsto \{takeBusM\}$, since both vehicles are expected to arrive in the next 5 minutes.

Even though in our future examples we will be using plain LARS, it is important to note that in [BDTEF15] Beck et al. enhance the window operator with the addition of a *stream choice* and vectors for window parameters. For instance, a *time-based* window function can be expressed with a parameter vector $x = (l, u, d)$ to specify how far in the past (l), how far in the future (u) and with what step size (d) the window function should operate. In addition, the stream choice ch can choose between the fixed input stream $S^\star$ and the currently considered window S.

Furthermore, in [BDTE16] Beck et al. define a novel logic called *Bi-LARS* that captures the FLP[3] [FLP04] based semantics of a large fragment of LARS programs.

This section is not meant to give exhaustive insight to the LARS Framework. For further reading on how LARS extends Truth Maintenance Systems, the Answer Update Algorithm, the Complexity of Reasoning in LARS and the full context behind Equivalent Stream Reasoning Programs refer to the materials [BDTE15, BDTEF15, BDTE16] cited in this section.

[3] A fully declarative, genuine generalization of the answer set semantics for disjunctive logic programming (DLP)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

2.3 Updating rule sets of logic programs

Before we analyze the state of the art and previous research regarding updates in logic programs we need to define and analyze the difference between what it means to *revise* and *update* a logic program. Initially the differences were not defined well until *rationality postulates* were defined in [AGM85] and then later revised in [KM91], which argues that there is no one fit all measure that can encapsulate all applications. It is further stated that both revision and update are two kinds of modifications to a knowledge base and then describes the differences by defining the two terms as follows:

- Revision: Is used when the goal is to infer new knowledge from an unchanging, static world (for example through binary resolution [Bun13]).
- Update: Is used when the world described by the knowledge base changes and consequently the knowledge base describing it is brought up to date.

We are interested in updating our knowledge base through the use of an interface description and update commands and will thus look closer on what attempts have been done towards this field of research.

The first attempts where made by Marek et al. in 1994 [MT94], Przymusinski et al. in 1997 [PT97] and Alferes and Pereira in 1996 [AP06] which were mainly concerned with finding *"interpretation updates"* [KM91]. This approach focuses on applying an update in the form of a knowledge base U to an existing **monotonic** knowledge base KB by finding the set of updated models of KB.

However these approaches were not well suited for non-monotonic applications. As [APPP02] points out, even though [AP06] managed to remove some of the drawbacks, it still was not possible to update logic programs comprised of logic rules and not just facts.

2.4 LUPS and LUPS*- A language for updating logic programs

A solution to the last drawback was finally proposed by Alferes et al. in the form of LUPS [APPP02] in 1999; a language for updating logic programs and its later improved version LUPS* [Lei01] in 2001 by Leite. Because later in Section 4.2 we will extend LUPS* in order to fit our purposes better, we now give an introduction to the LUPS concept as well as its syntax and semantics.

2.4.1 LUPS

LUPS's goal is not only to semantically describe how a logic program that is updated through the usage of update commands looks like, it also aims to describe the state transitions in between one state and the next of the KB. We can for example create

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

update commands of the form "add rule r_1 only if condition x applies"that will only add or remove a rule when certain conditions apply. It is also possible to define commands that will persist through the next transition states and not only affect the next state. These update commands are then treated as a logic program themselves, creating a sequence of updates through logic commands defined as *dynamic logic programming*, a paradigm presented in [ALP+00].

Definition 13. (Dynamic logic programming) A finite or infinite sequence of consecutive updates of a logic program by logic programs of the form $P_0 \bigoplus \ldots \bigoplus P_n \bigoplus \ldots$ with $P_0 = \{\}$.

Since dynamic logic programming does not provide an apposite language that can describe transitions in logic programs, the LUPS language was conceptualized. The following are the update commands as they are defined in [APPP02]:

(1) **assert** $L \leftarrow L_1, ..., L_k$ **when** $L_{k+1}, ..., L_m$
(2) **assert event** $L \leftarrow L_1, ..., L_k$ **when** $L_{k+1}, ..., L_m$
(3) **always** $L \leftarrow L_1, ..., L_k$ **when** $L_{k+1}, ..., L_m$
(4) **always event** $L \leftarrow L_1, ..., L_k$ **when** $L_{k+1}, ..., L_m$
(5) **cancel** $L \leftarrow L_1, ..., L_k$ **when** $L_{k+1}, ..., L_m$
(6) **retract** $L \leftarrow L_1, ..., L_k$ **when** $L_{k+1}, ..., L_m$
(7) **retract event** $L \leftarrow L_1, ..., L_k$ **when** $L_{k+1}, ..., L_m$

We briefly go over the semantics of the keywords, for this purpose let us assume that we are currently in knowledge state KS_i and that these update commands describe the transition to the states after that:

- **assert** is a non persistent command that will only be executed once in the transition from KS_i to KS_{i+1}. It adds rule $L \leftarrow L_1, ..., L_k$ to KS_{i+1} if condition $L_{k+1}, ..., L_m$ applies in KS_i. This rule will **remain by inertia** in the successive knowledge states KS_{i+1}, KS_{i+2}, ... until retracted.

- **assert event** is a combination using the **event** modifier, which will cause the rule to only remain for the next knowledge state KS_{i+1}. It will then be automatically removed from KS_{i+2}.

- **always** has the same result as the **assert** keyword, with one exception: while a normal assert command will only be executed once, an **always** command is persistent, which means it will be executed for every subsequent transition until the command is canceled through the **cancel** command. This type of keyword can be useful when we have a condition for which we are unsure when it is gonna change its truth value. By having a persistent command, we can make sure that as soon as the conditions are met in a knowledge state, the rule will be added in the next one and, depending on whether we have a command like (3) or (4), respectively remain by inertia or have the rule retracted in the following state.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

It is important to note that if a **when** condition is present, it is evaluated in each iteration to decide whether the command associated should be executed.

- **always event** again mirrors the behavior described for the **assert event** combination, except that the command will persist for future state transitions. This means as soon as the conditions are satisfied in the current or a future knowledge state, the rule will only be added for the subsequent knowledge state and then automatically retracted again.

- **cancel** removes a persistent update command previously defined through the **always** keyword.

- **retract** works the same way as **assert**, except the rule is removed from the knowledge state instead of added. This change is also inertial, i.e., the rule is removed from all future states.

- **retract event** acts like the **assert event** keyword, except that the rule is removed instead of added. Because of the **event** keyword, the rule is only deleted in the next knowledge state KS_{i+1}, the rule will then come back in KS_{i+2} automatically.

Example 4. The example given by Alferes et al. in their LUPS paper [APPP02] is very well suited to display the functionality of most of the above commands. It presents a real life situation where an attempt is made to formulate legal reasoning through logic programming. The example is based on the US political system and on the debate between Republican and Democratic parties on the legality of abortion. The scenario is constructed as follows:

- If the Republicans take the majority in both Congress and Presidency, then they will adopt a law punishing abortion by jail.
- If the Democrats take the majority in both Congress and Presidency, then they would remove said law.
- During any time where no faction holds both roles, no changes are made to this law.
- Abortion as an action will be treated as non-inertial, i.e., *abortion(mary)* is true only in the state where it is added as an update).

In LUPS, the scenario could be described with the following persistent update commands:

always $jail(X) \leftarrow abortion(X)$ **when** *repC, repP*
always *not* $jail(X) \leftarrow abortion(X)$ **when** *not repC, not repP*

These rules behave exactly as described in our scenario. When *repC* and *repP* are true, $jail(X) \leftarrow abortion(X)$ is asserted. Meanwhile, if both are false, *not repC* and *not repP* become true and the complement is asserted. By the nature of dynamic logic programming, the most recent assertion will take priority in case of conflicts (see Rejected rules Definition 15).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Note: Even though there are no such rules in this example, we observe that any other rule having $jail(X)$ as its head would also be rejected as follows by the definition of Rejected rules. Such a rule could be $jail(X) \leftarrow murder(X)$. This problem will later be referenced with LUPS* in Section 2.4.2.

Assume the following timeline:

(1) Democratic Congress and Republican President
(2) Mary performs an abortion
(3) Republic Congress elected and Republican President stays
(4) Kate performs an abortion
(5) Republic Congress stays and Democratic President elected
(6) Ann performs an abortion
(7) Democratic Congress elected and Democratic President stays
(8) Susan performs an abortion

Alferes et al. model this timeline with the following LUPS commands:

U_1 : **assert** *repP*
assert *not repC*
U_2 : **assert event** $abortion(mary)$
U_3 : **assert** *repC*
U_4 : **assert event** $abortion(kate)$
U_5 : **assert** *not repP*
U_6 : **assert event** $abortion(ann)$
U_7 : **assert** *not repC*
U_8 : **assert event** $abortion(susan)$

They summarise the rules asserted throughout the updates and the logic used to determine when an abortion is punished by jail as follows:

- At first there is no rule that implies jail in case of an abortion and none are asserted in U_1 either.
- In U_2, Mary decides to get an abortion, but will not be punished by jail since no rule implies it.
- A Republican Congress is elected in U_3, this then triggers our first persistent command since both *repC* and *repP* are true. We now have $jail(X) \leftarrow abortion(X)$.
- Now Kate's abortion in U_4 triggers *jail(kate)*. We observe that even though Mary opted for an abortion in U_2, the fact *abortion(mary)* was retracted in U_3 (semantics of **event**) causing her not to be punished by jail because of future changes.
- In U_5, a Democratic President takes over. This will not result in the retraction of the previously asserted rule since the Congress is still Republican.
- Since the same rule is in place, also Ann will be punished with jail when she opts for an abortion in U_6.
- Finally, in U_7, a Democratic Congress is elected as well and the second persistent command triggers asserting $not\ jail(X) \leftarrow abortion(X)$.
- When Susan opts for an abortion the rule asserted in U_7 takes precedence over the rule asserted in U_3 because it is more recent and Susan faces no punishment.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

The LUPS Semantics

We want to provide the set of commands seen above with a meaning. How are dynamic logic programs evaluated? What happens with conflicting rules? To answer these questions we translate the update program consisting of update commands to a dynamic logic program, defined in Definition 13. Since we can view the construct of dynamic logic programming as taking P_0, then updating it with P_1, then P_2 etc., we can observe it as an evolving knowledge base of rules. Intuitively, we can thus append new rules to the sequence without having to worry about conflicts with previous rules since the role of dynamic programming is to maintain previous rules active only as long as they do not conflict with a newer rule.
Before construction the Model M we give the following definition:

Definition 14. (**Generalized logic program**)[4] A generalized logic program P in the language $\mathcal{L}_\mathcal{K}$ is a (possibly infinite) set of propositional rules of the form

$$L \leftarrow L_1, ..., L_n$$

where $L_1, ..., L_n$ are literals and $\mathcal{K}$ is a set of propositional variables whose names do not start with "*not*". By $\mathcal{L}_\mathcal{K}$ we then mean the language with the set of propositional variables $\{A : A \in \mathcal{K}\}$ (called objective atoms) $\cup\{not\ A : A \in \mathcal{K}\}$ (called default atoms). We call a logic program P **normal** if none of the literals in the heads of the rules in P are default ones, i.e., do not begin with the word *not*.

To construct the stable model M of the sequence up to P_N, as a first step, given a Model M of the most recent program P_n, remove all rules from previous programs in the sequence, where its head appears as a complement in some later rule with a body that is true in M.

Definition 15. (**Rejected rules**)[4] Let $\bigoplus P_i \in S$ be a dynamic logic program, let $s \in S$, and let M be a model of P_s. Then:

$$Reject_s(M) = \{L_0 \leftarrow Body \in P_i | \exists\ not\ L_0 \leftarrow Body' \in P_j, i < j \leq s \wedge M \vDash Body'\}$$

As expected, *not* L_0 is the complement of L_0 and both *Body* and *Body'* are conjunctions of literals.

All rules that are not part of the *rejected rules* persist by inertia. To reach the stable model of a single generalized program, we use the Closed World Assumption principle and thus add facts *notA* for all the atoms A where no rules with a positive body exist. We hereby create the set of *Default rules*.

[4] As defined in [APPP02]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Definition 16. (**Default rules**)[4] Let M be a model of a generalized logic program P. Then:

$$Default(P, M) = \{not\ A\ |\nexists\ A \leftarrow L_1, ..., L_n \in P : M \vDash L_1, ..., L_n\}$$

We can then finally compute the stable model M of the P_n sequence.

Definition 17. (**Stable models of a DLP at state s**) [4] Let $\bigoplus \mathcal{P} = \bigoplus\{P_i : i \in S\}$ be a dynamic logic program, let $s \in S$, and let $\mathcal{U} = \bigcup_{i \leq s} P_i$. A model M of P_s is a stable model of $\bigoplus \mathcal{P}$ at state s iff:

$$M = least([\mathcal{U} - Reject_s(M)] \cup Default(\mathcal{U}, M))$$

least defines the smallest model that contains all logical consequences of the two subsets. The resulting model is then always a superset of the individual subsets.

Example 5. We now tie together Definitions 15 - 17. Consider the dynamic logic program $P_1 \oplus P_2$, where P_1 and P_2 are:

$$P_1 : a \leftarrow . \qquad b \leftarrow not\ c \qquad\qquad P_2 : not\ b \leftarrow a$$

Then the only stable model at P_2 is $M = \{a, not\ b, not\ c\}$. The reasoning is the following: $Default(P_1 \cup P_2, M) = \{not\ c\}$ since c cannot be derived from any rule. We also have $Reject_2(M) = \{b \leftarrow not\ c\}$ since the rule in P_2 has as its head the complement of b and the body a is true in M. We then get M by:

$$M = \{a, not\ b, not\ c\} = least((P_1 \cup P_2 - \{b \leftarrow not\ c\}) \cup \{not\ c\})$$

This concludes the semantics section after showing how we can reach a stable model from a set of update commands.

2.4.2 LUPS*

The author of LUPS* [Lei01] discovered an issue with how the **assert event** command added and retracted rules in the KB. Rules added through the command can not be distinguished from previous rules present in the KB with the same syntax. The inability to tell the rules apart then leads to both being retracted in the next state transition (due to the semantics of **event**), thus eliminating knowledge that should have remained intact. To combat this, in the author's perspective wrong behavior, [Lei01] introduced a way to uniquely identify rules through a pair of new propositional variables that solve the aforementioned problem.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Syntactic changes

In addition, due to other ambiguities and lack of functionality, some of the previous seven commands were altered:

- LUPS commands (3) and (4) become **always assert** [**event**] to distinguish from the new command **always retract** [**event**], where **event** is optional.
- LUPS command (5) becomes **cancel assert** to distinguish from the new command **cancel retract**.

and some new commands were introduced:

(8-9) **always retract** [**event**] $L \leftarrow L_1, ..., L_k$ **when** $L_{k+1}, ..., L_m$
(10) **cancel retract** $L \leftarrow L_1, ..., L_k$ **when** $L_{k+1}, ..., L_m$

The changes to the syntactical structure of the rules are summarised in Table 2.1.

LUPS	*LUPS**
assert [**event**]	**assert** [**event**]
retract [**event**]	**retract** [**event**]
always [**event**]	**always assert** [**event**]
non existing	**always retract** [**event**]
cancel	**cancel assert**
non existing	**cancel retract**

Table 2.1: Commands added in LUPS*

Semantic changes

For the largest part the semantics remained unchanged. The translation into a dynamic logic program follows the same procedure as for LUPS with the addition of a new propositional variable "$N(R)$" for each rule R, allowing targeted retraction of rules through the addition of the complement *not* $N(R)$. In addition, to combat the problem introduced by the retraction of rules asserted through the **event** keyword (non-inertial rules), a further propositional variable "$Ev(R, S)$" was introduced for each rule R asserted with an **event** keyword in state S. By doing so, in an arbitrary state $S+1$ all non inertial rules asserted in state S can be retracted safely through the negation *not* $Ev(R, S)$ without affecting any other rules that might share the same head.
The full translation into a dynamic logic program has been adapted for our version TLUPS and can be seen in Section 4.2.

The next example serves as a continuation to Example 4 and shows the issues described before and how the new commands allow to solve them.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Example 6. Assume that we have the following two rules active:

jail(*X*) ← *abortion(X)*
jail(*X*) ← *murder(X)*

If we then use the command **assert** *not jail*(*X*) ← *abortion(X)* to remove the rule punishing abortion, as suggested in Example 4, it will automatically invalidate the rule punishing murder as well. LUPS* commands (3-4) and (8-9) solve this issue:

always assert *jail*(*X*) ← *abortion(X)* **when** *repC, repP*
always retract *jail*(*X*) ← *abortion(X)* **when** *not repC, not repP*

The **always retract** statement will only remove the rule punishing abortion and will leave other rules implying jail with a different rule body untouched.

However, LUPS* still does not reach a level of expressiveness that we deem as sufficient. Furthermore, there are further problems with the **cancel assert** and **cancel retract** commands, which do not allow to specify which command needs to be canceled in an instance where two commands assert the same rule, but with different conditions. We will elaborate on these two problems in Section 4.2.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 3

Abstract Architecture and Components

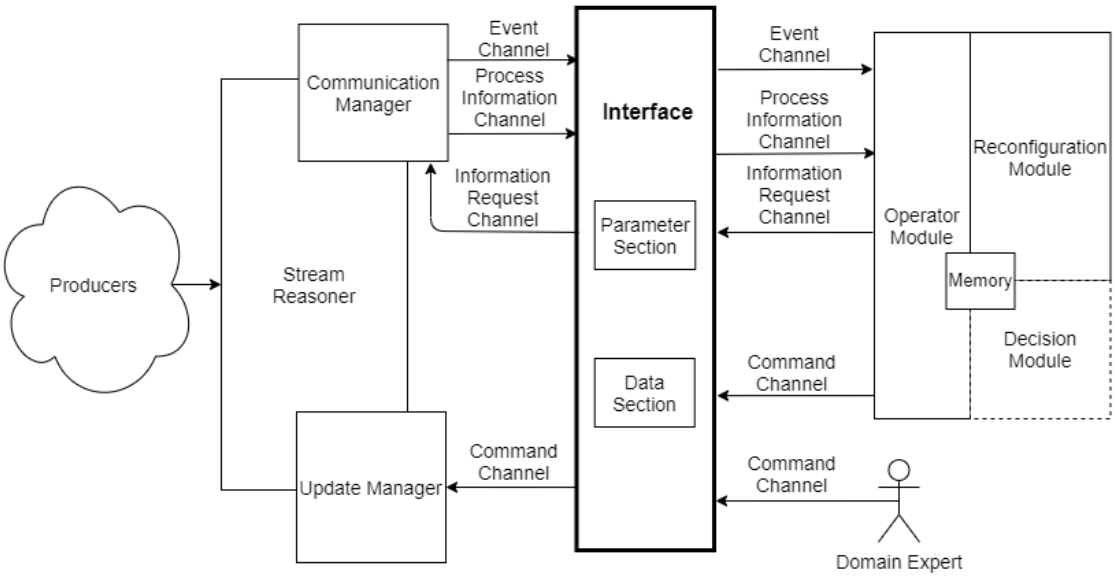


Figure 3.1: Dynamic Interface Architecture, readapted from DynaCon [EDTF+19]

As stated before, one of the goals of this thesis is to design an interface that can interact with different types of stream reasoners. We will thus make the general assumption that there will be some type of producer responsible for streaming unfiltered or prefiltered data directly to the stream reasoner. Such producers might be simple sensors responsible for measuring use case specific metrics or the output of other stream reasoners merging together. As in the nature of streamed data, the rate at which it will be transmitted is

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

variable and thus hard to predict. Furthermore, raw data is usually not well suited to be fed directly into a reconfiguration module. The stream reasoner thus intercepts the data stream and through the usage of a rule based language, groups low level data like quantitative temperature measures to high level abstractions like *hot*, *freezing*, etc. Not only should the stream reasoner be capable of detecting events through predefined rules acting on "instant" data, it should also be able to aggregate continuous data into accumulated process information that can be either sent regularly over the appropriate channel or requested by the reconfiguration module through pull requests. The accumulation of data is possible through certain logic based languages like LARS [BDTE15], C-SPARQL [BBCG10], and DLP$^{\mathcal{A}}$ [DFI+03].

