Christopher Maier, BSc

YAGI - An Easy and Light-Weighted Action-Programming Language for Education and Research in Artificial Intelligence and Robotics

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa

Institute for Software Technology

Ass. Prof. Dipl.-Ing. Dr. techn. Gerald Steinbauer

Graz, Jan. 2015

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

Date

Signature

Abstract (English)

The action-based imperative programming language Golog and its descendants have proved to be powerful and well-studied languages to model autonomous robots and agents. However, almost all Golog interpreters are implemented as a set of Prolog clauses, which eliminates Golog as a language of choice for any platform that lacks an implementation of a Prolog interpreter. Furthermore, the fuzzy boundary between features of Golog and side-effects of Prolog makes the process of implementing Golog programs cumbersome and error-prone. Various extensions and modifications of Golog have been proposed to serve a variety of different needs, yet previous work addressed the tight coupling of Golog to Prolog only to a certain extent. In this thesis, we introduce YAGI (an acronym for **Y**et **A**nother **G**olog **I**ntepreter), an action-based imperative programming language based on the theoretical foundations of situation calculus and IndiGolog, but with a clear separation of syntax and semantics that enables us to remove the tight binding to Prolog. We sketch a 3-tier architecture for a YAGI-based software system, provide a specification of the syntax and semantics of YAGI followed by a discussion of our proof-of-concept implementation and an analysis of how our implementation follows the specified semantics. The output of this thesis is the specification and implementation of a programming language that is specifically designed to avoid the pitfalls of Prolog-based implementations of Golog, hopefully allowing people to write less error-prone and more portable applications based on the semantics of situation calculus and IndiGolog.

Abstract (German)

Die aktionsbasierte imperative Programmiersprache Golog und dessen Abwandlungen sind leistungsfähige und gut erforschte Programmiersprachen zur Beschreibung von autonomen Systemen und Agenten. Allerdings sind fast alle Golog Interpreter als eine Menge von Prolog Klauseln implementiert, was Golog auf Plattformen beschränkt welche über einen Prolog Interpreter verfügen. Ein weiteres Problem ist die ungenaue Abgrenzung zwischen Eigenschaften von Golog zu Seiteneffekten von Prolog, was die Entwicklung von Gologprogrammen umständlich und fehleranfällig macht. In bisherigen Arbeiten wurden zahlreiche Erweiterungen und Abwandlungen von Golog präsentiert, welche die Sprache für unterschiedlichste Anforderungen erweitern. Allerdings wurde dabei die strikte Kopplung von Golog zu Prolog nur bis zu einem bestimmten Detailgrad behandelt. In dieser Masterarbeit stellen wir YAGI (ein Akronym für **Y**et **A**nother **G**olog **I**ntepreter) vor. YAGI ist eine aktionsbasierte imperative Programmiersprache aufbauend auf dem Situationskalkül und IndiGolog, jedoch hebt sie mit einer strikten Trennung von Syntax und Semantik die enge Kopplung zu Prolog auf. Wir skizzieren eine dreischichtige Architektur für ein YAGI-basiertes Softwaresystem, spezifizieren die Syntax und Semantik von YAGI, beschreiben unsere Proof of Concept Implementierung und diskutieren, warum diese Implementierung der Spezifikation entspricht. Das Ergebnis dieser Masterarbeit ist die Spezifikation und Implementierung einer Programmiersprache die bewusst entworfen wurde, um die Probleme von Prolog-basierten Implementierungen von Golog weitestgehend zu vermeiden. Wir erhoffen uns das diese Arbeit es Personen ermöglicht weniger fehleranfällige und leichter portierbare Programme basierend auf dem Situationskalkül und IndiGolog erstellen zu können.

Acknowledgements

I would like to thank a number of people for supporting me while working on this thesis.

First, I want to thank my adviser Prof. Gerald Steinbauer for his guidance and constant support and my supervisor Prof. Franz Wotawa for giving me the opportunity to work on this thesis.

Second, I want to thank Clemens Mühlbacher for his valuable comments and his constructive reviews of draft versions of this document.

Third, I want to thank the people from the Department of Computer, Control, and Management Engineering at the Sapienza University of Rome for their valuable input, in particular Stavros Vassos and Giuseppe De Giacomo. Further, I want to thank Alexander Ferrein from FH Aachen University of Applied Sciences for his insightful comments.

Finally, I want to thank my family and friends for their moral support.

Christopher Maier
Graz, 2015

Contents

Listings

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Motivation

The art of modeling autonomous robots and agents has been a highly compelling topic in the scientific community for the past decades. Up to the present day, teaching robots tasks that are trivial for human beings is a non-trivial undertaking. Consider the well-known example of an object delivery robot. The job of the robot is solely to deliver packages between different people in, say, an office building. To accomplish this task, the robot must be able to deal with a number of different issues. First of all, the robot must be able to perform all the actions that are needed to accomplish the task, e.g. moving to offices and picking up objects. Furthermore, the robot must be able to monitor its actions, e.g. track its position during movements and verify if an object has been delivered successfully. Moreover, the robot must be able to react to changes of its environment, e.g. it must be able to react to human requests to deliver some object. Note that even though this enumeration is non-exhaustive, it is not trivial (even sometimes not feasible) to consider all these issues when writing a robot program as a set of predefined rules.

To address these issues, several well-working approaches have been established over the years, one of the most successful ones is Golog (Levesque et al., 1994), a programming language based on situation calculus. Situation calculus is a formalism based on second-order logic with equality, introduced by (McCarthy, 1963) and (Reiter, 2001). Golog combines a formal domain specification with elements of imperative programming, hence the term *action-based imperative programming* has been chosen to describe this paradigm. Over the years, various dialects of Golog have been introduced to cope with different issues such as parallelism, sensing and exogenous events, making Golog a even more powerful programming language. Additionally, Golog was used to implement various different types of real-world applications and made its entrance into academic eduction as part of lectures held at technical universities around the world.

With the knowledge gathered from the intensified use of Golog researches discovered certain drawbacks, first and foremost the fact that almost all Golog interpreters have in common that their implementations are Prolog-based. The decision to pick Prolog as the language of choice for implementing a Golog interpreter is everything but surprising since the theoretical foundation of Golog is the situation calculus, a language based on second-order logic, hence picking a logic programming language seems to be the most natural decision. However, the decision to use Prolog also imposes some drawbacks, as lined out by (Ferrein et al., 2012). First of all, it implies that any platform that lacks of an implementation of a Prolog interpreter is not able to run a Golog program. Since one major domain of Golog-like languages is the field of robotics, a Prolog interpreter might not always be feasible due to resource constraints and/or performance limitations. Moreover, even if a Prolog implementation exists, integrating a Golog interpreter into a rather complex technical ecosystem might be a non-trivial task. Also, the distinction between what are features of Golog and what are (sometimes subtle) idiosyncrasies of (a specific implementation of) Prolog is especially

challenging for novices and sometimes even for experts. One of the reasons for this is that the boundary between Golog and Prolog tends to be fuzzy since Golog implicitly uses features of Prolog. To address these issues, (Ferrein et al., 2012) proposed YAGI, which stands for *Yet Another Golog Interpreter*. (Ferrein et al., 2012) sketched the syntax and semantics of YAGI, a novel approach to realize an action-based programming language inspired and based on Golog (and IndiGolog, respectively), but also specifically designed to address the issues described above.

1.2. Goal

The goal is to define a programming language based on the ideas from (Ferrein et al., 2012) that allows a programmer to build applications based on the semantics of situation calculus and IndiGolog. From a syntactical perspective, the goal is that the language should be easy to use for people who are aware of widely used programming concepts such as variables, loops and conditionals. Further, the syntax of the language should be as intuitive as possible to enable novices to build applications based on the foundations of situation calculus and IndiGolog without much burden. From a semantics perspective, the goal is to provide a solid and theoretically sound definition of a language that is based on situation calculus and IndiGolog. Furthermore, the language should lift the tight binding of Golog to Prolog. To accomplish such an abstraction it is necessary to specify a system architecture that is able to decouple the syntax and semantics of the language as cleanly as possible. Moreover, such an architecture should provide easy extensibility and a clear separation of concerns for easy maintainability.

1.3. Contributions of this Thesis

The contributions of this thesis are as follows:

- We present a 3-tier system architecture for YAGI-based applications that subdivides a YAGI application into front-end, back-end and system interface.
- We provide a formal specification of the syntax and semantics of YAGI, based on the ideas and goals outlined by (Ferrein et al., 2012). The focus is to provide a solid and theoretically sound definition of a language that is based on situation calculus and IndiGolog but can be used on a variety of different target systems independently of the existence of a Prolog interpreter.
- Based on the language specification and the system architecture we provide a proof-of-concept implementation of a YAGI interpreter.
- Based on the specification of the language and the proof-of-concept implementation we discuss how our implementation follows the specified semantics of the language.
- We provide an evaluation of our proof-of-concept implementation compared to a Prolog-based implementation of Golog.

1.4. Running Example

To illustrate our intentions, we use an object delivery robot as running example. The task of the robot is to pickup a certain object from a person, i.e. the sender and deliver it to another person, i.e. the receiver. Every person can possibly reside in a number of different offices, hence it is necessary for the robot to check whether or not the person is actually in the room before picking up/delivering an object.

1.5. Organization

This thesis is organized as follows. In the next chapter we present an overview of related work. Then, we describe the YAGI system architecture we propose for implementing YAGI-based systems in Chapter 3 and continue with an implementation of our running example in YAGI in Chapter 4. Next, we give a definition of syntax and semantics of YAGI in Chapter 5 followed by the description of our proof-of-concept implementation of a YAGI interpreter in Chapter 6. Further, we discuss why our implementation follows the specification in Chapter 7 and provide an evaluation of our implementation in Chapter 8. Finally, we finish with our conclusion and ideas about future work in Chapter 9.

Chapter 2

Related Work

In this chapter we give an overview of theoretical prerequisites and topics related to our work. We discuss the theoretical prerequisites of situation calculus and Golog in Section 2.1 and Section 2.2, respectively. Then, we discuss various extensions of Golog in Section 2.3 and continue with a description of some non-Prolog based implementations of Golog in Section 2.4. Next, we give a short summary of the initial proposal of YAGI in Section 2.5 and proceed with a short discussion about relational databases and their relation to basic action theories in Section 2.6. Subsequently, we give an overview of sensing and incomplete information in Section 2.7 and finish the chapter with a discussion of some other approaches that deal with the task of reasoning about actions and change in Section 2.8.

2.1. Situation Calculus

Situation calculus is a second-order language with equality which was designed to represent dynamic systems and to reason about actions and their effects. It was introduced by (McCarthy, 1963) and (Reiter, 2001) as a set of variables, constants, functions and predicates. The basic idea is that the world evolves from an initial situation (denoted by the special constant S_0) due to the execution of so-called *actions*. Such a sequence of actions is interpreted as a world history and is called a *situation*. To evolve into a new situation the special function symbol $do(a,s)$ is used to denote that the new situation is the result of executing the action a in a situation s. To state the fact that not every action might be able to be executed in every situation, a special predicate *Poss* (the so-called *action precondition*) of the form $Poss(a,s)$ is used to denote whether or not an action a can be executed in a situation s. To capture the changing properties of the world special relations called *fluents* of the form $F(x_1,\ldots,x_n,s)$ are used, where F is the name of the fluent, $x_1,\ldots,x_n$ are the parameters and s is a *situation*. Fluents can be further divided into *relational* fluents, i.e. $F(x_1,\ldots,x_n,s)$ is a predicate symbol that represents a truth value depending on a specific situation and *functional* fluents, i.e. $F(x_1,\ldots,x_n,s)$ is a function symbol that changes its result value depending on a specific situation.

To formalize this, (Reiter, 2001) defined a set of axioms that form a so-called *basic action theory* $\mathcal{D}$ as follows:

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$$

In a situation calculus basic action theory Σ denotes the set of domain independent axioms that define properties of a legal situation, $\mathcal{D}_{ssa}$ is the set of *successor state axioms* of the form $F(\vec{x},do(a,s)) \equiv \Phi_F(\vec{x},a,s)$ that specify for every fluent F under which conditions it holds in the situation $do(a,s)$. $\mathcal{D}_{ap}$ contains *action precondition axioms* of the form $Poss(A(\vec{y},s)) \equiv \Pi_A(\vec{y},s)$ that specify for every action A under which condition it can be executed in a situation s. $\mathcal{D}_{una}$ holds the *unique name axioms* for actions

of the form $A(\vec{x}) = A(\vec{y}) \supset \vec{x} = \vec{y}$[1], i.e. two distinct actions A and A' are equal iff they have the same name and the same parameter vectors. Finally, $\mathcal{D}_{S_0}$ contains axioms that describe the initial situation.

2.2. Golog

Golog (an abbreviation for al*GOL* in l*OG*ic) is a logic-based programming language that has been designed by (Levesque et al., 1994) based on the definitions of situation calculus. In addition to the execution of simple actions, Golog allows the definition of control structures (e.g. iteration and conditionals) as known from most of the common programming languages. Moreover, Golog allows the definition of procedures and introduces programming constructs to express non-determinism. A Golog program is expanded into a situation calculus formula and Golog language constructs like *while* and *if* can be seen as abbreviations for logical expressions of situation calculus. (Levesque et al., 1994) define a special abbreviation *Do* such $Do(\delta, s, s')$ holds if s' is a legal terminating situation of a given program δ in a starting situation s. Consequently, running a Golog program δ becomes a theorem proving task, i.e. the entailment $\mathcal{D} \models Do(\delta, S_0, do(\vec{a}, S_0))$ needs to be established. A sequence of actions that establish this entailment can be extracted from the proof as the *binding* for the situation term s' given by a successful proof.

2.3. Extensions of Golog

Based on Golog, (De Giacomo et al., 2000) proposed an extension that adds concurrent programming constructs to Golog. Consequently, this extension was named *ConGolog*, which stands for *Concurrent* Golog. Moreover, (De Giacomo et al., 2000) switched from a so-called *evaluation semantics* (i.e. evaluate $Do(\delta, s, s')$ for a complete program δ) to a so-called *transition semantics*, which is based on defining single steps of program execution in contrast to directly defining complete computations. More precisely, (De Giacomo et al., 2000) use two semantic predicates *Trans* and *Final* to specify transition of program states (or *configurations*) as well as legal termination states. We defer the exact definitions of the transition semantic predicates *Trans* and *Final* to Chapter 7 since any further details are of minor importance as of now.

Based on *ConGolog*, (De Giacomo et al., 2009) proposed IndiGolog (incremental deterministic Golog), which introduces *online execution* to allow the programmer to control the amount of planning performed and *sensing* to gain information about an agent's environment. Introducing online execution semantics is particularly interesting since it is a different mode of execution compared to the traditional offline execution semantics from Golog and ConGolog. Using offline execution semantics means that an executor must search over the whole program to find a legal sequence of actions before executing anything. This can be highly problematic when it comes to execution performance and reactivity of an agent. Therefore, IndiGolog introduces an *online execution semantics* that allows a program to execute actions without doing reasoning beforehand. However, this online execution semantics also comes with drawbacks, most importantly that if an action has been executed in the real world there might be no possibility to backtrack if - for example - a non-deterministic program construct has been resolved incorrectly. Consequently, online execution may fail in cases where offline execution may succeed. To control the balance between online and offline execution (De Giacomo et al., 2009) present the *search operator* Σ. The search operator applied to a program δ emulates offline execution for that given part of the program, i.e. it reasons offline to find a valid execution trace for the program δ before executing it.

For the specification of IndiGolog (De Giacomo et al., 2009) use the same transition semantics machinery that proved to be viable for the specification of ConGolog, which is particularly important for this thesis since the semantics of YAGI program execution is built on language constructs from IndiGolog and similar transition semantics predicates are used to show the relation between YAGI and IndiGolog.

[1] We use the superset operator $\supset$ to denote implications (if not explicitly stated otherwise) to stay consistent with the notation from (Reiter, 2001).

In addition to ConGolog and IndiGolog various other extensions of Golog such as DTGolog (Boutilier et al., 2000), sGolog (Lakemeyer, 1999), Legolog (Levesque and Pagnucco, 2000), ccGolog (Grosskreutz and Lakemeyer, 2003) and Readylog (Ferrein, 2007) exist. DTGolog focuses on the integration of decision theoretic planning (hence the name *DT*Golog) using Markov decision processes (MDPs) whereas sGolog incorporates sensing actions using *action trees* instead of linear action histories. Legolog illustrates how Golog can be used to control a LEGO® MINDSTORMS™ robot, including features like exogenous actions and sensing. ccGolog introduces the notion of continuous change to be able to evaluate fluents using a time scale, i.e. continuous fluents are fluents that are enhanced with a temporal component. The focus of Readylog is to combine various features of different Golog-like languages under the constraint of being viable for dynamic real-time domains such as robotic soccer.

2.4. Beyond Prolog-based Implementations of Golog

The idea to use a language other than Prolog to implement a Golog-like language was already discussed by (Ferrein, 2010) and (Ferrein and Steinbauer, 2010). (Ferrein, 2010) sketched *golog.lua*, an implementation of a Golog interpreter using the scripting language Lua. The decision to use Lua instead of Prolog was driven by the fact that Prolog was not available on their target platform (the robot platform Nao) and Lua offers several advantages like easy interfacing with C/C++ code-bases. Driven by similar motivations like lack of a Prolog interpreter for a specific platform and resource limitations (Ferrein and Steinbauer, 2010) extended the vanilla Golog implementation of *golog.lua* with features of IndiGolog with the specific goal to target the LEGO® MINDSTORM™ NXT platform. Besides those Lua-based implementations there also exists an implementation of Golog in Python called *pygolog* contributed by Ferri Federico from University La Sapienza of Rome[2].

The idea of implementing Golog as an answer set program (ASP) (Lifschitz, 2008) is discussed by (Ryan, 2014). Especially, (Ryan, 2014) discusses different encodings of Golog for ASP and shows how different encodings impact the run-time of the resulting ASP. Further, he shows how compilation of Golog programs to finite state machines further increases run-time performance.

2.5. YAGI

This thesis is based on the initial proposal of the YAGI language from (Ferrein et al., 2012). In the initial proposal (Ferrein et al., 2012) stated some functional and non-functional requirements that should be considered when designing a new action-based programming language and provided a first example of YAGI source code to illustrate the intentions behind YAGI. Further, (Ferrein et al., 2012) provided a definition of syntax and semantics of YAGI that we use as a foundation throughout this thesis. Moreover, (Ferrein et al., 2012) presented a basic action theory for YAGI called YAGI-BAT as

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{pres} \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{unc}.$$

Based on this YAGI-BAT we define a basic action theory $\mathcal{D}_{YAGI}$ as

$$\mathcal{D}_{YAGI} = \Sigma \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{unc},$$

where $\Sigma, \mathcal{D}_{ssa}, \mathcal{D}_{ap}, \mathcal{D}_{una}$ and $\mathcal{D}_{S_0}$ contain the same axioms as defined by (Reiter, 2001) and $\mathcal{D}_{unc}$ contains the unique-name axioms used to represent YAGI string tokens. $\mathcal{D}_{YAGI}$ is a reduced version of YAGI-BAT in a sense that (Ferrein et al., 2012) also added the set of Presburger arithmetic axioms (Enderton, 2001) $\mathcal{D}_{pres}$ to represent basic arithmetic operations, which we omit for the sake of simplicity.

[2] Source can be found at `https://github.com/fferri/pygolog`. Last visited on January 15th, 2015.

2.6. Basic Action Theories and Relational Databases

Basic action theories relate closely to the concept of relational databases. More precisely, the idea to use relational databases to represent the initial situation and to use operations that change the database to represent precondition and successor state axioms have been discussed by (Reiter, 2001). Based on the observation that basic action theories are closely related to relational databases, (De Giacomo and Palatta, 2000) illustrated how one can build a system for reasoning about actions exploiting relational database semantics. They illustrated how system states can be represented using database tables, how formulas can be translated to SQL queries and how actions that change the database can be encoded as SQL statements. (De Giacomo and Mancini, 2004) further distinguish *safe* and *unsafe* situation calculus basic action theories and show how efficient reasoning for both cases is possible using relational databases.

2.7. Sensing and Incomplete Information

One can imagine that there exist various cases where one wants to express information that is not yet known, but may (or may not) be *sensed* to its actual value during the lifetime of the agent. Due to the fact that sensing and incomplete information in YAGI are subject to future work we just briefly want to mention some related work regarding these topics. Various approaches of how to deal with incomplete information in different contexts have been discussed by (Etzioni et al., 1992), (Petrick and Bacchus, 2002), (Petrick and Bacchus, 2004), (Vassos and Levesque, 2007) and others. (Etzioni et al., 1992) proposed *UWL*, an extension of STRIPS (Fikes and Nilsson, 1972) and illustrated how UWL can be used to deal with incomplete information in a UNIX operating system domain. (Petrick and Bacchus, 2002) discuss a "knowledge level" approach for planning with incomplete knowledge and sensing, that is they represent incomplete knowledge as formulas of first-order modal logic and represent actions as updates of these formulas. Further, they present the PKS (Planning with Knowledge and Sensing) system that illustrates their approach of planning in the presence of incomplete knowledge and sensing. (Petrick and Bacchus, 2004) present further extensions like numerical evaluation of the PKS system. (Vassos and Levesque, 2007) discuss progression of situation calculus basic action theories with a restricted form of incomplete information. Therefore, they introduce the notion of *possible values*, that is they assume that for every functional fluent they may not now which value it has, but they *do* know the set of possible values for each functional fluent. Based on this assumption, they show that progression of restricted basic action theories with incomplete information is possible.

The problem of integrating sensing information into an action-based system was discussed by (De Giacomo and Levesque, 1999b), (De Giacomo and Levesque, 1999a) and (De Giacomo et al., 2009). In the context of planning and sensing (Levesque, 1996) introduced *sensed fluent axioms* of the form $SF(a,s) \equiv \phi_a(s)$, where *SF* is a distinguished predicate like *Poss*, relating an action to a fluent. For example, (Levesque, 1996) uses an airport scenario that shows how the action of checking a departure screen is connected to knowing where a certain plane is parked as $SF(check\_departures,s) \equiv Parked(Flight123,gateA,s)$. In other words, $\phi_a(s)$ gets asserted to a truth value by its corresponding sensing action. The basic action theory is therefore extended with the set of sensed fluent axioms $\mathcal{D}_{SF}$ and the task is to show that $\mathcal{D} \cup \mathcal{D}_{SF} \models \phi[s']$ for a goal formula ϕ in a situation s'.

Further, (Scherl and Levesque, 2003) specified *sensing result axioms* of the form $SR(\alpha(\vec{x}),s) = r \equiv \phi_\alpha(\vec{x},r,s)$, with α being the name of the action, $\vec{x}$ being the parameter vector, r being the result and s being the situation term. For example, (Scherl and Levesque, 2003) show a sensing result axiom to obtain information about the weather as $SR(sense\_weather,s) = r \equiv (r = "sunny" \vee r = "rainy" \vee r = "snow") \wedge weather(s) = r$. Note that the distinction between *SF* and *SR* is similar to the distinction between relational and functional fluents.

2.8. Other Approaches for Reasoning About Actions

Besides Golog and its dialects there exist other well-working approaches that deal with the task of reasoning about actions and change, among them are languages like fluent calculus (Thielscher, 1998), 3APL (Hindriks et al., 1999), $\mathcal{A}$ (Gelfond and Lifschitz, 1993) and its successors $\mathcal{B}$ and $\mathcal{C}$ (Gelfond and Lifschitz, 1998) as well as PREGO (Belle and Levesque, 2014).

The purpose of fluent calculus is to not only solve the *representational* frame problem (i.e. the problem of specifying all non-effects of actions) but also the *inferential* frame problem, i.e. the problem of inferring all these non-effects. Therefore, fluent calculus introduces the concept of so-called *state update axioms* to specify how an action modifies a state. A *state* relates a situation to the state of the world in that situation, that is, a *state* is the union of all relevant fluents that hold in a situation. This relation is reflected by a function $State(s)$ that relates a situation *s* with a corresponding state. Based on fluent calculus, (Thielscher, 2002) provides a programming method called FLUX, which stands for *FLU*ent e*X*ecutor.

3APL (pronounced "triple a-p-l") is an agent programming language that combines elements from imperative and logic programming. 3APL is based on the metaphor of *intelligent agents*, where (Hindriks et al., 1999) state three main properties that define an intelligent agent, namely *a*) having a complex internal mental state *b*) having the ability to act pro-actively and reactively and *c*) having the capability to reason reflectively. They show how 3APL supports those three properties and specify the semantics of 3APL using a transition-style semantics. The main difference between 3APL and ConGolog is that (Hindriks et al., 1999) argue that the execution model of 3APL is more dynamic compared to ConGolog's approach to extract a sequence of primitive actions from a given high-level program.

Finally, PREGO is a language based on situation calculus that focuses on dealing with uncertainty and noise (e.g., noisy sensors), which are constant companions in real-world robotics applications. Therefore, (Belle and Levesque, 2014) study formal and computational properties of projection when a *degree of belief* is incorporated.

Chapter 3

YAGI Software Architecture Specification

In this chapter we will define the software architecture of a YAGI-based software system. The goals of the architecture are to decouple the syntax and semantics of the language as cleanly as possible, provide easy extensibility and a clear separation of concerns for easy maintainability. Again, we want to emphasize that Golog is typically implemented as a set of Prolog clauses and Golog programs run inside a Prolog environment. As a consequence, the distinction between what are features of Golog and what are side-effects of Prolog is challenging for novices and sometimes even for experts. Due to this fuzzy separation of semantics writing correct Golog programs can be challenging and error-prone. Moreover, the tight coupling of Golog to Prolog is also problematic from a purely technical point of view since a Prolog interpreter might not always be feasible due to resource constraints and/or performance limitations of a target system and - even if a Prolog implementation exists - integrating a Golog interpreter into a rather complex technical ecosystem might be a non-trivial task.

To overcome these issues, we propose a 3-tier layered architecture for YAGI applications that allows a clear separation of syntax and semantics and that can be adopted for a variety of different target systems depending on the practical needs of a certain problem domain. We start with an overview of our architecture in Section 3.1 and proceed with the description of the purpose of each of the three layers, i.e. front-end (Section 3.2), back-end (Section 3.3) and system interface (Section 3.4). We finish this chapter with a description of the inter-layer communication between the formerly specified layers in Section 3.5.

3.1. Architecture Overview

In this section, we give an overview of the 3-tier architecture for YAGI applications, which consists of the following three layers:

1. **Front-End**: The front-end provides the YAGI user interface (UI) as well as the parser for YAGI source code. The front-end handles YAGI source code on a purely syntactical level, i.e. it is responsible for checking syntactical correctness of YAGI code entered by a user. Further, the front-end transforms YAGI code into a suitable intermediate representation that allows the back-end to process the YAGI program efficiently. The user interface can vary depending on the specific needs of a specific implementation, one can imagine a wide range of different user interfaces, from simple console-based interfaces to graphical user interfaces for mobile devices such as mobile phones or tablets.
2. **Back-End**: The back-end consumes data from the front-end, i.e. the front-end passes an abstract representation of a YAGI program (the so-called *abstract syntax tree*, or AST) to the back-end. The back-end stores the current state of the world (i.e. the YAGI basic action theory $\mathcal{D}_{YAGI}$) as well

as executable program elements (e.g. procedures, YAGI actions). The back-end handles the YAGI program on a semantics level, i.e. it modifies the YAGI basic action theory and/or the YAGI program elements according to the semantics of the given YAGI program. Further, the back-end responds to the front-end depending on the type of YAGI statement, which we will discuss in detail in Section 3.5.

3. **System Interface**: The system interface provides external data (e.g. data generated due to exogenous events) for the back-end and is responsible for executing actions, i.e. responding to YAGI *signal*-commands.

This 3-tier architecture is illustrated in Figure 3.1.

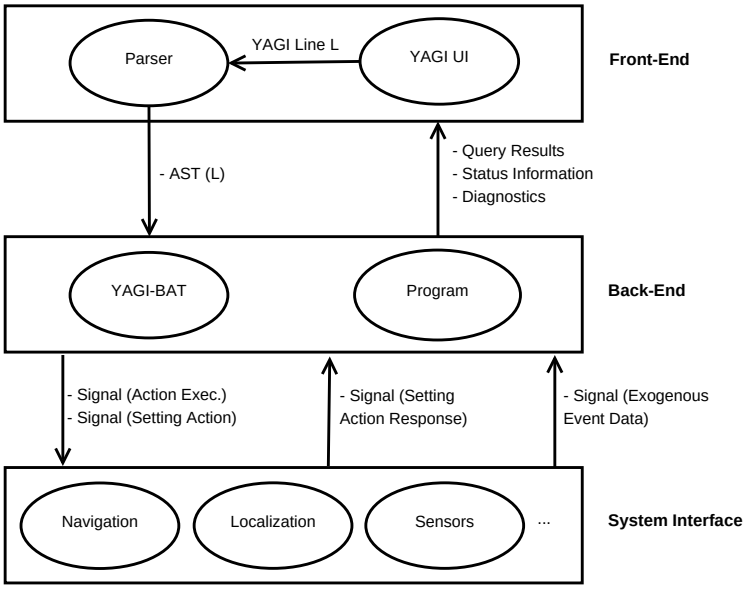

Figure 3.1.: YAGI 3-tier architecture

3.2. Front-End

In this section, the elements of the first layer (front-end) will be discussed.

3.2.1. YAGI User Interface

The YAGI user interface (UI) allows the user to specify a certain YAGI-domain as well as to interact with this domain, e.g. change the YAGI world of fluents or query world information. Depending on the scenario the user interface may vary, one can imagine console-based interfaces similar to console-based editors like *vi* or *emacs*, graphical user interfaces similar to popular integrated development environments (IDEs) like *Eclipse* or *Net Beans* or applications for mobile devices like mobile phones or tablets.

3.2.2. YAGI Parser

The YAGI parser transforms YAGI input source code into an abstract model of the input, i.e. into an abstract syntax tree (AST). This abstract representation ensures syntactical correctness of the input and can be used for further processing.

3.3. Back-End

In this section, the elements of the second layer (back-end) will be discussed.

3.3.1. YAGI Basic Action Theory

The back-end stores the state of the YAGI world, i.e. the YAGI basic action theory $\mathcal{D}_{YAGI}$. The back-end is also responsible for transforming the data provided by the front-end into a reasonable format that allows efficient storing and updating, hence the exact format depends on the specific implementation of the back-end.

3.3.2. Program

The back-end stores executable structures that can modify the state of the world. Executable structures can be rather complex and their execution happens solely in the back-end, hence there need to be a proper representation for these executable structures. Examples for complex structures are procedures, loops and non-deterministic choice of actions, among others defined in Chapter 5. Moreover, the back-end is responsible for properly executing YAGI statements provided by the front-end. *Properly* in this context means that the back-end shall implement the YAGI semantics as specified in Chapter 5.