3.1 Stream Reasoners

In the scope of this work, we aim to create a concept for the interface with the least amount of assumptions possible about the stream reasoner. Each use case requires and has different types of stream reasoners available. The latter can reach from simple rudimentary rule system implementations, like a pure ASP rule system, to more advanced reasoners that support updates to its rule system and implement functionalities such as time-based and tuple-based window functions, e.g., Laser* [BBU17].
Towards this goal, some parts of the description of the interface will assume specific capabilities on the stream reasoner side: ability to execute assert / retract statements, ability to push messages, ability to answer queries. These capabilities can either be native functionalities of the stream reasoner or additional modules, like a communication or update manager that can be combined with more basic stream reasoners. Since many stream reasoners have no built-in functions to send messages over communication channels, the first optional module that comes into play is the Communication Manager.

3.2 Communication Manager

The Communication Manager acts as a proxy module between the stream reasoner and the Operator. It is set in place to regulate the communication in order to only allow relevant and desired information to reach the Operator instead of having an unfiltered communication channel. The following tasks are managed by the Communication Manager:

1. *Event handling:* when event messages are generated by the stream reasoner, they are picked up by the Communication Manager. The interface description (covered below) is then used to determine whether an event should be sent, and what additional information (similar to process information) have to be added to the event message.

2. *Process information handling:* accumulated process information can either be obtained through pull requests sent by the Operator, or are sent automatically

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

to the Operator by the Communication Manager through push based messages. Different styles of communication involving buffering and delays are permitted to fine-tune the communication between the Communication Manager and the Operator module. Process information requests can either be dealt with by fetching the data from an internal cache or by calling the stream reasoner to complete the request.

3. *Communication handling:* as mentioned, the Communication Manager can employ several communication strategies. For instance, two examples for types of delay are: "buffer and burst" that buffers messages for later sending and "same message delay", where subsequent messages that are equal to the previous one received are discarded.

Event handler

In the default case, these messages fall under the category of push messages, since they are automatically fired by the stream reasoner and received by the Communication Manager. The Communication Manager then checks with the interface description to determine its behavior regarding the specific event. This process involves checking for the presence of the event in the interface and of any eventual exceptions associated to it. The process is explained in Section 3.5.2 by an example. If during this process it is decided that the event needs to be relayed to the Operator, then the Communication Manager can package metadata of the event into a predetermined message format and forward it to the Operator through the Event Channel.

Example 7. Event communication (Anticipating Example 12).
Taking as an example a stream reasoner placed on a roadside unit and monitoring traffic, we observe the following behavior. A rule in the stream reasoner triggers, pushing the event *trafficJam("U1","Karsplatz")* where *U1* represents a road and "*Karlsplatz*" the intersection where the traffic jam occurred. The Communication Manager receives this event and consults the interface to determine the appropriate communication behavior.

```
Predicate:
    name: trafficJam
    ...
    Communication:
      default:
        mode: push
        buffered: false
        delay: 0
```

Listing 3.1: excerpt of the interface defined in Section 3.5.2

The Communication Manager sees that the message should be pushed to the Operator without a delay. What if we need to define some exceptions to allow different types of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

communication within the same predicate? Our interface allows for fine tuning through boolean conditions. Exceptions allow us to change the communication behavior based on the parameters of the predicate (in this case *U1* and *Karlsplatz*). The syntax for these exceptions and the continuation to this example can be found in the Interface Description Section 3.5.2.

Process information handler

Taking an example based on LARS, a rule designed to accumulate information could look like this: Let us assume that there is a stream that at irregular intervals transmits information on the room temperature in the format $@_T\ temp(roomID, temperature)$, where T is the timestamp of the measurement. Then, the following rule would capture the average temperature of the measurements received during the last 2 minutes in its rule head $@_T\ avg\_temp\_in\_room(roomID, avgTemperature)$:

$$\begin{aligned} @_T\, avg\_temp\_in\_room(X, (A+B+C)/3) \quad \leftarrow \quad & @_T\, temp(X, A), \\ & @_{T-1}\, temp(X, B), \\ & @_{T-2}\, temp(X, C), \\ & avg\_temp\_in\_room\_request(X) \end{aligned} \tag{3.1}$$

This type of accumulated process information might both be useful to obtain in a pull based fashion through operator requests and be relevant enough to be pushed in periodical time intervals. To allow for different modes of communication and accommodate eventual buffering of data paired with delays, different settings can be made for each predicate in the *Data* section of the interface description. For example for pull requests, a flag at the rule body can be added, in this case $avg\_temp\_in\_room\_request(X)$ will only become true when the appropriate request has been received from the Operator module.

Delays in communication handling

As later described in Section 3.5.2, it is possible to define a delay for the messages of a specific predicate. Since "delay" is an ambiguous term we will now analyze the two main ways on how a delay can be implemented. The first is through a buffer & burst approach, while the second places a delay between messages of the same type:

- **buffer & burst**: This more specific type of delay will find use cases mostly depending on the environment instead of the specific predicate. It might be useful in cases where keeping a communication line open permanently is not possible, or where some degree of interception / recognition resistance is desired, as it is often the case in military use cases [Oet80]. The implementation of such buffers can be achieved by keeping track of a timer for each predicate using this type of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

delay. While the timer is ticking down, all incoming data regarding that predicate is stored in a buffer and once the timer runs out for that specific predicate, the data is transmitted all at once. The buffer & burst setting in our interface can be set in the following way:

```
...
  Communication:
    default:
      mode: push
      buffered: true
      delay: 10
```

The Communication Manager consults the interface and in this case creates a 10-seconds timer and holds all incoming messages until that timer is over. When the timer is over, all accumulated messages are sent and the timer restarts for the next burst.

- **Same message delay** Intuitively, if at a certain time point the body of a rule is satisfied, there is a good chance that the body will also be satisfied during the next iteration if the knowledge base does not change considerably. This would lead to a message being fired during each iteration. For example if the time interval between one iteration and the next is one second and the conditions making the body of a rule true do not change for a minute, it would result in 60 event type messages being sent to the Operator. While redundant, these messages would also slow down the Operator if we consider that the same could be true for dozens of other predicates. By implementing same message delay, we can make sure that the same type of message will be discarded by the Communication Manager if too little time has passed from the last message of the same type that has been transmitted.

3.3 The Communication Channels

In the previous section we mentioned the usage of communication channels between the stream reasoner, interface and Operator to deliver information. The type of communication protocol can be altered depending on the use case, ranging from UDP-, TCP- or a Websocket-based implementation. The split into 4 channels as suggested by the DynaCon [EDTF+19] architecture allows for a clear separation of the information exchange between the different modules.

- **Event Channel**: By its nature the event channel is used for sporadic messages whose occurrence is unpredictable. Events are previously defined occurrences of (unexpected) changes in the streamed data that the stream reasoner is able to detect. When a new event is detected, the Communication Manager is responsible for transferring it. To decide on the mode of transportation the Communication Manager checks the *Data* section of the interface in order to decide whether a message should be transmitted and which additional options are applied.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- **Process Information Channel**: Unlike the Event Channel where communication is mostly push based, process information can both be requested ad hoc as well as be set up for regular transmission. Such information will mostly be used to check conditions responsible for triggering eventual reconfigurations. Separating communication into an event and a process information channel also allows to use different communication protocols such as HTTP/REST or Websockets.

- **Information Request Channel**: If more information is required by the Operator it can be queried through this channel. The message format allows us to filter data according to some specifications, allowing to retrieve data meant to support the constant stream provided by the process information channel. Additionally, in the case where the event channel collapses, this channel can be used as a backup solution making the whole architecture more robust.

- **Command Channel**: Through this channel the Operator can issue changes to the rule set by adding, removing, activating, or deactivating specific rules. Moreover, the channel can be used to change entries in the interface description. This includes parameter values, but also modes of communication for predicates. While the other channels communicate directly with the stream reasoner or the Communication Manager, these commands are intercepted by the Update Manager. The Update Manager then executes the commands and stores eventual persistent commands.

3.3.1 Message Format

Depending on whether we are communicating by the occurrence of an event through the Event Channel or exchanging data through one of the others, different message formats apply depending on the channel:

- To communicate the occurrence of events the Communication Manager uses the tuple $m_e =< e, a, t, l, d, p >$ where:
  - e is the *EventType*, specifying the unique name or id of the event
  - a is the *Source*, specifying the source stream reasoner
  - t are the *Targets*, specifying the recipients of the message
  - l is the *Location*, specifying the location where the event occurred
  - d is the *Time*, specifying the time and date
  - p is a tuple with additional parameter names and values

  Example 8. Take the predicate *trafficJam(X)*, where X is the lane with a traffic jam. Additionally, the event occurs on January 31, 2020 at 14:30. The context is the same as seen before, we are modeling a traffic situation where roadside units (RSU) with sensors control the traffic. Then the message would have the following format:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

$m_{\textbf{\textit{trafficJam}}}$ = <*trafficJam*, *RSU#3*, *Operator*, *lane#231*, *310120201430*, ->

Where the time is simply encoded in the DD.MM.YYYY HH:MM format.

- For process information the tuple is defined as $m_p =< e, a, t, l, d, v >$ where v simply holds the value for the process information and e, a, t, l, d are defined as above.

  Example 9. The Operator requests details about the number of cars in lane 231 observed by RSU number 3, in response the following message is sent.

  $m_{\textbf{\textit{carInLane}}}$ = <*carInLane*, *RSU#3*, *Operator*, *lane#231*, *310120201430*, 10>

- To request information the message format for process information can be reused in the form $m_p =< e, a, t, l, d >$ as a filter, where e, a, t, l are as above and d again defines the time. Each parameter of the tuple can then act like it was part of a WHERE clause. By defining a parameter we look for process information matching that value, instead by leaving an asterisks ('*') all data is retrieved regardless of the value for that parameter.

  Example 10. If RSU number 3 produces information about the 4 lanes around the intersection (lane nr. 231, 232, 233 and 234) concerning the amount of cars in the lanes, a request could look as the following:

  $m_{\textbf{\textit{carInLane}}}$ = <*carInLane*, *RSU#3*, *, *, 10m>

  This query would be answered with all the *carInLane* messages that were generated in *RSU#3* for all the 4 lanes, regardless of the target and the location, in the last 10 minutes.

- Finally the format of messages required to issue commands depends on the implementation of the Stream Reasoner. Examples can be seen in Section 3.4.1 regarding Update Commands.

3.4 Update Manager

The Update Manager is the module responsible for managing the stream reasoner and the interface depending on update commands sent by the Operator. The Update Manager is essential since the stream reasoner alone "only" evaluates a given logic program. It cannot decide on its own, whether some rules need changing and it cannot update the program while the evaluation is running. The Update Manager can thus change the behavior of a running stream reasoner by changing the underlying program. Specifically, this happens through the commands that can be sent by the Operator through the command channel targeted at a specific stream reasoner instance. Then, depending on the command type, the Update Manager changes the interface or the logic program itself. We consider the following types of update commands for a program P

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- Interface:
  - Modify parameters, i.e., constants to program P
  - Modify communication settings for predicates, i.e., push or pull and delays
- Stream reasoner:
  - Add rules (including facts) to P
  - Delete rules from P
  - Activate and deactivate rules from P through comments

We also distinguish update commands by their temporal scopes. Some rules might be added permanently (inertial commands): this means a rule will remain in the logic program until it is removed by a future command. Instead, non-inertial commands like the ones specified through the **event** keyword in LUPS or with the **for** keyword in TLUPS, will automatically be retracted by the Update Manager after a given amount of time. This distinction is important on the update manager level, but on the operator level we distinguish two further type of commands: persistent and non persistent. Non persistent commands are evaluated and executed once by the Operator. For example the modification of an interface component or the assertion of a fact. Persistent commands stay in the Operator for future iterations too and will be evaluated each time. This allows us to state update commands with conditions, without knowing when the condition will become true. Examples for persistent commands can be seen in Section 3.4.2.

3.4.1 Update Commands

Update commands are a central part that allows for dynamic reconfiguration of the interface and modification of the rule set of a (logic) program in the stream reasoner. Through the usage of a command language similar to LUPS [APPP02], policies can be stated in the Operator, which can trigger update commands. These update commands can be used to change the parameters located in the interface or to change the mode of communication for specific predicates.

All update commands in this section are in the form in which they can be found as TLUPS policies in the Operator. Once a policy is triggered on the Operator, the command is sent to the Update Manager. Undesired information like the condition (`when ...`) are removed and the flags `interfaceCommand` or `srCommand` will be added depending on whether the `execute` or the `assert/remove/enable/disable` keywords, respectively are part of the policy.

Example 11. Take a rule written in LARS to be as the following:

$$room\_hot(X) \quad \leftarrow \quad \boxplus^{k} \Diamond room\_temperature(X, Y), Y > 30 \tag{3.2}$$

Where $room\_temperature(X, Y)$ states that the room X that was measured a temperature Y and $k = 60$ is a parameter specifying the time units, say in minutes. These

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

parameters are defined in the *Parameter* section of the interface description (elaborated in Section 3.5). Now assume that $room\_hot(X)$ triggers air conditioning in room X. The problem with $k = 60$, is that this rule will imply that the room is hot up until 60 minutes after the measurement that exceeded 30 degrees, even though the air conditioning might have lowered the temperature enough already. An update command can be used to change this parameter to 10:

```
execute setParameter(k, 10)
```
(3.3)

At this point we will preview a new function made available with the TLUPS keyword **for**, namely the timed addition of rules. This function is useful when we want to assert a rule for a given amount of time and then immediately retract it. The policy

```
assert for 60 rule <rule> when <condition>
```

will cause the rule `<rule>` to be asserted for 60 time units, for example seconds, if the condition is matched. The rule will then automatically be retracted after 60 seconds. A function that is not yet implemented in TLUPS but still could be useful is the option to assert a rule in the future. A keyword like **in** could be used in the following way:

```
assert in 30 for 60 rule <rule> when <condition>
```

If the condition is satisfied, the rule would be asserted after exactly 30 seconds and then retracted after a total of 90 seconds.

3.4.2 Persistent Updates

Now suppose the following scenario: A room is being used both as a lecture room and, during summer months, as a research laboratory that requires strict temperature monitoring. For such a laboratory it is not enough to update the state of the room (hot or not) every 10 minutes since temperature deviations of 2 degrees might already be detrimental to the experiments. Thus instead of changing the parameter twice every year through the command seen in (3.3) at the beginning and at the end of the summer period, the following conditional commands can be employed:

```
always execute rule setParameter(k, 1) when room_is_laboratory(X)
always execute rule setParameter(k, 10) when room_is_classroom(X)
```
(3.4)

These commands are so called persistent updates and make use of a fragment of the TLUPS language displayed in Section 4.2. With the **always** keyword, once these commands have gone through the command language interpreter, they will be preserved and checked during all successive iterations. During each iteration, the command language interpreter

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

will go through the commands stored as persistent updates and by using the information received through the event channel, and occasionally by requesting additional information through the information request channel, decide whether the commands should be sent to the Update Manager or not.

Another possibility for update commands is the addition or removal of rules from the (logic) program that is currently active in the stream reasoner. Going back to the previous example of the room that can function both as a classroom and a laboratory, we want the room to be locked at all times and are employing an NFC card reader to regulate access. While it is being used as a classroom, all students are granted access. Instead, while it is being used as a laboratory, it is desirable that only authorized people have access to the room. In this context we look at the following rules:

As a base rule, we always want authorized personnel to have access to the room through Rule (3.5), where $card\_scanned(X, Y)$ simply means that person X attempted to enter room Y through the NFC card reader and $has\_authorization\_for\_room(X, Y)$ is true if person X is authorized to access room Y:

$$open\_Door(Y) \leftarrow card\_scanned(X, Y), has\_authorization\_for\_room(X, Y) \tag{3.5}$$

Now take the rule r_1, which we do not want to be active at all times.

$$r_1 : open\_Door(Y) \leftarrow card\_scanned(X, Y), is\_Student(X) \tag{3.6}$$

This rule can be added through an update command with a precondition, both establishing when the rule should be added and retracted:

```
always assert rule r1 when room_is_classroom(X)
always retract rule r1 when room_is_laboratory(X)
```
(3.7)

The examples displayed above show the usage of update commands. The available options fully depend on the implementation of the command language and the command language interpreter. For instance, instead of having two separate rules for addition and retraction of a rule as seen in (3.7), a new keyword `when ... until ...` could both define the conditions for activation and deactivation. One could also add other parameters to the policies to indicate for how long (duration in time units) the rule should persist or the parameter should be changed. We can observe this functionality in the design of TLUPS in Section 4.2.

3.5 The Interface Description Language

For easy parsing, an XML format will be used to define the interface description. The XML format is well suited due to the numerous XML parsers and programming API's (DOM[1]) for Java, LINQ[2] for .NET, which allow for easy editing of the files. On the first

[1] https://javadoc.scijava.org/Java7/org/w3c/dom/package-summary.html
[2] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

level of tags encountered in the interface description we define Sections. These Sections include *Parameters* and *Data*. We will now look further into each Section to show what keywords can be used to define properties.

For clarity's and brevity's sake, the following snippets of the interface will be shown in YAML. Nonetheless, the actual implementation will be in XML as stated before.

3.5.1 Parameters

A typical entry in the *Parameters* Section would look like this:

```
1 parameters:
2   parameter:
3     id: 0
4     name: window_size_traffic
5     type: int
6     value: 5
7     comment: Sets the window size for rules that
8       detect traffic jams on roads
```

Listing 3.2: Example for a Parameter Section

The parameters defined here in the interface description can have their value changed by appropriate commands as seen in the previous section on update commands. At the same time, the stream reasoner implementation has access to these parameters, which will potentially influence the way it communicates facts or even how certain rules work (for example by changing window sizes). We now give a description for each of the attributes:

- **id** acts as a unique identifier. It allows for unique identification when requested through the Information Request Channel or when it is used by a command.
- **name** acts as a description.
- **type** defines the data type of the attribute *value*. We restrict ourselves to the two basic types of integer and string and the type collection.
- **value** represents the data held by this parameter. In the case of numerical parameters this would be either an integer or a natural number.
- **comment** serves as an annotation to describe what the parameter is used for.

3.5.2 Data

We use the *Data* Section to define what, when and how data is exchanged. We will explain the structure of this section through the Interface Description examples seen in Listing 3.3 and give the following example.

Example 12. Vienna Traffic. Take the map seen in Figure 3.2, which is a subset of the public transport grid in Vienna and will be use as a simplified version of traffic monitoring. In this network we have two main roads labeled with "U1" and "U4". Each have multiple

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

intersections with other streets (not shown on the map) that are marked with white nodes. In this simplified road example, we are concerned with two intersections, namely "Karlsplatz" and "Schwedenplatz". The predicate *trafficSlow(X,Y)* can become true when a significant delay is measured on road X at intersection Y.

Figure 3.2: Subset of Vienna transportation map used as traffic example, lines "U1" and "U4" treated as car lanes and white nodes as intersections

Assuming that traffic slows down enough to trigger a threshold in an intersection Y, say "Nestroyplatz", preceding "Schwedenplatz" on road "U1", triggering the event *trafficSlow("U1","Nestroyplatz")*. The Communication Manager is then responsible for packaging the event with meta data and sending it to the Operator. We now look at the interface shown in Listing 3.3 to see how the communication behavior is defined:

- **Stream Reasoner Default Behavior:** Checks the data section and looks for a predicate with the appropriate name and number of arguments. In this example it is the name *trafficSlow* and the number of arguments 2. If none is found, the default behavior defined in lines 2-5 will be used. For this example, no message regarding the occurrence of this event would be sent, since the default behavior for this stream reasoner is defined as pull only.

Now assume that the slow traffic leads to a traffic jam with cars having to wait in the middle of the intersection. This would trigger the event *trafficJam("U1","Nestroyplatz")* at the "Nestroyplatz" intersection. Again the Communication Manager tries to communicate the event as follows:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- **Predicate Default Behavior:** Again the data section is checked and the predicate with name *trafficJam* is found with the correct number of arguments, occupying line 6-26. The section for the exceptions in lines 18-26 is checked by iterating through all of them to find a matching one, in this case there is only 1 exception. Exceptions are defined in the same format used for SQL `WHERE` query conditions. To decide whether this exception matches, we have to check the conditions for the arguments, which in this case are:

  – The SQL syntax "`in`" checks whether a term is included in the comma separated collection. In this case, the first argument should either be "U1" **OR** "U4" (line 22)

  – **AND** the second argument should either be "Karlsplatz" **OR** "Schwedenplatz" (lines 23)

  The second condition does not apply because "Nestroyplatz" does not fulfil the conditions in line 22, thus we check whether a default behavior for this predicate is defined. This might be different from the default behavior of the stream reasoner seen before. We see that the default behavior for the predicate *trafficJam* defined in lines 15-18, is to push event data respecting a buffer of 10 time units.

Soon afterwards, the traffic jam also starts affecting one of the main intersections "Schwedenplatz" on the road "U1". This triggers the predicate *trafficJam("U1", "Schwedenplatz")* in the stream reasoner. We observe the following behavior:

- **Predicate Exception**: Again we check for an exception for the matching predicate and we see that this time the restrictions for both arguments match. As a consequence, the communication settings for this exception found in lines 24-26 trigger and the message is pushed without a buffer.

```
1  Data:
2    sr_default:
3      mode: pull
4      buffered: false
5      delay: 0
6    Predicate:
7      id: 0
8      name: trafficJam
9      number_of_arguments: 2
10     type: string
11     comment: "Takes as first Argument the road
12       identifier and as second Argument the
13       intersection on the road"
14     Communication:
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
15        default:
16          mode: push
17          buffered: true
18          delay: 10
19        Exceptions:
20          Exception:
21            id: 0
22            condition: arg1 in ("U1,U4") AND
23              arg2 in ("Karlsplatz, Schwedenplatz")
24            mode: push
25            buffered: false
26            delay: 0
```

Listing 3.3: Data Section for the Vienna Metro Example

Let us give a more detailed description of the communication keywords:

- **mode:** When *mode* is defined as **pull**, no automatic message transmission from the Stream Reasoner will occur. The status of the predicate will only be sent when requested through the Information Request Channel. Instead, the communication mode **push** can make use of the two other attributes. In essence, when a new fact or predicate is evaluated by the Stream Reasoner, its value will be transmitted to the Operator. How this is done depends on the next two attributes.
- **buffered:** If the newly evaluated facts and predicates are to be sent directly to the Operator without delay, then the *buffered* attribute would hold the value **false**. Instead, if information should be collected and only submitted in regular time intervals the attribute would hold value **true**. The last attribute defines when or how often buffered data has to be transmitted over the communication channel.
- **delay:** The time settings for the buffered delay communication. The type of delay can vary based on the implementation as laid out in Section 3.2.

Adding the Parameters and the Data part together results in an XML file incorporating the sections as named above:

```
1 InterfaceDescription:
2   Parameters: "..."
3   Data: "..."
```

3.5.3 Formal description in EBNF

To formalize the Interface Descriptions in XML, we present the following context free grammar using the Extended Backus-Naur Form (EBNF)[3]. The words written in bold

[3] http://matt.might.net/articles/grammars-bnf-ebnf/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

font are called tokens. Recall that these tokens are further expanded in the remaining parts of the grammar until all tokens that appear on the left side are expanded on the right side. The tokens placed between curly brackets can be repeated zero or more times while the text between quotation marks is to be matched verbatim.

Top Production rules:

int_desc ::= "<InterfaceDescription>" , **param_section** , **data_section**,
"</InterfaceDescription>";

param_section ::= "<Parameters>" , {**parameter**} , "</Parameters>";

data_section ::= "<Data>" , **sr_default** , {**predicate**} , "</Data>"; (g1)

As seen before, following the first production rule we can add as many **parameter** and **predicate** tokens as desired. These are the basic building steps for the interface description. As we further analyze the command language to edit the interface we will dive deeper into the construction of parameters and predicates. The complete EBNF grammar is presented in Appendix A.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

Command Languages

In this chapter, we will show both command languages that we designed. In Section 4.1 the commands are aimed at changing the interface and will be displayed in the format in which they are received by the Update Manager. Interface commands will also include a function that is later elaborated on in Section 4.2, namely the timed addition of rules. Through a duration parameter `<t>` also interface commands can be set to only execute their changes for a certain duration of time. After the duration `t` expires, the interface is reverted back to its state before the command was executed. Since we will have more examples later that deal with timed addition, all commands of this section will be viewed as permanent and thus carry the value -1 for parameter `<t>`.

In Section 4.2 we will introduce stream reasoner commands in connection with TLUPS, our modified version of LUPS [APPP02]. TLUPS allows to define policies that are evaluated in the Operator. Once a policy is triggered, the interface or stream reasoner command is extracted and sent to the Update Manager for execution.

4.1 Interface command language

First, we want to clearly define what role the TLUPS policies play with interface commands. TLUPS policies, as will be elaborated on in Section 4.2, are *only* used in the Operator. The policies wrap both interface and stream reasoner commands in order to give a framework to set conditions and timing options.

An interface command wrapped by a TLUPS policy in the Operator has the following form, where sr_0 identifies the targeted stream reasoner or interface:

sr_0 : `always execute` `for <t>` `rule` `editParamValue(<id>,<value>)`

(`always execute`: TLUPS policy; `for <t>`: Interface command; `rule`: TLUPS policy; `editParamValue(<id>,<value>)`: Interface command)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

$$\underbrace{\texttt{when <condition>;}}_{\text{TLUPS policy}}{}^{1}$$

For now we will ignore the meaning of TLUPS policies as they will be explained in the next section. The important part is that when the condition part of the TLUPS policy is satisfied and the command needs to be sent to the Update Manager, the TLUPS policy parts are removed and the keyword "`interfaceCommand`" is added creating the type of interface command that we will be analyzing in this section:

```
interfaceCommand(editParameterValue, <t>, <id>, <value>);
```

We also add the duration `<t>` which defines for how long the command should be active and get rid of the sr_0 identifier, because once the command has been sent through the command channel to the responsible update manager, the target has already been chosen. We now define a series of commands that allow us to make changes to the interface. It is clear that due to the size of the interface description (specifically the data part containing default mode, exceptions and multiple conditions) a lot of different commands could be created in order to satisfy specific use cases. To keep the command language concise, we will focus on the most important features.

Add a new "Parameter" entry

- `interfaceCommand(addParameter, <t>, <id>, <name>, <type>, <value>, <comment>);`

Parameters have small entries in the interface description and can thus be fully added through the usage of one single command. The EBNF grammar for a parameter is as follows:

parameter ::= "<Parameter><id>" , **integer** , "</id><name>" ,
parameter_name , "</name>" , **value_type_pair** , "<comment>" , **text** ,
"</comment></Parameter>"; (g2)

To differentiate between parameter names and predicates names, parameter names are all lower case and spaces are to be substituted by the '_' symbol. The **value_type_pair** token can have three different formats highlighted by the following grammar:

value_type_pair ::= "<type>string</type><value>" , **word** , "</value>"
|"<type>int</type><value>" , **integer** , "</value>"
|"<type>collection</type><value>" , **word_array** , "</value>"; (g3)

The **word_array** token represents a list of comma separated values. The definition for other basic tokens like **word**, **integer**, **text** etc. is given in Appendix A.

[1] In this example, the word `Parameter` has been substituted with `Param` for visualization purposes

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Example 13. For instance, assume that we want to add a parameter of type *collection*. The collection type can be used inside the interface description itself. In Example 12 one of the conditions in the communication exceptions was "*in ("Karlsplatz,Schwedenplatz")*". This can be substituted by using the placeholder **#1** where **1** is the id of the collection parameter defined in the interface. The placeholders allow for reusability of parameters within the interface description keeping it more consistent and easier to change.

```
interfaceCommand(addParameter, -1, 1, main_stops,
  collection, {Karlsplatz,Schwedenplatz}, Main stops on
  U1 metro line);
```

The resulting entry would then look like this:

```
1 Parameters:
2   Parameter:
3    id: 1
4    name: main_stops
5    type: collection
6    value: Karlsplatz,Schwedenplatz
7    comment: Main stops on U1 metro line
```

Listing 4.1: Result of using `addParameter` command

The condition *in("Karlsplatz,Schwedenplatz")* can then be rewritten as *in("#1")* and reused throughout the interface.

Add a new "Predicate" entry

– `interfaceCommand(addPredicateBase, <t>, <id>, <name>, <numberOfArguments>, <type>, <comment>, <mode>, <buffered>, <delay>);`

The construction of a predicate entry is more intricate than the one of a parameter. We thus split the commands to insert a predicate and its exceptions into multiple commands. By doing so, we can keep the base command to add a predicate without exceptions simple, but still allowing the addition of multiple exceptions later through additional commands. Depending on the use case, one could also consider different encoding methods in order to fit predicate and exceptions into one single command. The EBNF grammar for a predicate looks the following:

predicate ::= "<Predicate><id>" , **natural_number** , "</id><name>" , **predicate_name** , "</name><number_of_arguments>" , **natural_number**, "</number_of_arguments><type>" , **type** , "</type><comment>" , **text** , "</comment>" , **communication** , "</Predicate>"; (g4)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Example 14. We want to reproduce the predicate entry for *trafficJam* in the interface used previously in Example 12. We thus apply the `addPredicateBase` command in the following way:

```
interfaceCommand(addPredicateBase, -1, 0, trafficJam, 2,
  string, "Takes as first Argument...", push, true, 10);
```

As expected, the command creates an entry in the interface similar to the `addParameter` command. The default communication for the predicate is set using the last three arguments `mode, buffered, delay`. The resulting interface Data entry can be seen in Listing 4.2.

```
1  Data:
2    Predicate:
3     id: 0
4     name: trafficJam
5     number_of_arguments: 2
6     comment: "Takes as first Argument..."
7     Communication:
8       default:
9         mode: push
10        buffered: true
11        delay: 10
```

Listing 4.2: Result of using `addPredicate` command

Once the base predicate has been created, we can now add exceptions to fine tune the communication behavior.

```
– interfaceCommand(addException, <t>, <predicate_id>, <id>,
  <condition>, <mode>, <buffered>, <delay>);
```

In this command we define for what predicate we want to add the exception and state the unique identifier for this exception. Following is an SQL style condition that will define when this specific exception is triggered. Lastly, we add three arguments that describe the communication details for the case in which this exception applies.

Example 15. cont'd
We created the base predicate and now want to add the exception. If we wanted to achieve the exception defined in Example 12 for the *trafficJam* predicate, we would use the following command:

```
interfaceCommand(addException, -1, 0, 0, arg1 in ("U1,U4")
  AND arg2 in ("#1"), push, false, 0);
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Note that in the condition for the second argument, we make use of the parameter that we defined previously. This command will add an `Exception` entry to the `Communication` section of the *trafficJam* predicate. The result can be seen in Listing 4.3

```
Data:
  Predicate:
   id: 0
   ...
   Communication:
     ...
     Exceptions:
       Exception:
        id: 0
        condition: arg1 in ("U1, U4") AND
          arg2 in ("#1")
        mode: push
        buffered: false
        delay: 0
```

Listing 4.3: Result of using `addException` command to predicate with id 0. Predicate definition omitted since it is equal as in Listing 4.2

Editing parameters and predicates

Once again the necessity of editing commands strongly depends on the use case. The approach here will be minimalistic, we will thus take into consideration the cases in which the value of a parameter needs to be changed and the case in which the communication details for a predicate need to be changed (not for specific exceptions). The two commands are as follows:

- `interfaceCommand(editParameterValue, <t>, <id>, <value>);`
- `interfaceCommand(editPredicateCommunication, <t>, <id>, <mode>, <buffered>, <delay>);`

Example 16. If we wanted to change the value for the parameter `main_stops` that we added before we would use the command

```
interfaceCommand(editParameterValue, -1, 0, "Karlsplatz,
  Rathaus");
```

The Update Manager finds the parameter with id 0 in the interface and substitutes the old value "*Karlsplatz,Schwedenplatz*" with "*Karlsplatz,Rathaus*".

Example 17. We previously added the predicate *trafficJam* with default communication settings "`push, true, 10`". If we now decide that the default communication settings should be to send the message without any delay we would use the command

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
interfaceCommand(editPredicateCommunication, -1, 0, push,
  false, 10);
```

The delay value of "10" does not need to be set to 0 since the Communication Manager will ignore it after seeing that buffered is set to false. If greater changes need to be made to a specific parameter or predicate, it can always be deleted and re-added with the new information.

Deletion of interface components

In general the most likely use cases will simply involve deleting a parameter or a predicate. In some cases one might also wish to delete a certain exception from a predicate. The deletion commands will thus be limited to the following three commands:

- `interfaceCommand(deleteParameter, <t>, <id>);`
- `interfaceCommand(deletePredicate, <t>, <id>);`
- `interfaceCommand(deleteException, <t>, <pred_id>, <exc_id>);`

While deletion commands for parameters and predicates are straightforward (only require stating the duration and id of the parameter or predicate entry), the deletion of an exception needs two identifiers.

Example 18. In our previous example we defined an exception for the *trafficJam* predicate. If we wanted to change both the condition and communication mode attached to this exception, it might be easier to delete it and re-add it with the `addException` command. To uniquely identify the exception, we state both the predicate identifier and the exception identifier in the deletion command:

```
interfaceCommand(deleteException, -1, 0, 0);
```

4.2 TLUPS as a policy language and stream reasoner commands

As seen in the previous section, TLUPS can be used in the Operator to define conditions or timing options regulating update commands. To motivate the creation of TLUPS and show its syntax and semantics, we will use update commands aimed at a stream reasoner.

Just like with interface commands, stream reasoner commands are stripped of their TLUPS components before being sent to the Update Manager. A stream reasoner command wrapped by a TLUPS policy in the Operator has the following form, where sr_0 identifies the targeted stream reasoner:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

sr_0 : **always** (TLUPS policy) **assert** pc_1 (Sr command) **for** t (both) **rule** (TLUPS policy) a ← b (Sr command) **when** c (TLUPS policy) ;

When the condition part of the TLUPS policy is satisfied and the command needs to be sent to the Update Manager, the TLUPS policy parts are removed and the keyword "`srCommand`" is attached. Following is the final format for the message sent to the Update Manager:

```
srCommand(assert, t, a ← b);
```

We also get rid of the sr_0 identifier since once the command has been sent through the command channel to the responsible update manager the target has already been chosen. The time parameter `t` is also added to the stream reasoner command.

Assuming a cursory knowledge of the contents discussed in the *Updating rule sets of logic programs* Section of Chapter 2, we will now elaborate on the problems that we identified with LUPS*.

4.2.1 Motivation

Take the following persistent commands:

always assert a ← b **when** c (4.1)

always assert a ← b **when** d (4.2)

always assert event a ← b **when** e (4.3)

Following the semantic transformation defined in LUPS*, we would end up with the following set of persistent commands:

$$PC = \{\textbf{assert}\ \text{a} \leftarrow \text{b}\ \textbf{when}\ \text{c},\ \textbf{assert}\ \text{a} \leftarrow \text{b}\ \textbf{when}\ \text{d},\ \textbf{assert event}\ \text{a} \leftarrow \text{b}\ \textbf{when}\ \text{e}\} \quad (4.4)$$

Now if we want to delete the translation of the persistent command 4.1 from PC, we have to use the **cancel assert** statement:

cancel assert a ← b **when** ... (4.5)

We now observe two problems:

- (i) Since the **cancel** command cannot specify that it wants to delete the assertion rule with condition c, it will also remove the translation of the command 4.2.
- (ii) There is no way to specify whether we want to delete an inertial assertion (like the one seen in 4.1 and 4.2), or a non inertial one using the **event** keyword as used in 4.3. This would again cause the removal of both commands.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

We also note a further point of improvement that can be made through the following example.

Example 19. To avoid speeding at intersections or pedestrian crossings, a technology not unlike the one introduced in the UK[2] is implemented, where red lights are automatically triggered for a fixed period of time if an approaching vehicle exceeds the speed limit. Modeling this with the set of commands available in LUPS* can prove to be tricky. There is no way to specify that a rule should remain in place for a fixed period of time. Rules can either be asserted and retracted when a condition is matched or they can be asserted only for the next state through the **event** keyword.
In the first case, the event triggering the red light would be the detection of a speeding vehicle. However, the only condition for the traffic light to turn back to green is the passing of time. Using the **event** keyword in order to hold the red light for a specified amount of time, one would need to connect it to a condition that remains true over the whole time interval. However, the event of speeding that would trigger such a rule, is no longer true as soon as the vehicle slows down for the red light. We will thus define the third problem as:

- (iii) There is no way to assert a rule for a given amount of time, without being dependent on some event being active at the same time.

4.2.2 Our own solution: TLUPS - Timed LUPS

The root of the problem is that the persistent update commands are identified by the rule's syntax, which in some use cases might not be unique. To solve both problems (i) and (ii) we introduce an unique identifier for each persistent update command. We thus extend the command seen in (4.1) with an identifier and a new **rule** keyword to maintain a clear structure in the command syntax:

$$\textbf{always assert } pc_1 \textbf{ rule } \text{a} \leftarrow \text{b } \textbf{when } \text{c} \tag{4.6}$$

This identifier is then carried over to the semantic transformation and stored in the set of persistent commands PC as **assert** pc_1 **rule** a $\leftarrow$ b **when** c. The command **cancel assert/retract** can then optionally be extended with the identifier of the persistent command that one wishes to cancel. If left unspecified, the cancel statement will maintain its previous behavior of deleting all persistent assert or retract commands for the specified rule. Following is the syntax for the **cancel assert** command with an unique identifier:

$$\textbf{cancel assert } pc_1 \textbf{ rule } \text{a} \leftarrow \text{b } \textbf{when } \ldots \tag{4.7}$$

While the addition of a persistent command identifier solves problems (i) and (ii), we still do not have a solution to assert rules for a fixed period of time as described in problem (iii).

[2] https://www.itsinternational.com/its8/its2/feature/traffic-signals-turn-red-stop-speeding-drivers

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Ideally, we want a mechanism that can trigger a rule and then retract it after a given amount of time. This concept is not new in the LUPS* language, in fact the **event** keyword is nothing else but a modifier stating that the rule should be asserted or retracted for exactly one transition. We can thus use this concept to introduce the new keyword **for**. In our example, we would write:

always assert pc_1 **for** 60 **rule** trafficlight(TLid, red) ← **when** speeding(X) (4.8)

The command sets the traffic light with id TLid to red for 60 seconds if someone is detected speeding on road X. Using the **for** modifier with the value 1 would exactly mimic the previous behavior of the **event** keyword. At this point we briefly recall the stream reasoner command format discussed at the beginning of Section 4.2. We saw that the TLUPS components are stripped before the command is sent to the Update Manager. However, the **for** keyword is an exception. It is the Update Manager's responsibility to retract the rule once the time defined with the **for** keyword has passed. We thus need to add the timing information to the stream reasoner command:

```
srCommand(assert, 60, trafficlight(TLid, red).);
```

We summarize the original LUPS commands, the improvements done in LUPS* and our TLUPS version in Table 4.1.