3.4. System Interface

The system interface serves as the lowest level in the YAGI 3-tier architecture. The system interface varies depending on the area of application, one can think of a range of system interfaces from autonomous robots to video game bots. The purpose of the system interface is to serve as low-level communication layer that executes actions triggered by YAGI *signal*-commands and provides data acquired from external sources for the back-end.

3.5. Inter-Layer Communication

In this section, the communication mechanisms between each of the layers will be discussed.

3.5.1. Front-End $\rightarrow$ Back-End Communication

The front-end passes YAGI lines of code in the form of an abstract syntax tree (AST) to the back-end, i.e. for a YAGI line of code α the parser returns its abstract representation as a function *AST(α)*, which is passed to the back-end for further processing. The types of messages are as follows:

1. **Fluent Query**: The front-end can query information about the current state of the world, i.e. states of fluents.
2. **Program Specification**: The front-end can pass YAGI programs to the back-end. Such programs have no initial effect until they get executed.
3. **Program Execution**: The front-end can pass statements that initiate program execution to the back-end. There are various consequences depending on the program structure that gets executed, e.g. modifications of the state of the world or testing certain conditions. These various effects are described in detail in Chapter 5.

The exact format of the output of *AST(α)* as well as the communication mechanism between front-end and back-end depends on the particular implementation of the system, i.e. may vary depending on the requirements of a specific implementation. We describe our proof-of-concept implementation in Chapter 6.

3.5.2. Back-End → Front-End Communication

The back-end responds to the front-end depending on the type of message as follows:

1. **Fluent Query**: If the front-end queries the state of a fluent, the reply is a set of tuples representing the state of the fluent or *false* if the fluent is not defined.
2. **Program Specification**: The back-end returns *true* iff the program could be stored properly and *false* in any other case.
3. **Program Execution**: The back-end returns information about the program that is being executed. Such information can be status information, data produced by *signal*-blocks of YAGI actions or diagnostics in case any run-time errors occur.

The data exchange format for this part of the communication may also vary between specific implementations.

3.5.3. Back-End → System Interface Communication

The back-end communicates with the system interface via a string signaling mechanism. The content of the string can either be plain text in a natural language or executable code in an arbitrary programming language. This decision depends on how a specific system interface processes the contents of the string signal. Signals can for example trigger an action that executes a real-world action (e.g. motion of an autonomous robot) or query some information about the world (e.g. a synchronous/blocking request to a specific sensor).

The data exchange format as well as how to distinguish between action- and sensing-signals may vary between different implementations of the system.

3.5.4. System Interface → Back-End Communication

The system interface responds to a signal from the back-end accordingly, i.e. providing some status information about an executed action or returning an actively triggered status query result. Moreover, the system-interface asynchronously provides data from exogenous events via call-backs for the back-end. The back-end buffers exogenous event data in a queue if an action is being executed. These buffered values are then consumed by the back-end and the next action is executed.

The data exchange format may vary between different implementations of the system.

Chapter 4

YAGI By Example

In this chapter, we provide an implementation of our object delivery robot running example to illustrate one specific scenario we plan to use YAGI for, including a non-formal description of the intended semantics. We explain our running example in Section 4.1 and proceed with the definition of the fluents and facts in Section 4.2 and Section 4.3, respectively. Then, we describe the YAGI actions in Section 4.4, show the definition of an exogenous event in Section 4.5 and finish our example with the procedure definitions in Section 4.6.

4.1. Running Example

To illustrate our intentions, we use an object delivery robot as running example. The task of the robot is to pickup a certain object from a person, i.e. the sender and deliver it to another person, i.e. the receiver. Every person can possibly reside in a number of different offices, hence it is necessary for the robot to check whether or not the person is actually in the room before picking up/delivering an object.

In the following sections we provide an implementation of our running example written in YAGI. For the sake of simplicity, we define that every variable and every element in a fluent starting with lowercase ***r*** corresponds to a room, lowercase ***o*** corresponds to an object and lowercase ***p*** corresponds to a person.

We proceed with the definition of fluents, facts, actions and procedures including a non-formal description of the intended semantics. A detailed specification of the syntax and semantics of YAGI follows in Chapter 5. The complete listing can be found in Appendix A.

4.2. Fluents

The following listing specifies the fluents for the object delivery robot:

```
//location of the robot (room1, ..., room3)
fluent at [{"r1","r2","r3"}];
at = {<"r1">};

//location of objects (object1 in room1 etc)
fluent is_at[{"o1","o2","o3"}][{"r1","r2","r3"}];
is_at = {<"o1","r1">, <"o2","r2">, <"o3","r3">};

//object carried by robot
```

```
fluent carry[{"o1","o2","o3"}];

//requests moving an object (param 1) from a sender (param 2)
//to a receiver (param 3)
fluent request[{"o1","o2","o3"}][{"p1","p2","p3"}][{"p1","p2","p3"}];

//states what person has been detected in what room
fluent detectedPerson[{"p1","p2","p3"}][{"r1","r2","r3"}];
```

Listing 4.1: Object Delivery Robot Fluents

After the name of every fluent, one or more pairs of brackets follow. The number of pairs define the arity of the fluent. Inside every pair of brackets there need to be the specification of the domain inside a pair of braces, e.g. the fluent *at* has arity one and the domain is the set of available rooms, i.e. {"r1", "r2", "r3"}. Fluents can subsequently be assigned to their initial values. The fundamental type of a fluent assignment is a set of tuples, e.g. the fluent *is_at* is assigned with a set of object-room tuples. A tuple is denoted by enclosing angle brackets, whereas a set is denoted by enclosing braces. We decided to use tuples and sets as our basic concept because we believe that their semantics are widely familiar and easy to understand. Further, the concepts of sets and tuples closely relate to the semantics of relational databases, which is rather convenient as we will describe in detail in Section 6.2.2.

4.3. Facts

The following listing specifies the facts for the object delivery robot:

```
//one or more rooms are assigned to one person,
//i.e. the person's offices
fact office [{"p1","p2","p3"}][{"r1","r2","r3"}];
office = {<"p1","r1">, <"p1","r2">, <"p2","r2">, <"p3","r3">};
```

Listing 4.2: Object Delivery Robot Facts

Facts are similar to fluents, the only difference is that facts can only be assigned once and remain constant after the first assignment. Semantically, their is no difference between a fact and a fluent, its intended purpose is solely to enable a programmer to express constness of certain properties of the world.

4.4. Actions

The following listing specifies the action for moving the object delivery robot to a specific room:

```
//move robot to room $r
action move($r)
precondition:
  //robot is not in room $r
  not (<$r> in at);
effect:
  //now he is in room $r
  at = {<$r>};
signal:
  "Move to room " + $r;
end action
```

Listing 4.3: Object Delivery Robot Move Action

The action uses set-operators to describe the effects of moving a robot to a specific room. Furthermore, the precondition is defined using the binary operator *in*, which evaluates whether or not a concrete object (more specifically, a concrete tuple) is part of a set of tuples. The actions for picking up and putting down an object look similar:

```
//pickup object $o
action pickup($o)
precondition:
  //robot doesn't carry anything and is in the room where the object is
  (not(exists <$x> in carry) and exists <$y> in at such <$o,$y> in is_at);
effect:
  //now he carries $o
  carry += {<$o>};
signal:
  "Pickup object " + $o;
end action
```

Listing 4.4: Object Delivery Robot Pickup Action

```
//putdown object
action putdown($o)
precondition:
  //he carries the object stored in $o
  <$o> in carry;
effect:
  //now he's not
  carry -= {<$o>};

  //where ever it was, its now somewhere else...
  is_at -= {<$o,_>};

  //...namely: here!
  foreach <$r> in at do
    is_at += {<$o,$r>};
  end for

signal:
  "Put down object " + $o;
end action
```

Listing 4.5: Object Delivery Robot Putdown Action

The following listing specifies the action for detecting a person:

```
//"setting" action to detect a person, i.e.
//$p gets its value from an external src
action detectPerson( ) external ($p)
effect:

  //remove person
  detectedPerson -= {<$p, _>};

  //add the detected person + room tuple to the fluent
  foreach <$r> in at do
    detectedPerson += {<$p, $r>};
  end for

signal:
  "detect person";
end action
```

Listing 4.6: Object Delivery Robot Detect Person Action

The listing above illustrates an action that uses external information to *set* the value of a fluent. Consequently, we call these types of actions *setting actions*, denoted by the *external*-modifier in the first line of the action declaration. Note that every variable stated after the *external* keyword gets its value from an external source and can subsequently be used just like any other local variable. In contrast, an action without an *external*-modifier specifies projection effects without using any external data. Moreover, note the usage of the underscore character in the line that removes the detected person from the fluent. The underscore character serves as wild-card, i.e. can be replaced by any possible value of the domain. This feature resembles the pattern matching functionality from functional programming languages like Scala.

4.5. Exogenous Events

Exogenous events differ from *setting actions* in that they can't be actively triggered by a YAGI statement. Exogenous events are triggered by an external event, hence they modify the internal representation of the world based on some external input. The following listing specifies the exogenous event for receiving a request to transport an object from a sender to a receiver:

```
//exogenous event to initiate transportation
//of object $o from $sender to $receiver
exogenous-event receiveRequest($o, $sender, $receiver)
  //add request
  request += {<$o,$sender,$receiver>};
end exogenous-event
```

Listing 4.7: Object Delivery Robot Receive Request Exogenous Event

4.6. Procedures

The following listing specifies the procedure for serving a request:

```
//serves a request
proc serve($object, $sender, $receiver)

  pick <$sender,$roomSender> from office such
    move($roomSender);

    //search for person in the room
    detectPerson();

    //sender is actually in the room
    if (<$sender, $roomSender> in detectedPerson) then
      pickup($object);

      //deliver object to receiver
      pick <$receiver, $roomReceiver> from office such
        move($roomReceiver);

        //search for person in the room
        detectPerson();

        //receiver is actually in the room
        if (<$receiver, $roomReceiver> in detectedPerson) then
          putdown($object);
        end if
      end pick
    end if
  end pick
```

```
end proc
```

Listing 4.8: Object Delivery Robot Serve Request Procedure

Finally, the following listing specifies the main controller of the object delivery robot, which simply serves a randomly picked request:

```
proc main()
  //serve a random request
  pick <$object, $sender, $receiver> from request such
    serve($object, $sender, $receiver);
  end pick
end proc
```

Listing 4.9: Object Delivery Robot Main Procedure

Chapter 5

YAGI Language Specification

In this chapter, we specify the syntax and semantics of the YAGI language. Therefore, we describe the notation we use throughout this chapter in Section 5.1 and continue with the definition of some basic elements of the language in Section 5.2. Then, we define the semantics of YAGI for modeling the state of the world in the context of situation calculus in Section 5.3 and continue with the definition of YAGI program execution in the context of the semantics of IndiGolog in Section 5.4. Furthermore, we line out ideas about incomplete information and sensing in YAGI in Section 5.5 and Section 5.6, respectively. Subsequently, we specify exogenous events in YAGI in Section 5.7 and proceed with the description of some YAGI language elements that are neither related to situation calculus nor IndiGolog in Section 5.9. Finally, we finish this chapter with the definition of a YAGI program in Section 5.10.

5.1. Notation

To specify the syntax we use statements written in Backus-Naur Form (BNF) of the form $\langle a \rangle$ ::= **B**, where non-terminal symbols are denoted by enclosing $\langle$*angle brackets*$\rangle$ and syntactical elements (i.e., terminals) of the language are **bold**. Moreover, we use regular expressions to quantify occurrences of elements in BNF-formulas, applying default semantics of regular expression elements, i.e. one-or-more (+), zero-or-more (*), zero-or-one (?), negation ($\sim$). To specify semantics, we use logical connectives of propositional logic and first-order logic with their conventional meanings. Furthermore, we use the notation and semantics of situation calculus as defined by (McCarthy, 1963) and (Reiter, 2001) to model the state of the world and IndiGolog's programming constructs and their semantics defined by (De Giacomo et al., 2009) to specify program flow.

To specify the semantics of the situation calculus language $\mathcal{L}_{YAGI}$ over the basic action theory $\mathcal{D}_{YAGI}$ we define $\mathcal{L}_{YAGI}$ initially to be empty, i.e. no fluents, actions or constant symbols (except S_0) are defined and $\mathcal{D}_{YAGI}$ to only contain domain-independent information, i.e. everything except Σ and $\mathcal{D}_{unc}$ is empty. We consider this initial state as the interpretation of an empty YAGI program *null*[1]. For arbitrary sequences of YAGI lines of code $\langle l_1, \ldots, l_n \rangle$ the the language $\mathcal{L}_{YAGI}{}'$ over the resulting theory $\mathcal{D}_{YAGI}{}'$ is obtained by modifying their respective predecessors $\mathcal{L}_{YAGI}$ over $\mathcal{D}_{YAGI}$ obtained by the YAGI lines of code $\langle l_1, \ldots, l_{n-1} \rangle$, depending on the type of YAGI language construct of line l_n as specified in the following sections.

[1] We define *null* to be the empty YAGI program to avoid confusion with the empty Golog program, which is often denoted as *nil*.

5.2. Basic Language Elements

To be able to specify the semantics of the YAGI language (i.e. the mapping to situation calculus sentences and IndiGolog programming constructs) we need to briefly define a set of basic language elements that will be used throughout this chapter.

5.2.1. String

⟨*string*⟩ ::= **"**(∼ (**"** | *//*))***"**

Defines a valid sequence of characters, i.e. every concatenation of characters (except double quote and double slash) surrounded by a leading and trailing double quote character.

5.2.2. List of Strings

⟨*string_list*⟩ ::= ⟨*string*⟩(**,**⟨*string*⟩)*

A sequence of strings.

5.2.3. Identifier

⟨*id*⟩ ::= (**a**...**z**|**A**...**Z**)(**a**...**z**|**A**...**Z**|**0**...**9**|**_**)*

An identifier has no standalone semantics, it solely specifies the structure that a valid name of an entity in the YAGI language must fulfill. Under certain conditions, identifiers must be unique, e.g. the names of two different actions must not be equal. The exact conditions of name uniqueness will be discussed in later sections.

5.2.4. Variable

⟨*var*⟩ ::= **$** ⟨*id*⟩

Defines an identifier to which a value can be assigned to.

5.2.5. List of Variables

⟨*var_list*⟩ ::= ⟨*var*⟩ (**,** ⟨*var*⟩)*

A sequence of variables.

5.2.6. Value

⟨*value*⟩ ::= ⟨*string*⟩|⟨*var*⟩

A value is a shortcut for something that is either a string or a variable.

5.2.7. Value-Expression

⟨*valexpr*⟩ ::= ⟨*value*⟩ ((**+**) ⟨*value*⟩)*

Addition of two values. Due to the fact that variables can only hold string values (as specified in Section 5.3.4) each such expression ultimately boils down to an operator + being applied to string elements. Hence, we can define the semantics of a value expression as the concatenation of character sequences.

5.2.8. Tuple

⟨*tuple*⟩ ::= **<** (⟨*tuple_val*⟩ (**,** ⟨*tuple_val*⟩)*) | ε **>**

⟨*tuple_val*⟩ ::= ⟨*var*⟩ | ⟨*string*⟩ | ***** | **_**

Defines a (possibly empty) mathematical tuple $\langle x_1, \ldots, x_n \rangle$. Possible elements in such a tuple can be variables, strings, the star character (*****), which denotes *incomplete information* (as discussed in Section 5.5) and the underline character (**_**), which denotes pattern matching (as discussed in Section 5.3.5).

5.2.9. Set

⟨*set*⟩ ::= **{** (⟨*tuple*⟩ (**,**⟨*tuple*⟩)*) | ε **}**

Defines a finite (possibly empty) mathematical set $\{\langle x_1^1, x_1^2, \ldots, x_1^j \rangle, \ldots, \langle x_n^1, x_n^2, \ldots, x_n^k \rangle\}$ of tuples.

5.2.10. Set-Expression

⟨*setexpr*⟩ ::= ⟨*set*⟩ ((**+** | **-**) ⟨*set*⟩)*

Defines the set-based union and complement, i.e. let A and B be sets, then the YAGI expression `A + B` denotes the union $A \cup B$ and `A - B` denotes the complement $A \setminus B$. For the sake of conformance to the majority of well-known general purpose programming languages we define that both operators + and − have the same precedence and are both left-to-right associative.

5.3. YAGI and Situation Calculus

In this section, we describe how YAGI is mapped to elements of the situation calculus to model the state of the world.

5.3.1. Fluent Declaration

Syntax

⟨*fluent_decl*⟩ ::= **fluent** ⟨*id*⟩ (**[** (**String** | **{** ⟨*string_list*⟩**}**) **]**)* **;**

Semantics

Let l_n be a YAGI fluent declaration of a fluent F with arity m, where m denotes the number of square bracket pairs following the name of the fluent. Each pair of square brackets define the domain of its corresponding dimension, i.e. we say that the n-th dimension of fluent F with arity m and $0 \leq n \leq m$ has domain $\mathcal{S}_F^n$. Furthermore, we say that a fluent F has domain $\vec{\mathcal{S}_F}$, i.e. the fluent has domain $\mathcal{S}_F^1$ in dimension one, $\mathcal{S}_F^2$ in dimension two and so on, and $\vec{\mathcal{S}_F} = \langle \mathcal{S}_F^1, \mathcal{S}_F^2, \ldots, \mathcal{S}_F^m \rangle$. The sort of the n-th dimension of a fluent is defined as follows, where the term *sort* is used as in many-sorted first order languages and will from now on be used equivalently to the term *domain*. The sort string represents the countably infinite set of possible character sequences, i.e. the Kleene closure V^* over the alphabet $V = \{\text{A}\ldots\text{Za}\ldots\text{z1}\ldots\text{9\_}\}$. The axiomatization is achieved by mapping every string value to a constant with the same name as the value of the string and providing corresponding unique-name axioms for these constants. We call this set of axioms $\mathcal{D}_{unc}$. Note that the domain can either be the full range of V^* (denoted by `[String]`) or any finite subset $\{s_1, \ldots, s_n\} \subset V^*$ denoted by the enumeration of all the valid elements, i.e. `[{"s_1",...,"s_n"}]`.

The declaration of a fluent F with arity m extends $\mathcal{L}_{YAGI}$ by adding the corresponding $(m+1)$-ary predicate $F(\vec{x}, s)$ and two m-ary action symbols $addF(\vec{x})$ and $removeF(\vec{x})$, leading to $\mathcal{L}_{YAGI}{}'$, where $\vec{x}$ denotes the vector of fluent arguments $(x_1, \ldots, x_m)$ and s denotes the situation term. $\mathcal{D}_{YAGI}{}'$ is the same as $\mathcal{D}_{YAGI}$ except that all sentences that mention F, *addF* or *removeF* in $\mathcal{D}_{S_0}$, $\mathcal{D}_{ssa}$ and $\mathcal{D}_{ap}$ are removed and the axiom $\forall \vec{x}.F(\vec{x}, S_0) \equiv false$ is added to $\mathcal{D}_{S_0}$. This can be considered as some form of initialization of the theory for the fluent F. Moreover, the axiom $F(\vec{x}, do(a, s)) \equiv a = addF(\vec{x}) \vee F(\vec{x}, s) \wedge a \neq removeF(\vec{x})$ is added to $\mathcal{D}_{ssa}$. The purpose of the situation calculus simple actions *addF* and *removeF* are to make the fluent F *true* (or *false*, respectively) for a given parameter vector $\vec{x}$. Note that each element in $\vec{x}$ is an instantiation of an element of the sort of the corresponding dimension in the fluent declaration, i.e. $x_1 \in \vec{x}$ has domain $\mathcal{S}_F^1$ (the sort of the first dimension of the fluent declaration for the fluent F), $x_n \in \vec{x}, n \leq m$ has domain $\mathcal{S}_F^n$ and so on. To enforce this correspondence, we add the necessary preconditions $Poss(addF(\vec{x}), s) \equiv \bigwedge_{i=1}^{m} \tau(\mathcal{S}_F^i, x_i)$ and $Poss(removeF(\vec{x}), s) \equiv \bigwedge_{i=1}^{m} \tau(\mathcal{S}_F^i, x_i)$ to $\mathcal{D}_{ap}$[2], with $\tau(\mathcal{S}_F^i, x_i)$ being a binary predicate that holds iff x_i is an element of its corresponding sort $\mathcal{S}_F^i$. Also note that the initial database $\mathcal{D}_{S_0}$ is in closed form, according to the definition from (Reiter, 2001).

5.3.2. Fact Declaration

Syntax

⟨*fact_decl*⟩ ::= **fact** ⟨*id*⟩ (**[** (**String** | **{** ⟨*string_list*⟩**}**) **]**)* **;**

Semantics

The semantics of ⟨*fact_decl*⟩ is identical to the semantics of ⟨*fluent_decl*⟩, the only difference is that a fact can only be assigned once and becomes immutable after it has been assigned for the first time. According to this definition, we can simplify the underlying theory for facts by omitting the definitions of the situation calculus actions *add* and *remove* for each declared fact. This leads to a theory that makes the constness of facts more explicit since there exists no mechanism in the theory that is able to modify a fact. Initialization of facts is implemented by directly updating $\mathcal{D}_{S_0}$, i.e. adding $F(\vec{x}, S_0) \equiv \vec{x} = \vec{x_1} \vee \vec{x} = \vec{x_2} \vee \ldots \vee \vec{x} = \vec{x_n}$ for the fact F and the n parameter vectors which are used to initialize the fact. Further, we define that $F(\vec{x_1}, S_0) \equiv true$ and $\forall \vec{x}.F(\vec{x}, S_0) \equiv false$ for the special cases for $n = 1$ and $n = 0$, respectively.

Implementation Remarks

Any implementation shall check that there is no assignment to a *fact* after its initialization. Any further left-hand side appearance of a *fact* in an assignment shall lead to an error. Moreover, any implementation

[2] In the special case of a fluent F having arity 0, the action preconditions are defined as $Poss(addF, s) \equiv true$ and $Poss(removeF, s) \equiv true$.

shall ensure that a fact is subsequently assigned after its declaration, i.e. let l_n be a YAGI line of code that declares a fact, then the subsequent line l_{n+1} must be the initialization of the formerly declared fact. Any other type of statement shall lead to an error.

5.3.3. Formulas

Syntax

⟨*formula*⟩ ::= ⟨*atom*⟩
| **not (** ⟨*formula*⟩ **)**
| **(** ⟨*atom*⟩ ⟨*connective*⟩ ⟨*formula*⟩ **)**
| **exists** ⟨*tuple*⟩ **in** ⟨*setexpr*⟩ (**such** ⟨*formula*⟩)?
| **all** ⟨*tuple*⟩ **in** ⟨*setexpr*⟩ (**such** ⟨*formula*⟩)?
| ⟨*tuple*⟩ **in** ⟨*setexpr*⟩

⟨*atom*⟩ ::= ⟨*value*⟩ ⟨*comp_op*⟩ ⟨*value*⟩
| ⟨*setexpr*⟩ ⟨*comp_op*⟩ ⟨*setexpr*⟩
| (**true** | **false**)

⟨*comp_op*⟩ ::= == | **!=** | <= | >= | < | >

⟨*connective*⟩ ::= **and** | **or** | **implies**

Semantics

Instances of ⟨*formula*⟩ evaluate to a logical truth value. The elements allowed in such a first-order formula have the following semantics:

- **Truth Values**: **true** is *true*, **false** is *false*.
- **Comparisons**: On string values, two elements s_1 and s_2 are considered equal iff they have the same length and each character at the same position in both s_1 and s_2 are equal. If this equality relation holds the operator == returns *true*, otherwise it returns *false*. Consequently, the operator **!=** is the negation of ==. The ordering comparisons <=, >=, < and > are performed lexicographically. On sets, comparisons are element-based, i.e. two sets A and B are equal iff every element of A is in B and vice versa. Order comparisons are mapped to (proper) subset/superset relations, i.e. let X and Y be sets, then `X < Y` is *true* iff X is a proper subset of Y, i.e. $X \subsetneq Y$. Consequently, the operator > denotes the proper superset $\supsetneq$. The operators <= and >= follow intuitively as subset and superset without the strictness property, i.e. $\subseteq$ and $\supseteq$.
- **Logical Connectives**: The logical connectives **and** ($\wedge$), **or** ($\vee$) and **implies** ($\rightarrow$) have their usual meanings.
- **Negation**: The operator **not** ($\neg$) negates the truth value.
- **First-Order Quantifiers**: The operators **all** ($\forall$) and **exists** ($\exists$) have their usual meanings. Note that they operate on the sorts of the respective ⟨*setexpr*⟩, i.e. let F be a fluent of sort $\vec{S}_F$, then **exists** <$xj_1,$xj_2,...,$xj_n> **in** F translates to $\exists_{S_1^F} x_{j_1} \exists_{S_2^F} x_{j_2}, \ldots, \exists_{S_n^F} x_{j_n}.F(x_{j_1}, \ldots, x_{j_n}, s)$, where $\exists_{S_n^F}$ is the existential quantifier over the sort of the n-th dimension of fluent F. The **all**-quantifier follows similarly, with $\forall_{S_n^F}$ being the universal quantifier over the sort of the n-th dimension of fluent F. Note that the YAGI variables <$xj_1,$xj_2,...,$xj_n> must be *fresh* in a sense that they must not be bound to a value before they are used in an **all** or **exists** statement. The optional **such** ⟨*formula*⟩ is connected to the quantified formula either via a logical conjunction (in case of an existential quantifier, i.e. **exists** <$x> **in** F **such** <formula> translates to $\exists_{S_1^F} x.F(x,s) \wedge \varphi$) or a logical implication (in case of an all

quantifier, i.e. `all <$x> in F such <formula>` translates to $\forall_{S_1^F} x.F(x,s) \rightarrow \varphi$), where φ corresponds to a YAGI ⟨*formula*⟩ instance. Note that in case no **such**-block is present the semantics of **all** and **exists** are identical, i.e. `exists <$x> in F` and `all <$x> in F` hold iff there is at least one element for which the fluent F holds.

- **Operator in**: The keyword **in** is used to specify the domain of discourse when used with a first-order quantifier as defined above. Moreover, **in** can also be used without a quantifier, which changes its semantics as follows. `<$x1,$x2,...,$xn> in F` translates to $F(x_1,x_2,\ldots,x_n,s)$, i.e. the truth value of the Fluent F in situation s is evaluated for concrete elements $\langle x_1,\ldots,x_n\rangle$. Note that - contrary to YAGI variables used with first-order quantifiers as discussed above - the variables `<$x1,$x2,...,$xn>` must be bound to a value before being used on the left-hand side of the standalone operator **in**.

Implementation Remarks

Any implementation shall report different errors based on the following scenarios:

- **First-Order Quantification With Bound Variables**: Anything but unbound variables used in a first-order quantified formula shall result in an error, e.g.

```
fact floors[{"0","1","2","3","4","5","6"}];
floors = {<"0">, <"1">, <"2">, <"3">, <"4">, <"5">, <"6">};

exists <$x> in floors such $x < "1";    //valid, evaluates to 'true'

$y = "4";
exists <$y> in floors such $x < "1";    //invalid, $y is already bound

exists <"0"> in floors such "2" < "1";  //invalid, "0" is a constant
```

Listing 5.1: First-Order Quantification Examples

- **Unbound Variables on the Left-Hand Side of the Standalone Operator *in***: Any use of an unbound variable on the left-hand side of the operator *in* shall result in an error, e.g.

```
fact floors[{"0","1","2","3","4","5","6"}];
floors = {<"0">, <"1">, <"2">, <"3">, <"4">, <"5">};

$y = "6";
<$y> in floors; //valid, $y is already bound; evaluates to 'false'

<"0"> in floors; //valid, evaluates to 'true'

<$x> in floors; //invalid, $x is unbound
```

Listing 5.2: Operator 'in' Examples

5.3.4. Assignment

Syntax

⟨*assignment*⟩ ::= ⟨*assign*⟩ **;**
| ⟨*for_loop_assign*⟩
| ⟨*conditional_assign*⟩

⟨*assign*⟩ ::= ⟨*var*⟩ **=** ⟨*value*⟩

| ⟨*id*⟩ (= | += | -=) (⟨*id*⟩ | ⟨*setexpr*⟩)

⟨*for_loop_assign*⟩ ::= **foreach** ⟨*tuple*⟩ **in** (⟨*id*⟩ | ⟨*setexpr*⟩) **do** ⟨*assignment*⟩$^+$ **end for**

⟨*conditional_assign*⟩ ::= **if** ⟨*formula*⟩ **then** ⟨*assignment*⟩$^+$ (**else** ⟨*assignment*⟩$^+$)? **end if**

Semantics

The simplest case of ⟨*assign*⟩ is an assignment of a value to variable, which simply binds a single value to the name of the variable. Note that variables can only hold simple values, i.e. instances of sort string. Assigning more complex structures (i.e. tuples and sets) to variables is not permitted. Since this type of assignment solely maps a value to a name it has no influence on $\mathcal{D}_{YAGI}$ or $\mathcal{L}_{YAGI}$.

The second base case of ⟨*assign*⟩ is the assignment to an ⟨*id*⟩, i.e. the assignment to a fluent F'. Due to the fact that either another fluent F_σ or set of constants $\sigma_F = \{\langle x_1^1, x_1^2, \ldots, x_1^k\rangle, \ldots, \langle x_n^1, x_n^2, \ldots, x_n^k\rangle\}$ can be assigned to a fluent F' we need to look at both of these cases separately. In the first case, we already have a situation calculus representation we can use to formalize the assignment since the fluent F_σ must have been declared first. In the second case we need to construct a situation calculus representation from the set of constants σ_F, as follows. We transform σ_F to what we call a *shadow fluent*. A *shadow fluent* is the situation calculus representation of σ_F, i.e. a fluent $F_\sigma(\vec{x}, s)$ is created with $\vec{x}$ according to the elements in σ_F and s as situation term, the axiom $F_\sigma(\vec{x}, S_0) \equiv \vec{x} = \vec{x_1} \vee \vec{x} = \vec{x_2} \vee \ldots \vee \vec{x} = \vec{x_n}$ is added to $\mathcal{D}_{S_0}$ and the successor state axiom $\forall \vec{x}. F_\sigma(\vec{x}, do(a, s)) \equiv F_\sigma(\vec{x}, s)$ is added to $\mathcal{D}_{ssa}$. Note that each $\vec{x}$ in $\forall \vec{x}$ corresponds to one tuple in σ_F and the assignment is only valid iff the arity of the fluents are equal and each element of the assignment belongs to the same domain.