LUPS	*LUPS**	*TLUPS*
assert [**event**]	**assert** [**event**]	**assert** [**for**]
retract [**event**]	**retract** [**event**]	**retract** [**for**]
always [**event**]	**always assert** [**event**]	**always assert** *rID* [**for**]
non existing	**always retract** [**event**]	**always retract** *rID* [**for**]
cancel	**cancel assert**	**cancel assert** [*rID*]
non existing	**cancel retract**	**cancel retract** [*rID*]

Table 4.1: LUPS evolution

TLUPS: the **for** keyword substitutes the **event** keyword in LUPS*

4.2.3 Adapted semantics - translation to dynamic logic program

We adapt the semantics specified in LUPS* [Lei01] by keeping the same **Base step**, **Inductive step** and $NU_t = U_t \cup PC_t$ definition. We instead change the definitions for PC_t and P_t. The logic program resulting from the translation differs from the one in LUPS* only in the substitution of the Ev predicate by the Ti predicate (for assertions) and Re predicate (for retractions) and behaves in the same way after this modified translation procedure.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

$PC_t = PC_{t-1} \cup$
$\cup$ {**assert** pc_θ **rule** R **when** ϕ: **always assert** pc_θ **rule** R **when** $\phi \in U_t$}$\cup$
$\cup$ {**retract** pc_θ **rule** R **when** ϕ: **always retract** pc_θ **rule** R **when** $\phi \in U_t$}$\cup$
$\cup$ {**assert** pc_θ **for** s **rule** R **when** ϕ: **always assert** pc_θ **for** s **rule** R **when** $\phi \in U_t$}$\cup$
$\cup$ {**retract** pc_θ **for** s **rule** R **when** ϕ: **always retract** pc_θ **for** s **rule** R **when** $\phi \in U_t$}$-$
$-$ {**assert** pc_θ [**for** s] **rule** R **when** ϕ: **cancel assert** pc_θ **rule** R **when** $\psi \in U_t \wedge \bigoplus \mathcal{P}_{t-1} \vDash \psi$}$-$
$-$ {**assert** pc_θ [**for** s] **rule** R **when** ϕ: **cancel assert rule** R **when** $\psi \in U_t \wedge \bigoplus \mathcal{P}_{t-1} \vDash \psi$}$-$
$-$ {**retract** pc_θ [**for** s] **rule** R **when** ϕ: **cancel retract** pc_θ **rule** R **when** $\psi \in U_t \wedge \bigoplus \mathcal{P}_{t-1} \vDash \psi$}$-$
$-$ {**retract** pc_θ [**for** s] **rule** R **when** ϕ: **cancel retract rule** R **when** $\psi \in U_t \wedge \bigoplus \mathcal{P}_{t-1} \vDash \psi$}

For the **cancel assert** statement that specifies a command identifier pc_{id}, all assert statements with a matching pc_{id} are removed from the PC set, regardless of whether they have a **for** modifier. The second **cancel assert** statement instead does not specify a pc_{id} and will thus remove all assert commands with a matching rule R, regardless of command identifiers pc_{id} or **for** modifiers. The two **cancel retract** statements mirror this behavior.

$P_t =$
$= \{N(R) \leftarrow ;\ H(R) \leftarrow B(R),\ N(R)$:
assert pc_θ **rule** R **when** $\phi \in NU_t \wedge \bigoplus \mathcal{P}_{t-1} \vDash \phi\} \cup$

$\cup$ {*not* $N(R) \leftarrow$; *not* $Ti(R,\ t,\ X,\ Y) \leftarrow$:
retract pc_θ **rule** R **when** $\phi \in NU_t \wedge \bigoplus \mathcal{P}_{t-1} \vDash \phi\} \cup$

$\cup$ {$Ti(R,\ t,\ 1,\ s) \leftarrow ;\ H(R) \leftarrow B(R),\ Ti(R,\ T,\ X,\ Y)$:
assert pc_θ **for** s **rule** R **when** $\phi \in NU_t \wedge \bigoplus \mathcal{P}_{t-1} \vDash \phi\} \cup$

$\cup$ {$Re(R,\ t,\ 1,\ s) \leftarrow$; *not* $N(R) \leftarrow Re(R,\ T,\ X,\ Y)$; *not* $Ti(R,\ T',\ X',\ Y') \leftarrow Re(R,\ T,\ X,\ Y)$:
retract pc_θ **for** s **rule** R **when** $\phi \in NU_t \wedge \bigoplus \mathcal{P}_{t-1} \vDash \phi\} \cup$

$\cup$ {$Ti(R,\ t,\ X{+}1,\ Y) \leftarrow$: $\bigoplus \mathcal{P}_{t-1} \vDash Ti(R,\ t\text{-}1,\ X,\ Y),\ X{<}Y\} \cup$
$\cup$ {$Re(R,\ t,\ X{+}1,\ Y) \leftarrow$: $\bigoplus \mathcal{P}_{t-1} \vDash Re(R,\ t\text{-}1,\ X,\ Y),\ X{<}Y\} \cup$
$\cup$ {*not* $Ti(R,\ t\text{-}1,\ X,\ Y) \leftarrow$: $\bigoplus \mathcal{P}_{t-1} \vDash Ti(R,\ t\text{-}1,\ X,\ Y)\} \cup$
$\cup$ {*not* $Re(R,\ t\text{-}1,\ X,\ Y) \leftarrow$: $\bigoplus \mathcal{P}_{t-1} \vDash Re(R,\ t\text{-}1,\ X,\ Y)\}$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Why do timed commands work?

As seen in the LUPS* semantics, we extend normal asserts with the predicate $N(R)$, but instead extend our definition of timed asserts and retracts with the *Ti(R, T, X, Y)* and *Re(R, T, X, Y)* predicates respectively, where:

- R : The rule R for which the timing predicate is meant.
- T : Always carries the value of the current time point t. This value is necessary because during each iteration we can disable the Ti and Re predicates from the previous round.
- X : Is a counter resembling how long the rule has been asserted or removed for. This counter always starts at value 1 and is increased after each iteration as long as it is smaller than Y.
- Y : Resembles the total amount of time for which the rule should be asserted or removed. It is instantiated with value s carried by the **for** parameter and remains unchanged throughout.

In the following example we show that a rule added through a timed command, will remain active only as long as the **for** parameter specifies. Once the specified time elapses the rule is effectively disabled and cannot be derived any longer in the dynamic logic program unless it is asserted again in the future.

Example 20. Assume we have the following command in U_1 starting off with $P_0 = \{\}$ and $PC_0 = \{\}$:

$$U_1 = \{\textbf{assert } pc_1 \textbf{ for } 2 \quad \textbf{rule } \text{a} \leftarrow \text{b}\}$$

By the semantics of P_t, the command in U_1 would cause the following addition to P_1 at $t = 1$:

$$P_1 = \{\, Ti(a \leftarrow b, 1, 1, 2) \leftarrow ; \quad a \leftarrow b,\ Ti(a \leftarrow b, T, X, Y)\}$$

With these two additions the predicate $Ti(a \leftarrow b, T, X, Y)$ holds at P_1 since it can be instantiated to $T = 1$, $X = 1$, $Y = 2$. During the next iteration assume we get no additional update commands and thus $U_2 = \{\}$. Since

$$\bigoplus \mathcal{P}_{t-1} = \mathcal{P}_1 \vDash Ti(a \leftarrow b, 1, 1, 2) \ \textit{and}\ 1 < 2$$

we have to perform 2 operations to P_2, we add the fact $Ti(a \leftarrow b, 2, 2, 2)$ since $X < Y$ and we add $not\, Ti(a \leftarrow b, 1, 1, 2)$. We then end up with:

$$\begin{aligned}\bigoplus \mathcal{P}_2 = \{\ & Ti(a \leftarrow b, 1, 1, 2) \leftarrow ; \\ & a \leftarrow b,\ Ti(a \leftarrow b, T, X, Y); \\ & Ti(a \leftarrow b, 2, 2, 2); \\ & not\, Ti(a \leftarrow b, 1, 1, 2) \leftarrow ;\}\end{aligned}$$

We can see a conflict for the first and the last fact. This conflict is resolved by the semantics of dynamic logic programming, which places the first fact in the Rejection Set

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

since it has been added in P_1, whereas the last fast was added in P_2. We also observe that the rule $a \leftarrow b$ still holds, since the Ti predicate in the body can be instantiated to $T = 2$, $X = 2$, $Y = 2$.

We now iterate through one more step to observe how the rule will stop being enabled. Again we assume $U_3 = \{\}$. This time we have to only add 1 fact since $X \not< Y$ and

$$\bigoplus \mathcal{P}_{t-1} = \mathcal{P}_2 \models Ti(a \leftarrow b, 2, 2, 2)$$

we thus only add $not\, Ti(a \leftarrow b, 2, 2, 2)$ and get the following result (clashing clauses already removed):

$$\begin{aligned} \bigoplus \mathcal{P}_3 = \{\ & a \leftarrow b,\ Ti(a \leftarrow b, T, X, Y); \\ & not\, Ti(a \leftarrow b, 2, 2, 2) \leftarrow ; \\ & not\, Ti(a \leftarrow b, 1, 1, 2) \leftarrow ;\} \end{aligned}$$

We now see that there are no positive Ti atoms left to satisfy the atom in the rule body for the first rule. We observe that regardless of the value of b, the rule body can now not be satisfied anymore and the conclusion a also cannot be derived. The rule is now effectively disabled.

We now proved how a rule is added and removed from the dynamic logic program exactly as we would expect from the semantics of the **for** command. Due to the semantic intricacies of the translation to a dynamic program, we show a further example including persistent commands and timed **for** commands.

Example 21. We now present an example to show how to apply the semantics. Semicolon will be used as a separator between rules and the comma is used as a conjunction in the body of rules.
Let $P_0 = \{\}$ and $PC_0 = \{\}$. With the commands in U_1 we then get:

$$\begin{aligned} U_1 &= \{\textbf{assert rule } a \leftarrow b;\ \textbf{always assert } pc_1 \textbf{ rule } b \leftarrow \quad \textbf{when } c\} \\ PC_1 &= \{\textbf{assert } pc_1 \textbf{ rule } b \leftarrow \quad \textbf{when } c\} \\ NU_1 &= \{\textbf{assert rule } a \leftarrow b;\ \textbf{always assert } pc_1 \textbf{ rule } b \leftarrow \quad \textbf{when } c; \\ &\qquad \textbf{assert } pc_1 \textbf{ rule } b \leftarrow \quad \textbf{when } c\} \\ P_1 &= \{N(a \leftarrow b) \leftarrow \quad ; \quad a \leftarrow b, N(a \leftarrow b)\} \end{aligned}$$

At this state the model is $M = \{\}$ since no knowledge can be derived as we do not have b to satisfy the condition $b, N(a \leftarrow b)$.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

$$
\begin{aligned}
U_2 &= \{\textbf{assert for } 1 \textbf{ rule } c \leftarrow \quad\} \\
PC_2 &= \{\textbf{assert } pc_1 \textbf{ rule } b \leftarrow \quad \textbf{when } c\} \\
NU_2 &= \{\textbf{assert for } 1 \textbf{ rule } c \leftarrow \quad ; \quad \textbf{assert } pc_1 \textbf{ rule } b \leftarrow \quad \textbf{when } c\} \\
P_2 &= \{\, Ti(c \leftarrow \quad, 2, 1, 1) \leftarrow \quad ; \quad c \leftarrow Ti(c \leftarrow \quad, T, X, Y)\}
\end{aligned}
$$

Now the Model is $M = \{c\}$ since we can instantiate $T = 2$, $X = 1$, $Y = 1$. Now one could wrongly think that the rule **assert** pc_1 **rule** $b \leftarrow$ **when** c should trigger and thus granting us b too in the model, but the condition c needs to be satisfied in P_{t-1}.

$$
\begin{aligned}
U_3 &= \{\} \\
PC_3 &= \{\textbf{assert } pc_1 \textbf{ rule } b \leftarrow \quad \textbf{when } c\} \\
NU_3 &= \{\textbf{assert } pc_1 \textbf{ rule } b \leftarrow \quad \textbf{when } c\} \\
P_3 &= \{N(b \leftarrow \quad) \leftarrow \quad ; \; b \leftarrow N(b \leftarrow \quad); \mathit{not}\ Ti(c \leftarrow \quad, 2, 1, 1) \leftarrow \quad ;\}
\end{aligned}
$$

Finally the model is $M = \{b, a\}$ since the previously mentioned rule triggers granting us b in the model and consequently also satisfying $a \leftarrow b, N(a \leftarrow b)$ from P_1. We also observe that c no longer holds since we assert the negation of the literal $Ti(c \leftarrow \quad, 2, 1, 1)$ without adding a new Ti literal.

There are some further improvements that could be done, specifically regarding new use cases and diverse requirements. These will be further discussed in the Future Work Section 7.1.1.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 5

Dynamic Configuration System Prototype

We aim to provide a basic implementation of the architecture introduced in Figure 3.1. The prototype is developed in Java 13[1]. In Figure 5.1, we demonstrate on how the single components communicate using the communication channels described in Section 3.3.

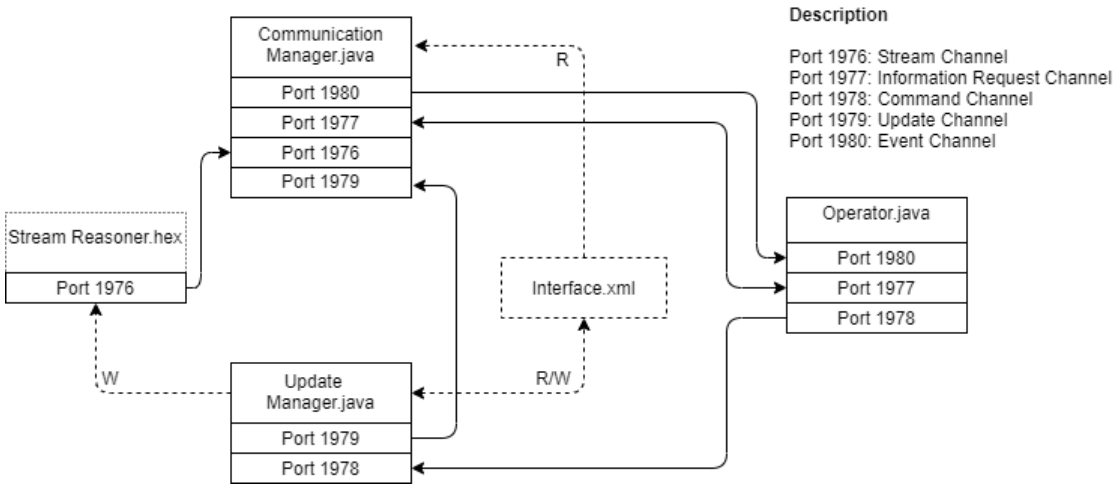


Figure 5.1: Connection structure for Java implementation

The prototype is implemented in Java 13 due to portability reasons. Portability is required because parts of the architecture will be run on a Linux operating system, since the HexLite stream reasoner, used in the prototype, is designed to run on a Linux system. For implementing the communication between the stream reasoner and

[1]https://openjdk.java.net/projects/jdk/13/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

our modules, we decided to use web sockets and the Jetty[2] library. For the internal communication between Update Manager, Communication Manager, and Operator, we use a basic Java Socket implementation (more details are given in Appendix B). With this implementation, a client only needs to know the IP address and the port of the server in order to connect. The web socket technology also works cross-platform allowing our HexLite Stream Reasoner to communicate with the java modules. The development of the Communication Manager is simple, but does require the implementation of efficient message handling techniques. Another challenge arises with the handling of pull-based reasoner, since they have different ways on how they can be called/activated. Extensions to deal with pull-based stream reasoners are discussed in Chapter 7.1.2.

The implementation of the Update Manager is more challenging, since it either needs to have direct access to the KB (including the rules) of the stream reasoner, or to have a communication channel to the stream reasoner to send update commands. Another challenge arises from different rule language "dialects". For instance, LARS rules might not be understood by the HexLite reasoner, therefore the Update Manager would need to convert the commands to a specific rule language. In the heart of the Operator lies the TLUPS decision maker module that currently covers a reduced version of the TLUPS language. We leave the following features for future work:

- Persistent: All TLUPS policies remain in the operator until removed manually.
- Conditions: Only event-based conditions are checked.

5.1 Overview

Before we cover the implementation of each part we give an overview of how we implemented the configuration cycle seen in the Introduction Chapter in Figure 1.1. We set the following scenario:

- All modules are running and connected to each other.
- The Communication Manager holds a cached copy of the Interface.
- The interface contains a parameter with id = **3**, which is currently set to 180 and is used by the stream reasoner to set the length in seconds of the green phase for traffic lights on road **l34**.
- The interface contains a predicate `trafficJam`, the communication behavior is defined as push without delay.
- In the Operator, we have 2 TLUPS policies that are as follows:
  - `execute for -1 rule (setParameter, 3, 300) when trafficJam(l34)`

[2] https://www.eclipse.org/jetty/about.html

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

– `disable for -1 rule <rule> when trafficJam(l34)`

The first policy targets the interface and the second targets the stream reasoner. Both are triggered when the message `trafficJam(l34)` is received. `<rule>` is a placeholder and is located in the stream reasoner's knowledge base. For instance, if the goal is to dissolve the traffic jam, the rule could be one that recommends other traffic participants to take road **l34**. In that case, we want to disable it when there is a traffic jam.

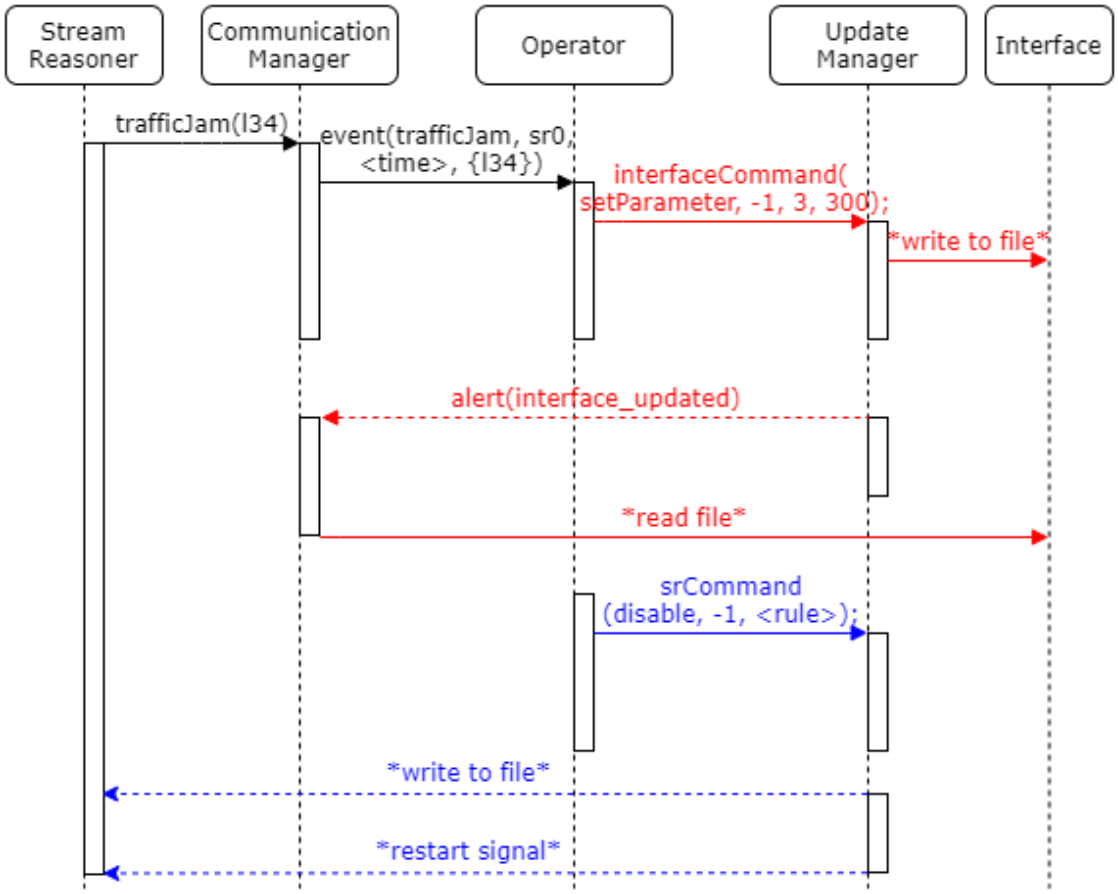


Figure 5.2: UML diagram representing configuration flow

In Figure 5.2, we show the configuration flow as an UML sequence diagram of the above scenario. The red part (starting from `interfaceCommand` until `*read file*` is triggered by the first (interface) policy while the blue part (everything including and after `srCommand`) is triggered by the second (stream reasoner) policy. In detail, the steps are as follows:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- Based on the rules in the KB that are evaluated by the stream reasoner, the event `trafficJam(l34)` is detected and sent exactly in a predefined format to the Communication Manager.
- The Communication Manager receives the message and checks the local copy of the interface to determine the communication behavior, which is push without delay. Next, the meta data is added: *sr0* is the source stream reasoner, *<time>* is a placeholder for the timestamp relating to the moment where the Communication Manager received the message and *{l34}* is the set of arguments for the event. The message is sent to the Operator.
- The Operator receives the message at which point both policies are triggered. As a consequence, the Operator creates two update commands and sends them to the Update Manager:

  1. `interfaceCommand(setParameter, -1, 3, 300);`
  2. `srCommand(disable, -1, <rule>);`

- Without loss of generality, we assume the interface command is executed first with the following steps:

  1. The changes of parameter with id=**3** are written to the interface file
  2. An alert is sent to the Communication Manager
  3. The Communication Manager loads the new version of the interface
  4. Stream Reasoner is updated with the new value
  5. A restart signal is sent.

- Following is the execution of the stream reasoner command:

  1. Stream Reasoner is updated with disabled rule
  2. A second restart signal is ignored (see Section 5.2.1)

- Update Manager sends restart signal to the Stream Reasoner
- After receiving the signal, the Stream Reasoner reloads the file containing the new settings.

5.2 Stream Reasoner

As a stream reasoner, we decided to use the HexLite solver that was already introduced in Section 2.2.2. Our choice was not driven by performance, in which case the Ticker [BEF17] engine would present a better choice. HexLite is well suited, since for our implementation a highly customizable implementation was preferred, allowing us to add new functionalities through plugins that can be written in Python[3] and then accessed from the stream reasoner through external atoms.

[3] https://www.python.org/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Example 22. For instance, connecting to the web socket server and sending messages can be done using the external atoms `@wsConnect` and `@wsSend`, respectively as follows:

```
1 server_adr("ws://localhost:8082").
2 @wsConnect(X) :- not &wsConnected[X], server_adr(X).
3 @wsSend("info(VehicleSpeed,",V0,",",V1,")") :-
4    trafficCount(V0,V1), V0 > 0.
```

The server address is stored in the fact `server_adr`. In line 2 we use the `@wsConnect` command to connect to the server, if a connection has not already been established. With the external command `@wsSend` we send messages from the stream reasoner through web sockets.

The stream reasoner fetches sensor data from PipelineDB[4] (a PostgreSQL extension that can run SQL queries on streams of data while storing the results in tables) through dynamic SQL commands with fixed window sizes. We recall that window functions allow to restrict the window of observation to avoid retrieving obsolete data. Through our generic approach, we are able to exchange the stream reasoner as long as the substitute provides the same communication methods.

5.2.1 Updates and restarts

For dealing with updates and restarts we decided to use **template** files for the stream reasoner. A template file has the same contents as the actual stream reasoner file representing an active program, with the exception that the template file has the parameter identifiers in the form of `!param!`, while the original file has the actual values for those parameters. We now describe how and when the original program file is generated from the template.

When the Update Manager starts, it opens the template and creates the original version of the stream reasoner by substituting all parameter values retrieved from the interface. The stream reasoner is then started by the Update Manager in order to obtain the process id, denoted as `pid`. In addition, when receiving commands targeting any of the rules (i.e., `assert`, `retract`, `enable`, `disable`) or any of the parameters (i.e., `addParameter`, `editParameterValue`, `deleteParameter`), the Update Manager will restart the stream reasoner with the following procedure.

1. In case of parameter commands, apply changes to the interface.
2. Commands targeting rules are performed directly on the template. This means that a rule will be disabled in the template file directly.
3. The Update Manager creates the original file by making a copy of the template and inserting the parameter values. By doing so, any rules that were added, removed, disabled or enabled in the template will remain so in the original file as well.

[4] https://www.pipelinedb.com/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

4. Once the original file has been created from the template, the Update Manager send the signal "`kill -SIGHUP pid`" to the process id of the stream reasoner to trigger its reload.

5. The stream reasoner receives the signal and reloads the newly created original file.

Since the reload happens after each of the commands, we decided to add a small buffer in the case that two such commands arrive simultaneously. Before restarting the stream reasoner, the Update Manager waits for 100 ms in which he ignores all further restart requests. By doing so, two commands that arrive simultaneously (maybe triggered by the same condition of a TLUPS policy) asserting two different rules will only cause one restart. Such an example can be seen in Section 6.2.3.
The 100 ms time interval is hard coded, but could easily be adjusted according to the use case in the following ways:

- The time interval could be dynamic and increase with each rule received in the time interval. For example, the Update Manager executes one rule and starts waiting before the restart. A new restart request arrives before the 100 ms elapse and is ignored. At this point, we could add another 100 ms to the timer to give more time for additional rules that might have been fired simultaneously.
- The time interval could be defined in the interface as a stream reasoner setting. When the Update Manager loads the interface it could simply read the value and use it as a restart timer. Having the value in the interface also means that it can be changed by the Operator through interface commands.

5.3 Communication Manager

When the Communication Manager is started, it reads the current interface description for the stream reasoner it is responsible for. This will be used as a reference point once the stream reasoner starts sending data. The communication settings contained in the interface description are loaded only at the start and whenever the Update Manager communicates that it performed changes to the interface. An alternative would be to check the interface each time a message is received by the Communication Manager, but that would quickly result in a performance bottleneck since multiple messages can arrive in very small time frames.

5.3.1 Data processing

In our prototype we decided to implement push and pull communication as well as the buffering of push-based messages. Immediate push-based communication and pull requests are handled as intended. If a predicate has the communication settings `push` with buffering set to false, then the message is packaged with a timestamp and the meta information from the sending stream reasoner. Similarly, if a predicate has pull

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

communication enabled, the messages are stored in the Communication Manager until a request for that predicate is sent by the Operator. Furthermore, we enter the more intricate type of communication, which is push-based with burst type communication.

Example 23. For instance, we have the following communication settings:

```
communication:
  mode: push
  buffered: true
  delay: 5
```

For this type of buffered data, a list of active predicates is maintained. Every data item received from the stream reasoner is added to the respective predicate entry in the list of active predicates. If there has not been any previous communication, we can send the message immediately. When we do, a timer starts with the number of seconds specified in the `delay` node. Until the timer expires, all messages received for the specific predicate are placed in the aforementioned list without being sent. Once the time delay expires, in this case 5 seconds, all accumulated messages for this predicate are sent.

```
Timer timer = new Timer();
timer.schedule(new SendMessagesTask(PredicateId,
    detectedEvent.ExceptionId),0, 5000);
```

Listing 5.1: Messages are scheduled to be sent every 5000ms for a specific predicate id and exception id

As shown in Listing 5.1, we also use an exception identifier, since inside of a predicate we can define exceptions for the communication behavior. Examples of such exceptions are given in Section 3.5.2. In order to allow different timings within a predicate there is one more distinction that happens after the first distinction based on the predicate identifier. The second distinction is based on the exception identifier. For a hypothetical predicate with identifier 0 we could have the following active communications:

- exception id 0 (urgent): push, 0 delay
- exception id 1 (important): push, 3 delay
- exception id 2 (warning): push 10 delay
- exception id -1 (default): pull

Note that each exception for each predicate spawns a thread of its own in order to upkeep responsiveness in our multi-threaded implementation. A negative aspect of this approach is that we end up with dozens of active threads that try to send messages regularly, even if there are no messages to be sent. We thus optimized the system by applying a check every time the timer expires. If the timer expires once and there are no messages to be sent, the thread is killed. The thread will only be restarted once a new message for the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

corresponding predicate and exception is received. Implementing this method allows us to limit the number of active threads to the bare minimum by also respecting the delay defined for each exception. This is possible because a thread is only killed once the delay has elapsed one time without sending messages.

Example 24. (Push 10s delay) Let us look at an example for a hypothetical exception id 2 with 10-second delays from the point of view of the Communication Manager.

0: message **received** from SR <event_message>
0: message **sent** to Operator <packaged_event>
1: message **received** from SR <event_message>
10: message **sent** to Operator <packaged_event>
20: no messages to send, **killing** thread
22: message **received** from SR <event_message>
22: message **sent** to Operator <packaged_event>

We can see that when the message arrives at time point 1 it is not sent until 10 seconds passed from the previous time we have sent a message. Then, no new messages are received until time point 20, at which we kill the thread responsible for this exception. At time point 22 we receive a new message and we are free to start the thread anew and immediately send the message.

Another approach could involve distinguishing the threads responsible for sending messages by their start time and delay instead of by their predicate and exception identifier. By doing so, messages could be managed by the same thread if two predicates have the same timing regarding their delays and when the message transmissions started off.

Example 25. The Communication Manager receives two messages from the Stream Reasoner at the time point `t`. In the interface, the same communication behavior is defined for both: push based with 5 seconds delay. Instead of creating two separate threads to deal with the communication, they could be merged into one thread responsible for sending messages that coincide with starting point `t` (or any increment of `t` by 5 seconds) and a 5-second delay.

5.3.2 Determining exceptions

Specific exceptions are defined in the interface and are thus unknown to the stream reasoner. This means that when an event message reaches the Communication Manager,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

there is no exception or predicate identifier attached to it. To determine these identifiers, which will later define how the message is communicated, we use Algorithm 5.1.

Algorithm 5.1: Determining predicate and exception Id

Data: predicateName, [predicateArgs], interfacePredicates

Result: (predicateId, exceptionId)

```
1  foreach p ∈ interfacePredicates do
2      if p.name==predicateName then
3          if p.numberOfArgs == predicateArgs.count then
4              predicateId ← p.id;
5              foreach e ∈ p.exceptions do
6                  if eval(e.condition, [predicateArgs]) then
7                      exceptionId ← e.id;
8                      return (predicateId, exceptionId);
9              exceptionId ← -1;
10             return (predicateId, exceptionId);
11 predicateId ← -1;
12 exceptionId ← -1;
13 return (predicateId, exceptionId);
```

When we get a message from the stream reasoner, the information we have are the predicate name and the arguments. The Communication Manager also has a local copy of the interface, and thus knows all the predicates and exceptions stored therein. In line 1 we start iterating through all the predicates stored in the interface, checking whether the names match (line 2) and the number of arguments match (line 3). When a match is found, the predicate id is assigned and we iterate through the exceptions of that predicate. The function *eval* in line 6 takes the SQL condition for the exception e and checks whether it is satisfied with the given predicate arguments. If *eval* returns true, we also found the exception and can return from execution. If no exception matches, we return -1 for the `exceptionId`, which is synonymous with the default communication mode for the predicate. Lastly, we have the case in which the predicate could not be found in the interface file. In this case we return -1 to both identifiers, which will cause the default stream reasoner communication mode to trigger.

Once both identifiers are known, the Communication Manager can easily retrieve the communication mode for the exception, the default mode for the predicate or the most general for the stream reasoner.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

5.3.3 Information requests

Currently the Communication Manager can receive information requests from the Operator over port 1977. In our prototype it is currently limited to requesting messages of a specific predicate with the option to narrow down the results to an exception. These messages can be sent at any time and the Communication Manager will answer immediately by sending all messages corresponding to the request through this communication channel and not through the event channel. Optionally more parameters could be added for the Operator in order to restrict the delivered results to location or time intervals.

5.4 Operator

When the Operator starts, it launches the threads that listen for incoming connections from Communication and Update Managers. Once an incoming connection is established, a separate thread is launched to deal with all incoming messages. Incoming connections are stored in the *active connections* list, where the respective Communication and Update Manager are mapped to their stream reasoner. Each Update Manager and Communication Manager pair is responsible for one stream reasoner and consequently also one interface. The connections to the Communication Manager are twofold: one of the connections represents the event channel on port 1980 and the other the information request channel on port 1977. If the Operator receives an incoming connection request, an object in the *active connections* list is created as follows:

- `srID`: The stream reasoner identifier uniquely identifies the stream reasoner that will be managed by this *active connection* object.
- `cmThread`: The Communication Manager thread is responsible for information requests. When required, the Operator can request information by sending a message through port 1977 and will then obtain the response on this communication channel. By doing so, it is independent from the `eventThread` and the event channel on port 1980. Given the correct configuration, this setup theoretically allows to handle a situation where communication from the event thread dies down by obtaining all necessary information through the `cmThread`.
- `eventThread`: The event thread is also used for simplex communication from the Communication Manager to the Operator and is used for event related messages that are sent according to `push` communication settings, with or without delay.
- `umThread`: The Update Manager thread allows for simplex communication. It functions as the command channel where the Operator can send commands directed to change the interface description or the stream reasoner.
- `tlupsRules`: This is a set of TLUPS rules that defines what commands will be fired based on what conditions. In the following section we will give more details on how the rules are evaluated.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

5.4.1 The TLUPS Decision Maker

We implemented a slightly different and reduced version of the conceptualized version in Section 4.2. We implemented TLUPS policies with two type of commands, interface and stream reasoner commands. They are distinguished mainly by the first keyword, where `execute` is reserved for interface commands and `assert`, `remove`, `enable` and `disable` are reserved for stream reasoner commands. A TLUPS policy with an interface command looks as follows:

```
execute for <t> rule <command> when <condition>;
```

A TLUPS policy with commands aimed at the stream reasoner looks as follows:

```
[assert|remove|enable|disable] for <t> rule <rule> when
  <condition>;
```

If we want to enforce permanent changes, we can either set `<t>` to -1 or omit the `for <t>` part completely.

In our current implementation we omitted the following functionalities of the TLUPS language as described in Section 4.2:

- Persistent & non persistent: all TLUPS policies remain in the Operator until removed by a user (thus treated as persistent) and will be checked each time a new event message arrives. For this reason, the when condition is required, otherwise the policy would trigger each iteration. If a command does not require a condition it can be entered manually.

- Conditions: only simple conditions that require fixed equality assignments are supported. For example a condition could be **when** `TrafficJam(U5, Karlsplatz)`, which becomes true if exactly that message is sent by the respective stream reasoner and Communication Manager.

As shown in Algorithm 5.2, whenever an event message arrives at the Operator, all TLUPS policies aimed at that stream reasoner are checked. To do this, the information regarding event name and arguments are extracted from the message sent by the Communication Manager (line 1-3) and then compared to the policy conditions (line 5). If any of them are matching, the command specified in the policy is extracted and transformed into the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

update command format (line 6) before being sent to the Update Manager.

Algorithm 5.2: Checking if TLUPS rules trigger

Data: eventMessage, [tlupsPolicies]

Result: [commands]

```
1 eventName ← extractName(eventMessage);
2 eventArgs ← extractArgs(eventMessage);
3 eventCompare ← eventName + "(" + eventArgs + ")";
4 foreach policy ∈ tlupsPolicies do
5     if eval(policy.condition, eventCompare) then
6         commands.add(transform(policy));
7 return commands;
```

Example 26. For example, we have the following TLUPS policy:

```
execute for -1 rule editPredicateCommunication(4, push,
  false, 0) when <condition>
```

its transformation to the update command format is:

```
interfaceCommand (editPredicateCommunication, -1, 4, push,
  false, 0).
```

We will later see that in our implementation the comma is substituted by a ^ symbol, which facilitates parsing.

5.5 The Update Manager

The Update Manager receives all commands for a certain stream reasoner instance that are sent by the Operator. It deals both with commands aiming to change the interface description and the stream reasoner's KB. As a result, the Update Manager loads an instance of the interface in a Java class structure, making use of the `org.wc3.dom`[5] package, which provides useful methods for XML processing. The commands sent by the Operator using the format described in Section 4.1 are then received by the Update Manager and elaborated with the usage of regular expressions and string manipulation.

5.5.1 Interface commands

We now look at an example targeting the interface configuration.

[5] https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/package-summary.html

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Example 27. We assume the following inertial (second parameter is -1) interface command (`cm`), adding an exception with id 2 to predicate 0 with condition `[(3, [value1, value2]), (4, [value3, value4])]` and 10-second burst delayed communication that is sent by the Operator as:

```
interfaceCommand(addException, -1, 0, 2, [(3, [value1,
  value2]), (4, [value3, value4])], push, true, 10)
```

Note: For the conditions in exceptions, we use a simplified version in our implementation. Instead of an SQL condition as suggested before, we simply check if an argument is included in a set of values. For example, the above condition `[(3, [value1, value2]), (4, [value3, value4])]` checks whether argument 3 is equal to either `value1` or `value2` and argument 4 is either equal to `value3` or `value4`.

The command `cm` would then be dissected in the following way:

```
1 String command = cm.substring(0, cm.indexOf("("));
```

A substring command extracts the "`interfaceCommand`" string and splits the execution to a thread responsible for changing the interface. To extract all parameters one option would be to split the command message after every comma, but this does not work for all commands as seen in this example since some commas are included inside the argument array `[(3, [value1, value2]), (4, [value3, value4])]` defining the conditions for the exception.