Now, having a situation calculus representation for both of the valid assignment cases, we can proceed with the specification of the assignment semantics. Assignments to fluents expand to YAGI programs as follows. Let F be the fluent at the left-hand side of an assignment and let F_σ be the fluent at the right-hand side of the assignment, then we need to distinguish between the following types of assignment:

- **Add-Assignment**: An add-assignment (assignment operator **+=**) adds all the tuples from F_σ to F, leaving all other elements in F unchanged. That is, if $\mathcal{D}_{YAGI} \models F(\vec{x})$ and $\mathcal{D}_{YAGI} \models F_\sigma(\vec{x})$ then it holds after the assignment `F += F_sigma;` that $\mathcal{D}_{YAGI} \models F_\sigma(\vec{x}) \rightarrow F(\vec{x})$. Consequently, given the YAGI assignment `F += F_sigma;` we can construct a YAGI program as follows:

```
foreach <$x1,...,$xn> in F_sigma do
  addF($x1,...,$xn);
end for
```

  This YAGI-loop essentially adds all the elements from F_σ to the fluent F using the situation calculus simple action *addF*. Recall that the situation calculus simple actions *addF* and *removeF* are created for every YAGI fluent F at its declaration, see Section 5.3.1. The exact semantics of *foreach* (i.e. mapping of a YAGI-*foreach* to IndiGolog) are discussed in Section 5.4.6.

- **Remove-Assignment**: A remove-assignment (assignment operator **-=**) removes all tuples in F_σ from F, leaving all other elements in F unchanged. That is, if $\mathcal{D}_{YAGI} \models F(\vec{x})$ and $\mathcal{D}_{YAGI} \models F_\sigma(\vec{x})$ then the assignment `F -= F_sigma;` leads to $\mathcal{D}_{YAGI} \not\models F(\vec{x})$ if $F_\sigma(\vec{x})$ holds. Similar to the add-assignment, given the YAGI assignment `F -= F_sigma;` we can construct a YAGI program as follows:

```
foreach <$x1,...,$xn> in F_sigma do
  removeF($x1,...,$xn);
end for
```

  Note that the only difference to the YAGI program for the add-assignment is the different situation calculus action *removeF*.

- **Override Assignment**: An override assignment (assignment operator **=**) makes the fluent F *true* for all and only all tuples in F_σ. In other words, an override assignment removes all elements from F and

adds all the tuples from F_σ to it. Consequently, we can express an override assignment as a remove-assignment **F -= F** followed by an add-assignment **F += F**$_\sigma$. Hence, we can specify the override assignment by applying the construction rules for add- and remove-assignment specified above.

Based on the specification of assignments to fluents we can proceed with complex assignment statements. ⟨*for_loop_assign*⟩ defines an iteration over all tuples in one ⟨*setexpr*⟩ with multiple assignments in the loop body. The intention is to provide a convenient way to assign (potentially multiple) fluents to some value that is determined by iterating over a set of tuples. Note that the semantics of the different types of assignments specified above still apply since ⟨*for_loop_assign*⟩ is basically just a less verbose way to formulate a list of consecutive assignments. The mapping of a ⟨*for_loop_assign*⟩ to an IndiGolog program works in the same way as the mapping of a ⟨*for_loop*⟩ discussed in Section 5.4.6, the only difference is that in a ⟨*for_loop_assign*⟩ only multiple instances of ⟨*assignment*⟩ can be executed in the loop body whereas the loop body in a ⟨*for_loop*⟩ consists of an arbitrary ⟨*block*⟩. Due to the fact that we specify that it is not permitted to modify the set the loop iterates over inside the loop body we are not be able to perform rewriting for override assignments since the expression **F -= F** would violate this specification. To avoid this specification violation we remove the modification restriction for loop assignments and specify the execution semantics of assignment for-loops as follows. The ⟨*setexpr*⟩ the assignment loop iterates over is evaluated once (and only once) before the loop gets executed. Using this semantics we can make sure that assignment rewritings for assignments like **F -= F** work correctly.

⟨*conditional_assign*⟩ is driven by a similar motivation as ⟨*for_loop_assign*⟩, i.e. to provide a convenient way to formulate (potentially multiple) assignments based on the evaluation of some ⟨*formula*⟩. One can think of it as conditional branching like *if-then-else* constructs known from most of the common programming languages, with the restriction that in each of the branches the only type of statement allowed is ⟨*assignment*⟩. The mapping to an IndiGolog program works in the same way as the mapping of a ⟨*conditional*⟩ discussed in Section 5.4.4

Having defined the semantics and rewriting rules of YAGI assignment statements we want to emphasize that the most complex construct we get from rewriting is a loop that iterates over a finite set of tuples and performs adding and removing elements to/from fluents. We can guarantee that even at worst we always deal with finite sets of tuples since we specified a set to always contain finitely many elements and any operation that adds elements to a set (i.e. *add-assignment* and operator *plus*) can only occur finitely many times in a program. Hence, any set produced by these operations can only contain a finite number of tuples. Note that the pattern matching extension discussed in Section 5.3.5 does not contradict that observation in any way. This conclusion becomes immensely important for work we plan to do in the near future, which is to prove that one can compile arbitrary YAGI action effects to situation calculus successor state axioms.

Implementation Remarks

Any implementation shall check that assignments of a ⟨*setexpr*⟩ are semantically valid, i.e. that every element in every tuple of the ⟨*setexpr*⟩ is an element of the sort of the corresponding dimension of the fluent at the left-hand side of the assignment. If an ⟨*id*⟩ (i.e., another fluent) is assigned to the fluent at the left-hand side the assignment is only valid if both fluents (left-hand side and right-hand side of the assignment) have the same arity and the same domains in each dimension. Any other case shall lead to an error.

5.3.5. Pattern Matching

For YAGI assignments that contain interactions with sets of any kind we introduce a pattern matching functionality inspired by functional programming languages like Scala. Syntactically, we use *underscore* "_" as wildcard character. The set-theoretic semantics of pattern matching is defined as follows. Let F be a fluent of sort $\vec{\mathcal{S}_F}$ and $\sigma = \{\langle x_1, \ldots, x_n \rangle\}$ the set that is assigned to F using a YAGI assignment operator as specified in Section 5.3.4. Then, it must hold that each element $x_i, 1 \leq i \leq n$ of σ is an element of the sort of the i-th dimension of F, i.e. $x_i \in \mathcal{S}_F^i$ and the number of elements in the n-tuple of σ

must be equal to the number of dimensions of the fluent F. Then, assignment works as specified in the section above. Now let σ' be the same as σ except that the i-th element in the n-tuple of σ is the wildcard character, i.e. $\sigma' = \{\langle x_1, \ldots, x_{i-1}, \_, x_{i+1}, \ldots, x_n \rangle\}$. Now pattern matching applied to σ' leads to $\sigma'' = \{\langle x_1, \ldots, x_{i-1}, \chi_1, x_{i+1}, \ldots, x_n \rangle, \langle x_1, \ldots, x_{i-1}, \chi_2, x_{i+1}, \ldots, x_n \rangle, \ldots, \langle x_1, \ldots, x_{i-1}, \chi_m, x_{i+1}, \ldots, x_n \rangle\}$, i.e. for each of the m elements in the i-th domain of the fluent F a new n-tuple is added to σ'', with the wildcard character replaced with a concrete element $\chi_j \in \mathcal{S}_F^i, 1 \leq j \leq m$. In some sense, such a replacement of a wildcard symbol with a set of concrete elements resembles *grounding* (i.e., replacing a program with variables with an equivalent program without variables) for finite domains in answer set programming (ASP), as discussed by (Gelfond and Lifschitz, 1988) and (Lifschitz, 2008).

The general case with an arbitrary number of wildcards in a single assignment statement follows the same principle, the difference being that the expansions are equal to the Cartesian product of their corresponding domains, e.g. let the wildcard character be present at two arbitrary positions i and j in a tuple that is assigned to a fluent F. Then, pattern matching generates tuples with all elements of $\mathcal{S}_F^i \times \mathcal{S}_F^j$ and all the non-wildcarded elements of the original tuple. Having defined the set-theoretic semantics of pattern matching we want to construct YAGI code that implements the expansion as specified above. To be able to express this in YAGI, we need to introduce a new construct called *shadow fact*.

Shadow Facts

Essentially, *shadow facts* are ordinary YAGI facts as specified in Section 5.3.2, with the additional property that they're internally created and hence not accessible for the developer of a YAGI program. Note that they're conceptually similar to *shadow fluents* as specified in Section 5.3.4. A *shadow fact* is internally created if a fluent is involved in a pattern matching assignment, as follows. Let F be a fluent of sort $\vec{S}_F$ and $\sigma = \{\langle x_1, \ldots, x_{i-1}, \_, x_{i+1}, \ldots, x_n \rangle\}$ a set that should be assigned to F, with the wildcard character at the i-th position. Then, a YAGI fact $\mathcal{F}_i^*$ is created according to the semantics of fluent/fact declaration discussed in Section 5.3.1 (and 5.3.2, respectively) and is subsequently assigned to $\mathcal{F}_i^* = \{\langle x_1 \rangle, \langle x_2 \rangle, \ldots, \langle x_n \rangle\}, \forall x \in \mathcal{S}_F^i$. That is, the shadow fact $\mathcal{F}_i^*$ holds for all values of the sort of the i-th dimension of the fluent F. Note that this assignment can be expressed in YAGI using the rules specified in Section 5.3.4. Having a definition for a *shadow fact*, we can continue with the specification of the YAGI pattern matching expansion.

YAGI Pattern Matching Expansion

Let there be a YAGI assignment of the form

```
fluent F[{"a","b"}];
F += {<_>};  //equal to F = {<"a">, <"b">};
```

Then, we can rewrite the pattern matching expansion as YAGI code of the form

```
foreach <$chi1> in F*_1 do //F*_1 is the 'shadow fact' of dimension 1 of the fluent F.
  F += <$chi1>;
end for
```

The expanded YAGI code essentially iterates over the *shadow fact* of the domain of the fluent mentioned at the left-hand side of the assignment and executes the assignment with each value from the domain. Due to the fact that every *shadow fact* gets assigned with all the values of its corresponding domain and becomes immutable afterwards (according to the definition of a YAGI *fact*) we can argue that the YAGI *foreach* loop from above iterates over the complete domain at any given point in time during program execution. The rewriting for the assignment operators `-=` and `=` follow similarly. The general case follows the same principle, adding one nested loop for every wildcard character in the assignment. Consider an assignment with two wildcard characters at arbitrary positions i and j of the form

```
F += {<$x1,...,_,...,_,...,$xn>};
```

Then, the expansion leads to two nested loops of the form

```
foreach <$chi_i> in F*_i do
  foreach <$chi_j> in F*_j do
    F += {<$x1,...,$chi_i,...,$chi_j,...,$xn>};
  end for
end for
```

Note that these nested loops express exactly the Cartesian product, which we used to specify the set-theoretic semantics of pattern matching. In the general case (i.e. an arbitrary number of wildcards in a single assignment) for each of the wildcards an additional loop that iterates over the corresponding *shadow fact* is added to the nesting as outlined above. Moreover, note that any variables from the original assignment (`$x1` and `$xn` in the example above) remain untouched by the pattern matching rewriting.

For the time being, we restrict pattern matching assignments to fluents that have a user-defined set of strings as domain. This restriction is driven by the fact that in the case that the domain of a fluent is the full (countably infinite) set of strings (as defined in Section 5.3.1) we would induce a loop iterating over countably infinitely many elements, which we are not able to express in YAGI. For the time being, pattern matching over finite domains suffices our needs, even though we plan to loosen that restriction in future work.

Implementation Remarks

The same remarks as for assignments (see Section 5.3.4) apply, i.e. any implementation shall check that assignments of a ⟨*setexpr*⟩ are semantically valid, i.e. that every element in every tuple of the ⟨*setexpr*⟩ is an element of the sort of the corresponding dimension of the fluent at the left-hand side of the assignment. Any mismatch shall lead to an error. Furthermore, any use of the wildcard character outside of an assignment statement shall result in an error. Lastly, any attempt to use pattern matching on a dimension of a fluent that has the countably infinite set of strings as domain shall result in an error.

5.4. YAGI and IndiGolog

In this section, we specify the syntax and semantics of YAGI language constructs that are responsible for program execution. Recall that in the earlier section of this chapter we exclusively specified YAGI constructs responsible for modeling the state of the YAGI world. In the following sections, we specify program execution semantics using programming constructs from IndiGolog. The semantics of IndiGolog's programming constructs has been defined by (De Giacomo et al., 2009), using transition semantic predicates *Trans* and *Final*. Hence, we can map YAGI statements to IndiGolog language constructs to specify their intended semantics. We discuss the relation between IndiGolog transition predicates and YAGI program execution semantics in more detail in Chapter 7 as this relation becomes particularly important when we argue about specification conformance of our implementation.

5.4.1. Test

Syntax

⟨*test*⟩ ::= **test** ⟨*formula*⟩ **;**

Semantics

Tests whether or not a corresponding formula holds. Semantically, it's the counterpart of IndiGolog's *test action* $\phi?$.

5.4.2. Choose

Syntax

⟨*choose*⟩ ::= **choose** ⟨*block*⟩ (**or** ⟨*block*⟩)$^+$

Semantics

Non-deterministically chooses one of the given blocks for execution. Semantically, it's the counterpart of IndiGolog's *nondeterministic branch* $\delta_1 \mid \delta_2$.

5.4.3. Pick

Syntax

⟨*pick*⟩ ::= **pick** ⟨*tuple*⟩ **from** ⟨*setexpr*⟩ **such** ⟨*block*⟩ **end pick**

Semantics

Non-deterministically picks a ⟨*tuple*⟩ from a given ⟨*setexpr*⟩ and executes the subsequent block using the picked tuple as parameter, i.e. non-deterministically take a tuple from ⟨*setexpr*⟩, bind its values to fresh variables in the tuple provided by the ⟨*tuple*⟩-expression and execute the ⟨*block*⟩ with this variable assignment. Any attempt to state something different than a variable in the ⟨*tuple*⟩ of the *pick*-statement is not permitted[3]. Semantically, it's the counterpart of IndiGolog's *non-deterministic choice of argument* $\pi v.\delta$.

Note that besides fresh variables that will be bound to a value by the *pick* statement as described above a ⟨*tuple*⟩ may also contain variables that are already bound to a value. In this case we simply use the already available value instead of binding the variable via the *pick*-statement. To clarify the semantics, we provide examples for the different cases using the domain of our running example, as follows.

```
at = {<"r1">,<"r2">,<"r3">};

//$x is unbound...
pick <$x> from at such move($x); end pick;
//...hence its value after the pick is either "r1", "r2", or "r3"
```

Listing 5.3: Pick With Unbound Variable

```
//Bind $x to a constant
$x = "r1";

//$x is already bound...
pick <$x> from at such move($x); end pick;
//...hence its value stays exactly the same after the pick
```

Listing 5.4: Pick With Bound Variable

[3] One might argue that it could make sense to allow constants and/or pattern matching in the ⟨*tuple*⟩ of pick. Due to the fact that the exact semantics are not yet clear we delay this idea to future work.

```
is_at = {<"o1","r1">, <"o2","r2">, <"o1","r3">};

//Bind $x to a constant
$x = "o1";

//$x bound, $y unbound
pick <$x,$y> from is_at such pickup($x); end pick;
//tuple can be <"o1","r1"> or <"o1","r3">
```

Listing 5.5: Pick With Bound and Unbound Variables

Implementation Remarks

Any implementation shall check that only variables are stated in the ⟨*tuple*⟩-expression. Any attempt of stating a constant shall result in an error.

5.4.4. Conditional

Syntax

⟨*conditional*⟩ ::= **if (**⟨*formula*⟩**) then** ⟨*block*⟩ (**else** ⟨*block*⟩)? **end if**

Semantics

Executes one of two given blocks based on the evaluation of ⟨*formula*⟩. Semantically, it's the counterpart of IndiGolog's *synchronized conditional* **if** ϕ **then** δ_1 **else** δ_2 **endIf**.

5.4.5. While Loop

Syntax

⟨*while_loop*⟩ ::= **while** ⟨*formula*⟩ **do** ⟨*block*⟩ **end while**

Semantics

Executes a block as often as ⟨*formula*⟩ holds. Semantically, it's the counterpart of IndiGolog's *synchronized loop* **while** ϕ **do** δ **endWhile**.

5.4.6. For Loop

Syntax

⟨*for_loop*⟩ ::= **foreach** ⟨*tuple*⟩ **in** ⟨*setexpr*⟩ **do** ⟨*block*⟩ **end for**

Semantics

Executes a ⟨*block*⟩ for every tuple in a given ⟨*setexpr*⟩. Due to the fact that IndiGolog has no specification for this kind of program flow construct, we rewrite ⟨*for_loop*⟩ into ⟨*while_loop*⟩ as follows. Let F be an n-ary fluent and consider

```
foreach <$x1,$x2,...,$xn> in F do
  <block>
end for
```

to be a YAGI for loop over F. We transform this loop into the following YAGI code:

```
F* = F;
while (exists <$x1,$x2,...,$xn> in F*) do
  pick <$x1,$x2,...,$xn> from F* such
    <block>
    F* -= {<$x1,$x2,...,$xn>};
  end pick
end while
```

with F* being a copy of the fluent F. Note that all the identifiers ending with a star (*) are no valid names according to the specification of ⟨*id*⟩. These names were purposely chosen to contradict the syntactical specification to illustrate that these names are chosen internally by the interpreter, i.e. it's syntactically impossible for a programmer to access these elements. The transformation checks if a tuple exists in F* via the *while*-condition and subsequently binds a tuple from F* to the variables `$x1,$x2,...,$xn` via the `pick`-statement, making it semantically equivalent to an execution of the statement **`foreach`** `<$x1,$x2,...,$xn>` **`in`** `F` **`do`**. Then, the same ⟨*block*⟩ as in the for-loop gets executed. Finally, the chosen tuple is removed from the fluent F*. Note that this transformation works correctly if and only if the value of the fluent the *foreach*-loop iterates over is not modified in its loop body, hence we don't permit any modifications of the fluent the *foreach*-loop iterates over in the loop body.

To justify the claim that the rewritten loop above satisfies the specified semantics we argue inductively, as follows:

- For the base case, let F be a fluent that doesn't hold for any parameter vector, i.e. $\forall\vec{x}.F(\vec{x},s) \equiv False$. Since F* is specified to be a copy of the fluent F it also holds that $\forall\vec{x}.F^*(\vec{x},s) \equiv False$. Consequently, the YAGI formula **`exists`** `<$x1,$x2,...,$xn>` **`in`** `F*` evaluates to *False* because it translates to $\exists\vec{x}.F^*(\vec{x},s)$ according to the specification of YAGI formulas in Section 5.3.3. Thus, the while loop becomes **`while`** `(False)` **`do`** and the code in the while-block doesn't get executed.
- For the inductive step we assume that for every fluent with k tuples the transformation is correct. For any arbitrary Fluent F that holds for a set of parameter vectors $S = \{\vec{x_1},\ldots,\vec{x_{k+1}}\}$ the YAGI formula **`exists`** `<$x1,$x2,...,$xn>` **`in`** `F*` evaluates to *True* because it holds that $\exists\vec{x}.F^*(\vec{x},s)$ and `<$x1,$x2,...,$xn>` corresponds to one parameter vector $\vec{x_i} \in S$. Consequently, the ⟨*pick*⟩ statement in the loop body gets executed, i.e. a tuple is non-deterministically picked from F*. Due to the fact that $\exists\vec{x}.F^*(\vec{x},s)$ holds ⟨*pick*⟩ is guaranteed to succeed. Subsequently, an arbitrary ⟨*block*⟩ gets executed for the picked tuple and the very same tuple is removed from F* in the last statement of the ⟨*pick*⟩-block. Due to the fact that we don't permit any modifications of the fluent the *foreach*-loop iterates over in the loop body the line `F* -= {<$x1,$x2,...,$xn>};` is guaranteed to be the only line that modifies F^*. Hence, after one iteration of the while-loop it is guaranteed that the number of tuples for which F^* holds is decreased by one, thus the claim holds by induction. If no elements remain it holds that $\forall\vec{x}.F^*(\vec{x},s) \equiv False$, which is exactly the base case described in the section above.

Finally, we want to emphasize an important consequence of our specified semantics of a YAGI for-loop. Due to the fact that a YAGI for-loop iterates over a *set* of tuples there can be no statement made about any kind of order of the iteration. For clarification, consider the following YAGI code:

```
at = {<"r1">,<"r2">,<"r3">};

//No guarantee that the order of execution is r1-r2-r3!
```

```
foreach <$x> in at do
  move($x);
end for
```

Here, `move($x);` gets executed for every tuple in the set `{<"r1">,<"r2">,<"r3">}`, whereas the *order* of execution is non-deterministic. Note that this is the only consistent semantics because a mathematical set has - by definition - no notion of order of elements.

Implementation Remarks

Any implementation shall check that the fluent the *foreach*-loop iterates over is not modified inside the loop body. Any modification attempt shall result in an error.

5.4.7. Procedure Declaration

Syntax

⟨*proc_decl*⟩ ::= **proc** ⟨*id*⟩ **(** ⟨*var_list*⟩? **)** ⟨*block*⟩ **end proc**

Semantics

Declares a procedure with a name and a (possibly empty) list of parameters, leaving $\mathcal{D}_{YAGI}$ and $\mathcal{L}_{YAGI}$ unchanged. Semantically, *procedure declaration* is the counterpart of IndiGolog's *procedure definition* **proc** $P(\mathbf{x})$ δ **endProc**, i.e. the same semantics and restrictions as defined by (Levesque et al., 1994) apply.

Implementation Remarks

Any implementation shall ensure that procedures are unique. We define uniqueness for procedures as follows. Given a procedure P with arity m we say that the name-arity tuple $\langle P, m \rangle$ must be unique, i.e. two procedures are equal iff they have the same name and the same arity. Any redeclaration of an already declared procedure overrides the former with the latter and shall result in a warning.

5.4.8. YAGI Action Declaration

Syntax

⟨*action_decl*⟩ ::= **action** ⟨*id*⟩ **(** ⟨*varlist*⟩? **)** (**external (**⟨*varlist*⟩**)**)?
(**precondition:** ⟨*formula*⟩)?
(**effect:** ⟨*assignment*⟩$^+$)?
(**signal:** ⟨*valexpr*⟩ **;**)?
end action

Semantics

Let α be a YAGI action declaration for an action A with arity m, where m denotes the number of parameters for that respective action, i.e. the number of elements in ⟨*varlist*⟩. Then $\mathcal{L}_{YAGI}$ and $\mathcal{D}_{YAGI}$ remain unchanged and a Golog procedure of the form **proc** $A(\vec{x})$ δ **endProc** is added to the set of Golog procedures. We choose the name of the Golog procedure to be the same as the name of the action and $\vec{x}$ as the vector of the m parameters passed to the YAGI action. The Golog program δ consists of a *test*-action as first

statement that evaluates the formula constructed from the YAGI *precondition* as specified in Section 5.3.3. If no *precondition*-block is present, the *test*-action in the corresponding Golog procedure can be omitted. The YAGI *effect*-block is mapped to a (possibly empty) sequence of Golog statements constructed from the sequence of ⟨*assignment*⟩-statements as discussed in Section 5.3.4. The optional **signal**-block is solely responsible for communication with the system interface as described in Section 3.5.3, i.e. it has no influence on $\mathcal{L}_{YAGI}$ and $\mathcal{D}_{YAGI}$ and can therefore be omitted in the Golog procedure. A schematic representation of the correspondence between a YAGI action and an IndiGolog procedure is sketched in the listing below.

```
action A($x1,...,$xm)
  precondition:
    ϕ;
  effect:
    assignment_1;
    assignment_2;
    ...
    assignment_n;
  signal:
    "some data";
end action
```

Listing 5.6: YAGI Action Declaration Schematic

```
proc A(x1,...,xm)
  %precondition:
    ϕ?;
  %effect:
    δ1;
    δ2;
    ...
    δn;
  %signal:
  %  "some data";
endProc
```

Listing 5.7: Corresponding Golog Procedure Schematic

Additionally, a YAGI action declaration can be augmented with an optional *external* modifier, followed by a non-empty list of variables. The semantics of this extension is that the variables listed after the *external* modifier are *set* to a value based on some data from external sources. Consequently, we call actions with an *external* modifier present *setting actions*. The activity of setting values from external sources (e.g. cameras, motion sensors, distance sensors) to variables is actively triggered by calling a YAGI *setting action*. We claim that the semantics (i.e. the mapping to situation calculus and IndiGolog) of ordinary YAGI action declarations and setting action declarations are equivalent. This claim can be justified by the observation that *setting actions* solely assign values to variables, i.e. bind a value to an identifier. Due to the fact that assignments to variables have no influence on the underlying domain theory (see Section 5.3.4) variable assignments can be considered transparent from a theoretical point of view. Furthermore, note that the activity of setting values to variables is triggered via the *signal*-expression of the action declaration. A schematic representation of the correspondence between a YAGI setting action and an IndiGolog procedure is sketched in the listing below.

```
action B($x1,...,$xm) external ($y1,...,$yk)
  precondition:
    ϕ;
  effect:
    assignment_1;
    assignment_2;
    ...
    assignment_n;
  signal:
    "trigger setting action";
end action
```

Listing 5.8: YAGI Setting Action Declaration Schematic

```
proc B(x1,...,xm,y1,...,yk)
  %precondition:
    ϕ?;
  %effect:
    δ1;
    δ2;
    ...
    δn;
  %signal:
  %  "trigger setting action";
endProc
```

Listing 5.9: Corresponding Golog Procedure Schematic

Relating YAGI Actions, Situation Calculus Actions and Golog Procedures

Considering the fact that one of the basic elements of situation calculus are *actions* the question might arise why we decided to map YAGI actions to Golog procedures and not directly to situation calculus actions. To answer that question consider how a mapping from a YAGI action to a situation calculus action

might look like. For each YAGI action A_{YAGI} we would create a situation calculus action $A_{sitcalc}$ with the same name and the same parameters as the YAGI action. Furthermore, we would construct the *action precondition* of the form $Poss(A_{sitcalc}(\vec{y},s)) \equiv \Pi_{A_{sitcalc}}(\vec{y},s)$ from the YAGI action *precondition* $\langle formula \rangle$ and *successor state axioms* of the form $F(\vec{x},do(A_{sitcalc},s)) \equiv \Phi_F(\vec{x},A_{sitcalc},s)$ for each fluent involved in an assignment in the YAGI action *effect* block. Constructing the action precondition formula $\Pi_{A_{sitcalc}}$ from the YAGI $\langle formula \rangle$ is straight-forward according to the definition of $\langle formula \rangle$ in Section 5.3.3 whereas the construction of successor state axiom formulas from a sequence of YAGI assignments requires deeper analysis. First of all, YAGI supports *for-loop assignments* of the form

```
foreach <$x1,$x2,...,$xn> in <setexpr> do
  <assignment>
end for
```

which, loosely speaking, means that the loop iterates over each tuple in a set and uses those tuples for arbitrary assignments inside the loop body. Note that <$x1,$x2,...,$xn> in the example above is not a syntactically valid YAGI tuple. When we use this notation in YAGI code in this chapter we actually mean that instead of "..." all the concrete elements in the tuple are explicitly stated. Furthermore, the exact semantics of such a loop is discussed in Section 5.3.4 and are of minor importance for the further discussion in this section. Note that the syntax of the *foreach*-loop above closely resembles iteration constructs from general purpose languages like Java, C++ and C#, hence people familiar with such languages might assume similar (i.e. iterational) semantics just based on the syntax. When compiling such a loop directly into a successor state axiom (i.e. a formula) we would lose any sequential/iterational semantics since the evaluation of a formula is inherently "parallel".

We strongly believe that removing iterational semantics from such a loop would lead to a huge level of confusion among people who are not aware of the exact semantics of situation calculus. Moreover, we claim that rewriting arbitrary sequences of YAGI assignments to a single successor state axiom formula is a non-trivial task, even though we want to emphasize that we're positive that it is possible to prove that one could rewrite YAGI *effect* blocks to successor state axioms, which is something we plan to show in the near future.

Additionally, we want to mention that the YAGI basic action theory is always *progressable* when we map YAGI actions to Golog procedures since the only situation calculus actions involved are *add*- and *remove*-actions for each fluent (as defined in Section 5.3.1), which makes the YAGI basic action theory *strong local-effect* and for strong local-effect basic action theories a *first-order strong progression* always exists according to the work done by (Vassos et al., 2008). We will explain local-effect basic theories and their impact on progression in more detail in Chapter 7, for now we only want to mention that being first-order progressable is an important property of our basic action theory.

Lastly, we want to mention that the decision to map YAGI actions to IndiGolog procedures instead of situation calculus actions may also have an impact from a purely practical point of view. Here, by *practical* we mean a concrete implementation of a YAGI software system. More precisely, one could argue that the rewriting to IndiGolog procedures induces a performance (i.e. run-time) overhead compared to a direct mapping to situation calculus simple actions since IndiGolog procedures are more complex constructs. For the sake of completeness we want to mention that we also think that this is a valid argument and needs proper discussion, even though we consider it to be of minor importance at this point in time and - hence - delay it to future work.

Implementation Remarks

Any implementation shall ensure that the process of setting values to the variables listed after the *external*-keyword happens synchronously, i.e. the execution of the YAGI program shall block until the sensing process has finished. Furthermore, any implementation shall provide a timeout mechanism to prevent the application from waiting indefinitely. Moreover, any attempt to put a variable that is passed as parameter to the action after the *external*-keyword shall result in an error.

5.4.9. Procedure Call

Syntax

⟨*proc_call*⟩ ::= ⟨*id*⟩ **(**⟨*arglist*⟩?**);**

⟨*arglist*⟩ ::= ⟨*value*⟩ (**,** ⟨*value*⟩)*

Semantics

The execution of a YAGI procedure is the counterpart of IndiGolog's *procedure call* $P(\theta)$. Since we map both YAGI actions and YAGI procedures to IndiGolog procedures the concept of a *YAGI action call* vanishes, hence we need no additional specification for calling YAGI actions. Arguments (i.e. elements in ⟨*arglist*⟩) are passed in a call-by-value manner. Note that IndiGolog also specifies the semantics of calling a *primitive action*, i.e. a situation calculus action. Due to the fact that the only primitive actions in YAGI are the actions *add* and *remove* (which are automatically generated for each declared fluent, see Section 5.3.1) and neither of these types of actions should be invoked explicitly by a YAGI programmer we don't need a syntactic construct that maps to IndiGolog's *primitive action* call.

Atomicity of YAGI Action Execution

We specify the execution of a YAGI action (or more precisely, a procedure that has been generated from a YAGI action) to be *atomic*. The atomicity of the execution of a YAGI action is particularly important in the context of *search* since our implemented search strategy considers the execution of a YAGI action as fundamental step that shall not be interrupted. Moreover, data generated by *exogenous events* is assimilated after a YAGI action has been executed (as specified in Section 5.7.2), which guarantees that one single YAGI action always gets executed with respect to one specific model of the world. Lifting the restriction of atomicity of YAGI action executions might lead to inconsistent and/or undefined behavior. Finally, we want to note that we consider this level of atomicity as the most natural from a user's perspective, which influenced that decision as well. Still, there might be arguments for making the atomicity level more fine-grained than the execution of a single YAGI action, but due to the fact that we are not able to foresee the theoretical and practical implications of such a decision we defer this discussion to future work.

Recursion

Even though ConGolog (De Giacomo et al., 2000) as well as IndiGolog (De Giacomo et al., 2009) have definitions for unbounded recursive calls we decided to forbid recursive procedure calls in YAGI for the time being. Unbound recursive calls require second-order logic extensions for *Trans* and *Final* as discussed by (De Giacomo et al., 2000), which we want to avoid for the sake of simplicity of our specification.

Implementation Remarks

To avoid any ambiguities about whether to call a procedure that has been automatically created from a YAGI action or a procedure that has been explicitly declared by the programmer any implementation shall check that names of YAGI actions and YAGI procedures are distinctly and mutually unique, i.e. any two YAGI actions must not be equal, any two YAGI procedures must not be equal and any YAGI procedure and any YAGI action must not be equal. For equality comparison we use name-arity tuples as defined for procedure uniqueness in Section 5.4.7. Any violation of this uniqueness property shall result in an error.

5.4.10. Sequence

Syntax

⟨*block*⟩ ::= ⟨*statement*⟩$^+$

⟨*statement*⟩ ::= ⟨*test*⟩
| ⟨*proc_call*⟩
| ⟨*choose*⟩
| ⟨*pick*⟩
| ⟨*conditional*⟩
| ⟨*while_loop*⟩
| ⟨*for_loop*⟩
| ⟨*search*⟩
| ⟨*fluent_query*⟩

Semantics

A sequence of YAGI statements. Semantically, it's the counterpart of IndiGolog's *sequence* $\delta_1;\delta_2$. Note that the only valid statements in a YAGI ⟨*block*⟩ are exactly the control flow statements (with their defined IndiGolog counterparts) as specified in the sections above, hence we can establish this correspondence between a YAGI ⟨*block*⟩ and an IndiGolog *sequence*.

5.5. Incomplete Information

The assignments discussed in Section 5.3.4 exclusively deal with information that is *known*. One can imagine that there exist various practical cases where one wants to express information that is not yet known, but may (or may not) be *sensed* to its actual value during the lifetime of the agent. Consider for example a fluent that stores the location (e.g. the room) a robot resides in. Initially, (i.e. on start-up) the robot might not know in what room he is currently residing, but he might be able do narrow down the possibilities during his lifetime. Various approaches of how to deal with incomplete information in different contexts have been discussed by (Etzioni et al., 1992), (Petrick and Bacchus, 2004), (Vassos and Levesque, 2007) and others. For the time being, we're not able to express something like incomplete knowledge in YAGI. The ingredients we need are on the one hand a syntactical element to denote incomplete information and on the other hand a mechanism to eliminate possible values due to some (external) information. We discuss the latter issue in Section 5.6 and continue with our proposed solution to the former.

To syntactically express incomplete information we use the character *star* (*****) at the right-hand side of an assignment to a fluent. Loosely speaking, an assignment of the form `F = {<*>};` expresses that the value of the fluent F is not yet known. Using our running example, the assignment `at = {<*>};` expresses the fact that we don't know where the robot is, but we *do* know all the valid assignments, i.e. the powerset of all tuples that can be generated from the finite domain $\mathcal{S}^1_{at} = \{r1, r2, r3\}$. Due to the fact that the precise semantics of incomplete information in YAGI is not yet clear, we stick with the syntactical specification for the time being and defer the specification of the semantics to future work.

5.5.1. Implementation Remarks

Any attempt to assign incomplete information to a fluent shall be ignored and shall result in a warning.

5.6. Sensing

5.6.1. Syntax

⟨*sensing_decl*⟩ ::= **sense** ⟨*id*⟩ **(** ⟨*varlist*⟩? **)** (**external (**⟨*varlist*⟩**)**)? ⟨*formula*⟩ **end sense**

5.6.2. Semantics

Sensing actions are specified by (Scherl and Levesque, 1993), (Levesque, 1996), (De Giacomo and Levesque, 1999a) and others as actions that can be taken by the agent or robot to obtain information about the state of certain fluents, rather than to change them. Sensing actions are particularly relevant when the initial state of the world is incompletely specified, which is something YAGI allows us to do, as discussed in Section 5.5. Similar to the distinction between YAGI *actions* (without an *external* modifier) and YAGI *setting actions* (with an *external* modifier, see Section 5.4.8) we distinguish between *binary sensing actions* (without an *external* modifier) and *n-ary sensing actions* (with an *external* modifier). Loosely speaking, the difference is that binary sensing only provides information about whether or not a certain condition holds, i.e. returns a truth value (hence the term *binary*) and n-ary sensing returns a list of entities rather than a truth value. The idea is similar to the distinction between relational and functional fluents. This distinction is necessary because we plan to use different formalizations for binary and n-ary sensing actions, namely *sensed fluent axioms* for binary sensing actions as defined by (Levesque, 1996) and *sensing result axioms* for n-ary sensing actions as defined by (Scherl and Levesque, 2003).

First, we consider the case of binary sensing. (Levesque, 1996) introduced *sensed fluent axioms* of the form $SF(a,s) \equiv \phi_a(s)$, where SF is a distinguished predicate like *Poss*, relating the action to the fluent. For example, (Levesque, 1996) use an airport scenario that shows how the action of checking a departure screen is connected to knowing where a certain plane is parked as $SF(check\_departures,s) \equiv Parked(Flight123, gateA, s)$. In other words, $\phi_a(s)$ gets asserted to a truth value by its corresponding sensing action. The basic action theory is therefore extended with the set of sensed fluent axioms $\mathcal{D}_{SF}$ and the task is to show that $\mathcal{D} \cup \mathcal{D}_{SF} \models \phi[s']$ for a goal formula ϕ in a situation s'. We can map YAGI sensing actions to sensed fluent axioms as follows. Let a_y be the name of a YAGI sensing action (i.e. the value of ⟨*id*⟩) with arity *m*, where *m* denotes the number of parameters for that respective sensing action, i.e. the number of elements in ⟨*varlist*⟩. Then we construct the sensed fluent axiom as $SF(a_y,s) \equiv \phi_{a_y}(s)$, with $\phi_{a_y}(s)$ being the formula constructed from ⟨*formula*⟩ as discussed in Section 5.3.3.

Having defined the binary case, we continue with the n-ary case. (Scherl and Levesque, 2003) specified *sensing result axioms* of the form $SR(\alpha(\vec{x}),s) = r \equiv \phi_\alpha(\vec{x},r,s)$, with α being the name of the action, $\vec{x}$ being the parameter vector, *r* being the result and *s* being the situation term. For example, (Scherl and Levesque, 2003) show a sensing result axiom to obtain information about the weather as $SR(sense\_weather,s) = r \equiv (r = \text{"}sunny\text{"} \vee r = \text{"}rainy\text{"} \vee r = \text{"}snow\text{"}) \wedge weather(s) = r$. As of yet, there exists no mapping from a YAGI sensing action with an *external* modifier to a *sensing result axiom* due to the fact that - as of today - there exists no precise and theoretically sound description of the intended semantics. Moreover, *SR* is strongly coupled to functional fluents and there is no syntactical construct to express a functional fluent in YAGI until today. Based on these issues, we decided to stick with the syntactical specification of n-ary sensing for the time being and defer the specification of the exact semantics to future work.

5.6.3. *Setting*- and *Sensing*-Actions Revisited

After having defined both *setting*-actions (setting values of fluents based on externally generated data) and *sensing*-actions (obtaining information about the state of a fluent) we explicitly want to outline their difference regarding their semantics. Recall that *setting*-actions (and also exogenous events, for that matter) change the state of the world, i.e. modify the underlying theory, whereas sensing can be considered as a form of cutting down on possible models generated by incomplete information. To clarify this difference in semantics we provide a simple example illustrated in Figure 5.1, as follows.

Consider two fluents f and g both declared over the same domain $\{a,b\}$. Initially, we assign f to be *unknown* and g to a concrete value of the domain. Subsequently, we execute the assignment $g = f$, leading to the successor state S_0'. Note that in S_0' we end up with four models $\mathcal{M}_1,\ldots,\mathcal{M}_4$ due to the fact that we assign incomplete information to the fluent g, hence we need to generate models for all the possible sets of tuples of the domain of fluent f. Now we can analyze two different successor states, namely S_0'' generated by *sensing* the concrete value a of fluent g and S_0''' generated by *setting* the fluent g to the same concrete value a. In the first case we obtained information about the fluent g, namely we *sensed* that its value is a, hence we eliminate all models that don't match the sensed information. Consequently, we end up with the single model $\mathcal{M}_2$ remaining. In the second case we explicitly *set* the fluent g to a new value, resulting in a state where still four different models exist and each of which gets updated with the new value from the *setting action*.

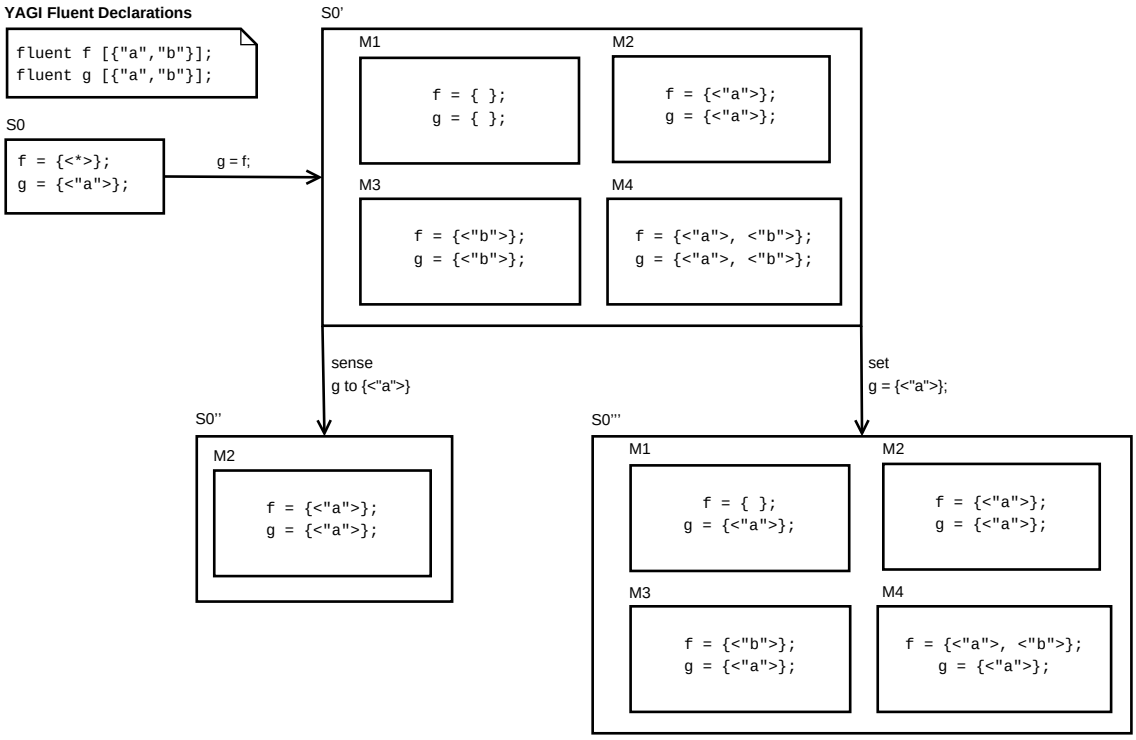

Figure 5.1.: Evolution of S_0 With Setting and Sensing Actions

This simple example illustrates that *setting* and *sensing* are fundamentally different things even though they might look similar at first sight. Moreover, we hope that this motivational example emphasizes the importance of having both mechanisms in YAGI.

5.7. Exogenous Events

5.7.1. Syntax

⟨*exogenous_event_decl*⟩ ::= **exogenous-event** ⟨*id*⟩ **(**⟨*var_list*⟩**)** ⟨*assignment*⟩$^+$ **end exogenous-event**

5.7.2. Semantics

Semantically, exogenous events are equivalent to YAGI actions with an *external*-modifier, the only difference of exogenous events is the fact that the point in time where an exogenous event gets executed

is non-deterministic, i.e. depends on arbitrary external (real-world) events. To cope with this issue, we define the following mode of execution, similar to IndiGolog's *sense-think-act* main cycle described by (De Giacomo et al., 2009):

1. Assimilate all pending data generated by exogenous events.
2. Update the underlying domain theory using the data from exogenous events according to the semantics of ⟨*assign*⟩ discussed in Section 5.3.4.
3. Progress $\mathcal{D}_{S_0}$ by executing the next YAGI action in the program.
4. Go back to 1.

5.7.3. Implementation Remarks

Any attempt to actively call an exogenous event via a YAGI statement shall result in an error. Furthermore, any implementation shall guarantee that exogenous events are processed as specified above. Moreover, any implementation shall prevent the loss of data provided by exogenous events, i.e. some kind of buffering mechanism as mentioned in Section 3.5.4 shall be implemented.

5.8. Search

5.8.1. Syntax

⟨*search*⟩ ::= **search** ⟨*block*⟩ **end search**

5.8.2. Semantics

Like IndiGolog, YAGI uses an online execution semantics as defined by (De Giacomo and Levesque, 1999a) and (De Giacomo et al., 2009). To be able to introduce offline execution semantics for certain parts of a YAGI program, we specify the operator *search*. The operator *search* applies offline execution semantics to a YAGI ⟨*block*⟩ it is applied to. In offline execution mode, YAGI searches for an appropriate sequence of actions *before* actually executing any of it. Note that *search* can - syntactically - be applied to an arbitrary ⟨*block*⟩, which imposes several issues. For example, recall that YAGI supports *sensing actions* that can potentially be called in such a ⟨*block*⟩. In the context of offline execution this implies that the system must be able to take potential sensing results into account during offline deliberation. Due to the fact that dealing with incomplete knowledge during offline execution is a non-trivial task (potential approaches have already been discussed by (Levesque, 2005), (Vassos and Levesque, 2007) and others) we're not able to provide a sound solution on how to approach this issue in YAGI for the time being. Other constructs that would further increase the complexity of *search* are *setting actions* and *exogenous events* since they both deal with data from external sources. This implies that there need to be a strategy of how to model these external influences when doing offline deliberation. For the time being, we decided to exclude *setting actions*, *sensing* and *exogenous events* from search and defer the specification of their exact offline deliberation semantics to future work. Moreover, note that with these restrictions we stay consistent with IndiGolog's *search operator* Σ, which we consider to be the semantic counterpart of *search* in YAGI. More formally, (De Giacomo et al., 2009) define the semantics of offline execution as $Do(\delta, s, s') = \exists \delta'.Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s')$, where $Trans^*$ is the reflexive transitive closure of Trans[4]. We discuss the semantics of *Trans* and *Final* in more detail in Chapter 7.

[4] $Trans^*$ can be defined as a situation calculus second-order formula. For the sake of simplicity we omit the details here and refer to (De Giacomo et al., 2009) for more details.

5.8.3. Implementation Remarks

Any implementation shall check that no sensing-/setting-action or exogenous event is part of a search-⟨*block*⟩. Any appearance of any of these constructs in a search-⟨*block*⟩ shall result in an error.

5.9. Miscellaneous Language Elements

5.9.1. Fluent/Fact Query

Syntax

⟨*fluent_query*⟩ ::= ⟨*id*⟩ **;**

Semantics

The fluent query command has no influence on the underlying domain theory or program execution whatsoever, its purpose is solely to echo the state (i.e. the assignment) of the fluent that is being queried. Note that due to the fact that ⟨*fluent_query*⟩ is a ⟨*statement*⟩ it could be part of any program ⟨*block*⟩. For the time being, we only define that if the whole program consists of just a ⟨*fluent_query*⟩ its semantics is that it returns the state of the queried fluent to the caller (i.e. the front-end) or *false* if the fluent (or fact) doesn't exist. Any use of a fluent query inside a more complex YAGI program should be ignored gracefully.

5.9.2. Include

Syntax

⟨*include*⟩ ::= **@include** ⟨*string*⟩ **;**

Semantics

The include command has no direct influence on the underlying domain theory or program execution whatsoever, its purpose is solely to import YAGI code from different files into a single file. More precisely, the semantics of ⟨*include*⟩ is that the whole include command gets replaced by the YAGI code from the file which name is provided via the ⟨*string*⟩ value. This semantics is identical to the semantics of macro replacement performed by the preprocessor in the C programming language. Note that even though a purely textual replacement semantics is sufficient for the time being we aim for more sophisticated include mechanism similar to the *import* command in Java in the future.

5.10. A YAGI Program

5.10.1. Syntax

⟨*program*⟩ ::= (⟨*declaration*⟩ | ⟨*block*⟩ | ⟨*include*⟩)$^+$

⟨*declaration*⟩ ::= ⟨*fluent_decl*⟩
| ⟨*fact_decl*⟩
| ⟨*action_decl*⟩

| ⟨*proc_decl*⟩
| ⟨*exogenous_event_decl*⟩
| ⟨*sensing_decl*⟩
| ⟨*assignment*⟩

5.10.2. Semantics

Finally, we call arbitrary sequences of YAGI lines of code $\langle l_1, \ldots, l_n \rangle$ a YAGI *program*. A line in a YAGI program can either be a *declaration* modifying the state of the YAGI world (except ⟨*proc_decl*⟩), i.e. the underlying theory (as specified in the first part of this chapter) or any sequence of program flow statements (as specified in the second part of this chapter) that specify the program, i.e. a ⟨*block*⟩ or a ⟨*proc_decl*⟩. The semantics of a YAGI program is then given by the consecutive execution of its lines of code in their given order according to the transition semantics of IndiGolog defined by (De Giacomo et al., 2009). We restate the exact definitions of the IndiGolog transition semantics and show their correspondence to YAGI in Chapter 7.

Chapter 6

Implementation

In this chapter, we describe our implementation of the YAGI software stack and the YAGI language specified in the previous chapters. We discuss our fundamental design decisions in Section 6.1 and describe our system architecture in Section 6.2. Finally, we briefly explain how the specified YAGI language elements have been implemented in Section 6.3.

6.1. Fundamental Design Decisions

Due to the fact that our application is the first implementation of a YAGI software system as specified in this thesis the main focus was to deliver a viable proof-of-concept implementation that can be easily extended later on. Since the YAGI software stack already strongly favors a loosely coupled software system due to its layered architecture we also tried to decouple the components in our implementation as cleanly as possible without inducing too much overhead. Furthermore, we tried to exclusively use programming languages, components and libraries that are considered well-known among people in the computer science community to encourage people to extend and enhance our implementation.

We decided to use C++ as our implementation language of choice because we envision YAGI also to be used in real-world robotics applications, which are often subject to computation- and/or memory-restrictions. Hence, using a language that induces as little overhead as possible was a major requirement. Moreover, easy interfacing with robot operating systems like ROS (Quigley et al., 2009) is important when it comes to the implementation of the system interface of the YAGI software stack. By its nature, C++ is the most suitable language of choice for such a task because it provides the low-level capabilities of a language like C enriched with modern concepts that make the codebase more maintainable and less error-prone.

6.2. System Architecture

In this section, we explain the system architecture of our implementation and describe the ideas and goals and how they influenced the various design decisions.

6.2.1. Front-End

The front-end is implemented as a console application that allows the user to interactively enter and run YAGI code.

Parser Implementation

The parser for the YAGI language is generated using Terence Parr's popular parser generator *ANTLR* (*AN*other *T*ool for *L*anguage *R*ecognition). We briefly describe ANTLR and the compiler construction theory behind ANTLR in this section based on information from the ANTLR reference manual from (Parr, 2007).

ANTLR uses an extended version of the Backus-Naur Form (BNF) as a notation for writing the grammar rules for the desired target language, similar to the notation used in this thesis to describe the syntax of YAGI. ANTLR is a parser generator for so-called *LL(*)* languages, i.e. it is able to create parsers for languages that are in the set of LL(*) parsable languages. A language is said to be LL(*) parsable iff it can be parsed with a top-down LL parser (parses input ***L***eft to right and performs ***L***eftmost derivations) that is not restricted to a finite *k* tokens look-ahead. In the case of YAGI, we restrict the ANTLR parsing algorithm to use LL(1) parsing, a more restricted version of LL(*) parsing that decides which production rule to apply by looking only at the next input symbol (Aho et al., 2007). The complete ANTLR grammar for YAGI can be found in Appendix B.

ANTLR is able to create the parser and lexer code for a given grammar file in different target languages, e.g. Java, Python and C#. The fundamental design decision for implementing YAGI was to use C/C++ as language of choice, hence the automatically generated output from ANTLR needs to be either in C or C++ to allow easy integration into the YAGI codebase. Due to the fact that the newest version ANTLR 4.x. does not have support for C or C++ code generation yet[1] we needed to fall back to ANTLR v3 to get support for C code generation[2]. The output of the generated ANTLR parser is the *abstract syntax tree* (AST) of the YAGI input program. The AST is an abstract hierarchical structure that represents the source program and is often used as an intermediate representation for further processing (Aho et al., 2007). In YAGI, we use the AST for steps like type checking, code rewriting and interpretation.

Architecture Overview

An overview of the software-design of the front-end is presented in the class diagram in Figure 6.1. The purpose of the presented classes is as follows.

- **YAGIMain**: The YAGIMain class provides the console front-end of the YAGI shell. It reads YAGI programs from either a file or the console and passes the code for further processing to the ANTLR-Lexer class.
- **ANTLRLexer**: The class *ANTLRLexer* is responsible for lexing the YAGI source code and passing the resulting stream of tokens to the class *YAGIParser*. Note that the ANTLRLexer class contains no handcrafted code, it is purely auto-generated from the YAGI grammar file.
- **ANTLRParser**: The class *ANTLRParser* takes a stream of tokens as input and outputs the resulting AST structure. Due to the fact that *YAGILexer* and *YAGIParser* are written in the C language and further processing steps based on this C data structures tend to be cumbersome an additional class *YAGITreeWalker* is introduced. Note that the ANTLRParser class contains no handcrafted code, it is purely auto-generated from the YAGI grammar file.
- **YAGITreeWalker**: The class *YAGITreeWalker* is the auto-generated output of a separate grammar file, a so-called *tree grammar*. A *tree grammar* operates on a stream of tree nodes (i.e., the AST) rather than on tokens (Parr, 2007). The purpose of the YAGITreeWalker is to traverse the C representation of the AST and to send signals with information about the visited AST nodes to some external receiver. Note that the main advantage of this approach is that no handcrafted C code for tree traversal is necessary because the code is automatically generated based on the ANTLR tree grammar.

[1] See `http://www.antlr.org/download.html` for further information. Last visited on December 3rd, 2014.
[2] See `http://www.antlr3.org/download.html` for further information. Last visited on December 3rd, 2014.

- **CToCppBridge**: The class *CToCppBridge* is responsible for providing callbacks (i.e. function pointers) as hooks that the YAGITreeWalker can use to signal information about visited AST nodes to some external receiver.
- **YAGICallbackConnector**: The class *YAGICallbackConnector* is the concrete bridge between the C part of the implementation and the C++ codebase responsible for building an object-oriented representation of the AST and any further processing (e.g. type checking, rewriting, execution) performed on the AST.
- **ASTBuilder**: The purpose of the factory-like class *ASTBuilder* is to build an object-oriented representation of the AST based on the information provided from the traversal of the C AST. To accomplish this task, the ASTBuilder reacts to the signals from the YAGITreeWalker and builds a tree structure based on this data. The resulting C++ representation of the AST is then passed back to the caller (i.e. the instance of YAGIMain) for further processing.

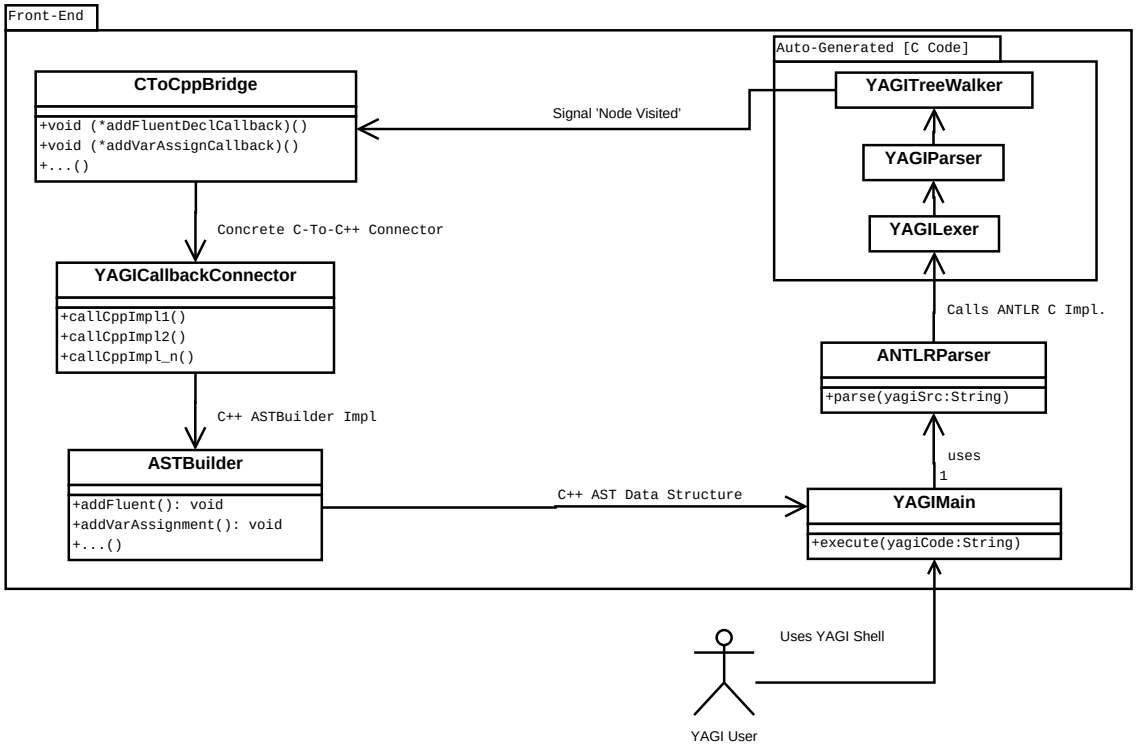

Figure 6.1.: Schematic Class Diagram of the Front-End Implementation

6.2.2. Back-End

Due to the fact that the basic concepts for describing the state of the world in YAGI are sets and tuples we decided to use a relational database management system to represent the YAGI state of the world, i.e. fluents and facts. The concepts from relational databases naturally resemble fluents and sets of tuples as we can interpret a database table as a fluent, a tuple as a row in a table and a set of tuples as the set of all rows in a table. Moreover, databases and their relations to action theories have been discussed by (Lin and Reiter, 1997), (De Giacomo and Palatta, 2000), (De Giacomo and Mancini, 2004), (Vassos and Sardina, 2011) and others, which even more encouraged us to use relational database technology for our back-end implementation. We decided to use *SQLite* as our relational database management system (RDMS) of

choice. The decision to prefer SQLite over other database systems was driven by a number of reasons, a non-exhaustive list of reasons[3] is presented as follows, in no particular order:

- SQLite is self-contained: SQLite requires minimal support from external libraries and the operating system and is written in ANSI-C. Therefore, it is highly portable to a huge variety of different platforms without much effort.
- SQLite is server-less and zero-configuration: A lot of database engines are implemented as separate server processes, whereas SQLite is not. In our case, the advantage of being serverless is that we can easily enable people who want to use YAGI (e.g. students attending a lecture at university) to do so without needing to install, setup and configure a RDMS.
- SQLite is the most widely deployed SQL database: SQLite is used by a vast number of well-known and widely used software products such as Mozilla Firefox, Skype and McAfee anti-virus software. The SQLite developers estimate that there are are at least 500 million SQLite deployments in use.

The second major task of the back-end is to store and execute YAGI programs. To accomplish this task we decided to simply store the program (or, to be more precise, its abstract representation in form of an abstract syntax tree (AST)) in appropriate data structures and execute (i.e. interpret) these structures on demand. An overview of the software-design of the back-end is presented in the class diagram in Figure 6.2. The purpose of the presented classes is as follows.

- **ASTNodeVisitorBase**: The base class implementation of the visitor design pattern (Gamma et al., 1994). The visitor implementations are used to visit nodes of the AST and perform certain operations based on the data from the AST nodes. The operations depend on the concrete implementations of the ASTNodeVisitorBase base class. The implementation is based on the *acyclic visitor* implementation from (Alexandrescu, 2001).
- **TypeCheckVisitor**: The first visitor that is applied to the AST provided by the front-end. Its purpose is to check the AST for type errors. The YAGI program executions continues iff no type errors occurred. The current implementation of type checking is very rudimentary and only checks some very simple cases, e.g. referring to an undefined fluent or re-assigning a fact.
- **RewritingVisitor**: After type-checking, the rewriting visitor performs syntactical rewriting according to the specification of the YAGI language, e.g. pattern matching is rewritten as specified in Section 5.3.5.
- **ExecutionVisitor**: The ExecutionVisitor class is responsible for the execution of the YAGI program, i.e. the interpretation of the AST. It holds various members of polymorphic types (e.g. for database access, formula evaluation and signal handling) that are parametrized with concrete implementations by the caller.
- **IExogenousEventConsumer**: The interface for consuming data provided by exogenous events.
- **ExogenousEventConsumer**: A concrete consumer of exogenous event data. In the proof-of-concept implementation exogenous event data is directly consumed by the interpretation visitor.
- **DatabaseConnectorBase**: The base class responsible for connecting to a database and executing SQL queries.
- **SQLiteConnector**: A concrete implementation of the *DatabaseConnectorBase* class for the SQLite database back-end.
- **IFormulaEvaluator**: An interface responsible for the evaluation of a YAGI ⟨*formula*⟩.
- **FormulaEvaluator**: A concrete implementation of the *IFormulaEvaluator* interface that evaluates YAGI formulas as discussed in Section 7.2.5.
- **IYAGISignalHandler**: An interface responsible for handling signals triggered by YAGI actions.
- **CoutCinSignalHandler**: A concrete implementation of the *IYAGISignalHandler* interface that simply displays YAGI signals textually on the YAGI shell.

[3] Summary of a list of advantages presented at `http://www.sqlite.org/`. Last visited on December 3rd, 2014.

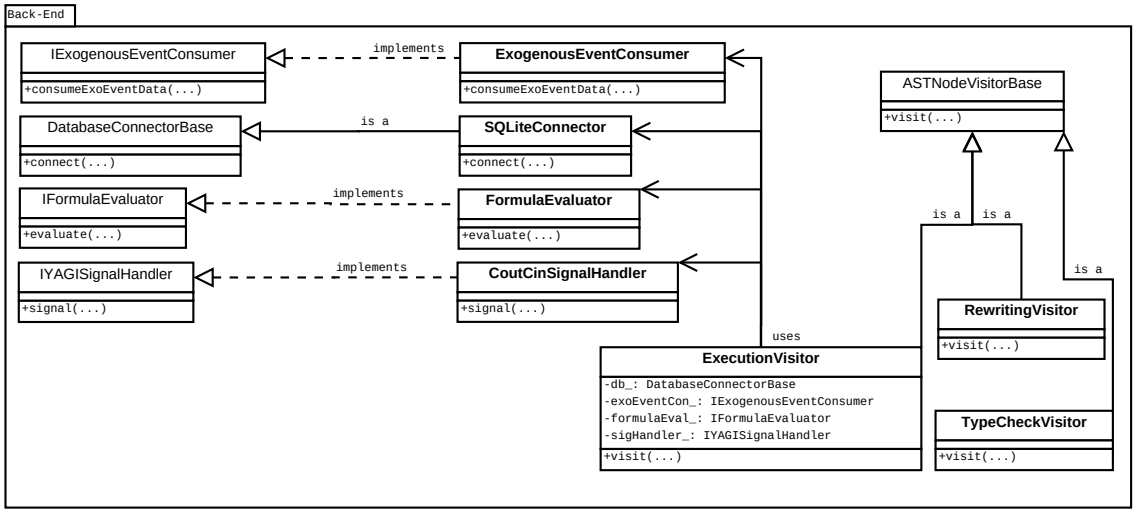

Figure 6.2.: Schematic Class Diagram of the Back-End Implementation

6.2.3. System Interface

In our first implementation, the system interface is solely designed for testing and simulation. When a YAGI actions sends a signal to the system interface our implementation simply echoes the signal data back and its content is displayed in the YAGI interpreter command shell window. When a setting action is triggered the system interface prompts the user to enter the data that should be passed to the respective setting action. Lastly, exogenous events are implemented file-based, i.e. data for exogenous events can be written into a specific plain-text file. When the file is saved the currently running YAGI program consumes this data and processes it as specified in Section 5.7.2.

6.2.4. Inter-Layer Communication

Due to the fact that front-end, back-end and system interface are compiled into one single binary the communication between the layers is implemented via simple method calls.

6.3. YAGI Language Constructs

Having described the implementation of the three layers of the YAGI software stack we proceed with a brief description of how the different YAGI language constructs are implemented. Note that we only give a description of the software-engineering aspects of the implemented YAGI language constructs in the following sections and defer any discussion about how the implementation relates to the specification to Chapter 7.

6.3.1. Fluent- and Fact-Declaration

Each fluent and each fact corresponds to one distinctive SQLite database table. The number of columns in such a table is determined by the arity of the fluent. Additionally, each domain of each dimension of each fluent is stored in a separate, distinct table. A simple example of a fluent declaration and its effects on the database is illustrated in the figure below. Fact declarations follow analogously.

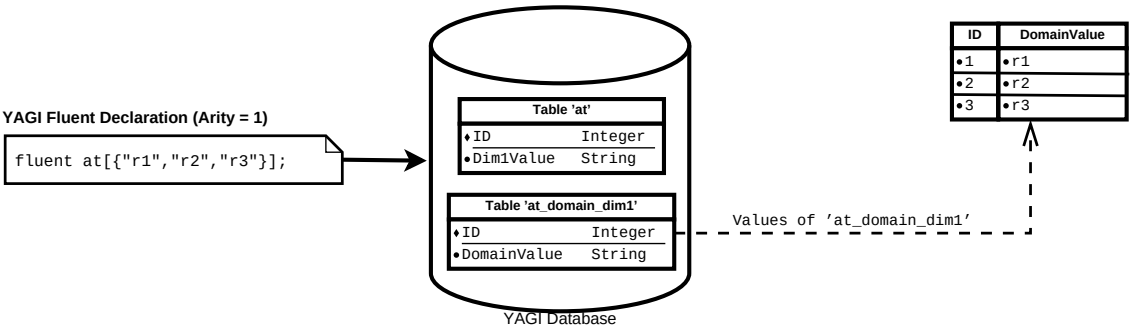

Figure 6.3.: Schematic of Fluent Declaration Implementation

6.3.2. Action Declaration

The AST of each declared action is stored in an associative container that has the name and arity of the action as key and the AST of the action as value. On execution, the AST of the action to be executed is interpreted using an instance of the *ExecutionVisitor*. By specification, YAGI actions are rewritten into procedures to accomplish sequential behavior as specified in Section 5.4.8. We omit this step in our implementation and simply interpret both actions and procedures in a sequential manner. Omitting the rewriting of YAGI actions is no violation of the specification since our implementation treats the evaluation of the action precondition (semantically) like a *test* statement (to which the precondition gets rewritten according to the specification), i.e. the remaining part of the action is executed if and only if the precondition holds. This semantics is exactly the same as the semantics of *test*, i.e. it *guards* the execution of some part of a YAGI program depending on the evaluation of a truth value. We illustrate the declaration of two actions in Figure 6.4. Note that we purposely omitted the bodies of the actions because they are of no importance during this discussion.

6.3.3. Formulas

Formulas are evaluated using the semantics of tuples and sets on appropriate C++ data structures and the built-in C++ operators for value comparisons[4]. In any case a fluent is involved in a formula all the necessary values are fetched from the database and stored in C++ data structures on which the evaluation of the formula is performed. We discuss the evaluation of formulas in more detail in Section 7.2.5. A possible optimization could be to evaluate formulas directly using SQL queries as described by (De Giacomo and Palatta, 2000).

6.3.4. Assignments

As specified in Section 5.3.1, fluents (i.e. the model) can only be modified using the situation calculus actions *add* and *remove* that are created for each declared fluent. These actions are implemented via SQL statements *insert* (in case of *add*) and *delete* (in case of *remove*). The inserted (or deleted) values for these statements are inferred from the parameters of the corresponding situation calculus action and the affected table name corresponds to the name of the fluent to which the situation calculus action belongs to. All types of fluent assignment statements ultimately expand to a set of *add* and *remove* statements (see Section 5.3.4), i.e. a sequence of SQL *insert* and *delete* statements. Assignments to variables are implemented via a variable table that holds an associative container with the name of the variable as key and a stack of values

[4]Recall that we specified ordering comparisons for strings to be performed lexicographically. Due to the fact that we currently don't support numbers in YAGI we compare strings that represent valid integral numbers using integer comparison semantics to avoid counter-intuitive behavior, e.g. "10" < "2" returns *true* when compared lexicographically but one might expect *false* as result.

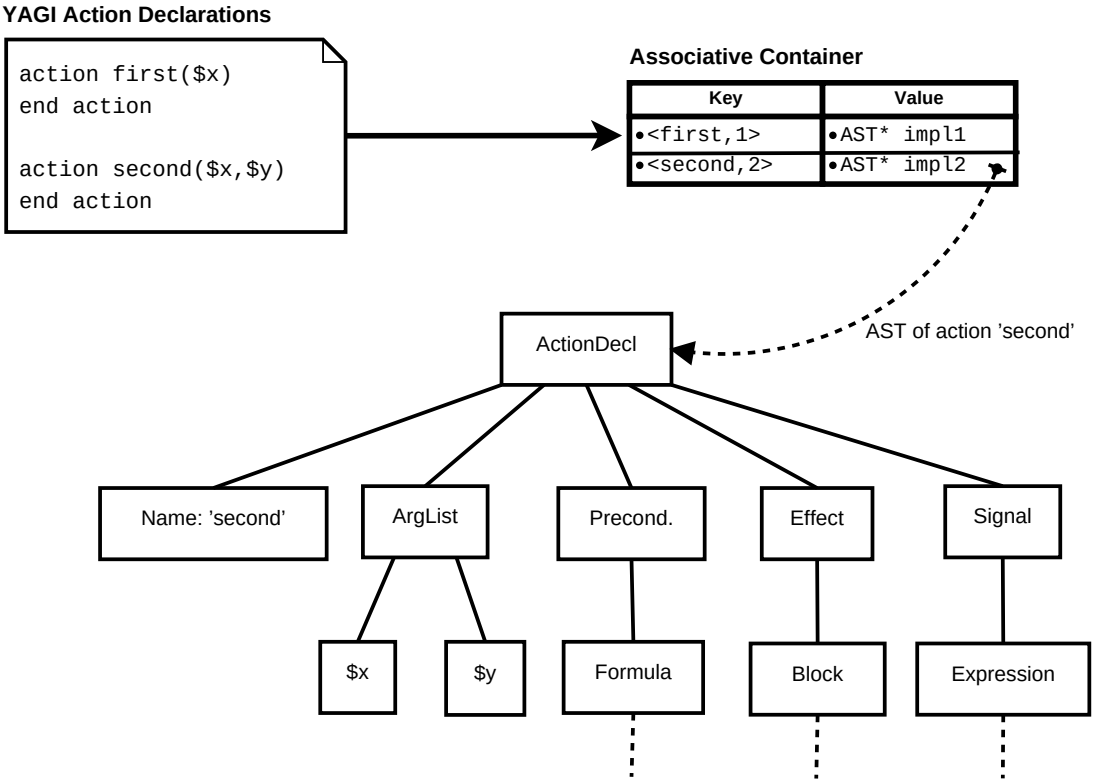

Figure 6.4.: Schematic of Action Declaration Implementation

that represents the values of the variable in possibly different scopes. We illustrate a simple assignment to a fluent and its effect in Figure 6.5.

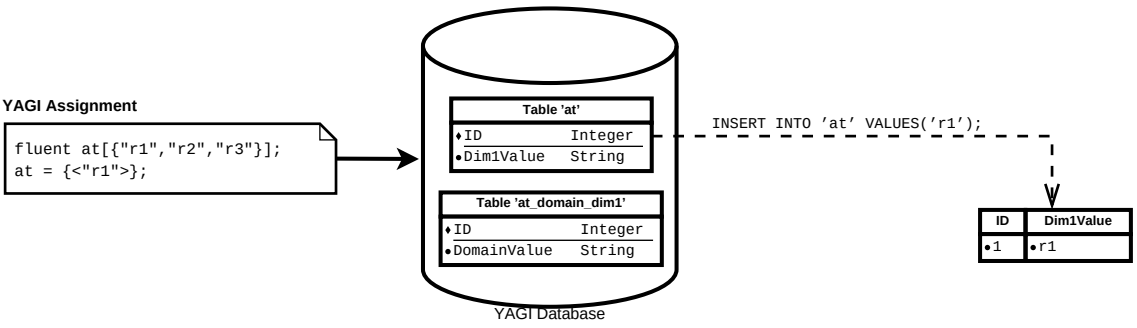

Figure 6.5.: Schematic of a Fluent Assignment

6.3.5. Incomplete Information

Incomplete information is only implemented on a syntactical level, i.e. parsing of YAGI statements that use incomplete information. Any attempt to use incomplete information beyond parsing results in an error. Again, we want to emphasize that there exists no specification of the intended semantics of incomplete information in YAGI for the time being. Since we base YAGI on IndiGolog we also want to mention that IndiGolog uses a so-called *possible value* semantics to handle some limited version of incomplete information via the predicates *settles* and *rejects*, as discussed by (De Giacomo et al., 2009).

6.3.6. Pattern Matching

Pattern matching constructs are rewritten by the *RewritingVisitor* class according to the specification in Section 5.3.5, i.e. the rewriting visitor returns a modified AST that represents the specified rewriting. Consequently, the concept of pattern matching is transparent for any further processing steps.

6.3.7. Exogenous Events

Analog to action declarations, the AST of each declared exogenous event is stored in an associative container that has the name and arity of the exogenous event as key. When an exogenous event gets triggered the AST of the event to be executed is interpreted using an instance of the *ExecutionVisitor* class.

6.3.8. Sensing

Sensing is only implemented on a syntactical level, i.e. parsing of declarations of sensing actions. Any attempt to use sensing beyond parsing results in an error.

6.3.9. *Test* Statement

The *Test* statement is implemented using a C++ conditional ("if-statement") that tests the truth value of the evaluated YAGI ⟨*formula*⟩ and either continues with the execution of the remaining YAGI program (in case ⟨*formula*⟩ holds) or aborts execution (in case ⟨*formula*⟩ does not hold).

6.3.10. Non-deterministic Programming Constructs

The non-deterministic YAGI statements *pick* and *choose* are implemented in a hit-or-miss like manner, i.e. an argument (in case of *pick*) or a block (in case of *choose*) is chosen pseudo-randomly[5] and the program execution continues. Since YAGI is specified to use an *online execution semantics* by default, an unfortunately chosen pseudo-random number can lead to a state where the program is unable to continue. In case of *offline execution semantics* (i.e. using a *search*-block) *pick* and *choose* are resolved using *planning* semantics, as described in Section 6.3.13.

6.3.11. Conditionals

YAGI conditionals are implemented via conditional constructs ("if-statements") from the C++ programming language. The truth value of the condition (i.e. the YAGI ⟨*formula*⟩) is evaluated and either the if-block or the else-block (if present) gets executed depending on the evaluation result of ⟨*formula*⟩.

6.3.12. Loops

YAGI *while* loops correspond to C++ *while* loops. As for conditionals, the truth value of the condition of the loop is evaluated and the loop body gets executed as long as the condition holds. YAGI *for* loops essentially iterate over a finite set of values, which makes them similar to C++ *range-based for loops* (since C++11), which were consequently used for the implementation of YAGI *for* loops.

[5] The pseudo-random values are generated uniformly distributed (via `std::uniform_int_distribution`) using a Mersenne Twister pseudo-random number generator (via `std::mt19937`) (Overland, 2013).

6.3.13. Search-Operator

As described in Section 5.8.2, the purpose of *search* is to find a valid execution trace of a YAGI program *before* actually executing it. Our implementation of *search* works as follows. If a *search* statements gets executed it creates a *shadow world*. A *shadow world* is a copy (or a snapshot) of the current state of the world at one specific point during the execution of a YAGI program. A *shadow world* consists of a copy of all the fluents and facts (i.e. the database) and a copy of all the variables and their values. Consecutively, the YAGI program the *search* operator is applied to gets *mentally executed* on the previously created *shadow world*. By *mentally executed* we mean that it is not a real-world execution but the attempt to find an execution trace that is guaranteed to succeed if executed in the real world. Consequently, if the execution of the program on the *shadow world* was successful the resulting execution trace gets executed in the "real world", otherwise an appropriate error message gets displayed and the program does not get executed.

The execution of *search* becomes particularly interesting if non-deterministic choices are part of the YAGI program the *search* operator is applied to. If - during the mental execution of a YAGI program on a *shadow world* - a *pick* or *choose* statement gets executed a different execution branch (i.e. a new *shadow world*) for each possible value in *pick* (or for each possible block in *choose*, respectively) gets created. Each of these branches is responsible for searching over the program for one possible value of *pick* or for searching over one possible block in case of *choose*. The search over these branches happens as follows. Each branch executes exactly one YAGI action and waits until all other branches finished their execution[6]. After all branches performed their executions we check whether or not the execution was successful. If a YAGI action could not be successfully executed due to a violation of the action precondition we remove the branch from the set of possible result branches. For all other branches (i.e. branches where the execution was successful) we continue with the execution of the next action. All branches that lead to a valid result after all actions have been executed are possible result states and need to be taken into consideration for further program execution.

The idea behind this approach is to mimic the behavior of breadth first search (BFS). In our context, the equivalent of visiting a node using BFS on a finite graph is the execution of a single YAGI action. We start at the root node (i.e. the point where a *pick* or *choose* gets executed) and visit all its direct neighbors, i.e. execute the first YAGI action. If all direct neighbors have been visited (i.e. every execution branch finished its execution of one YAGI action) we progress to the next level, i.e. execute the next YAGI action in each execution branch. The choice to implement the *search* operator in this manner was driven by two major aspects:

- **Completeness of BFS**: Building a BFS-like strategy as described above enables us to argue that our implementation of *search* has the same properties as BFS regarding *completeness* and *optimality*[7]. The decision to focus on the completeness property and to ignore the disadvantages of BFS for now was directly driven by the second major aspect.
- **The Curse of Prolog DFS**: Due to the fact that the majority of Prolog systems use a depth-first strategy (Nilsson and Małuszyński, 1990) also most of the Prolog-based implementations of Golog are bound to DFS. Experience showed that this fact plays a huge role in why using a Prolog-based Golog interpreter is challenging for novices and sometimes even for experts. Since one of the design goals of YAGI was to remove the tight coupling to Prolog it was also an important aspect to implement a different search strategy.

Finally, we want to mention that we are convinced that search - as it is implemented now - is sufficient for a first proof-of-concept implementation but is still far from being ideal. One possible optimization (among possibly many) could be the implementation of iterative deepening depth-first search (IDDFS) instead of BFS, but we decided to stick to BFS for the time being and defer the analysis of possible optimizations of the search strategy to future work.

[6] Due to the fact that we specified the execution of a YAGI action to be guaranteed to terminate this mode of operation does not impose any problems regarding termination.

[7] This is only true under the assumption that the execution of one YAGI action is guaranteed to terminate. Due to the fact that only iteration over finite domains can appear in a YAGI action effect and YAGI actions are - by specification - unable to call other YAGI actions (i.e. recursion is impossible) it is guaranteed that the execution of a YAGI action always terminates.

Search - Program Flow Example

To illustrate the program flow between the multiple units of execution involved when a search-operator is present in a YAGI program consider the following example:

```
//Fluents 'carry', 'office' and 'at' from our running example
carry = {<"o1">};
office = {<"p1","r1">, <"p2","r2">};
at = {<"r1">};

proc searchSample()
  search
    pick <$p, $r> from office such  //either <"p1","r1"> or <"p2","r2">
      move($r);                     //<"p1","r1"> violates precondition of 'move'
      choose
        pickup("o1");               //violates precondition of 'pickup'
      or
        putdown("o1");              //execution possible
      end choose
    end pick
  end search
end proc
```

Listing 6.1: YAGI Search Sample

Without a search-block the program above may or may not get executed successfully, depending on the pseudo-randomly chosen element from *pick* and the pseudo-randomly chosen block from *choose*. Using a search-block we are able to *plan* the execution ahead, i.e. find a valid execution trace - if one exists. The program flow of the YAGI code snippet illustrated above is visualized in Figure 6.6.

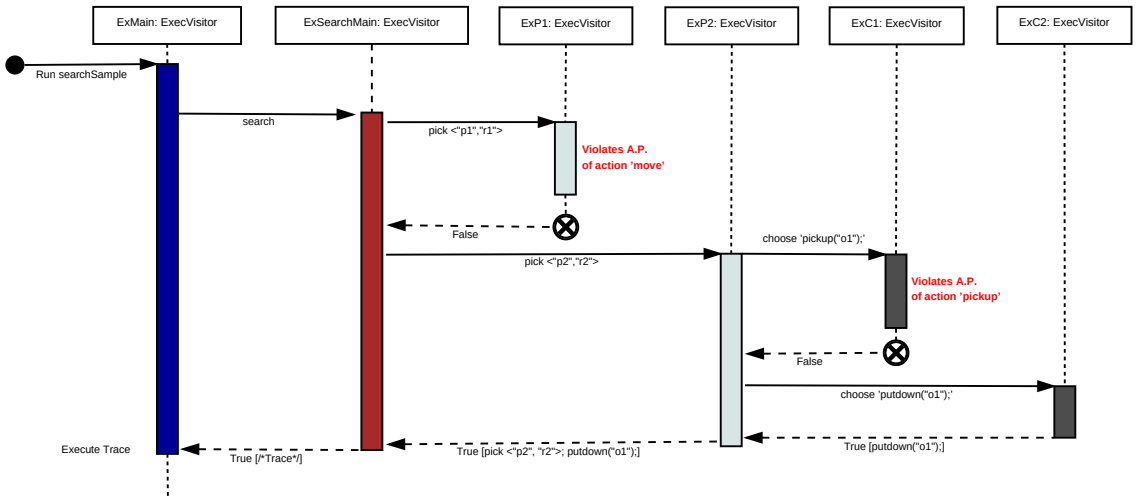

Figure 6.6.: Schematic Sequence Diagram of the Execution of a Search Block

When the YAGI code illustrated above gets executed an instance of the execution visitor (ExMain) traverses the AST and executes the assignments to *carry*, *office* and *at*. As soon as the node in the AST that represents the search-block gets executed a second instance of the execution visitor (ExSearchMain) is instantiated that *mentally* executes the code inside the search-block, blocking the execution of ExMain. For each possible tuple of the *pick*-statement a separate execution instance (ExP1 and ExP2) gets instantiated, each of them searching for a possible execution trace. If execution is possible for a tuple from *pick* the corresponding execution instance continues (ExP2), otherwise the execution instance returns *False* to its caller (ExP1). The execution for *choose* follows similarly (ExC1 and ExC2). If a valid trace of the program

inside the search-block could be found the trace is returned to the main execution unit, which subsequently executes the trace in the "real world".

6.3.14. Procedure Declaration

Analog to action declarations, the AST of each declared procedure is stored in an associative container that has the name and arity of the procedure as key. When a procedure is executed the AST of the procedure to be executed is interpreted using an instance of the *ExecutionVisitor* class.

Chapter 7

Specification Conformance

In this chapter we discuss why our implementation conforms the specification of the YAGI language described in Chapter 5. Recall that one important observation in the introduction of this thesis was that most of the Golog implementations are Prolog-based. Even though Prolog-based implementations have a number of drawbacks (as outlined in the introduction of this thesis) using Prolog as language of choice for implementing basic action theories and Golog programs comes with a notable advantage regarding the discussion of specification conformance: One can prove properties like termination and correctness of Golog programs directly within the situation calculus[1] (Reiter, 2001). Due to the fact that we decided not to use a Prolog back-end in this implementation and correctness proofs of programs written in general purpose programming languages are only feasible under certain conditions[2] we base our discussion on showing semantic equivalence between the specification of YAGI and our proof-of-concept implementation. Specifically, we have to show a correspondence between the following elements:

1. **Situation Calculus (BATs) $\leftrightarrow$ Databases, Ground Formula Evaluation**: Our specification of YAGI uses elements of situation calculus to specify the state of the world, reflect how the world evolves and answer questions about the state the world. Our implementation makes use of database semantics to implement parts of situation calculus. Consequently, we have to show how and why database semantics reflect the semantics of situation calculus basic action theories and how formulas (more precisely, formulas *without* free variables) can be evaluated based on this semantics.

2. **IndiGolog $\leftrightarrow$ YAGI Program Execution**: The second aspect of our discussion is program flow. Recall that we used the semantics of IndiGolog to specify the execution of YAGI programs and that our implementation interprets an abstract representation of YAGI source code (i.e. an abstract syntax tree) to implement program flow. Hence, we have to show how the program flow semantics of IndiGolog relates to the program flow semantics of YAGI.

We start with providing a set of definitions we need throughout this chapter in Section 7.1 and continue with showing the connection between situation calculus and database semantics in Section 7.2. Subsequently, we discuss the program flow semantics of IndiGolog and how this semantics relates to YAGI program execution in Section 7.3 and finish this chapter by summarizing our results in Section 7.4.

[1] This proof is based on the idea that one can prove that a Prolog implementation of a basic action theory is correct under certain assumptions (properties of the basic action theory (most notably closed initial database and no functional fluents) and *properness* of the Prolog interpreter) and Golog programs are *macro expanded* into situation calculus sentences.

[2] Possible approaches are to restrict the language to a certain subset that allows formal verification and/or to use a *Design by Contract* model to provide formal semantics of the program (Meyer, 1992).

7.1. Definitions

To discuss about specification conformance we need to establish a connection between situation calculus (especially fluents and situations) and our database semantics and a connection between the execution of an IndiGolog program (i.e. the evaluation of the transition semantic predicates *Trans* and *Final*) and the execution of a YAGI program, i.e. the interpretation of the AST. To be able to establish such a connection we need to define a set of predicates and functions, as follows.

Definition 7.1 (Relational Database)**.** We define a relational database *db* as a set of database tables

$$db = \mathcal{T}_1 \cup \mathcal{T}_2 \cup \ldots \cup \mathcal{T}_n,$$

where a database table in a relational database can be interpreted as set-theoretic *relation*, a row in a table can be interpreted as a mathematical *tuple* and a column can be interpreted as an *attribute* (Gossett, 2009). For the sake of clarity we omit the detailed definitions of relational databases, relational models and relational algebra and refer to the work of (Codd, 1970). Throughout this chapter we use the term *database* synonymously to *relational database*.

Definition 7.2 (Function Symbol *exec*)**.** The binary function symbol *exec* is defined as

$$exec : action \times db \rightarrow db,$$

where *action* is a situation calculus action (i.e. an element of the sort *actions*, where an *action* is the only entity of situation calculus that can change situations and fluents) and *db* is a database according to the definition above. We provide a detailed description of the *exec* function when we discuss the correspondence between successor state axioms and the database semantics of our implementation in Section 7.2.4.

Definition 7.3 (Transition Semantics Predicate *Trans*)**.** The transition semantics predicate *Trans* is the 4-ary predicate

$$Trans(\delta, s, \delta', s')$$

with δ being an executable program in a starting situation s leading to situation s' by executing one elementary step of δ, resulting in the remaining program δ'. *Trans* holds iff there exists a transition from (δ, s) to (δ', s'). This form of transition semantics has been used by (De Giacomo et al., 2000) to specify the transition semantics of *ConGolog* and later on by (De Giacomo et al., 2009) to describe the transition semantics of *IndiGolog*.

Definition 7.4 (Transition Semantics Predicate *Final*)**.** The transition semantics predicate *Final* is the binary predicate

$$Final(\delta, s)$$

with δ being a program that is allowed to successfully terminate in situation s. Together with *Trans*, *Final* was also used by (De Giacomo et al., 2000) and (De Giacomo et al., 2009) to define the transition semantics of *ConGolog* and *IndiGolog*, respectively.

Definition 7.5 (YAGI Transition Semantics Predicate *YagiTrans*)**.** The transition semantics predicate *YagiTrans* is the 4-ary predicate

$$YagiTrans(\alpha, b, \alpha', b')$$

with α being a YAGI program and b being a database. The execution of α w.r.t. the database b leads to a new database b' and results in the remaining program α'. *YagiTrans* holds iff there exists a transition from (α, b) to (α', b').

Definition 7.6 (YAGI Transition Semantics Predicate *YagiFinal*)**.** The transition semantics predicate *YagiFinal* is the binary predicate

$$YagiFinal(\alpha, b)$$

with α being a program that is allowed to successfully terminate w.r.t the database b.

Definition 7.7 (Program Translation Function *yagiToGolog*)**.** The unary function symbol *yagiToGolog* is defined as

$$yagiToGolog : \alpha \rightarrow \delta$$

where α is a valid YAGI program and δ is a valid situation calculus / IndiGolog program. The interpretation is that the function translates YAGI programs to situation calculus / IndiGolog programs.

Definition 7.8 (Uniform Formulas)**.** According to (Reiter, 2001) a formula is said to be *uniform in* σ if σ is a situation term and it holds for the formula that

- it doesn't mention the predicates *Poss* or $\sqsubset$, where $\sqsubset$ denotes the concept of a *proper subsequence* of situations.
- it doesn't quantify over situations.
- it doesn't mention equality on situations.
- it doesn't mention any other term of sort *situation* than σ as situation argument for a fluent.

Definition 7.9 (Function Symbol *do*)**.** The binary function symbol *do* is defined as

$$do : action \times situation \rightarrow situation$$

The interpretation is that $do(a,s)$ denotes the successor situation resulting from the execution of action a in situation s (Reiter, 2001).

Definition 7.10 (Projection Problem)**.** According to (Reiter, 2001) the projection problem in the situation calculus is defined as follows: Given a basic action theory $\mathcal{D}$, a sequence of ground action terms $[a_1,\ldots,a_n]$ and a goal formula $G(s)$ that is uniform in s, determine whether or not

$$\mathcal{D} \models G(do([a_1,\ldots,a_n],S_0)).$$

That is, find out whether or not a goal G holds in world resulting from performing a certain sequence of actions.

Definition 7.11 (Regression)**.** (Reiter, 2001) has shown that regression is a mechanism to solve the projection problem. Therefore, he formulated the *regression theorem*, as follows. Given a regressable sentence W of the language $\mathcal{L}_{sitcalc}$ that mentions no functional fluents and a basic action theory $\mathcal{D}$ then it holds that

$$\mathcal{D} \models W \text{ iff } \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \mathcal{R}[W],$$

with $\mathcal{R}$ being the regression operator that returns a sentence logically equivalent to W, but uniform in S_0. That is, a sentence W is transformed into a logical equivalent sentence that mentions *only* the initial situation S_0, which is a much simpler entailment because it reduces the evaluation of a regressable sentence to a theorem proving task in the initial theory $\mathcal{D}_{S_0}$.

Definition 7.12 (Progression)**.** Progression is the alternative to regression to solve the projection problem. (De Giacomo et al., 2009) provided a definition as follows. Given a basic action theory $\mathcal{D}$ and any goal formula ϕ, find a $\mathcal{D}_0'$ such that

$$\mathcal{D} \models \phi[do([a_1,\ldots,a_n],S_0)] \text{ iff } \mathcal{D}_{una} \cup \mathcal{D}_0' \models \phi[S_0],$$

where $\mathcal{D}_0'$ is a database transformed by the progression operator $\mathcal{P}$ such that $\mathcal{D}_0' = \mathcal{P}(\mathcal{D}_{S_0},[a_1,\ldots,a_n])$. Further, note that (Lin and Reiter, 1997) showed that progression is not always feasible. However, (Vassos, 2009) discusses various action theories that are restricted in some form and shows how such action theories can be progressed.

7.2. Situation Calculus (BATs) $\leftrightarrow$ Databases, Ground Formula Evaluation

To establish a connection between the semantics of relational databases and situation calculus, we rely on the work done by (De Giacomo and Palatta, 2000). Our basic action theory $\mathcal{D}_{YAGI}$ underlies the following constraints:

- The sort *String* is countably infinite at worst and each element of the sort *String* is represented as a constant for which the unique name assumption holds. Since we restricted $\langle assignment \rangle$ to finite sets any fluent can hold only for a finite number of different parameter vectors at any given point in time.
- The initial database $\mathcal{D}_{S_0}$ is specified to be in closed form, i.e. the theory either logically implies $F(\vec{x}, S_0)$ or $\neg F(\vec{x}, S_0)$ for each fluent F and each parameter vector $\vec{x}$.

Under similar constraints, (De Giacomo and Palatta, 2000) showed how fluents can be represented, formulas can be evaluated and the state of the system can be changed using database semantics. We will introduce a similar model for YAGI in the next sections.

7.2.1. Progression in YAGI

To solve the projection task in YAGI we use progression according to Definition 7.12. Even though regression has proved to be a powerful mechanism to reason about actions, it imposes the serious drawback that it always has to regress back to the initial situation. If some agent performed lots of actions during its lifetime the history of actions that must be taken into account can be considerably huge, which implies lots of computational work (De Giacomo and Palatta, 2000). Since progression changes its database of the initial situation it doesn't suffer from this drawback. However, one major problem with progression is that it is not always feasible. (Lin and Reiter, 1997) demonstrated relatively simple basic action theories where no progression operator exists.

However, we rely on the work done by (Vassos et al., 2008) who defined the following terminology:

A successor state axiom is said to be *local effect* if for an action $A(\vec{x})$ that changes the truth value of a fluent $F(\vec{y}, s)$ it holds that and $\vec{y}$ is contained in $\vec{x}$. If all successor state axioms in $\mathcal{D}_{ssa}$ are *local effect*, then the basic action theory is *local effect*. Further, a successor state axiom is said to be *strictly local effect* if it is *local effect* and if the change of the fluent $F(\vec{y}, s)$ also depends on a fluent $G(\vec{z}, s)$ then $\vec{z}$ is also contained in $\vec{x}$. Consequently, a basic action theory is *strictly local effect* if all successor state axioms in $\mathcal{D}_{ssa}$ are *strictly local effect* and the basic action theory includes a set with uniqueness of names axioms for constants. Ultimately, (Vassos et al., 2008) showed that for a *strictly local effect* basic action theory a first-order strong progression always exists and it can also be computed, i.e. it is guaranteed to be finite.

What remains for discussion is whether or not the YAGI basic action theory $\mathcal{D}_{YAGI}$ is *strictly local effect*. Recall that the successor state axiom for each fluent F in YAGI is of the form $F(\vec{x}, do(a, s)) \equiv a = addF(\vec{x}) \vee F(\vec{x}, s) \wedge a \neq removeF(\vec{x})$ and all successor state axioms in $\mathcal{D}_{ssa}$ are of this form. Since the only parameter vector involved is $\vec{x}$, the only fluent involved is F and the YAGI basic action theory enforces uniqueness of names for constants via the set of axioms in $\mathcal{D}_{unc}$ the *strictly local effect* condition holds and we conclude that $\mathcal{D}_{YAGI}$ is always finitely first-order progressable.

7.2.2. Fluent Representation

A fluent (or fact) F with arity m is represented as a database table T_F composed by m columns $(f_1, \ldots, f_m)$. Further, we define $R_F[x_1, \ldots, x_m]$ to be a row in the database table for the fluent F that represents a parameter vector $\vec{x} = \langle x_1, \ldots, x_m \rangle$. Then, we define that for all fluents F and all parameter vectors $\vec{x}$ it holds that

$$\mathcal{D} \models F(\vec{x}, s) \Leftrightarrow R_F[x_1, \ldots, x_m] \in T_F$$

$$\mathcal{D} \not\models F(\vec{x},s) \Leftrightarrow R_F[x_1,\ldots,x_m] \notin T_F$$

That is, if the theory $\mathcal{D}$ entails that the fluent $F(\vec{x},s)$ holds then the row that corresponds to $\vec{x}$ is stored in the database table (and vice versa), otherwise the fluent does not hold for that given parameter vector.

7.2.3. Fluent- and Fact-Declaration

Initially, we start with an empty database, i.e. a database without any tables. This corresponds to an initial situation S_0 where no fluent holds, i.e. $\forall\vec{x}.F(\vec{x},S_0) \equiv False$ holds for all fluents F. The execution of $\langle$*fluent_decl*$\rangle$ or $\langle$*fact_decl*$\rangle$ for a fluent (or fact) F results in a new database table representing the fluent F as discussed in the section above. Note that the sole existence of a new database table for F doesn't make the fluent hold for any given parameter vector $\vec{x}$, i.e. it still holds that $\forall\vec{x}.F(\vec{x},S_0) \equiv False$.

7.2.4. Successor State Axioms

For each declared fluent F we specified that a successor state axiom of the form $F(\vec{x},do(a,s)) \equiv a = addF(\vec{x}) \vee F(\vec{x},s) \wedge a \neq removeF(\vec{x})$ is added to $\mathcal{D}_{ssa}$. We have to show that the semantics of this type of successor state axiom is reflected in our database implementations. Therefore, we use the function *exec* specified in Definition 7.2. The interpretation is that $exec(a,b)$ denotes the *successor database* resulting from the execution of an situation calculus action a w.r.t. the database b. Due to the fact that YAGI is specified to only use two types of situation calculus actions (namely *addF* and *removeF* as used in the successor state axiom above) we specify *exec* as

$$exec(a,b) = \begin{cases} b' = \mathcal{T}_F^b \cup R_F[\vec{x}], & \text{if } a = addF(\vec{x}). \\ b' = \mathcal{T}_F^b \setminus R_F[\vec{x}], & \text{if } a = removeF(\vec{x}), \end{cases}$$

where by $\mathcal{T}_F^b$ we mean the table that corresponds to the fluent F in the database b and $R_F[\vec{x}]$ is the row that corresponds to the parameter vector $\vec{x} = \langle x_1,\ldots,x_m\rangle$ in the database table representing the fluent F. The interpretation is that depending on the action a either a row is added or removed from the database table of the corresponding fluent, leading to the new database b'. Now, recall that we specified that a fluent $F(\vec{x},s)$ holds if $\mathcal{D} \models F(\vec{x},s) \Leftrightarrow R_F[x_1,\ldots,x_m] \in T_F$ holds and $R_F[x_1,\ldots,x_m]$ is exactly the row that is added or removed depending on the action (*addF* or *removeF*) from the successor state axiom. Hence, our database semantics reflects exactly the successor state axiom described above. Since (De Giacomo and Palatta, 2000) showed correspondence between successor state axioms and SQL commands in general we consider our definition to be a more restricted version of that since it deals only with specific types of successor state axioms.

Furthermore, we want to mention that adding and removing rows is implemented in a straight-forward manner using the corresponding SQL *insert*- and *delete*-statements, as follows:

```
INSERT INTO T_F
VALUES ('x1',...,'xm')
```

Listing 7.1: SQL Schematic For Action $addF(\vec{x})$

```
DELETE FROM T_F
WHERE (f1 = 'x1' AND f2 = 'x2' AND ... AND fm = 'xm');
```

Listing 7.2: SQL Schematic For Action $removeF(\vec{x})$

Based on this definition of how successor state axioms relate to our database semantics we analyze the different types of YAGI assignments.

Add-Assignment

Let there be an add-assignment of the form `F += F_sigma;`, with F and F_σ being fluents. Then such an assignment gets - by specification - transformed into a YAGI loop of the form

```
foreach <$x1,...,$xn> in F_sigma do
  addF($x1,...,$xn);
end for
```

Given this YAGI loop, we have to show that the fluent F holds for all parameter vectors $\vec{x}$ of F_σ after the loop body has been executed. Formally, we have to show that $\forall \vec{x}.F_\sigma(\vec{x},d) \rightarrow F(\vec{x},d')$, with d being the database *before* that loop gets executed and d' being the database *after* the loop got executed. Depending on the number of parameter vectors for which F_σ holds (denoted as $|F_\sigma|$) we argue inductively, as follows:

- $|F_\sigma| = 0$: Based on the specification of YAGI *For-Loops* in Section 5.4.6 the loop doesn't get executed. Hence, *addF* doesn't get executed and the fluent F stays unaffected. Thus, the argument holds trivially.
- $|F_\sigma| = 1$: The loop gets executed exactly once, i.e. one single *add* action gets executed. This is the exact semantics of the successor state axiom as described above. Thus, the argument holds according to the specification of the successor state axiom.
- $|F_\sigma| > 1$: Every time the loop gets executed one single *add* action gets executed for a parameter vector $\vec{x}$. Due to the specification of the successor state axioms it holds that $F(\vec{x},d') \equiv True$ for the chosen parameter vector $\vec{x}$. That exact vector gets removed from F_σ according to the specification of the semantics of YAGI *For-Loops*, hence it holds that $|F_\sigma|$ gets decreased by one every loop iteration. Consequently, the base case of $|F_\sigma| = 1$ is guaranteed to be reached after a finite number of iterations. Thus, the argument holds due to the specification of the successor state axioms and the induction hypothesis.

Note that the assignment expansion into a loop as described above ultimately expands to a sequence of executions of situation calculus simple actions $addF$, so the effect of the YAGI assignment is equivalent to $\mathcal{D} \models \phi[do([addF(\vec{x_1}), addF(\vec{x_2}), \ldots, addF(\vec{x_k})], S_0)]$, for each of the k parameter vectors the fluent F_σ holds for.

Remove-Assignment

Let there be a remove-assignment of the form `F -= F_sigma;`, with F and F_σ being fluents. Then such an assignment gets - by specification - transformed into a YAGI loop of the form

```
foreach <$x1,...,$xn> in F_sigma do
  removeF($x1,...,$xn);
end for
```

The only difference to the add-assignment discussed above is that a different situation calculus action (*remove* instead of *add*) gets executed, other than that the transformation is exactly the same. Consequently, the inductive argument can be build exactly the same way as for the add-assignment.

Override Assignment

By specification, an override assignment is a remove-assignment followed by an add-assignment. That is, we specified that an override assignment of the form $F = F_\sigma$ makes the fluent F *true* for all and only all tuples in F_σ. In other words, an override assignment removes all elements from F and adds all the tuples from F_σ to it. Consequently, we can express an override assignment as a remove-assignment **F -= F** followed by an add-assignment $\mathbf{F \mathrel{+}= F_\sigma}$. Since we already discussed both of these types of assignments above we don't need to build a special case for override assignments.

Loop Assignment

Loop assignments ultimately collapse to sequences of add- and remove-assignments. That is, a YAGI loop assignment of the form

```
foreach <$x1,$x2,...,$xn> in F do
  F' += {<$x1,$x2,...,$xn>};
end for
```

can be interpreted as a sequence of assignments of the form

```
//foreach <$x1,$x2,...,$xn> in F do
F' += {<$x11,$x21,...,$xn1>};
F' += {<$x12,$x22,...,$xn2>};
F' += {<$x1m,$x2m,...,$xnm>};
//end for
```

for each of the m tuples the fluent F holds for. Hence, the same inductive argument used for *add-assignment* above applies.

Conditional Assignment

Conditional assignments are simply two possible sequences of assignments that get executed depending on the evaluation of a formula, i.e. the conditional. Consequently, the same reasoning as above applies.

7.2.5. Ground Formula Evaluation

Having defined how fluents are represented and how the truth value of fluents can be changed we proceed with the evaluation of YAGI formulas. More precisely, we deal with first-order formulas that are in closed form (or *ground*), i.e. first-order formulas without free variables. Whereas (De Giacomo and Palatta, 2000) showed how every situation calculus formula of arbitrary complexity can be directly translated to SQL we decided to evaluate formulas using C++ machinery rather than to translate formulas directly to SQL. If a fluent is involved in a formula we fetch its data from the database using a SQL *select* statement and store its result in appropriate C++ data structures. We interpret the data from a row in a database table as a tuple of strings, hence we build an instance of `std::vector<std::string>` for each row in a table. The set of all such tuples (i.e. an instance of `std::vector< std::vector<std::string> >`) represents the state of the fluent, which we subsequently use for formula evaluation[3].

For the discussion of formula evaluation semantics we argue inductively on the structure of the formula. Since an empty formula is forbidden according to the syntactical specification of $\langle formula \rangle$ we use the evaluation of a fluent according to the definition of fluent representation in Section 7.2.2 for the base case, i.e. let φ be a formula of the form $\varphi = F(\vec{t})$ for a fluent F and a vector of terms $\vec{t} = \langle t_1, \ldots, t_m \rangle$, then we can decide whether or not φ holds according to the semantics in Section 7.2.2. Note that we can omit the situation term since we only deal with a single situation, i.e. the database as a snapshot of the world generated via progression. For the inductive step we treat each case separately, as follows:

- **Truth Values**: *true* and *false* are constants, hence their truth value is independent of a specific model. Truth values are implemented using the C++ datatype *bool*, hence its evaluation is trivial.
- **Comparisons**: String values are constants, hence comparisons of string values work independently of a specific model, according to the specification in Section 5.3.3. For comparisons of sets of tuples we need to consider two separate cases. In the first case, the compared sets solely consist of tuples with constant string values. In this case comparison works independent of a specific model. In the

[3] We want to mention that using a vector of strings as intermediate representation to evaluate formulas implies an overhead in execution time compared to the idea lined out by (De Giacomo and Palatta, 2000) to evaluate formulas directly using SQL. We plan to pursue the formula evaluation ideas discussed by (De Giacomo and Palatta, 2000) in a future implementation.

second case, a fluent is involved in a comparison. In this case, the set is deduced from the database table as described above. On this deduced set, comparison works exactly like it does for a set of tuples with constants.

- **Logical Connectives**: The logical connectives **and** ($\wedge$) and **or** ($\vee$) are implemented using their C++ equivalents && and ||. The **implies** ($\rightarrow$) connective is rewritten into a logically equivalent term of the form $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$.
- **Negation**: The operator $\neg$ is implemented using the equivalent C++ negation operator !.
- **First-Order Quantifiers**: All-quantified formulas are evaluated for all tuples in a certain set, i.e. for each tuple in a set its values are bound to variables and the formula gets evaluated. The formula holds iff it holds for all bindings, i.e. for all tuples in the set. Exists-quantified formulas follow similarly, with the difference that the formula holds if one binding that fulfills the formula is found. Note that quantifiers operate on the domain of the involved fluent and we currently deal with finite domains only. Furthermore, we handle the special case with no **such**-block present separately, i.e. `exists <$x> in F` and `all <$x> in F` hold iff there is at least one element for that the fluent *F* holds.
- **Operator *in***: The evaluation of **in** is implemented as a simple search of a tuple in a set, i.e. `<"a"> in {<"a">,<"b">,<"c">}` is true iff the left-hand side tuple is an element of the right-hand side set.

7.2.6. Action Preconditions

Since YAGI actions get rewritten into IndiGolog procedures a YAGI action declaration has no effect on $\mathcal{D}_{ap}$. More precisely, the precondition of a YAGI action gets rewritten into an IndiGolog *test*-statement of the form *test* ϕ;, where *phi* is the formula that corresponds to the YAGI action precondition. Hence, the YAGI action precondition becomes a program execution statement rather than a situation calculus action precondition. Note that the formula ϕ from the *test*-statement gets evaluated according to the semantics discussed in Section 7.2.5 above. Hence, the formula ϕ holds if and only if the YAGI action precondition formula holds. Consequently, we want to emphasize that according to this specification a YAGI action declaration *never* affects $\mathcal{D}_{ap}$.

What remains for discussion regarding situation calculus action preconditions are the action preconditions of *add* and *remove* for each declared fluent. Note that *add* and *remove* are the only situation calculus actions that can occur, consequently we only need to discuss their preconditions and how they are implemented. Essentially, the preconditions of *add* and *remove* are specified to ensure that only elements of the correct sort can be added and removed. Due to the fact that we store the sort of each domain of each declared fluent in separate database tables we enforce the preconditions by checking if each element of the parameter vector $\vec{x}$ that is passed to *add* or *remove* exists in the corresponding database table. The corresponding action is executed if and only if all elements of $\vec{x}$ belong to the respective sort of the declared fluent.

7.3. IndiGolog $\leftrightarrow$ YAGI Program Execution

Having defined how our database semantics relates to situation calculus in the previous section, we proceed with the execution of YAGI programs and how such an execution semantics relates to IndiGolog transition semantics. To accomplish this task we restate the transition semantics specified by (De Giacomo et al., 2009), state our YAGI transition semantics (i.e. *YagiTrans* and *YagiFinal*) and show the relation between these transition semantics predicates for each YAGI program element.

7.3.1. YAGI Program Representation

When giving definitions for *YagiTrans* and *YagiFinal* we used terms such as α being a *program* and α' being a *remaining program*. Here, we give a definition of what these terms mean in context of our implementation. A YAGI program is internally represented by its AST, i.e. a tree with a root node «program» and children «$stmt_1$»,...,«$stmt_n$» that represent the statements and declarations[4] in a YAGI program as specified by the syntax of $\langle program \rangle$ in Section 5.10.1. This is what will refer to as *program* α in this context. *YagiTrans* then describes the execution of one elementary step in the AST according to its specification, leading to the *remaining program*, i.e. the remaining AST that represents the program α' that remains to be executed[5]. If all AST nodes have been visited the empty YAGI program (denoted as *null*) remains. The execution of a simple YAGI program is illustrated in Figure 7.1 below. The subtree of the AST colored in blue denotes the current statement that is being executed. After the execution of the first statement the subtree that represents the first statement vanishes, denoted via dotted lines. The *remaining program* is the conditional statement, which is also the last statement in the YAGI program. Consequently, the empty program *null* remains and the program is allowed to terminate. Note that in the figure below we assume that both transition predicates (i.e. YagiTrans for the test-statement and YagiTrans for the conditional) hold, i.e. both transitions can be executed successfully.

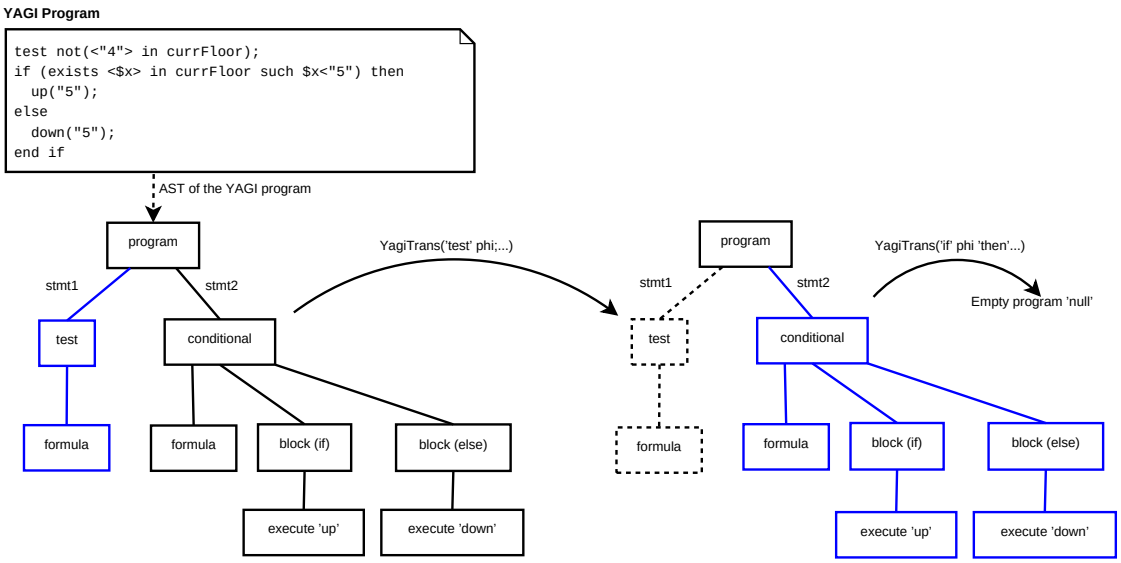

Figure 7.1.: AST Execution Schematic

Having defined our structure of a program we proceed with the mapping of YAGI programs to situation calculus and IndiGolog programs.

7.3.2. YAGI Program Mapping

To accomplish a mapping from an arbitrary YAGI program to situation calculus and IndiGolog we provide a mapping function for each of the YAGI language constructs (except procedures, which are discussed separately in Section 7.3.11) to IndiGolog and situation calculus, as follows:

1. Empty program

$$yagiToGolog(null) = nil$$

[4]Note that declarations of YAGI actions, procedures etc. are also part of the AST. Due to the fact that those statements don't have any transitional semantics they are not discussed in the context of transition semantics in the following sections.

[5]We don't actually remove already executed statements from the AST because it has no effect but to give a slight hit in execution performance. We simply traverse the AST and execute the statements in their given order.

2. Primitive Action

$$yagiToGolog(a) = a$$

3. Test

$$yagiToGolog(\textbf{test } \phi \textbf{ ;}) = \phi?$$

4. Choose

$$yagiToGolog(\textbf{choose } \alpha_1 \textbf{ or } \alpha_2)^6 = yagiToGolog(\alpha_1) \mid yagiToGolog(\alpha_2)$$

5. Pick

$$yagiToGolog(\textbf{pick } \langle v_1, \ldots, v_n \rangle \textbf{ from } F \textbf{ such } \alpha) = \pi v.yagiToGolog(\alpha)$$

6. Conditional

$$yagiToGolog(\textbf{if } \phi \textbf{ then } \alpha_1 \textbf{ else } \alpha_2 \textbf{ end if}) = \textbf{if } \phi \textbf{ then } yagiToGolog(\alpha_1) \textbf{ else } yagiToGolog(\alpha_2) \textbf{ endIf}$$

7. While

$$yagiToGolog(\textbf{while } \phi \textbf{ do } \alpha \textbf{ end while}) = \textbf{while } \phi \textbf{ do } yagiToGolog(\alpha) \textbf{ endWhile}$$

8. Sequence

$$yagiToGolog(\alpha_1; \alpha_2) = yagiToGolog(\alpha_1) \text{ ; } yagiToGolog(\alpha_2)$$

Note that we didn't provide a program mapping for a YAGI for-loop to IndiGolog. The reason for this is that IndiGolog has no language construct that has the intended semantics of a YAGI for-loop. Therefore, we specified the semantics of a YAGI for-loop in terms of a rewriting into *while* and *pick* in Section 5.4.6. Because for-loops are rewritten into *while* and *pick* we need no separate program mapping for a YAGI for-loop since we can express its mapping using *while* and *pick*.

Based on the mappings from above we provide YAGI transition semantics for each of these language constructs and show their relation to the IndiGolog transition semantics. In the following sections free variables are assumed to be universally quantified.

7.3.3. Empty Program

Let *nil* be the empty IndiGolog program. Then *Trans* and *Final* for this empty program are specified as

$$Trans(nil, s, \delta', s') \equiv False$$

$$Final(nil, s) \equiv True$$

Essentially, this means that an empty program is always allowed to legally terminate and that an empty program is under no circumstance able to evolve into anything. Then, we define *YagiTrans* and *YagiFinal* as

$$YagiTrans(null, b, \alpha', b') \equiv False$$

$$YagiFinal(null, b) \equiv True,$$

which are exactly the same transitions as for an empty IndiGolog program.

[6] Note that we syntactically allow an arbitrary number of blocks to choose from, i.e. **choose** α_1 **or** α_2 **or** ... **or** α_n. The transformation works identically for all of these blocks, so we can - without loss of generality - reduce our analysis to the case with two blocks.

7.3.4. Primitive Actions

For situation calculus primitive actions *Trans* and *Final* are specified as

$$Trans(a,s,\delta',s') \equiv Poss(a[s],s) \wedge \delta' = nil \wedge s' = do(a[s],s)$$

$$Final(a,s) \equiv False$$

Essentially, this means that (a,s) evolves to $(nil, do(a[s],s))$ iff the execution of the action a is possible in the situation s and after the execution of a nothing remains to be executed. Moreover, the execution of a primitive action can never be final, i.e. the action must be executed before the program can legally terminate. For YAGI, we specify *YagiTrans* and *YagiFinal* as

$$YagiTrans(a,b,\alpha',b') \equiv \varphi_{AP}(a) \wedge \alpha' = null \wedge b' = exec(a,b)$$

$$YagiFinal(a,b) \equiv False$$

$\varphi_{AP}(a)$ is the action precondition formula for the primitive action a (which can only be either *addF* or *removeF* for a fluent F), so $\varphi_{AP}(a)$ is guaranteed to hold according to the discussion in Section 7.2.6. Again, note that we don't have a situation term in $\varphi_{AP}(a)$ since we always progress our database, hence always deal just with the current situation. Further, the progression to s' is reflected by the execution of the function $exec(a,b)$ because it implements exactly the successor state axioms specified for *addF* and *removeF* according to Definition 7.2.

7.3.5. Test

For IndiGolog's *Test* actions *Trans* and *Final* are specified as

$$Trans(\phi?,s,\delta',s') \equiv \phi[s] \wedge \delta' = nil \wedge s' = s$$

$$Final(\phi?,s) \equiv False,$$

which essentially means that *Trans* holds iff the formula under test holds in the current situation and after executing the test nothing remains to be executed. Furthermore, ϕ? can never be final, i.e. its execution is mandatory. For YAGI, we define

$$YagiTrans(\textbf{test}\ \phi\ \textbf{;}, b, \alpha', b') \equiv \phi[b] \wedge \alpha' = null \wedge b' = b$$

$$YagiFinal(\textbf{test}\ \phi\ \textbf{;}, b) \equiv False,$$

where $\phi[b]$ is the evaluation of the YAGI formula w.r.t. the database b and $\phi[s]$ is the mapping of the IndiGolog representation of the condition ϕ to the corresponding situation calculus formula, see (De Giacomo et al., 2009) for details. Recall that we already showed how the evaluation of YAGI formulas follows the specification in Section 7.2.5. Consequently, test also behaves according to the specification. Further, note that ϕ? does not change the successor situation as **test** ϕ **;** doesn't change the database.

7.3.6. Choose

For a non-deterministic branch *Trans* and *Final* are specified as

$$Trans(\delta_1 \mid \delta_2, s, \delta', s') \equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s')$$
$$Final(\delta_1 \mid \delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s),$$

meaning that $(\delta_1 \mid \delta_2, s)$ can evolve to (δ', s') iff either of the two branches can do so. Consequently, we define *YagiTrans* and *YagiFinal* as

$$YagiTrans(\textbf{choose } \alpha_1 \textbf{ or } \alpha_2 \textbf{ end choose}, b, \alpha', b') \equiv YagiTrans(\alpha_1, b, \alpha', b') \vee YagiTrans(\alpha_2, b, \alpha', b')$$
$$YagiFinal(\textbf{choose } \alpha_1 \textbf{ or } \alpha_2 \textbf{ end choose}, b) \equiv YagiFinal(\alpha_1, b) \vee YagiFinal(\alpha_2, b),$$

which expresses the same semantics in terms of YAGI.

7.3.7. Pick

For a non-deterministic choice of argument *Trans* and *Final* are specified as

$$Trans(\pi v.\delta, s, \delta', s') \equiv \exists x.Trans(\delta_x^v, s, \delta', s')$$
$$Final(\pi v.\delta, s) \equiv \exists x.Final(\delta_x^v, s),$$

meaning that there exists an x such that (δ_x^v, s) can evolve to (δ', s') and δ_x^v is a program where v is substituted with the variable x. Analogously, we specify that

$$YagiTrans(\textbf{pick } \vec{\nu} \textbf{ from } F \textbf{ such } \alpha \textbf{ end pick}, b, \alpha', b') \equiv \exists_{\vec{S_F}} \vec{x}.YagiTrans(\delta_{\vec{x}}^{\vec{\nu}}, b, \alpha', b')$$
$$YagiFinal(\textbf{pick } \vec{\nu} \textbf{ from } F \textbf{ such } \alpha \textbf{ end pick}, b) \equiv \exists_{\vec{S_F}} \vec{x}.YagiFinal(\delta_{\vec{x}}^{\vec{\nu}}, b)$$

where $\exists_{\vec{S_F}}$ is the existential quantifier over the sort of the fluent F (which is a more restricted quantification than $\exists x$ in *Trans* and *Final*) and $\delta_{\vec{x}}^{\vec{\nu}}$ is a YAGI program where $\vec{\nu}$ is substituted with the variable vector $\vec{x}$.

7.3.8. Conditional

For synchronized conditionals *Trans* and *Final* are specified as

$$Trans(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf}, s, \delta', s') \equiv \phi[s] \wedge Trans(\delta_1, s, \delta', s') \vee \neg\phi[s] \wedge Trans(\delta_2, s, \delta', s')$$
$$Final(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf}, s) \equiv \phi[s] \wedge Final(\delta_1, s) \vee \neg\phi[s] \wedge Final(\delta_2, s),$$

saying that the conditional can evolve to (δ', s') if $\phi[s]$ holds and (δ_1, s) can do so (if-clause) or $\phi[s]$ does not hold and (δ_2, s) can do so (else-clause). *YagiTrans* and *YagiFinal* are defined equivalently as

$$YagiTrans(\textbf{if } \phi \textbf{ then } \alpha_1 \textbf{ else } \alpha_2 \textbf{ end if}, b, \alpha', b') \equiv \phi[b] \wedge YagiTrans(\alpha_1, b, \alpha', b') \vee \neg\phi[b] \wedge YagiTrans(\alpha_2, b, \alpha', b')$$
$$YagiFinal(\textbf{if } \phi \textbf{ then } \alpha_1 \textbf{ else } \alpha_2 \textbf{ end if}, b) \equiv \phi[b] \wedge YagiFinal(\alpha_1, b) \vee \neg\phi[b] \wedge YagiFinal(\alpha_2, b),$$

where $\phi[b]$ is the evaluation of a YAGI formula w.r.t. the database b.

7.3.9. While

For a synchronized loop *Trans* and *Final* are specified as

$$Trans(\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile}, s, \delta', s') \equiv \exists \gamma.(\delta' = \gamma; \textbf{while } \phi \textbf{ do } \delta) \wedge \phi[s] \wedge Trans(\delta, s, \gamma, s')$$

$$Final(\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile}, s) \equiv \neg\phi[s] \vee Final(\delta, s)$$

Analogously, we specify *YagiTrans* and *YagiFinal* as

$$YagiTrans(\textbf{while } \phi \textbf{ do } \alpha \textbf{ end while}, b, \alpha', b') \equiv \exists \gamma.(\alpha' = \gamma; \textbf{while } \phi \textbf{ do } \alpha \textbf{ end while}) \wedge \phi[b] \wedge YagiTrans(\alpha, b, \gamma, b')$$

$$YagiFinal(\textbf{while } \phi \textbf{ do } \alpha \textbf{ end while}, b) \equiv \neg\phi[b] \vee YagiFinal(\alpha, b)$$

Note that the AST remains unchanged in case of a transition, i.e. the execution of *YagiTrans* for a while-statement leaves the remaining AST equal to the AST before its execution.

7.3.10. Sequence

For a sequence of IndiGolog statements *Trans* and *Final* are specified as

$$Trans(\delta_1; \delta_2, s, \delta', s') \equiv \exists \gamma.\delta' = (\gamma; \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s')$$

$$Final(\delta_1; \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s),$$

stating that $(\delta_1; \delta_2, s)$ can either evolve to $(\delta_1'; \delta_2', s')$ given that (δ_1, s) can evolve to (δ_1', s') or to (δ_2', s') given that (δ_1, s) is a final configuration and (δ_2, s) can evolve to (δ_2', s') (De Giacomo et al., 2009). In YAGI, we reflect this semantics as

$$YagiTrans(\alpha_1; \alpha_2, b, \alpha', b') \equiv \exists \gamma.\alpha' = (\gamma; \alpha_2) \wedge YagiTrans(\alpha_1, b, \gamma, b') \vee YagiFinal(\alpha_1, b) \wedge YagiTrans(\alpha_2, b, \alpha', b')$$

$$YagiFinal(\alpha_1; \alpha_2, b) \equiv YagiFinal(\alpha_1, b) \wedge YagiFinal(\alpha_2, b).$$

7.3.11. Procedures

Traditional Golog and ConGolog/IndiGolog use different approaches when it comes to the formalization of procedures. In traditional Golog, procedures are *macro expanded* to situation calculus formulas, i.e. a procedure call is replaced by its definition and parameters of the procedure call are evaluated w.r.t. the current situation and then passed in a *call-by-value* manner. However, there is no straight-forward way to macro expand recursive procedure calls and macro expansion leads to a less expressive formalism, as discussed by (Levesque et al., 1994).

ConGolog and IndiGolog use a different formalization technique that is able to deal with unbound recursive procedure calls. In ConGolog, procedure calls are handled in a standard way with call-by-value semantics and lexical scoping, which is a fundamentally different approach compared to macro expansion. The price to pay is that this approach requires *Trans* and *Final* to be defined as a second-order formula. A detailed description of those second-order predicates can be found in (De Giacomo et al., 2000).

In YAGI, we treat procedures similar to ConGolog. That is, procedures don't get macro expanded, they are treated like procedures in a traditional manner, i.e. parameters are passed via call-by-value and the procedure gets subsequently executed in its own environment. This modus operandi resembles a typical way to deal with parameter passing and procedure execution from a compiler construction point of view (Aho et al., 2007).

7.4. Consequence

Based on the specification of how the YAGI database semantics relates to situation calculus and how YAGI transition semantics relates to IndiGolog transition semantics we conclude that the implementation follows the specified semantics. This follows directly from the definition of the base cases (i.e. YAGI program mappings for *empty program*, *primitive action* and *test*) and the inductive definition for all other cases. Further, recall that we use the *visitor* design pattern in our implementation to execute YAGI programs, as mentioned in Section 6.2.2. The AST traversal via the visitor implementation reflects exactly the semantics of *YagiTrans* and *YagiFinal* as specified in this chapter.

Chapter 8

Evaluation

In this chapter, we provide an evaluation of our YAGI implementation. We start with a description of our evaluation setting in Section 8.1 and continue with a description of the mechanisms we use for measuring program execution time in Section 8.2. Then, we present the evaluation results for the elevator domain and the blocks world in Section 8.3 and Section 8.4, respectively. Finally, we finish with a discussion of our evaluation results in Section 8.5.

8.1. Evaluation Setting

Due to the fact that the interpreter implementation of IndiGolog is considered to be in "alpha" stage[1] we use the classical SWI-Prolog-based Golog interpreter provided by the University of Toronto[2] for comparison. We compare the performance of YAGI and Golog using two different domains, namely *Elevator Controller* and *Blocks World*. For each of these domains we randomly generate ten initial situations for each test case, i.e. we run each program with ten random initial situations to increase the precision of the measured run-time information for each test case. For each test case and each program the result is the 4-tuple $\langle \mu[s], \sigma[s], to[\%], succ[\%] \rangle$, where $\mu[s]$ and $\sigma[s]$ are the mean and standard deviation over the run-times[3] of the program given ten random initial situations in seconds, $to[\%]$ is the percentage of timeouts (i.e. the program was not able to find a solution for a given initial situation consuming a certain amount of resources[4]) and $succ[\%]$ is the percentage of initial situations where the given program was able to find a valid solution. For each of the domains we use different implementations (i.e. different YAGI- and Golog-programs) for comparison, which are as follows:

I **YAGI (non-deterministic, no planning)**: The YAGI program makes use of non-deterministic constructs *pick* and *choose* and executes the program online, i.e. *without* searching for a valid trace beforehand. As a consequence, this implementation may or may not find a solution based on the output of the random number generator, i.e. depending on the output of the random number generator there might arise a situation where YAGI program execution can't continue, leading to the termination of the program. The listings for elevator and blocks world can be found in Appendix C.1.1 and Appendix C.2.1, respectively.

II **YAGI (conditional, no planning)**: The YAGI program makes use of non-deterministic constructs *pick* and *choose* and executes the program online, i.e. *without* searching for a valid trace beforehand. The

[1] According to `http://www.cs.toronto.edu/cogrobo/main/systems/`. Last visited on November 12th, 2014.
[2] `http://www.cs.toronto.edu/cogrobo/Systems/golog_swi.pl`. Last visited on November 12th, 2014.
[3] Only programs that did *not* time out contribute to the mean and standard deviation calculations.
[4] We define that no solution can be found if a program takes more than 10 minutes to execute or runs out of resources (e.g. memory, threads) before that time.

difference to the program described above is that the program uses conditional constructs to prevent violations of *YagiTrans* and *YagiFinal*. As a consequence, the program should always terminate successfully. The listings for elevator and blocks world can be found in Appendix C.1.2 and Appendix C.2.2, respectively.

III **YAGI (non-deterministic, full planning)**: The YAGI program makes use of non-deterministic constructs *pick* and *choose* and executes the program offline, i.e. it searches for a valid trace beforehand. As a consequence, the program executed online should always terminate - if the search procedure is able to find a valid trace. The listings for elevator and blocks world can be found in Appendix C.1.3 and Appendix C.2.3, respectively.

IV **YAGI (conditional, full planning)**: The mode of execution is identical to the program described above, the difference is that the program uses conditionals to prevent execution traces from failing. The listings for elevator and blocks world can be found in Appendix C.1.4 and Appendix C.2.4, respectively.

V **Golog**: The 'classic' Golog implementation of the respective problem domain. It can be considered as the Golog counterpart of *YAGI (non-deterministic, full planning)* since Golog always performs full planning and the input program is non-deterministic. The listings for elevator and blocks world can be found in Appendix C.1.5 and Appendix C.2.5, respectively.

VI **Golog (conditional)**: A modified version of the 'classic' Golog implementation that uses conditionals to preemptively eliminate execution paths that are guaranteed to fail. Therefore, it can be considered as Golog counterpart of *YAGI (conditional, full planning)*. The listing can be found in Appendix C.1.6.

VII **Golog (reordered)**: A Golog program with the order of the statements of the non-deterministic branch operator switched. The intention behind such a modification is that we want to investigate if the order of statements influences the result of the program in any way. The listing can be found in Appendix C.2.6.

8.2. Measurement Techniques

For timing the run-time of the Golog program we use the built-in predicate *statistics(cputime, T)* from SWI Prolog (Wielemaker et al., 2014). More precisely, we capture the CPU time *before* executing the Golog program (*statistics(cputime, T1)*, run the Golog program, capture the CPU time again (*statistics(cputime, T2)* and take the difference $T2 - T1$ as execution time. Similarly, we use the high-resolution timing functions from the *chrono* namespace of C++ (Gregoire et al., 2011) to measure the run-time of the YAGI program, i.e. we capture the current time point via chrono's *high_resolution_clock*, execute the YAGI program, capture the time point again and take the difference of time points as execution time.

8.3. Elevator Controller

We use a slightly modified version[5] of the well-known elevator example from (Reiter, 2001). For the different test cases we use a different number of total and active floors, i.e. there exist n floors and for a random $m < n$ floors the fluent *on* initially holds. We state the concrete numbers of total and active floors for each test case T_n in parentheses, i.e. for a test case T_i the interpretation of $T_i(n,m)$ is that for the i-th test case the total number of floors is n and m floors are active. Further, we randomize which floors are active and on which floor the elevator initially resides for each of the ten iterations per test case. We present the measurement results for each test case in the tables below.

[5] Taken from `http://www.eecs.yorku.ca/course_archive/2006-07/W/3402/asg3/simple_elevator.swipl`. Slightly adapted version for SWI Prolog. Last visited on November 12th, 2014.

Test Case $T_1(7,2)$				
	Mean μ	Std. Dev. σ	Timeout	Success
YAGI (non-deterministic, no planning)	0.08s	0.03s	0%	10%
YAGI (conditional, no planning)	0.11s	0.01s	0%	100%
YAGI (non-deterministic, full planning)	1.10s	0.07s	0%	100%
YAGI (conditional, full planning)	0.69s	0.06s	0%	100%
Golog	3.10^{-4}s	4.10^{-5}s	0%	100%
Golog (conditional)	3.10^{-4}s	4.10^{-5}s	0%	100%

Table 8.1.: Evaluation Results for the Elevator Example Test Case $T_1(7,2)$

Test Case $T_2(20,10)$				
	Mean μ	Std. Dev. σ	Timeout	Success
YAGI (non-deterministic, no planning)	0.12s	0.08s	0%	0%
YAGI (conditional, no planning)	0.98s	0.35s	0%	100%
YAGI (non-deterministic, full planning)	∞[a]	0s	100%	0%
YAGI (conditional, full planning)	32s	24.4s	0%	100%
Golog	3.10^{-3}s	2.10^{-3}s	0%	100%
Golog (conditional)	3.10^{-3}s	10^{-3}s	0%	100%

Table 8.2.: Evaluation Results for the Elevator Example Test Case $T_2(20,10)$

[a] Program could not find a solution for any of the ten initial situations.

Test Case $T_3(50,25)$				
	Mean μ	Std. Dev. σ	Timeout	Success
YAGI (non-deterministic, no planning)	0.29s	0.20s	0%	0%
YAGI (conditional, no planning)	3.90s	1.40s	0%	100%
YAGI (non-deterministic, full planning)	∞[a]	0s	100%	0%
YAGI (conditional, full planning)	∞[a]	0s	100%	0%
Golog	0.03s	0.01s	0%	100%
Golog (conditional)	0.03s	0.01s	0%	100%

Table 8.3.: Evaluation Results for the Elevator Example Test Case $T_3(50,25)$

[a] Program could not find a solution for any of the ten initial situations.

Test Case $T_4(70,60)$				
	Mean μ	Std. Dev. σ	Timeout	Success
YAGI (non-deterministic, no planning)	0.32s	0.23s	0%	0%
YAGI (conditional, no planning)	8.80s	0.72s	0%	100%
YAGI (non-deterministic, full planning)	∞[a]	0s	100%	0%
YAGI (conditional, full planning)	∞[a]	0s	100%	0%
Golog	0.35s	0.09s	0%	100%
Golog (conditional)	0.28s	0.06s	0%	100%

Table 8.4.: Evaluation Results for the Elevator Example Test Case $T_4(70,60)$

[a] Program could not find a solution for any of the ten initial situations.

Test Case $T_5(100,100)$				
	Mean μ	Std. Dev. σ	Timeout	Success
YAGI (non-deterministic, no planning)	0.66s	0.31s	0%	0%
YAGI (conditional, no planning)	23.6s	6.72s	0%	100%
YAGI (non-deterministic, full planning)	∞^a	0s	100%	0%
YAGI (conditional, full planning)	∞^a	0s	100%	0%
Golog	1.72s	0.42s	0%	100%
Golog (conditional)	1.63s	0.46s	0%	100%

Table 8.5.: Evaluation Results for the Elevator Example Test Case $T_5(100,100)$

[a] Program could not find a solution for any of the ten initial situations.

8.4. Blocks World

The second example we use is an implementation of blocks world, a famous planning problem in the area of artificial intelligence (Nilsson, 1982) (Nilsson, 1998) (Russell and Norvig, 2014). The blocks world domain is described by (Russell and Norvig, 2014), as follows:

> "This domain consists of a set of cubic blocks sitting on a table. The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can only pick up one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top of what other blocks."

An example of a problem instance of the blocks world domain is illustrated in Figure 8.1.

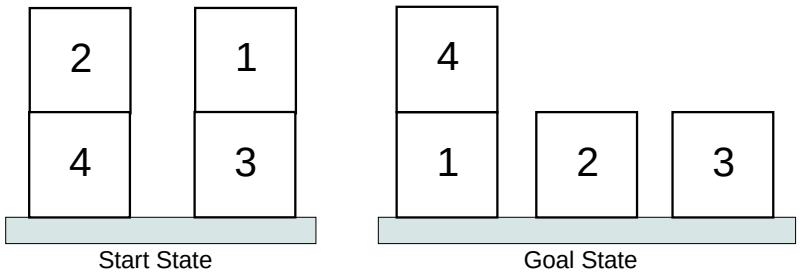

Figure 8.1.: Problem Instance of the Blocks World Domain

The corresponding Golog code that describes the start state from the example above is illustrated in the listing below. The YAGI corresponding code follows similarly.

```
%Fluent 'ontable' describes blocks that sit directly on the table
ontable(4,s0). ontable(3,s0).

%Fluent 'on' describes what block is on top of another
on(2,4,s0). on(1,3,s0).

%Fluent 'clear' describes blocks that are moveable
clear(2,s0). clear(1,s0).
```

Listing 8.1: Golog Fluents for Blocks World Example

Our Golog implementation is based on a version from University of Mainz[6], extended with a set of Golog procedures. For the different test cases we use a different number of blocks and stacks, i.e. there exist n blocks forming $m \leq n$ stacks randomly. We state the concrete numbers of blocks and stacks for each test case T_n in parentheses, i.e. for a test case T_i the interpretation of $T_i(n,m)$ is that for the i-th test case the total number of blocks is n, forming m different stacks.

Further, we randomize the goal state for every iteration. More precisely, we randomly pick one of the predicates `on/2`, `onTable/1` or `clear/1` and randomly pick one or two blocks, depending on the arity of the predicate. The predicate that holds for the picked block(s) is our goal state[7]. We present the measurement results for each test case in the tables below.

Test Case $T_1(4,1)$				
	Mean μ	Std. Dev. σ	Timeout	Success
YAGI (non-deterministic, no planning)	0.08s	0.03s	0%	40%
YAGI (conditional, no planning)	0.74s	0.92s	0%	100%
YAGI (non-deterministic, full planning)	19s	25.8s	0%	100%
YAGI (conditional, full planning)	49s	69s	0%	100%
Golog	10^{-4}s	6.10^{-5}s	40%	60%
Golog (reordered)	10^{-4}s	5.10^{-5}s	40%	60%

Table 8.6.: Evaluation Results for the Blocks World Example Test Case $T_1(4,1)$

Test Case $T_2(5,1)$				
	Mean μ	Std. Dev. σ	Timeout	Success
YAGI (non-deterministic, no planning)	0.12s	0.07s	0%	20%
YAGI (conditional, no planning)	1.07s	1.39s	0%	100%
YAGI (non-deterministic, full planning)	26.7s	37.4s	10%	90%
YAGI (conditional, full planning)	45.4s	62.8s	10%	90%
Golog	2.10^{-4}s	7.10^{-5}s	30%	70%
Golog (reordered)	2.10^{-4}s	8.10^{-5}s	30%	70%

Table 8.7.: Evaluation Results for the Blocks World Example Test Case $T_2(5,1)$

Test Case $T_3(6,3)$				
	Mean μ	Std. Dev. σ	Timeout	Success
YAGI (non-deterministic, no planning)	0.22s	0.14s	0%	0%
YAGI (conditional, no planning)	3s	3.33s	0%	100%
YAGI (non-deterministic, full planning)	47.7s	68.5s	40%	60%
YAGI (conditional, full planning)	29.9s	40.0s	40%	60%
Golog	10^{-4}s	4.10^{-5}s	50%	50%
Golog (reordered)	5.10^{-4}s	6.10^{-4}s	70%	30%

Table 8.8.: Evaluation Results for the Blocks World Example Test Case $T_3(6,3)$

[6] Taken from `http://www.informatik.uni-mainz.de/arbeitsgruppen/informationssysteme/studium/wintersemester-2012/einfuerung-in-die-kuenstliche-intelligenz/uebungszettel/blocksworld-in-prolog/at_download/file`. Last visited on December 1st, 2014.

[7] Additionally, we check that the randomly generated goal does *not* hold initially because that would imply that the initial situation is already a valid solution and nothing needs to be executed.

Test Case $T_4(10,1)$				
	Mean μ	Std. Dev. σ	Timeout	Success
YAGI (non-deterministic, no planning)	0.12s	0.11s	0%	0%
YAGI (conditional, no planning)	2.4s	2.9s	0%	100%
YAGI (non-deterministic, full planning)	6.5s	6.7s	60%	40%
YAGI (conditional, full planning)	13.5s	13.8s	60%	40%
Golog	3.10^{-4}s	2.10^{-4}s	30%	70%
Golog (reordered)	3.10^{-4}s	2.10^{-4}s	70%	30%

Table 8.9.: Evaluation Results for the Blocks World Example Test Case $T_4(10,1)$

Test Case $T_5(10,5)$				
	Mean μ	Std. Dev. σ	Timeout	Success
YAGI (non-deterministic, no planning)	0.27s	0.27s	0%	10%
YAGI (conditional, no planning)	6.4s	6.8s	0%	100%
YAGI (non-deterministic, full planning)	67.00s	78.10s	60%	40%
YAGI (conditional, full planning)	36.30s	25.00s	60%	40%
Golog	2.10^{-4}s	7.10^{-5}s	50%	50%
Golog (reordered)	7.10^{-4}s	undef.[a]	90%	10%

Table 8.10.: Evaluation Results for the Blocks World Example Test Case $T_5(10,5)$

[a] Standard deviation is undefined if there is just one sample.

8.5. Discussion

8.5.1. Runtime

Both scenarios show similar tendencies that Golog outperforms YAGI regarding the measured execution times. The reason for this is that the current implementation of YAGI is considered to be *proof-of-concept*, hence no optimizations regarding run-time and memory consumption have been applied yet. Further, the specification of YAGI inherently includes certain definitions that imply performance overhead compared to Golog, e.g. features like pattern matching and assignment rewriting to for-loops impose a performance overhead by nature. We illustrate the run-times of a Golog and a YAGI implementation of the elevator example in Figure 8.2.

Further, most of the YAGI programs of blocks world show a much higher run-time standard deviation than the Golog programs and the elevator examples for Golog and YAGI. In fact, in many cases the standard deviation is higher than the mean value, sometimes even by a factor of 1.25 - 1.35. The reason for these high standard deviation values is that - depending on the randomly generated initial situation and goal state - a different number of actions must be executed to reach the goal state. In case of planning the number of actions necessary to reach the goal state makes the most difference since one more action to be executed means that BFS will find the goal one level deeper in the BFS tree. Since BFS traverses the tree level-wise a shift in one layer increases the run-time drastically. The high standard deviation values (especially in the YAGI planning programs) illustrate exactly this connection between number of actions to execute and run-time of the program.

8.5.2. Planning vs. No Planning

The run-time differences between *no planning* and *full planning* are quite severe. The reason for this is that the implemented search algorithm is an unoptimized textbook version of a BFS-like strategy. Breadth-first search strategies have the inherent disadvantage of being very memory consuming (Russell and Norvig, 2014), hence using an unoptimized textbook version in YAGI even amplifies this drawback. Further, BFS

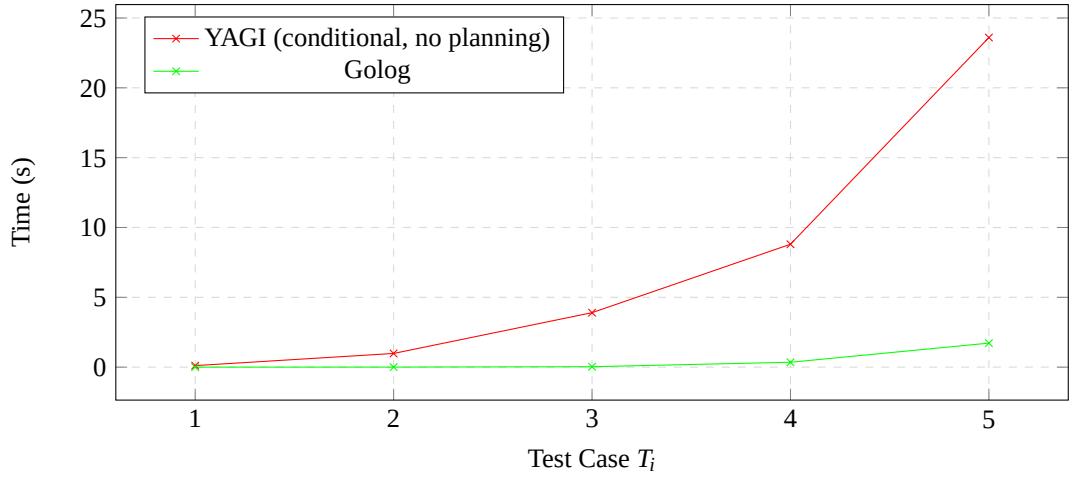

Figure 8.2.: Comparison of Test Case Run-Times of the Elevator Example

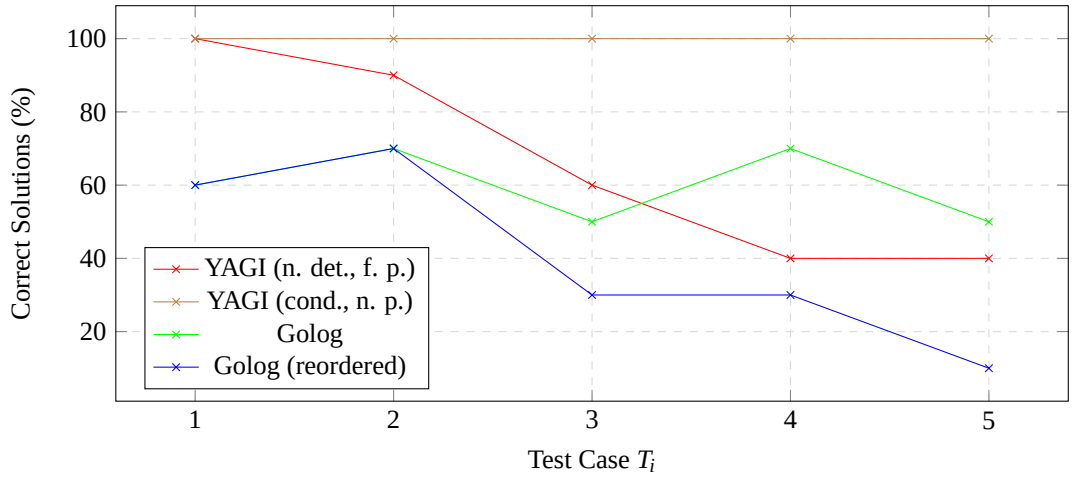

Figure 8.3.: Comparison of Valid Solution Percentages of the Blocks World Example

can be very time-consuming if the solution is far away from the initial state (Russell and Norvig, 2014). The elevator example exploits exactly this weakness of BFS since the solution is guaranteed to be always in the last level of the BFS tree because the goal of the elevator example holds iff all active floors have been served. This implies that any search attempts in a higher level (i.e. a level closer to the initial state) than the last level of the BFS tree are guaranteed to deliver no successful result. This also explains why Golog outperforms YAGI heavily in the elevator domain since Prolog's underlying DFS-like approach finds a solution for the elevator example immediately.

8.5.3. Conditionals and Non-Determinism

In both scenarios the program *YAGI (non-deterministic, no planning*) illustrates that non-determinism in case of online execution is highly unreliable and in most cases unable to find a correct solution. Due to the fact that it is purely random whether or not the YAGI program can finish successfully these results are not surprising. When guarded via conditionals (program *YAGI (conditional, no planning)*) *YagiTrans* or *YagiFinal* are forced to hold, meaning that the program is able to successfully terminate. The evaluation results show exactly this circumstance as 100% of the *YAGI (conditional, no planning)* programs delivered correct results.

8.5.4. Golog Order of Statements

Blocks world results clearly show that changing the order of statements in the Prolog-based implementation of Golog influences whether or not a result can be found. Such an order-sensitive behavior can be very counter-intuitive for people who are not aware of the precise semantics of Prolog. Further, our experiments show that the YAGI counterpart of the 'classic' Golog implementation of blocks world (i.e. *YAGI (non-deterministic, full planning)*) outperforms Golog in many cases in a sense that it is able to deliver correct results more often. Note that reordering statements in a YAGI program for the sake of comparison makes no sense since in the case of online execution the statements are picked pseudo-randomly (i.e. the order of statements doesn't matter) and in offline execution planning is executed in a BFS-like manner which is guaranteed to find a solution - if one exists. Note that even though BFS is guaranteed to find a solution (if one exists) the program *YAGI (non-deterministic, full planning)* didn't find a solution in 100% of the cases. This is due to the fact that YAGI ran out of resources during the search process, hence the reason that a solution could not be found lies in the implementation rather then the search strategy. We illustrate the percentages of valid solutions found for the blocks world example in Figure 8.3.

Chapter 9

Conclusion

9.1. Summary

The goal of this thesis was to define the syntax and semantics of an action-based programming language based on the theoretical foundations of situation calculus and IndiGolog, but especially designed for easy usability and strict separation of syntax and semantics to enable an implementation to be completely decoupled from any specific programming environment like Prolog.

Therefore, we defined a 3-tier system architecture for a YAGI-based software system that clearly separates syntax and semantics of the language with the primary goal to avoid immanent pitfalls that result from the tight coupling of the vast majority of Prolog-based Golog interpreter implementations.

Further, we presented a motivating YAGI example of an object delivery robot to illustrate the syntax of YAGI and to introduce a specific scenario we plan to use YAGI for.

Subsequently, we provided a formal specification of the syntax and semantics of YAGI. We specified the mapping of YAGI fluents and facts to situation calculus basic action theories and presented how assignments to fluents and facts can be mapped to situation calculus simple actions. Moreover, we discussed more sophisticated language features of YAGI like pattern matching and *setting actions* and illustrated their relation to situation calculus. Additionally, we outlined ideas of how to further extend YAGI with features like sensing and incomplete information.

Having defined the mapping of YAGI to situation calculus to represent the world of a specific problem domain we proceeded with the specification of the semantics of YAGI program execution. We used the execution semantics of IndiGolog as a theoretical foundation to build the YAGI execution semantics on and explained how YAGI program constructs relate to IndiGolog.

Having a formal specification of the syntax and semantics of YAGI we continued with the description of our proof-of-concept implementation of a YAGI software system. We explained our fundamental design decisions and discussed the software architecture of our implementation of each of the layers in the YAGI software systems. We explained that our back-end uses a relational database to represent the state of the world (i.e., fluents and facts), a decision that was motivated by the fact that the semantics of situation calculus relates closely to the semantics of relational databases. Subsequently, we discussed how the individual elements of the YAGI language had been implemented and illustrated how our BFS-like planning approach is integrated in the implementation.

With a language specification and a description of our proof-of-concept implementation we continued with the discussion how our implementation follows the specification. To be able to discuss the connection of specification and implementation we provided a formal description of our database semantics and how

this semantics relate to the specified situation calculus mappings of YAGI. We further discussed how logical sentences in YAGI are evaluated w.r.t the specified database semantics. To round out the discussion of specification conformance we argued inductively on the structure of YAGI programs to demonstrate how the execution semantics of YAGI relate to the execution semantics of IndiGolog. Specifically, we introduced a function that maps YAGI code to IndiGolog code and provided transition semantics predicates *YagiTrans* and *YagiFinal* as counterparts of IndiGolog's transition semantics predicates *Trans* and *Final* to discuss their relation.

Finally, we provided an evaluation of our implementation, comparing YAGI programs to Golog programs regarding their execution times and whether or not a solution could be found. The results showed that Golog outperforms our current YAGI implementation in many cases due to the fact that our implementation is an unoptimized proof-of-concept implementation and needs further performance tuning to be able to compete, especially when *search* is applied to a YAGI program. This rather huge performance hit is a result of our textbook implementation of a BFS-like search strategy. Still, the BFS-like approach showed its strength in a sense that it was able to find solutions for blocks world problem instances where the Prolog-based implementation of Golog failed to deliver a correct solution.

We conclude that we achieved our goal to design an easy to use action-based programming language to be used for education and research in the fields of artificial intelligence and robotics.

9.2. Future Work

The definition of syntax and semantics of YAGI in this thesis lays the groundwork for various possible future extensions of the language regarding its syntax and semantics as well as its formal background. Furthermore, our presented proof-of-concept implementation of YAGI can be improved and extended in various ways. We present a non-exhaustive list of reasonable extensions as follows, in no particular order:

- **Relating YAGI Action *Effect* Blocks to Situation Calculus Successor State Axioms**: As discussed in Section 5.4.8, we are positive that it is possible to prove that one can rewrite arbitrary YAGI *effect* blocks directly to situation calculus successor state axioms. Proving this claim will be necessary to be able to discuss theoretical properties of YAGI that are based on this observation.
- **Incomplete Information and Sensing Actions**: To this day, YAGI only has syntactical constructs for incomplete information and sensing actions, but lacks of a precise semantic definition of these features. We believe that incomplete information and sensing actions are of great importance when it comes to modeling real-world application domains, hence a precise specification of the intended semantics would be highly valuable.
- **Extension of *Search***: Setting actions, sensing and exogenous events have been deliberately excluded from occurring inside a *search* block. Besides the fact that there exists no semantics for sensing actions (as discussed above) the reason for these restrictions is that we're not able to bring up a viable way to model sensing actions, setting actions and exogenous events in an offline execution mode.
- ***Search* Strategy Optimization**: Our current implementation of *search* is a textbook BFS-like algorithm and the evaluation in Chapter 8 clearly shows the weaknesses of this approach. Different search strategies might make sense in different scenarios, so a possible optimization (among many) could be to implement different search strategies and let the developer of a YAGI program control which of the implemented strategies should be used for each *search*-block separately. We believe that being able to use multiple different search strategies in a single YAGI program would be highly valuable since having the flexibility of choosing an appropriate search strategy based on some a priori knowledge about the nature of the problem domain can be used to pick the best fitting strategy.
- **Implementations for Various Platforms**: Our proof-of-concept implementation has currently been tested only on Linux-based operating systems. We think that bringing YAGI to other operating systems like Microsoft Windows or Apple Mac OS would be a valuable goal. Also, we hope that

YAGI will be used for educational purposes. Hence, having YAGI implementations for platforms used in the *educational robotics* environment like the LEGO® MINDSTORM™ is an important aspect.

- **ROS Binding**: In our proof-of-concept implementation the system interface simply returns the data it gets passed by the back-end to its caller. In real-world robotics, one would need an implementation of the system interface that interacts with the software framework the concrete real-world robot uses. On popular example is the robot operating system ROS (Quigley et al., 2009). Therefore, it would be highly valuable to have an implementation that allows YAGI to communicate with ROS-based real-world robotics applications.
- ***Break* in Loops**: Until today there exists no concept of breaking a loop depending on a certain condition. The concept of breaking loops is a widely spread feature that can be found in many well-known general purpose programming languages, hence it might also be a valuable feature in YAGI. How such a concept would impact our program execution semantics is a question that requires deeper analysis.
- ***Return*-Statement in Actions and Procedures**: Actions and procedures are currently unable to return any information to the caller. Still it might make sense to provide the ability to return information from the caller to the callee via return statements.
- **Macros**: A macro-system like in the C programming language (i.e. textual substitution) could be useful in certain scenarios, e.g. defining reoccurring domains for fluents and facts as a macro to avoid code duplication.
- **Inclusion of External Resources**: It might be pleasant to load external libraries (possibly written in languages other than YAGI) into a YAGI environment to extend the set of features a YAGI program can use without modifying the core language.
- **Error Detection and Error Recovery**: In certain situations, the program may end up in a situation where execution may not be able to continue, e.g. the *pick*-operator is not able to find a variable binding that leads to a solution. Currently, the application displays an error message and terminates. This could be refined using some form of exception handling and/or recovery mechanisms. Possible approaches to actively diagnose and repair inconsistencies between the knowledge of an agent and the real world have already been discussed by (Mühlbacher and Steinbauer, 2014). We strongly believe that similar approaches could also be applied to YAGI and would be a highly valuable extension.
- **Action Inheritance**: Practical tasks showed that their exists actions that share some form of common behavior and/or state. To avoid code duplication some form of action inheritance could be defined.
- **Syntactic Sugar**: For the time being, YAGI lines of code require a vast amount of different types of braces to precisely specify sets, tuples and formulas. To simplify the syntax, braces should be omissible as often as possible without introducing any ambiguities.

Bibliography

Aho, A., Lam, M., Sethi, R., and Ullman, J. (2007). *Compilers: Principles, Techniques and Tools, 2nd Edition*. Pearson Education.

Alexandrescu, A. (2001). *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley.

Belle, V. and Levesque, H. (2014). Prego: An action language for belief-based cognitive robotics in continuous domains. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*.

Boutilier, C., Reiter, R., Soutchanski, M., Thrun, S., et al. (2000). Decision-theoretic, high-level agent programming in the situation calculus. In *AAAI/IAAI*, pages 355–362.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

De Giacomo, G., Lespérance, Y., and Levesque, H. J. (2000). Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1):109–169.

De Giacomo, G., Lespérance, Y., Levesque, H. J., and Sardina, S. (2009). Indigolog: A high-level programming language for embedded reasoning agents. In *Multi-Agent Programming:*, pages 31–72. Springer.

De Giacomo, G. and Levesque, H. J. (1999a). An incremental interpreter for high-level progra with sensing. In *Logical Foundations for Cognitive Agents*, pages 86–102. Springer.

De Giacomo, G. and Levesque, H. J. (1999b). Progression and regression using sensors. In *Proc. of IJCAI*, volume 99, pages 160–165. Citeseer.

De Giacomo, G. and Mancini, T. (2004). Scaling up reasoning about actions using relational database technology. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 245–250. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

De Giacomo, G. and Palatta, F. (2000). Exploiting a relational db for reasoning about actions. *Proc. of CogRob 2000*.

Enderton, H. B. (2001). *A mathematical introduction to logic*. Academic press.

Etzioni, O., Hanks, S., Weld, D. S., Draper, D., Lesh, N., and Williaon, M. (1992). An approach to planning with incomplete information. *KR*, 92:115–125.

Ferrein, A. (2010). Golog. lua: Towards a non-prolog implementation of golog for embedded systems. In *AAAI Spring Symposium: Embedded Reasoning*.

Ferrein, A. and Steinbauer, G. (2010). On the way to high-level programming for resource-limited embedded syste with golog. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 229–240. Springer.

Ferrein, A., Steinbauer, G., and Vassos, S. (2012). Action-based imperative programming with yagi. In *Proceedings of the 8th International Cognitive Robotics Workshop at AAAI-12*.

Ferrein, A. A. (2007). *Robot Controllers for Highly Dynamic Environments With Real-time Constraints.* PhD thesis, Rheinisch-Westfälischen Technischen Hochschule Aachen.

Fikes, R. E. and Nilsson, N. J. (1972). Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3):189–208.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080.

Gelfond, M. and Lifschitz, V. (1993). Representing action and change by logic progra. *The Journal of Logic Programming*, 17(2):301–321.

Gelfond, M. and Lifschitz, V. (1998). Action languages. *Electronic Transactions on AI*, 3(16).

Gossett, E. (2009). *Discrete mathematics with proof*. John Wiley & Sons.

Gregoire, M., Solter, N. A., and Kleper, S. J. (2011). *Professional C++*. John Wiley & Sons.

Grosskreutz, H. and Lakemeyer, G. (2003). cc-golog–an action language with continuous change. *Logic Journal of IGPL*, 11(2):179–221.

Hindriks, K. V., De Boer, F. S., Van der Hoek, W., and Meyer, J.-J. C. (1999). Agent programming in 3apl. *Autonomous Agents and Multi-Agent Syste*, 2(4):357–401.

Lakemeyer, G. (1999). On sensing and off-line interpreting in golog. In *Logical Foundations for Cognitive Agents*, pages 173–189. Springer.

Levesque, H. J. (1996). What is planning in the presence of sensing? In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*, AAAI'96, pages 1139–1146. AAAI Press.

Levesque, H. J. (2005). Planning with loops. In *IJCAI*, pages 509–515.

Levesque, H. J. and Pagnucco, M. (2000). Legolog: Inexpensive experiments in cognitive robotics. In *Cognitive Robotics Workshop at ECAI*, pages 104–109.

Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. B. (1994). Golog: A logic programming language for dynamic domains. *J. LOGIC PROGRAMMING*, 19(20):1–679.

Lifschitz, V. (2008). What is answer set programming?. In *AAAI*, volume 8, pages 1594–1597.

Lin, F. and Reiter, R. (1997). How to progress a database. *Artificial Intelligence*, 92(1):131–167.

McCarthy, J. (1963). Situations, actions, and causal laws. Technical report, DTIC Document.

Meyer, B. (1992). Applying'design by contract'. *Computer*, 25(10):40–51.

Mühlbacher, C. and Steinbauer, G. (2014). Knowledge-aware execution of programs in indigolog. *Proc. of CogRob 2014*.

Nilsson, N. J. (1982). *Principles of artificial intelligence*. Springer.

Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis*. Elsevier.

Nilsson, U. and Małuszyński, J. (1990). *Logic, programming and Prolog*. Wiley Chichester.

Overland, B. (2013). *C++ for the Impatient*. Addison-Wesley.

Parr, T. (2007). *The definitive ANTLR reference: building domain-specific languages*. Pragmatic Bookshelf.

Petrick, R. P. and Bacchus, F. (2002). A knowledge-based approach to planning with incomplete information and sensing. In *AIPS*, pages 212–222.

Petrick, R. P. and Bacchus, F. (2004). Extending the knowledge-based approach to planning with incomplete information and sensing. In *KR*, pages 613–622.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5.

Reiter, R. (2001). *Knowledge in action: logical foundations for specifying and implementing dynamical syste*. MIT press.

Russell, S. and Norvig, P. (2014). *Artificial Intelligence: A Modern Approach*. Always learning. Pearson.

Ryan, M. (2014). Efficiently implementing golog with answer set programming. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*.

Scherl, R. and Levesque, H. (1993). The frame problem and knowledge-producing actions. In *Proceedings of AAAI-93*, pages 689–695, Washington, DC. AAAI Press/The MIT Press.

Scherl, R. B. and Levesque, H. J. (2003). Knowledge, action, and the frame problem. *Artificial Intelligence*, 144(1):1–39.

Thielscher, M. (1998). Introduction to the fluent calculus. *Electronic Transactions on Artificial Intelligence (http://www. etaij. org)*, 3.

Thielscher, M. (2002). Programming of reasoning and planning agents with flux. In *PRINCIPLES OF KNOWLEDGE REPRESENTATION AND REASONING-INTERNATIONAL CONFERENCE-*, pages 435–448. Morgan Kaufmann Publishers; 1998.

Vassos, S. (2009). *A reasoning module for long-lived cognitive agents*. PhD thesis, University of Toronto.

Vassos, S., Lakemeyer, G., and Levesque, H. J. (2008). First-order strong progression for local-effect basic action theories. In *KR*, pages 662–672.

Vassos, S. and Levesque, H. J. (2007). Progression of situation calculus action theories with incomplete information. In *IJCAI*, volume 7, pages 2024–2029.

Vassos, S. and Sardina, S. (2011). A database-type approach for progressing action theories with bounded effects. *Knowing, Reasoning, and Acting: Essays in Honour of Hector J. Levesque. College Publications*.

Wielemaker, J., De Koninck, L., Fruehwirth, T., Triska, M., and Uneson, M. (2014). *SWI Prolog Reference Manual 7.1*. BoD–Books on Demand.

Appendices

Appendix A

Object Delivery Robot YAGI Source Code

```
//location of the robot (room1, ..., room3)
fluent at [{"r1","r2","r3"}];
at = {<"r1">};

//location of objects (object1 in room1 etc)
fluent is_at[{"o1","o2","o3"}][{"r1","r2","r3"}];
is_at = {<"o1","r1">, <"o2","r2">, <"o3","r3">};

//object carried by robot
fluent carry[{"o1","o2","o3"}];

//requests moving an object (param 1) from a sender (param 2)
//to a receiver (param 3)
fluent request[{"o1","o2","o3"}][{"p1","p2","p3"}][{"p1","p2","p3"}];

//states what person has been detected in what room
fluent detectedPerson[{"p1","p2","p3"}][{"r1","r2","r3"}];
```

```
//one or more rooms are assigned to one person,
//i.e. the person's offices
fact office [{"p1","p2","p3"}][{"r1","r2","r3"}];
office = {<"p1","r1">, <"p1","r2">, <"p2","r2">, <"p3","r3">};
```

```
//move robot to room $r
action move($r)
precondition:
  //robot is not in room $r
  not (<$r> in at);
effect:
  //now he is in room $r
  at = {<$r>};
signal:
  "Move to room " + $r;
end action
```

```
//pickup object $o
action pickup($o)
precondition:
  //robot doesn't carry anything and is in the room where the object is
  (not(exists <$x> in carry) and exists <$y> in at such <$o,$y> in is_at);
effect:
  //now he carries $o
  carry += {<$o>};
signal:
  "Pickup object " + $o;
end action
```

```
//putdown object
action putdown($o)
precondition:
  //he carries the object stored in $o
  <$o> in carry;
effect:
  //now he's not
  carry -= {<$o>};

  //where ever it was, its now somewhere else...
  is_at -= {<$o,_>};

  //...namely: here!
  foreach <$r> in at do
    is_at += {<$o,$r>};
  end for

signal:
  "Put down object " + $o;
end action
```

```
//"setting" action to detect a person, i.e.
//$p gets its value from an external src
action detectPerson( ) external ($p)
effect:

  //remove person
  detectedPerson -= {<$p, _>};

  //add the detected person + room tuple to the fluent
  foreach <$r> in at do
    detectedPerson += {<$p, $r>};
  end for

signal:
  "detect person";
end action
```

```
//exogenous event to initiate transportation
//of object $o from $sender to $receiver
exogenous-event receiveRequest($o, $sender, $receiver)
  //add request
  request += {<$o,$sender,$receiver>};
end exogenous-event
```

```
//serves a request
proc serve($object, $sender, $receiver)

  pick <$sender,$roomSender> from office such
    move($roomSender);

    //search for person in the room
    detectPerson();

    //sender is actually in the room
    if (<$sender, $roomSender> in detectedPerson) then
      pickup($object);

      //deliver object to receiver
      pick <$receiver, $roomReceiver> from office such
        move($roomReceiver);

        //search for person in the room
        detectPerson();

        //receiver is actually in the room
        if (<$receiver, $roomReceiver> in detectedPerson) then
```

```
          putdown($object);
        end if
      end pick
    end if
  end pick
end proc
```

```
proc main()
  //serve a random request
  pick <$object, $sender, $receiver> from request such
    serve($object, $sender, $receiver);
  end pick
end proc
```

Appendix B

YAGI Grammar

```
//************************************************************************
//Grammar for the YAGI programming language
//Author: Christopher Maier
//Date: 2014-07-01
//************************************************************************
grammar YAGI;

options
{
  language = C;
  output = AST;
  ASTLabelType=pANTLR3_BASE_TREE;
  k = 1;
}

//************************************************************************
//Imaginary tokens
//************************************************************************
tokens
{
TOKEN_EOL = ';';

TOKEN_COLON = ':';

TOKEN_PICK = 'pick';
TOKEN_END_PICK
        =       'end pick'
        ;

TOKEN_FROM = 'from';
TOKEN_SUCH = 'such';

TOKEN_TEST = 'test';
TOKEN_IN = 'in';
TOKEN_DO = 'do';

TOKEN_IF = 'if';
TOKEN_END_IF
        =       'end if'
        ;

TOKEN_THEN = 'then';
TOKEN_ELSE = 'else';

TOKEN_CHOOSE = 'choose';
TOKEN_END_CHOOSE
```

```
        =       'end choose'
        ;

TOKEN_WHILE
        =       'while'
        ;

TOKEN_END_WHILE
        =       'end while'
        ;

TOKEN_END_FOR
        =       'end for'
        ;

TOKEN_DOMAIN_STR
        =       'String'
        ;

TOKEN_ACTION = 'action';
TOKEN_END_ACTION
        =       'end action'
        ;

TOKEN_PRECOND = 'precondition:';
TOKEN_EFFECT = 'effect:';
TOKEN_SIGNAL = 'signal:';
TOKEN_SENSING = 'sense';
TOKEN_END_SENSING = 'end sense';
TOKEN_EXTERNAL = 'external';

TOKEN_EXO_EVENT = 'exogenous-event';
TOKEN_END_EXO_EVENT
        =       'end exogenous-event'
        ;

TOKEN_NOT= 'not';
TOKEN_EXISTS= 'exists';
TOKEN_ALL= 'all';
TOKEN_IMPLIES= 'implies';

TOKEN_ASSIGN = '=';
TOKEN_ADD_ASSIGN= '+=';
TOKEN_REMOVE_ASSIGN= '-=';

TOKEN_EQUALS= '==';
TOKEN_NEQUALS= '!=';
TOKEN_LE= '<=';
TOKEN_GE= '>=';
TOKEN_LT= '<';
TOKEN_GT= '>';
TOKEN_PLUS = '+';
TOKEN_MINUS = '-';

TOKEN_AND= 'and';
TOKEN_OR= 'or';
TOKEN_TRUE= 'true';
TOKEN_FALSE= 'false';

TOKEN_PATTERN_MATCHING
        =       '_';

TOKEN_INCOMPLETE_KNOWLEDGE
        =       '*';

TOKEN_SET_START= '{';
TOKEN_SET_END= '}';
```

```
TOKEN_DOMAIN_START= '[';
TOKEN_DOMAIN_END= ']';

TOKEN_VAR_DECL_START= '$';
TOKEN_FLUENT = 'fluent';


TOKEN_END_SEARCH
        =       'end search';

TOKEN_SEARCH
        =       'search'
        ;

TOKEN_OPEN_PAREN
        =       '('
        ;

TOKEN_CLOSE_PAREN
        =       ')';

TOKEN_COMMA
        =       ','
        ;

TOKEN_END_PROC
        =       'end proc';

TOKEN_PROC =    'proc'
          ;

TOKEN_FACT
        =       'fact'
        ;

TOKEN_FOR_EACH
        =       'foreach'
        ;

TOKEN_INCLUDE = '@include';

  IT_FLUENT_DECL;
  IT_STRING_SET;
  IT_TUPLE_SET;
  IT_FACT_DECL;
  IT_PROGRAM;
  IT_ASSIGN;
  IT_ADD_ASSIGN;
  IT_REMOVE_ASSIGN;
  IT_PLUS;
  IT_MINUS;
  IT_TUPLE;
  IT_VAR;
  IT_EXO_EVENT;
  IT_VAR_LIST;
  IT_BLOCK;
  IT_ACTION_DECL;
  IT_SIGNAL;
  IT_SENSING;
  IT_EXTERNAL_VARS;
  IT_NOT;
  IT_AND;
  IT_OR;
  IT_IMPLIES;
  IT_ALL;
  IT_EXISTS;
```

```
  IT_IN;
  IT_EFFECT;
  IT_FORMULA;
  IT_PROC_DECL;
  IT_SEARCH;
  IT_PICK;
  IT_PROC_EXEC;
  IT_FLUENT_QUERY;
  IT_VALUE_LIST;
  IT_CONDITIONAL;
  IT_FORALL;
  IT_WHILE;
  IT_CHOOSE;
  IT_TEST;
  IT_EQ;
  IT_NEQ;
  IT_GT;
  IT_LT;
  IT_GE;
  IT_LE;
  IT_ATOM_SETEXPR;
  IT_ATOM_VALEXPR;
  IT_INCLUDE;
}

//*************************************************************************
//Basic program structure
//*************************************************************************
program
        :       (declaration | statement | include)+
        ;

block
        :       statement+

                -> ^(IT_BLOCK statement+)
        ;

include :       TOKEN_INCLUDE STRING TOKEN_EOL

                -> ^(IT_INCLUDE STRING)
        ;

//*************************************************************************
//Declarations
//*************************************************************************
declaration
        :       fluent_decl
        |       fact_decl
        |       action_decl
        |       proc_decl
        |       exo_event_decl
        |       sensing_decl
        ;

fluent_decl
        :       TOKEN_FLUENT ID (TOKEN_DOMAIN_START domain TOKEN_DOMAIN_END)* TOKEN_EOL

                -> ^(IT_FLUENT_DECL ID domain*)
        ;

fact_decl
        :       TOKEN_FACT ID (TOKEN_DOMAIN_START domain TOKEN_DOMAIN_END)* TOKEN_EOL

                -> ^(IT_FACT_DECL ID domain*)
        ;
```

```
domain
        :       TOKEN_DOMAIN_STR -> TOKEN_DOMAIN_STR
        |       TOKEN_SET_START STRING (TOKEN_COMMA STRING)* TOKEN_SET_END -> ^(
            IT_STRING_SET STRING+)
        ;

action_decl
        :       TOKEN_ACTION ID TOKEN_OPEN_PAREN v1=var_list? TOKEN_CLOSE_PAREN (
            TOKEN_EXTERNAL TOKEN_OPEN_PAREN v2=var_list TOKEN_CLOSE_PAREN)?
                (TOKEN_PRECOND formula_outerMost)?
                effect?
                (TOKEN_SIGNAL valexpr TOKEN_EOL)?
                TOKEN_END_ACTION

                -> ^(IT_ACTION_DECL ID ^(IT_VAR_LIST $v1?) (^(IT_EXTERNAL_VARS $v2))?
                    formula_outerMost? effect? (^(IT_SIGNAL valexpr))? )
        ;

effect
        :       TOKEN_EFFECT block
                -> ^(IT_EFFECT block)
        ;

var_list
        :       var (TOKEN_COMMA var)*

                -> var+
        ;


proc_decl
        :       TOKEN_PROC ID TOKEN_OPEN_PAREN var_list? TOKEN_CLOSE_PAREN block
            TOKEN_END_PROC

                -> ^(IT_PROC_DECL ID (^(IT_VAR_LIST var_list))? block)
        ;

exo_event_decl
        :       TOKEN_EXO_EVENT ID TOKEN_OPEN_PAREN var_list TOKEN_CLOSE_PAREN block
            TOKEN_END_EXO_EVENT

                -> ^(IT_EXO_EVENT ID ^(IT_VAR_LIST var_list) block)

        ;

sensing_decl
        :       TOKEN_SENSING ID TOKEN_OPEN_PAREN v1=var_list? TOKEN_CLOSE_PAREN (
            TOKEN_EXTERNAL TOKEN_OPEN_PAREN v2=var_list TOKEN_CLOSE_PAREN)? formula
            TOKEN_END_SENSING

                -> ^(IT_SENSING ID ^(IT_VAR_LIST $v1?) (^(IT_EXTERNAL_VARS $v2))?
                    formula)
        ;

//*****************************************************************************
//Statements
//*****************************************************************************
statement
        :       id_term
        |       var_assign
        |       test
        |       choose
        |       pick
        |       for_loop
        |       conditional
        |       while_loop
        |       search
```

```
        ;

id_term
        :       ID (
                     TOKEN_OPEN_PAREN value_list? TOKEN_CLOSE_PAREN TOKEN_EOL -> ^(
                         IT_PROC_EXEC ID (^(IT_VALUE_LIST value_list))?)
                   | TOKEN_EOL                                                 -> ^(
                       IT_FLUENT_QUERY ID)
                   | ass_op setexpr TOKEN_EOL                                  -> ^(
                       ass_op ID setexpr)

                   )
        ;

value_list
        :       value (TOKEN_COMMA value)*

                -> value+
        ;

test
        :       TOKEN_TEST formula TOKEN_EOL

                -> ^(IT_TEST formula)
        ;

choose
        :       TOKEN_CHOOSE block ( TOKEN_OR block )+ TOKEN_END_CHOOSE

                -> ^(IT_CHOOSE block+)
        ;

pick
        :       TOKEN_PICK tuple TOKEN_FROM setexpr TOKEN_SUCH block TOKEN_END_PICK

                -> ^(IT_PICK tuple setexpr block)
        ;

for_loop
        :       TOKEN_FOR_EACH tuple TOKEN_IN setexpr TOKEN_DO block TOKEN_END_FOR

                -> ^(IT_FORALL tuple setexpr block)
        ;

conditional
        :       TOKEN_IF TOKEN_OPEN_PAREN formula TOKEN_CLOSE_PAREN TOKEN_THEN b1=block
            (TOKEN_ELSE b2=block)? TOKEN_END_IF

                -> ^(IT_CONDITIONAL formula $b1 $b2?)
        ;

while_loop
        :       TOKEN_WHILE formula TOKEN_DO block TOKEN_END_WHILE

                -> ^(IT_WHILE formula block)
        ;

search
        :       TOKEN_SEARCH block TOKEN_END_SEARCH

                -> ^(IT_SEARCH block)
        ;


//*************************************************************************
//Assignments
//*************************************************************************
```

```
var_assign
        :       var TOKEN_ASSIGN value TOKEN_EOL  -> ^(IT_ASSIGN var value)
        ;

ass_op
        :       (TOKEN_ASSIGN -> IT_ASSIGN
                  | TOKEN_ADD_ASSIGN -> IT_ADD_ASSIGN
                  | TOKEN_REMOVE_ASSIGN -> IT_REMOVE_ASSIGN
                )
        ;

//****************************************************************************
//Formulas
//****************************************************************************
formula_outerMost
        :       formula TOKEN_EOL

                -> ^(IT_FORMULA formula)
        ;

formula
        :       atom
        |       TOKEN_NOT TOKEN_OPEN_PAREN formula TOKEN_CLOSE_PAREN -> ^(IT_NOT formula
            )
        |       TOKEN_OPEN_PAREN f1=formula formula_connective f2=formula
            TOKEN_CLOSE_PAREN -> ^(formula_connective $f1 $f2)
        |       TOKEN_EXISTS tuple TOKEN_IN setexpr (TOKEN_SUCH formula)? -> ^(IT_EXISTS
             tuple setexpr formula?)
        |       TOKEN_ALL tuple TOKEN_IN setexpr (TOKEN_SUCH formula)? -> ^(IT_ALL tuple
             setexpr formula?)
        |       tuple TOKEN_IN setexpr -> ^(IT_IN tuple setexpr)
        ;

formula_connective
        :       TOKEN_AND -> IT_AND
        |       TOKEN_OR  -> IT_OR
        |       TOKEN_IMPLIES -> IT_IMPLIES
        ;

atom
        :       v1=value atom_connector v2=value -> ^(IT_ATOM_VALEXPR ^(atom_connector
            $v1 $v2))
        |       s1=setexpr atom_connector s2=setexpr -> ^(IT_ATOM_SETEXPR ^(
            atom_connector $s1 $s2))
        |       (TOKEN_TRUE | TOKEN_FALSE)
        ;

atom_connector
        :       TOKEN_EQUALS -> IT_EQ
        |       TOKEN_NEQUALS -> IT_NEQ
        |       TOKEN_LE -> IT_LE
        |       TOKEN_GE -> IT_GE
        |       TOKEN_LT -> IT_LT
        |       TOKEN_GT -> IT_GT
        ;

//****************************************************************************
//Sets
//****************************************************************************
set     :       TOKEN_SET_START
                (
                  tuple (TOKEN_COMMA tuple)* -> ^(IT_TUPLE_SET tuple+)
                | /*eps*/                    -> ^(IT_TUPLE_SET)
                ) TOKEN_SET_END
        |       ID
        ;
```

```
setexpr :       set (expr_op^ set)*
        ;


//*****************************************************************************
//Tuples
//*****************************************************************************
tuple
        :       TOKEN_LT (
                  tuple_val (TOKEN_COMMA tuple_val)* -> ^(IT_TUPLE tuple_val+)
        |       /*eps*/                              -> ^(IT_TUPLE)
                ) TOKEN_GT
        ;

tuple_val
        :       STRING
        |       TOKEN_PATTERN_MATCHING
        |       TOKEN_INCOMPLETE_KNOWLEDGE
        |       var
        ;

//*****************************************************************************
//Variables
//*****************************************************************************
var
        :       TOKEN_VAR_DECL_START ID -> ^(IT_VAR ID)
        ;

value
        :       STRING
        |       var
        ;

valexpr
        :       value (expr_op^ value)*
        ;


expr_op
        :       TOKEN_PLUS -> IT_PLUS
              | TOKEN_MINUS -> IT_MINUS
        ;

//*****************************************************************************
//Lexer Rules
//*****************************************************************************
WS  :   (' '|'\t'|'\f'|'\n'|'\r')+ { $channel=HIDDEN; };

ID  :   ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
    ;

STRING
    :  '"' ( ~('//'|'"') )* '"'
    ;

COMMENT
    :   '//' ~('\n'|'\r')* (EOF|'\r'? '\n') {$channel=HIDDEN;}
    ;

ML_COMMENT
    :   '/*' (options {greedy=false;} : .)* '*/' {$channel=HIDDEN;}
    ;
```

Appendix C

Evaluation Programs

C.1. Elevator

C.1.1. YAGI (non-deterministic, no planning)

```
@include "fluents.y";

action turnoff($n)
precondition:
  <$n> in fon;
effect:
  fon -= {<$n>};
signal:
  "Turn-off button at floor " + $n;
end action

action open()
signal:
  "Open door";
end action

action close()
signal:
  "Close door";
end action

action up($n)
precondition:
  exists <$x> in floors
  such (currFloor == {<$x>} and $x < $n);
effect:
  currFloor = {<$n>};
signal:
  "Move up to floor " + $n;
end action

action down($n)
precondition:
  exists <$x> in floors
  such (currFloor == {<$x>} and $x > $n);
effect:
  currFloor = {<$n>};
signal:
  "Move down to floor " + $n;
```

```
end action

proc park()
  if (currFloor == {<"0">}) then
    open();
  else
    down("0");
    open();
  end if
end proc

proc goto($n)
    choose
      test currFloor == {<$n>};
    or
      up($n);
    or
     down($n);
    end choose
end proc


proc serve($n)
  goto($n);
  turnoff($n);
  open();
  close();
end proc

proc serveafloor()
    pick <$n> from fon such
      serve($n);
    end pick
end proc

proc control()
  while exists <$n> in fon do
    serveafloor();
  end while

  park();
end proc

control();
```

C.1.2. YAGI (conditional, no planning)

```
@include "fluents.y";

action turnoff($n)
precondition:
  <$n> in fon;
effect:
  fon -= {<$n>};
signal:
  "Turn-off button at floor " + $n;
end action

action open()
signal:
  "Open door";
end action

action close()
signal:
  "Close door";
end action
```

```
action up($n)
precondition:
  exists <$x> in floors
  such (currFloor == {<$x>} and $x < $n);
effect:
  currFloor = {<$n>};
signal:
  "Move up to floor " + $n;
end action

action down($n)
precondition:
  exists <$x> in floors
  such (currFloor == {<$x>} and $x > $n);
effect:
  currFloor = {<$n>};
signal:
  "Move down to floor " + $n;
end action

proc park()
  if (currFloor == {<"0">}) then
    open();
  else
    down("0");
    open();
  end if
end proc

proc goto($n)
  if (exists <$x> in currFloor such $x < $n) then
    up($n);
  else
    down($n);
  end if
end proc

proc serve($n)
  if (exists <$x> in currFloor such $x != $n) then
    goto($n);
  end if

  turnoff($n);
  open();
  close();
end proc

proc serveafloor()
    pick <$n> from fon such
      serve($n);
    end pick
end proc

proc control()
    while exists <$n> in fon do
      serveafloor();
    end while

    park();
end proc

control();
```

C.1.3. YAGI (non-deterministic, full planning)

```
@include "fluents.y";
```

```
action turnoff($n)
precondition:
  <$n> in fon;
effect:
  fon -= {<$n>};
signal:
  "Turn-off button at floor " + $n;
end action

action open()
signal:
  "Open door";
end action

action close()
signal:
  "Close door";
end action

action up($n)
precondition:
  exists <$x> in floors
  such (currFloor == {<$x>} and $x < $n);
effect:
  currFloor = {<$n>};
signal:
  "Move up to floor " + $n;
end action

action down($n)
precondition:
  exists <$x> in floors
  such (currFloor == {<$x>} and $x > $n);
effect:
  currFloor = {<$n>};
signal:
  "Move down to floor " + $n;
end action

proc park()
  if (currFloor == {<"0">}) then
    open();
  else
    down("0");
    open();
  end if
end proc

proc goto($n)
    choose
      test currFloor == {<$n>};
    or
      up($n);
    or
     down($n);
    end choose
end proc


proc serve($n)
  goto($n);
  turnoff($n);
  open();
  close();
end proc
```

```
proc serveafloor()
    pick <