Consequently, we attempted to extract all the arguments by using the following regular expression:

```
1  List<String> arguments = new ArrayList<String>();
2  cm=cm.substring(cm.indexOf("(")+1, cm.indexOf(")"));
3  Matcher m = Pattern.
4      compile("([\\w+|\\s*]|\\[.+\\])+(,?)").matcher(cm);
5  while (m.find()) {
6      String raw = m.group();
7      arguments.add((raw.endsWith(",") ?
8          raw.substring(0, raw.length() - 1) : raw).trim());
9  }
10 String[] ar =  arguments.toArray(new String[0]);
```

Firstly, we extract only the contents of the interface command by using the substring function in line 2. Following is the regular expression that is designed to match words, or the whole content inside of a squared bracket, followed by a comma. This approach would not work when trying to send commands aimed at disabling rules spanning over multiple lines. Hence, we simply substitute the commas with "^" symbols. By doing so, the command can be transformed into an array by simply splitting after each occurrence of the "^" symbol. The command becomes

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
interfaceCommand(addException^-1^0^2^[(3, [value1,
  value2]),(4, [value3, value4])]^push^true^10)
```

The specific interface command is then contained in the first slot of the string array and the corresponding method can be started through a simple `switch` statement by passing the corresponding parameters. The information on how long the command should be active for is in the seconds parameter, in this case -1.

For example in the following code listing, we check if the command is equal to "addParameter" and if it is we pass the parameters needed for the method signature `AddParameter(duration, id, name, type, value, comment)`, where `IE` is the Interface Editor object:

```
switch (cm) {
  case "addParameter":
    IE.AddParameter(Integer.parseInt(ar[1]),
      Integer.parseInt(ar[2]), ar[3], ar[4], ar[5],
      ar[6]);
    break;
  ...
}
```

Once a change has been made to the interface description, it is saved to the XML file and a message is sent over port 1979 to the Communication Manager, warning it that some changes to the interface have been performed. More details on the implementation can be found in the Communication Manager Section.

5.5.2 Stream reasoner commands

Similarly to an interface command, the input string is split to determine which command should be executed. Additions and removals of rules are performed by exactly matching the syntax of the submitted rule. When considering the option of activating and deactivating rules we have constructed a test that is shown in Section 6.2.1. The test demonstrates on how to deactivate a predicate, which will result into deactivating any rules that contain it. As described in Section 5.2.1, the Update Manager performs the changes to the template, creates the original copy for the stream reasoner and then sends a reload signal.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

CHAPTER 6

Use Cases and Functionality Showcase

In this chapter we will use the previously describes prototype to showcase some of its functionalities in a use case environment. As the main use case, we will continue with the traffic related use case that was briefly mentioned in section 2.1.1.

The Siemens C-ITS use case

The C-ITS use case as described in [ESS19] and [EDTF+19] is as follows:

> Cooperative Intelligent Transportation Systems (C-ITS) are the setting for this use case. The edge devices are roadside units (RSU) that observe V2X communication messages of traffic participants (i.e., cars and traffic lights). A traffic control centre that is responsible for managing the overall traffic and the different RSUs acts as the configuration unit. The initial goal is the detecting of undesired traffic events, i.e., accidents or traffic jams, and the second goal is finding a reconfiguration of traffic light signal plans that ameliorates the effects of the undesired event. Concretely, we aim to improve vehicle throughput compared to an unmodified system for undesired traffic events. [ESS19]
>
> In Figure 6.1, we show an example for a C-ITS instantiation of our general architecture. The producers (P) are in this setting either vehicles that communicate via V2X messages to the surrounding or traffic light (TL) installations that manage their assigned intersection. Each intersection has one edge device, the RSU, which is connected to the traffic lights installations and receives the V2X messages from the vehicles. The RSUs acts as monitor (M) and actuator (A) by analyzing the V2X messages streams, detecting events such as accidents, and updating the (new) signal plans to the connected TL installation. Resulting, the stream reasoner would be located on the RSU, and sending (detected) events and (local) process

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

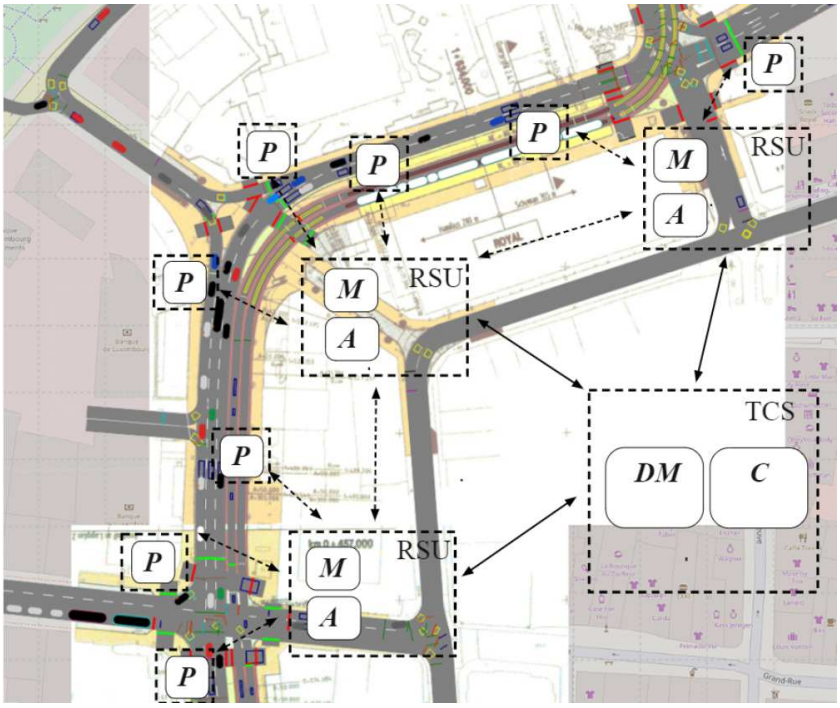


Figure 6.1: C-ITS Architecture Example [ESS19]

information to the traffic control centre (TCS), which is our cloud server and has the tasks of re-configuring the TL installations using the decision module (DM) to trigger the re-configuration. Note that the configurator (C) should be on the TCS, since only there all the events and local information is available. Since, we have several RSUs deployed, and vehicles have their own capabilities of monitoring and changing the surrounding, we naturally have also multi-agent environment.[ESS19]

Relating this back to our work, we can draw some parallels. As the use case states, in an RSU we have the stream reasoner that filters data. In our implementation, the RSU would also contain the Communication Manager and the Update Manager. The interface file is also located on the RSU. Furthermore the role of the traffic control center (TCS) is taken by the Operator in our implementation.

Note that the tests can be replicated by following the instructions outlined in Appendix C.

6.1 Experiment 1: Change in communication behavior

In this example we will perform two tests that showcase the ability of changing the communication behavior of information derived in the stream reasoner without restarting it. This is done through changes to the interface that are then adapted by the Communication Manager.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

6.1.1 Test 1: Change existing communication behavior

In this test, we will change the default communication behavior of predicate `carInLane` through a TLUPS policy that triggers when the message `trafficJam(l34)` is received. No stream reasoner restarts are required.

Setting:

A deployed RSU on a given intersection monitors the number of cars in the lanes, and reports it every second. The initial setting in our interface states that this type of information should be delivered once every 10 seconds. This setting would be as follows:

```
Predicate:
  id: 0
  name: carInLane
  number_of_arguments: 2
  type: string
  comment: "First Argument is lane identifier and
    second Argument the number of cars."
  communication:
    default:
      mode: push
      buffered: true
      delay: 10
```

The transmission of this type of data is in 10-second intervals, which is fine during a normal operation. The configurator does not need to get messages regarding the number of cars in a lane more often if there is no situation that requires reconfiguration. Suppose that from one of the RSU's overlooking an intersection we get the message `trafficJam(l34)`, where `l34` is the lane identifier. At this point the configurator could take action by trying to change the traffic light plan to alleviate the traffic jam, or possibly even try to change the traffic flow to be redirected through other roads. Hence it becomes important to obtain more information regarding the number of cars in all of the lanes merging up to the intersection. A 10-second interval is too long and we now want to obtain this information instantly without a delay. A TLUPS policy to trigger such a change could be the following (implemented without the **always** keyword but maintains functionality of inertia):

```
execute rule editPredicateCommunication(0, push, false, 0)
  when trafficJam(l34)
```

Ideally, the operator would then send a message to the Update Manager to change the interface and change the communication behavior. In this example let us assume that for the first 20 seconds, every second the stream reasoner will send data on the number of cars in the lanes seen by a RSU. Then it will send the message `trafficJam(l34)` and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

resume submitting data on the number of cars in the lanes. Let us see how our prototype reacts to such a situation:

Expected outcome:

After a delay of 6 seconds after starting the system, we expect to receive the following outcome:

6 : Communication Manager sends burst to Operator with 1 message
16 : Communication Manager sends burst to Operator with 10 message
26 : Communication Manager sends burst to Operator with 10 message
26 : Communication Manager sends trafficJam(l34) message to Operator
26 : TLUPS policy triggers and Operator sends command to Update Manager
26 : Update Manager changes interface and warns Communication Manager
26 : Communication Manager reloads the interface
27 : Each second, Communication Manager sends `carInLane` message to Operator.

Outcome:

In Figure 6.2, we show the command-line output of the Communication Manager for our test. The first 22 seconds are omitted as they simply repeat in a 10-second cycle, namely 6-16, 17-26 and are shown again at time point 26. As we can see, each second the Communication Manager receive a message from the stream reasoner. The buffered messages are sent for the second time in a 10 messages burst at time point 26. After the burst we observe the `trafficJam(l34)` message and the `carInLane` message for the 26th time point. We then see that at second 26 a message is sent to the Operator for a predicate with the id **1**, which corresponds to the entry for `trafficJam` in the interface. We can then take a look at the output of our Operator to see what happens when the `trafficJam` message arrives. In Figure 6.3, we show that right after the `trafficJam` message is received, we receive a confirmation that the message triggered one of our policies. In the next line a command is sent to the responsible update manager. The Update Manager receives the command, performs the changes specified by setting the `buffered` field to false and then saves the changes to the interface.

Finally, we look back to our Communication Manager in Figure 6.2 and observe the alert that is sent by the Update Manager. As a result the interface is reloaded and we can immediately observe a change in how the messages are communicated. The messages are now relayed instantly. In time point 27, we show that two messages are sent, this is because the `carsInLane` message for time point 26 was received after the Communication Manager started sending the messages for the 26th second. It thus ended up in the buffer and was sent along with the next burst.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

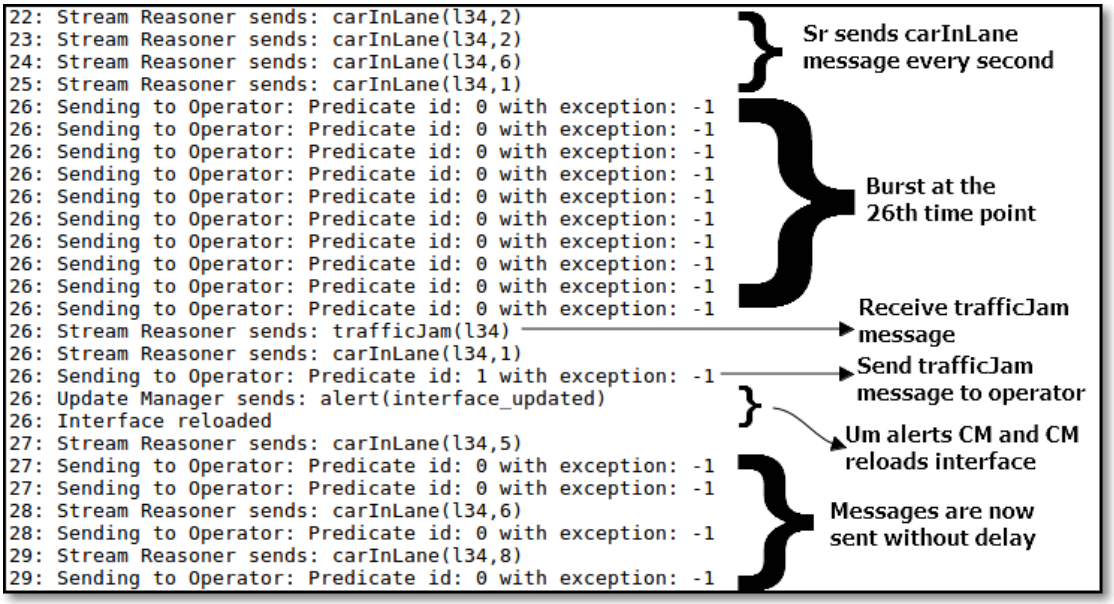


Figure 6.2: Communication Manager for Test 1

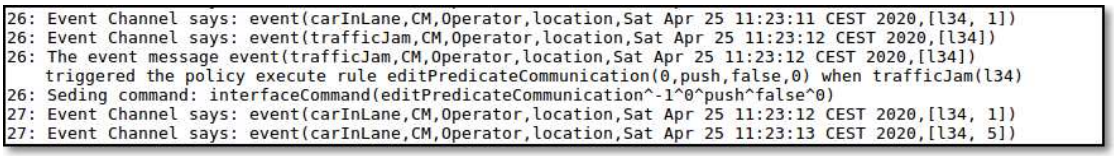


Figure 6.3: Operator for Test 1

Problems:

In a previous implementation on a Windows operating system and with a previous version of this test (hence the different timestamps and capitalization of event names), we ran into an unexpected behavior: We sent multiple predicate messages from the SR to the CM and then to the Operator. Since it is a multithreaded environment and the messages are sent towards the same receiving port simultaneously, this can lead to problems in the code shown in Appendix B. The `DataOutputStream` object uses the `writeBytes` method, which means that messages are received byte by byte, and this can lead to the problem illustrated in the Figures 6.4 and 6.5.

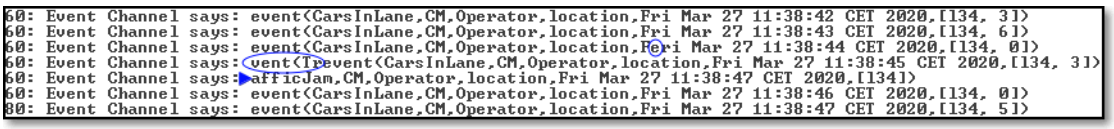


Figure 6.4: Split message instance 1

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

In Figure 6.4, we show how this problem manifests: Each letter is represented by 1 byte and we can see that when the two messages are sent simultaneously by two threads they can arrive at different time points. In this case the `TrafficJam` message arrives contemporarily to two other `CarsInLane` messages. The letter "`e`" appears two messages before and the sub string "`vent (Tr`" appears in the previous message. The actual message is then missing exactly those letter. This problem is not given by the output to the console; the event message is exactly received as that. As a consequence, the TLUPS policies will not recognize the name of the event since it is incomplete. The problem is

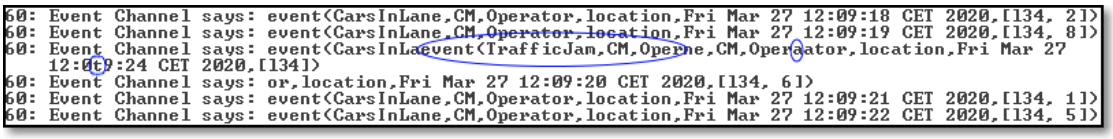


Figure 6.5: Split message

accentuated even more in Figure 6.5 where the `TrafficJam` message is scattered across the previous message. Fortunately, modern programming languages such as Java offer a solution to deal with these kind of problems.

Solution:

The reason for this problems is that messages are sent asynchronously. By doing so, the bytes have a chance of arriving in mixed order at the destination. Ideally, we would like for the message sending part to be synchronous. In this way, a new message would only be sent if no other message was being sent at the same moment. Java allows this through the usage of the `synchronized` keyword, which introduces a Monitor. The Monitor checks which program parts access the synchronized part of code. By definition, "*...a block of code that is marked as synchronized in Java tells the JVM: "only let one thread in here at a time"*"[1]. The improved code looks like this, where `out` is the `DataOutputStream`:

```
1 synchronized(out) {
2     out.writeBytes(message + "\n");
3     out.flush();
4 }
```

The compiler places a lock on the `out` object, appending instructions for the threads on how to obtain and release the lock. When a thread obtains a lock it becomes the owner and any other threads attempting to access the object need to wait until the lock is release by the other thread. By doing so, messages are never sent simultaneously and the problem of scrambled messages is solved.

[1] https://www.javamex.com/tutorials/synchronization_concurrency_synchronized1.shtml

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

6.1.2 Test 2: Add communication exceptions

In this test we take the predicate `carInLane` that has no exceptions defined and where all messages use the default communication of push with delay. We then change the behavior only for certain messages by adding an exception to the interface. Again, no stream reasoner restarts are required.

Setting:

We start from the same setting as in Test 1; we have a predicate `carInLane` that periodically sends information about the number of cars in a lane. In the last test, we changed the communication of the whole predicate from push based with 10-second delays to immediate communication. This change influenced all `carInLane` messages sent by the stream reasoner. Now assume that we are interested in the number of cars in lane `l20` and want to receive immediate messages only for this lane. We can manually input the command:

```
interfaceCommand(addException,-1, 0, 0, [(0,[l20])], push,
  false, 0)
```

In order of appearance the values mean the following:

- `addException`: defines the specific command to be executed
- `-1`: is the timing variable. -1 means that the changes done by the command should be permanent (inertial)
- `0` is the unique id for predicate `carInLane`
- `0` is the new unique identifier for the added exception
- `[(0,[l20])]` is the condition defining when the exception triggers. In this case it triggers if the argument `0` is contained in the array `[l20]`
- `push` is the communication mode for this exception
- `false` defines whether messages should be sent in bursts
- `0` is the number of seconds for the interval between bursts

Note that the used syntax is different from the command used in Test 1. In Test 1 we saw a TLUPS policy for an interface command. In this test we skip the TLUPS policy since the command will be entered manually by the system user, we thus immediately use the format of an interface command since this command will not pass through the Operator.

Expected outcome:

Before the above command is sent, the Operator should receive messages for lanes `l10` and `l20` every 10 seconds in bursts of 10 messages each. After we input the manual command adding the exception at time point 27, we should start receiving the messages for `l20` every second, while the messages for `l10` keep arriving in 10 second bursts.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

7 : All modules are running and connected to eachother

7 : Communication Manager sends burst to Operator with 1 message each for `carInLane(l10)` and `carInLane(l20)`.

17 : Communication Manager sends burst to Operator with 10 messages each for `carInLane(l10)` and `carInLane(l20)`.

27 : Communication Manager sends burst to Operator with 10 messages each for `carInLane(l10)` and `carInLane(l20)`.

28 : Update Manager receives our manual command and updates the interface with the new exception, also warning the Communication Manager

28 : Communication Manager reloads the interface

28-: Each second, Communication Manager sends `carInLane` message for argument `l20`

37 : Communication Manager sends burst to Operator with 10 messages only for `carInLane(l10)`

Outcome:

The Communication Manager sends the first message pair at time point 7, after all modules had loaded up and connected. The Operator receives the two messages at time point 7 and then receives nothing until time point 17, where he receives a burst of mixed messages for both `l10` or `l20`. They are mixed since they have been sent by two separate threads and thus can arrive in random order. As shown in Figure 6.6 at time point 27, the Operator receives the burst of mixed messages, where the text at the end of each message indicates if it was for `l10` or `l20`. At time point 27 we insert our manual command and immediately after that we can see that at time points 28-30 the information regarding `l20` is sent without a delay. The messages for `l10` arrived in 10

```
27: Event Channel says: event(carInLane,CM,Operator,location,Sat Apr 25 12:59:30 CEST 2020,[l20, 9])
27: Event Channel says: event(carInLane,CM,Operator,location,Sat Apr 25 12:59:30 CEST 2020,[l10, 9])
27: Event Channel says: event(carInLane,CM,Operator,location,Sat Apr 25 12:59:31 CEST 2020,[l10, 2])
27: Event Channel says: event(carInLane,CM,Operator,location,Sat Apr 25 12:59:31 CEST 2020,[l20, 2])
interfaceCommand(addException^-1^0^0^[(0,[l20])]^push^false^0)
27: Seding command: interfaceCommand(addException^-1^0^0^[(0,[l20])]^push^false^0)
28: Event Channel says: event(carInLane,CM,Operator,location,Sat Apr 25 12:59:33 CEST 2020,[l20, 4])
29: Event Channel says: event(carInLane,CM,Operator,location,Sat Apr 25 12:59:34 CEST 2020,[l20, 3])
30: Event Channel says: event(carInLane,CM,Operator,location,Sat Apr 25 12:59:35 CEST 2020,[l20, 8])
```

Figure 6.6: Operator with manual message in green ("^" substitutes ",")

message bursts at time point 37 and in addition there was also one more messages for `l20`. This message is the one generated at time point 27. It has been waiting in the buffer since the exception was only added after the Communication Manager had placed that message in the 10 second buffer queue.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

As planned, we now obtain immediate communication for `carInLane` messages from `l20` while the communication for `l10` remains burst based with 10-second intervals. If we check the interface, we indeed see the exception that has been added[2]:

```
Predicate:
  id: 0
  name: carInLane
  number_of_arguments: 2
  type: string
  comment: "First Argument is lane identifier and
    second Argument the number of cars."
  communication:
    default:
      mode: push
      buffered: true
      delay: 10
    Exceptions:
      Exception:
        id: 0
        condition: arg1 in ("l20")
        mode: push
        buffered: false
        delay: 0
```

6.2 Experiment 2: Changes to stream reasoner's KB

In this experiment we will perform three tests, which will each require a restart of the stream reasoner. We will conduct the following tests:

1. Based on two TLUPS policies, we disable and enable a predicate in the stream reasoner's KB.
2. We change a parameter in the stream reasoner's knowledge base (KB) that controls which information is sent to the Communication Manager.
3. We use three TLUPS policies that trigger simultaneously adding three rules to the stream reasoner's KB, which allows us to retrieve new information from a database and send it to the Communication Manager.

6.2.1 Test 1: Disabling and enabling rules

Depending on the information collected by the stream reasoner, it might be desirable or necessary to add or remove certain rules from the KB. If rules are removed, the stream

[2]Here displayed using YAML and the SQL format for the condition. In our implementation the interface is an XML file and the condition is stated for each argument

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

reasoner will have to check less of them and automatically work faster (assuming that he has to evaluate the condition for each one).

Additionally, removing or disabling rules that cause events or process information will automatically reduce the number of messages sent to the Communication Manager and thus to the Operator. As seen before, the Operator controls all TLUPS policies, every time a message is received we also lower the burden on the Operator.

Setting:

Once a year in Vienna (during summer) the Pride Parade is held organised by HOSI Wien[3]. The parade goes through the "Innerer Ring" (the Ring Road) of the city and as such, disrupts the normal flow of traffic since no cars are allowed on the affected streets. Let us assume that there is a setting like the one in the previous example. RSUs are located at selected intersections and observe traffic and relay events of significance to the Communication Manager. Clearly, events regarding traffic jams, the number of cars in a lane or which vehicles stopped are not used during the Pride Parade. We observe three options:

1. **Shut down:** the easiest option would be to turn off all stream reasoners that are located on the parade route.
2. **Remove:** undesired rules can be removed from the stream reasoner's KB.
3. **Disable:** since the rules are stored in a KB represented by a text file, rules can be disabled by commenting them out.

We observe that all three options achieve the goal of lessening the strain on the Stream Reasoner, Communication Manager and Operator.

Option 1 is the easiest to implement, however, there might be some relevant information that can be transmitted despite of the parade. For instance, there could be temperature sensors that detect high temperatures. In case one of the parade wagons malfunctions, these sensor could detect a fire and immediately trigger a warning to the fire brigade.

Option 2 can be implemented by removing all rules in the stream reasoner that contain the unimportant rules. The problem with this approach is that in order to re-add the rule, it must be stated inside of a policy to its entirety matching the exact syntax.

Option 3 allows us to comment out the affected rules. Depending on the implementation this can be achieved by simply adding the % character. The rules are then effectively disabled and thus achieving the same result as if they were removed. Later, these rules can be enabled again by simply removing the commenting character. To identify the lines to be commented out we can simply use the predicate name and comment all rules containing it in the rule head. Optionally, we can also comment all rules that have

[3]https://viennapride.at/en/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

that predicate name as a positive atom in the rule body, since it will never be satisfied. Following are the TLUPS policies responsible for the disabling:

– `disable for -1 rule trafficCount when prideParade(l34)`

– `enable for -1 rule trafficCount when prideParadeEnd(l34)`

Expected outcome:

All rules containing the to be disabled predicate in the head, or with a positive atom in the rule body, should be disabled by comments. For rules that extend over multiple lines this means commenting each one of the lines in order not to corrupt the file with invalid data. All non-affected rules should remain unchanged and keep working as expected. In addition, enabling the predicate again should also behave as expected. The comment symbols should be removed from each line belonging to an affected rule.

Outcome:

In Listing 6.1 a subset of the stream reasoner rules is presented and we see that both rules contain the `trafficCount` predicate. In addition, both rules span over multiple lines (end of a rule is marked by the "." symbol).

```
1  ...
2  trafficCount(V0,V1) :-
3    &sql2["SELECT iid, to_char(ROUND(x, 2), 'fm000.00') as x1
4    FROM v_speed_mrel GROUP BY iid, x, tp HAVING (iid, tp) IN (
5    SELECT iid, MAX(tp) FROM v_speed_mrel GROUP BY iid) ORDER
6    BY iid"](V0,V1).
7
8  @wsSend("vehicleStop(",V0,",",V1,")") :- trafficCount(V0,V1),
9    V1 <= "000.00", V0 > 0.
10 ...
```

Listing 6.1: Snippet of the Stream Reasoner before disabling

We start our Stream Reasoner and send a `prideParade(l34)` message at time point 7. We can see in Figure 6.7 that the Operator receives the message at time point 7 and that it triggers the correct policy. The command is sent to the Update Manager that is responsible for performing the changes on the stream reasoner's KB.

```
7: Event Channel says: event(prideParade,CM,Operator,location,Sat Apr 25 14:33:33 CEST 2020,[l34])
7: The event message event(prideParade,CM,Operator,location,Sat Apr 25 14:33:33 CEST 2020,[l34]) triggered
   the policy disable for -1 rule trafficCount when prideParade(l34)
```

Figure 6.7: Policy on the Operator is triggered by `prideParade(l34)` message

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

In Figure 6.8, we can see that the Update Manager receives the command; it substitutes any parameter in the template with their actual values and then restarts the Stream Reasoner. To disable the rules, the Update Manager reads the stream reasoner's KB

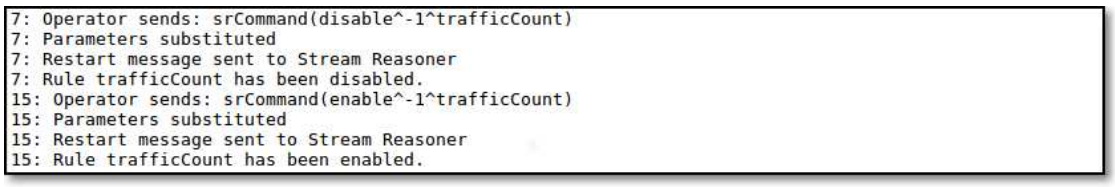


Figure 6.8: Policy on the Operator is triggered by `prideParade(134)` message

(represented by a file) and attributes each line to a rule. For example the rule starting line 2 extends over multiple lines, all of those lines are mapped to rule 0. All lines pertaining to a rule are then checked to see if the substring `trafficCount` is included. If any line is found matching, all lines belonging to the analyzed rule will be modified by adding the comment symbol `%`. Finally, the contents of the file are overwritten with the updated version. Listing 6.2 shows the effects of this change on the stream reasoner.

```
1 ...
2 %trafficCount(V0,V1) :-&sql2["SELECT iid, to_char(ROUND(x, 2)
      % , 'fm000.00') as x1 FROM v_speed_mrel GROUP BY iid, x,
      % tp HAVING (iid, tp) IN (SELECT iid, MAX(tp) FROM
      % v_speed_mrel GROUP BY iid) ORDER BY iid"](V0,V1).
3
4 %@wsSend("vehicleStop(",V0,",",V1,")") :- trafficCount(V0,V1)
      % , V1 <= "000.00", V0 > 0.
5 ...
```

Listing 6.2: Snippet of the Stream Reasoner after the rules have been disabled

As expected, when sending the `prideParadeEnd(134)` message, the second policy triggers on the Operator and sends the corresponding command to the Update Manager. In Figure 6.8, we can again see the command arriving at the Update Manager. Following the same procedure as before, the stream reasoner file is parsed and the lines containing comments and belonging to a rule that contains the `trafficCount` string have their comments removed.

Problems:

There is currently no history that allows us to tell which lines have been commented by a previous command and which were already commented before for other reasons. So in this example, if the lines containing the event `trafficCount` were already commented before executing any changes, the lines would end up being uncommented at the end of the enable command. This might lead to unwanted behavior. To solve this, one could

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

implement a trace log to keep track of which lines were affected by the commands or attribute an identifier to each command. For example, a previous rule deactivated a rule and we mark it with the new comment symbol %1%. The new command disabling all `trafficCount` rules would then carry the comment %2%. When the rules have to be enabled again we would then only remove the comments of the form %2% and maintain the previous comment.

6.2.2 Test 2: Timed parameter value change

In this test, we change a parameter located in the stream reasoner's KB through the interface and showcase the ability to seamlessly restart the stream reasoner. In addition the change to the parameter only lasts for 60 seconds, requiring an additional edit with restart.

Setting:

We define an event `vehicleStop` which triggers when a vehicle slows down below a certain speed threshold. The rule in the stream reasoner is defined as follows:

```
@wsSend("vehicleStop(",V0,",",V1,")") :-
  trafficCount(V0,V1), V1<= !speed_for_vehicle_stop!, V0
  >780.
```

`trafficCount(V0,V1)` is information retrieved from PipelineDB. It simply gives information about each car `V0` and the speed they are travelling at `V1`. If a car slows down under the threshold defined by the parameter `!speed_for_vehicle_stop!`, a message `vehicleStop` is sent to the stream reasoner with the car id and velocity. For this test, we also reduce the number of incoming messages to reduce clutter and limit the information to vehicles with id greater than 780.

Normally, the value for `!speed_for_vehicle_stop!` is set at "010.00" which is equivalent to 10 km/h. During normal traffic conditions, a car slowing down to less than 10 km/h can be assumed to be in the process of stopping. However, during a traffic jam most vehicles move slower than 10 km/h even though they are not stopping. We thus send a `trafficJam` message after a random time interval, which will trigger the following TLUPS policy:

```
execute for 60 rule editParameterValue(0,001.00) when
  trafficJam(l20)
```

This policy will change the value of the parameter with id `0` to the value `001.00` (1 km/h) when the message `trafficJam(l20)` is received. Since we also have the for keyword, these changes will last for 60 seconds.

Expected outcome:

The test should unfold as follows:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

1. The stream reasoner submits multiple `vehicleStop` messages each second reporting on the vehicle id and speed of cars travelling at less than 10 km/h.

2. At a certain point, the Operator will receive the `trafficJam(l20)` message and trigger the TLUPS policy. The command will then be executed by the Update Manager, which restarts the stream reasoner.

3. We should then see a change in the message that we receive: only information about cars travelling at less than 1 km/h should be submitted.

4. After 60 seconds, the Update Manager should reverse the changes made to the parameter and set the value back to 10 km/h, before restarting the stream reasoner once more. The Operator should then go back to receiving all messages like before for vehicles travelling at less than 10 km/h.

Outcome:

Before we observe the messages that arrive at the Operator we show the succession of events inside the Update Manager. In Figure 6.9, we show the console output of the Update Manager: at time point 17 it receives the interface command extracted from the TLUPS policy described in the Settings part of this experiment. It then changes the interface entry for the parameter with id `0` to the following:

```
parameters:
  parameter:
    id: 0
    name: speed_for_vehicle_stop
    type: string
    value: 001.00
    comment: Speed under which a vehicle is deemed
      as stopped.
```

The changes are saved and the parameter values are inserted into the running stream reasoner. To apply the changes, the stream reasoner needs to be restarted on-the-fly. After the restart of the stream reasoner, the Update Manager is idle for 60 seconds.

```
17: Operator sends: interfaceCommand(editParameterValue^60^0^001.00)
17: Changes saved.
17: Parameters substituted
18: Restart message sent to Stream Reasoner
78: Changes saved.
78: Parameters substituted
78: Restart message sent to Stream Reasoner
```

Figure 6.9: Actions taken by Update Manager

Note that it could still receive and execute other commands from the Operator due to our multi-threaded implementation. After 60 seconds, at time point 78, the interface is changed back to the previous value `010.00`, the changes are saved and inserted into

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the stream reasoner, which then gets triggered to restart. When we now look at the Operator, we expect the communication behavior to change soon after time point 18 and after time point 78. In Figure 6.10,[4] we observe three time intervals and the output

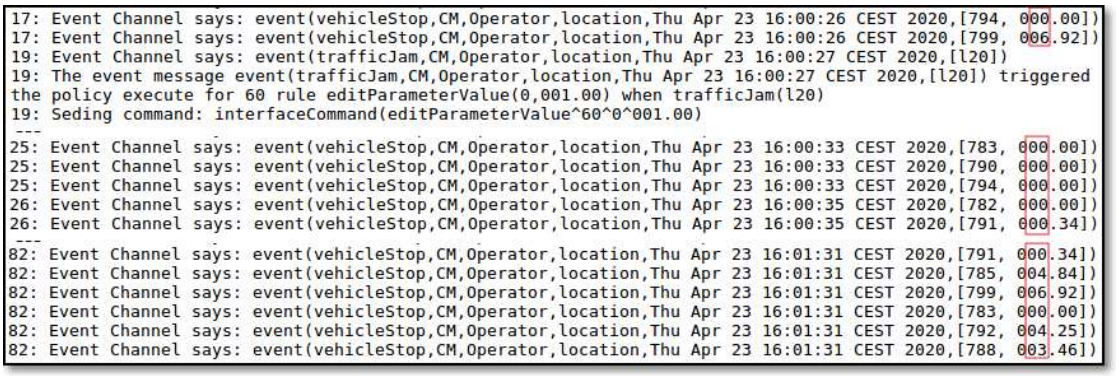


Figure 6.10: Operator sections showing the change in communication.

generated by our Operator.

- At time point 17, we observe that the `vehicleStop` messages contain information for cars travelling at 0 km/h and at 6.92 km/h.
- At time point 19, we receive the `trafficJam` message and the policy is triggered sending the interface command to the Update Manager.
- We then skip ahead to time point 25 and see that all messages that are received are for cars travelling at less than 1 km/h.
- We skip ahead one more time to point 82, which is after the changes to the interface have been retracted. We observe that messages are now being sent also for vehicles driving faster than 1 km/h.

We thus changed the interface settings twice and restarted the stream reasoner twice as well. These restarts took less than a second and would seem nonexistent from the Operator's perspective, we only spot them through the update manager's output.

6.2.3 Test 3: Addition of data source

In this example, we will trigger three TLUPS policies simultaneously with the goal of including a new PipelineDB data source to enable the communication of traffic light signal states.

[4]The timestamps of the Operator are 2 seconds ahead of the Operator since it gets started slightly later.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Setting:

In the past examples, we retrieved data from a database to obtain information about the number of cars in lanes and their speed. The stream reasoner connects at startup to the PipelineDB database, which is kept open during the evaluation of the KB.

Assume that we receive a message regarding a traffic jam and we are unsure if this might be caused by a malfunction of some traffic light. We now want to add a new data source that gives us information on the state of the traffic lights. With an appropriate implementation, the Operator could then decide based on the new data whether there are any abnormalities in the traffic lights behavior or whether the cause lies elsewhere, e.g., an accident. The three rules that we will need in our stream reasoner are the following:

- r1:
```
hasSignalGroup(L,T) :-
    &sql2["SELECT a,b FROM object_role_assertion WHERE
    object_role=151"](L,T).
```
- r2:
```
trafficLighState(T,S) :-
    &sql2["SELECT DISTINCT ON (iid) iid, x FROM
    v_signalstate_mrel WHERE tp > 0 ORDER BY iid, tp
    DESC"](T,S).
```
- r3:
```
@wsSend("trafficLightState(",T,",",S,")") :-
    hasSignalGroup(L,T), trafficLighState(T,S).
```

For asserting these rules, we use the following TLUPS policies:

- `assert for 600 rule r1 when trafficJam(l20)`
- `assert for 600 rule r2 when trafficJam(l20)`
- `assert for 600 rule r3 when trafficJam(l20)`

As stated before, the ^ symbol substitutes the comma due to easier parsing. This example in particular shows the difficulty of parsing a comma separated command when a rule involving SQL requests is involved.

Expected outcome:

The three TLUPS policies trigger and arrive at the Update Manager simultaneously, and are added to the stream reasoner template file before the Update Manager creates the original file and triggers stream reasoner restart. At this point, the concept described in Section 5.2.1 should come into play. Without intervention, the Update Manager would trigger the stream reasoner to restart three times. This causes unnecessary downtime. Instead, by adding a small delay the Update Manager waits 100 ms before it restarts ignoring any other restart commands. After the restart, we will be receiving information from the new data source on the Operator.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Outcome:

We can see the actions of the Update Manager in Figure 6.11. Specifically, we can observe that all steps are repeated for each command except for the restart message. The multiple restart prevention method worked and only caused one restart causing less downtime. The three commands come in at time point 28, at which point any eventual

```
28: Operator sends: srCommand(assert^15^hasSignalGroup(L,T) :- &sql2["SELECT a,b FROM object_role_assertion
    WHERE object_role=151"](L,T).)
28: Operator sends: srCommand(assert^15^trafficLighState(T,S) :- &sql2["SELECT DISTINCT ON (iid) iid, x FROM
    v_signalstate_mrel WHERE tp > 0 ORDER BY iid, tp DESC"](T,S).)
28: Operator sends: srCommand(assert^15^@wsSend("trafficLightState(",T,",",S,")") :- hasSignalGroup(L,T), tr
    afficLighState(T,S).)
28: Parameters substituted
28: Parameters substituted
28: Rule hasSignalGroup(L,T) :- &sql2["SELECT a,b FROM object_role_assertion WHERE object_role=151"](L,T). h
    as been asserted.
28: Parameters substituted
28: Rule @wsSend("trafficLightState(",T,",",S,")") :- hasSignalGroup(L,T), trafficLighState(T,S). has been a
    sserted.
28: Restart message sent to Stream Reasoner
28: Rule trafficLighState(T,S) :- &sql2["SELECT DISTINCT ON (iid) iid, x FROM v_signalstate_mrel WHERE tp >
    0 ORDER BY iid, tp DESC"](T,S). has been asserted.
```

Figure 6.11: Operator section showing the assertion of rule `r1`, `r2` and `r3`

parameters present in the template are inserted and a restart message is sent. Right after the execution of the three commands the Communication Manager starts receiving new message concerning traffic lights and their current status. Note that if no predicate for the `trafficLightState` event is defined in the interface, its message will use the default behavior of the stream reasoner. Optionally, another TLUPS policy could be added to add the predicate `trafficLightState` to the interface and define the communication behavior there.

With these experiments we demonstrated how the communication settings can be changed in the interface and how the communication behavior changes without touching the Stream Reasoner. For changes involving the modification of rules and thus requiring a stream reasoner restart, we exemplified the restart behavior showing the seamless interaction through stream reasoner, Update Manager and Operator.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 7

Conclusion

In this thesis, we presented an approach that allows declarative adaptive interface monitoring. We extended the DynaCon [EDTF+19] framework by adding specialized modules: the Communication Manager and the Update Manager. In combination came the creation of an interface description language, introducing a way to dynamically adapt communication with the stream reasoner. For a formal underpinning of the interface, we created TLUPS, an extension of *LUPS* [APPP02] (a language for updating logic programs), which in addition also allows us to assert rules for a given amount of time. We combined TLUPS with our interface command language to not only allow changing the stream reasoner through TLUPS policies, but also change the interface configuration, directly influencing communication behavior and stream reasoner parameters. For interacting with the interface and defining the format for parameter and predicate entries, we created an interface description language, which we then formalized using the EBNF grammar described in Appendix A.

To showcase declarative adaptive interface monitoring, we created a Java prototype including modules for the Communication Manager, the Operator and the Update Manager. Lastly, in combination with the HexLite [Sch19] stream reasoner we performed some experiments in the scope of a C-ITS use case in order to demonstrate the capabilities of our approach.

7.1 Future Work

The concepts and ideas presented in Chapter 3 and 4 still dwarf in comparison to the myriads of use cases in real world scenarios. The evolution of LUPS over LUPS* [Lei01] to TLUPS is only a small example showing the necessity for more functionalities in update policies for temporal rule languages. In this section we will mention some improvements and future extensions connected to this thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

7.1.1 TLUPS

While TLUPS has many functionalities like conditional, persistent, and timed commands we identified some features that could be added:

- Support for asserting rules at future time points, i.e., assert a rule or perform a change after a specific time interval has elapsed.

  Example 28. Suppose we want to assert a rule with the delay t when a certain event occurs. We would extend the current assert command with the `in` keyword as follows:

  ```
  assert in 60 rule <rule> when <condition>
  ```

  The policy would then be triggered as soon as `<condition>` is satisfied, but `<rule>` would only be asserted after 60 seconds passed.

- When we assert a rule, the only way to retract it automatically is through the **for** keyword that can only take a time unit as an input. What if we wanted to retract a rule that we asserted once a certain condition is achieved?

  Example 29. In Experiment 2 Test 2, we set the speed threshold to 1km/h with the intention of reverting it after 60 seconds. What if the traffic jam is still ongoing after 1 minute? Then we would have to assert the rule again. Instinctively, as soon as cars start circulating at higher speeds the traffic jam is over. The stream reasoner could detect this through a window function checking how many cars are travelling above a certain speed. So we could set a condition of the type:

  ```
  execute rule editParameterValue(0,001.00) when
     trafficJam(l20) until trafficJamEnd(l20)
  ```

- Adding more functions to commands makes the semantic transformation to a logic program more intricate. We could see in Section 4.2 that the transformation to a logic program became harder with LUPS* and even more complicated with the substitution of the **event** keyword by the timed **for** keyword in our TLUPS extension.

7.1.2 Prototype functions

As is often the case with prototypes, ours also is not a full fledged implementation of the concepts described in Chapter 4.1. Within the given time frame, some parts had to be omitted in favor of a quicker implementation while others present new challenges that we could not address in the scope of this thesis. We subdivide the omitted features by the responsible modules in the following sections.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Communication Manager

Our implementation of the Communication Manager had a limited expressibility for conditions in exceptions. Ideally, we would want our prototype to be able to evaluate arbitrary SQL-like or Datalog-like conditions in order to fine-tune the exceptions as envisioned in the concept in Chapter 3. Furthermore, a particularly hard challenge is given by pull based stream reasoners where data has to be retrieved by polling. Our Communication Manager can only work with push based stream reasoners that send messages on their own.

Operator

Right now all TLUPS policies in the Operator are persistent due to the lack of a ticker system. The Operator does not work in iterations, it immediately responds as soon as a message is received by the Communication Manager. While this approach is more responsive compared to a ticker system where the Operator has to wait for the next iteration before acting, it is harder to deal with non persistent commands since there is not a succession of clearly defined states where conditions can be evaluated also for non persistent commands.

Our implementation currently supports one simultaneous stream reasoner. More communication and update managers can connect to the Operator but both messages and commands are hard coded to only work with the first stream reasoner that connects.

The conditions for TLUPS policies also had to be restricted to exact matching of event messages. For example we can wait for the arrival of certain event messages through

```
... when trafficJam(l34);
```

This condition will only be satisfied if the exact matching message `trafficJam(l34)` arrives at the Operator. Clearly, more expressiveness is desired. Once again SQL or Datalog type syntax would offer far more elaborate tools to fine tune conditions. The minor change of introducing a wildcard symbol would already increase the flexibility of defining conditions. By using `trafficJam(_)` or `trafficJam(*)` as a condition we could accept any `trafficJam` message.

Information requests sent by the Operator can only specify a specific predicate and exception. Ideally, the Operator should be able to filter by locations, time stamps, arguments and more to obtain the information that it needs.

Example 30. Let `carsInLane(X,Y)` be a predicate that contains information about the number of cars X in lane Y. This predicate also is defined to work on pull based communication and has no exceptions in the interface. The Operator is interested in knowing whether there was any instance in the last 10 time points, where the number of cars in lane **l34** were above 20. The only tool available right now is to ask for predicate `carsInLane`. The Operator would then receive all messages stored in the Communication Manager related to the predicate and check by himself whether one of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the messages is helpful. Ideally, we would want a filter to retrieve messages created in the last 10 minutes, where $X = l34$ and $Y > 20$. The same functionality should be available for the Communication Manager when evaluating the conditions defined for predicate exceptions.

Update Manager

Currently to perform any changes through update commands, data has to travel from the Stream Reasoner through the Communication Manager to the Operator and then finally to the Update Manager. While we have not constructed any performance benchmarks, undeniably there would be a gain in speed if the cycle was shortened somewhere along the path. For example, some TLUPS policies could be stored with the Update Manager and some of the information delivered by the stream reasoner could be immediately routed to the Update Manager as well. By doing so, the Communication Manager and Operator are skipped in the workflow, which would speed up the process of performing changes to the interface and Stream Reasoner.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

APPENDIX A

Full EBNF grammar

int_desc ::= "<InterfaceDescription>" , **param_section** , **data_section**, "</InterfaceDescription>";

param_section ::= "<Parameters>" , {**parameter**} , "</Parameters>";

parameter : := "<Parameter><id>" , **natural_number** , "</id><name>" , **parameter_name** , "</name>" , **value_type_pair** , "<comment>" , **text** , "</comment></Parameter>";

parameter_name ::= **low_case_letter** , {["_"] , **low_case_letter**};

data_section ::= "<Data>" , **sr_default** , {**predicate**} , "</Data>";

sr_default ::= "<sr_default>" , **comm_detail** , "</sr_default>";

predicate ::= "<Predicate><id>" , **natural_number** , "</id><name>" , **predicate_name** , "</name><number_of_arguments>" , **natural_number**, "</number_of_arguments><type>" , **type** , "</type><comment>" , **text** , "</comment>" , **communication** , "</Predicate>";

predicate_name ::= **up_case_letter** , {**low_case_letter** | **up_case_letter** };

communication ::= "<communication>" , [**default**] , **exc_section**, "</communication>";

default ::= "<default>" , **comm_detail**, "</default>";

exc_section ::= "<exceptions>" , {**exc**} , "</exceptions>";

exc ::= "<exception><id>" , **natural_number** , "</id><arguments>" , **argument** , {**argument**} , "</arguments>" , **comm_detail** , "</exception>";

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

argument ::= "<argument nr='" , **natural_number** , "'>" , **word_array** , "</argument>";

comm_detail ::= "<mode>" , **comm_mode** , "</mode><buffered>" , **boolean** , "</buffered><delay>" , **natural_number** , "</delay>";

value_type_pair ::= "<type>string</type><value>" , **word** , "</value>"
|"<type>int</type><value>" , **integer** , "</value>"
|"<type>collection</type><value>" , **word_array** , "</value>";

boolean ::= "true" | "false";

comm_mode ::= "push" | "pull";

type ::= "int" | "string";

text ::= {word|" "};

word ::= (**digit** | **low_case_letter** | **up_case_letter**) , { **digit** | **low_case_letter** | **up_case_letter**};

word_array ::= **word** , {","**word**};

low_case_letter ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z";

up_case_letter ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z";

integer ::= "0"|["-"], **natural_number**;

natural_number ::= "0"| (**digit_excl_zero**, {**digit**});

digit ::= "0"| **digit_excl_zero**;

digit_excl_zero ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

APPENDIX B

Implementation code

The basic Java Socket implementation for communication between internal modules (Communication Manager, Update Manager and Operator)

```
1  import java.io.BufferedReader;
2  import java.io.InputStreamReader;
3  import java.net.ServerSocket;
4  import java.net.Socket;
5
6  ServerSocket server= new ServerSocket(1979);
7  Socket client= server.accept();
8
9  InputStreamReader inp = null;
10 BufferedReader bReader = null;
11
12 inp = new InputStreamReader(client.getInputStream());
13 bReader = new BufferedReader(inp);
14
15 String text= bReader.readLine();
```

Listing B.1: Java server implementation

```
1  import java.io.DataOutputStream;
2  import java.net.Socket;
3
4  Socket socket = new Socket("localhost", 1979);
5  DataOutputStream out =
6      new DataOutputStream(socket.getOutputStream());
7
8  out.writeBytes("message" + "\n");
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
9  out.flush();
```

Listing B.2: Java client implementation

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

APPENDIX C

Experiment replication

On a Linux system, install HexLite by following the procedure outlined in https://github.com/hexhex/hexlite.

The Java prototype can be found in https://github.com/patrik999/adaptive-stream-reasoning-monitoring/tree/master/src/LinuxThesis .

To run one of the tests from the thesis follow these steps:

- Inside the `LinuxThesis` folder are subfolders for each test containing the interface file for the test, the stream reasoner template file and the original stream reasoner file.
- For example, to execute Experiment 1 Test 2, open the startup files
  - `OperatorStartup.java`
  - `UmStartup.java`
  - `CmStartup.java`

  and set the variables `exp=1` and `test=2`. By doing so, the Operator will assert the correct TLUPS commands and the Communication Manager and Update Manager will know in what folder to find the interface and stream reasoner template.
- Start the Modules in the order: Operator → Communication Manager → Update Manager. The Update Manager will then take care of starting the Stream Reasoner.

Note that some tests like Experiment 1 Test 1 will change the interface after it is run once. To replicate the experiments multiple times you will need to revert the interface back to its original state.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[AGM85] Carlos Alchourrón, Peter Gärdenfors, and David Makinson. On the logic of theory change: Partial meet contraction and revision functions. *J. Symb. Log.*, 50:510–530, 06 1985.

[ALP+00] J.J. Alferes, J.A. Leite, L.M. Pereira, H. Przymusinska, and T.C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1):43 – 70, 2000.

[AP06] José Alferes and Luís Pereira. *Update-programms can update programs*, pages 110–131. Springer, 04 2006.

[APPP02] José Júlio Alferes, Luís Moniz Pereira, Halina Przymusinska, and Teodor C. Przymusinski. Lups—a language for updating logic programs. *Artificial Intelligence*, 138(1):87 – 116, 2002. Knowledge Representation and Logic Programming.

[AS13] Antonio Alberti and Dhananjay Singh. Internet of things: Perspectives, challenges and opportunities. In *International Workshop on Telecommunications (IWT 2013), Volume: 1*, 05 2013.

[Bal09] Martin Baláž. Answer set programming: Syntax and semantics. url: `http://dai.fmph.uniba.sk/~siska/asp/asp01.pdf`, [Online; accessed 03-May-2020], 2009.

[BBCG10] Davide Barbieri, Daniele Braga, Stefano Ceri, and Michael Grossniklaus. An execution environment for c-sparql queries. In *13th International Conference on Extending Database Technology (EDBT 2010)*, Lausanne, Switzerland, 01 2010.

[BBU17] Hamid R. Bazoobandi, Harald Beck, and Jacopo Urbani. Expressive stream reasoning with laser. In Claudia d'Amato, Miriam Fernandez, Valentina Tamma, Freddy Lecue, Philippe Cudré-Mauroux, Juan Sequeda, Christoph Lange, and Jeff Heflin, editors, *The Semantic Web – ISWC 2017*, pages 87–103, Cham, 2017. Springer International Publishing.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[BDTE15] Harald Beck, Minh Dao-Tran, and Thomas Eiter. Answer update for rule-based stream reasoning. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 2741–2747. AAAI Press, 2015.

[BDTE16] Harald Beck, Minh Dao-Tran, and Thomas Eiter. Equivalent stream reasoning programs. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, page 929–935. AAAI Press, 2016.

[BDTEF15] Harald Beck, Minh Dao-Tran, Thomas Eiter, and Michael Fink. Lars: A logic-based framework for analyzing reasoning over streams. In *AAAI15: Twenty-Ninth Conference on Artificial Intelligence*, Austin, Texas, USA, 01 2015.

[BEF17] Harald Beck, Thomas Eiter, and Christian Folie. Ticker: A system for incremental asp-based stream reasoning. *Theory and Practice of Logic Programming*, 17(5-6):744–763, Aug 2017.

[BJC05] Thaís Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. *Lecture Notes in Computer Science*, 98:1–17, 06 2005.

[Bun13] Alan Bundy. The interaction of representation and reasoning. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 469(2157):20130194, 2013.

[CHVF09] S. Ceri, F. Harmelen, E. Valle, and D. Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(06):83–89, nov 2009.

[DFI+03] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate functions in DLV. In Marina De Vos and Alessandro Provetti, editors, *Answer Set Programming, Advances in Theory and Implementation, Proceedings of the 2nd Intl. ASP'03 Workshop, Messina, Italy, September 26-28, 2003*, volume 78 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[dH16] D. de Leng and F. Heintz. Dyknow: A dynamically reconfigurable stream reasoning framework as an extension to the robot operating system. In *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 55–60, Dec 2016.

[DVvHB17] Daniele Dell'Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. Stream reasoning: A survey and outlook. *Data Sci.*, 1:59–83, 2017.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[EDTF+19] Thomas Eiter, M. Dao-Tran, A. Falkner, P. Ogris, K. Schekotihin, P. Schneider, P. Schüller, and A. Weinzierl. Stream reasoning and multi-context systems. *Stream Reasoning Workshop 2019*, 2019. url: `https://sr2019.on.liu.se/slides/eiter-keynote-sr2019ws.pdf` , [Online; accessed 19-April-2020].

[EFI+15] Thomas Eiter, M. Fink, G. Ianni, T. Krennwallner, C. Redl, and P. Schüller. A model building framework for answer set programming with external computations. *Theory and Practie of Logic Programming*, 07 2015.

[EFTW18] Thomas Eiter, Gerhard Friedrich, Richard Taupe, and Antonius Weinzierl. Lazy grounding for dynamic configuration: Efficient large-scale (re)configuration of cyber-physical systems with asp. *KI - Künstliche Intelligenz*, 32, 05 2018.

[EIST05] Thomas Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 90–96, Edinburgh, Scotland, UK, 01 2005.

[Eit16] Thomas Eiter. Answer set programming and extensions. url: `http://www.kr.tuwien.ac.at/staff/eiter/courses/vtsa16/unit1.1nup.pdf`, [Online; accessed 03-May-2020], 2016.

[ESF+19] Thomas Eiter, Patrik Schneider, Andreas Falkner, Konstantin Schekotihin, and Weinzierl Antonius. Deliverable D2.1: Requirements and architecture (v9). internal report, Project DynaCon (FFG 861263), 2019.

[ESS19] Thomas Eiter, Patrik Schneider, and Peter Schüller. Deliverable D5.1: Interface between stream reasoner and configurator (v4). internal report, Project DynaCon (FFG 861263), 2019.

[FLP04] Wolfgang Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004*, volume 3229, pages 200–212, 01 2004.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski, Bowen, and Kenneth, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.

[KM91] Hirofumi Katsuno and Alberto O. Mendelzon. On the difference between updating a knowledge base and revising it. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, KR'91, pages 387–394, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[KM14] S.K. Khaitan and James Mccalley. Design techniques and applications of cyberphysical systems: A survey. *IEEE Systems Journal*, 9:1–16, 07 2014.

[Kow74] Robert A. Kowalski. Predicate logic as programming language. In Jack L. Rosenfeld, editor, *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, pages 569–574. North-Holland, 1974.

[KWH13] Henning Kagermann, W. Wahlster, and J. Helbig. Recommendations for implementing the strategic initiative industrie 4.0 – securing the future of german manufacturing industry. Final report of the industrie 4.0 working group, acatech – National Academy of Science and Engineering, Muenchen, 04 2013.

[Lee06] Edward Lee. Cyber-physical systems - are computing foundations adequate? *NSF Workshop On Cyber-Physical Systems*, 01 2006.

[Lee08] E. A. Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, May 2008.

[Lee17a] Kent Lee. *Foundations of Programming Languages*. Springer Verlag, 01 2017.

[Lee17b] Kent Lee. *Logic Programming*, pages 277–304. Springer Verlag, 12 2017.

[Lei01] João Alexandre Leite. A modified semantics for LUPS. In Pavel Brazdil and Alípio Jorge, editors, *Progress in Artificial Intelligence, Knowledge Extraction, Multi-agent Systems, Logic Programming and Constraint Solving, 10th Portuguese Conference on Artificial Intelligence, EPIA 2001*, volume 2258 of *Lecture Notes in Computer Science*, pages 261–275, Porto, Portugal, 2001. Springer.

[MSLM09] Joao Marques-Silva, Inês Lynce, and S. K. Malik. Conflict-driven clause learning sat solvers. In *Handbook of Satisfiability*, 2009.

[MT94] Victor W. Marek and Mirosław Truszczyński. Revision specifications by means of programs. In Craig MacNish, David Pearce, and Luís Moniz Pereira, editors, *Logics in Artificial Intelligence*, pages 122–136, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[OC99] Ming Ouyang and Vasek Chvatal. *Implementations of the DPLL Algorithm*. PhD thesis, Rutgers University, USA, 1999. AAI9947888.

[Oet80] J. Oetting. An analysis of meteor burst communications for military applications. *IEEE Transactions on Communications*, 28(9):1591–1601, Sep. 1980.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34(3), September 2009.

[PT97] Teodor C. Przymusinski and Hudson Turner. Update by means of inference rules. *The Journal of Logic Programming*, 30(2):125 – 143, 1997.

[Ray13] A. Ray. Autonomous perception and decision-making in cyber-physical systems. In *2013 8th International Conference on Computer Science Education*, pages 1–10, April 2013.

[RG18] Jacqueline Reis and Rodrigo Gonçalves. *The Role of Internet of Services (IoS) on Industry 4.0 Through the Service Oriented Architecture (SOA): IFIP WG 5.7 International Conference, APMS 2018, Seoul, Korea, August 26-30, 2018, Proceedings, Part II*, pages 20–26. Springer, 08 2018.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[Roj17] Andreja Rojko. Industry 4.0 concept: Background and overview. *International Journal of Interactive Mobile Technologies (IJIM)*, 11. 77. 10.3991/ijim.v11i5.7072, 2017.

[Sch19] Peter Schüller. The hexlite solver. In Francesco Calimeri, Nicola Leone, and Marco Manna, editors, *Logics in Artificial Intelligence*, pages 593–607, Cham, 2019. Springer International Publishing.

[SWD+14] Pradhan Subhav, O. William, A. Dubey, C. Szabo, G. Aniruddha, and G. Karsai. Towards a self-adaptive deployment and configuration infrastructure for cyber-physical systems. Technical report, Institute for Software Integrated Systems, Nashville, 06 2014.

[TAB+17] M. Törngren, F. Asplund, S. Bensalem, J. McDermid, R. Passerone, H. Pfeifer, A. Sangiovanni-Vincentelli, and B. Schätz. Chapter 1 - characterization, analysis, and recommendations for exploiting the opportunities of cyber-physical systems. In *Cyber-Physical Systems*, Intelligent Data-Centric Systems, pages 3 – 14. Academic Press, Boston, 2017.

[VCB+09] Emanuele Valle, Stefano Ceri, Davide Francesco Barbieri, Daniele Braga, and Alessandro Campi. A first step towards stream reasoning. In John Domingue, Dieter Fensel, and Paolo Traverso, editors, *Future Internet FIS 2008*, pages 72–81. Springer-Verlag, Berlin, Heidelberg, 2009.

[YZL+19] Xifan Yao, Jiajun Zhou, Yingzi Lin, Yun Li, Hongnian Yu, and Ying Liu. Smart manufacturing based on cyber-physical systems and beyond. *Journal of Intelligent Manufacturing*, 30(8):2805–2817, December 2019.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN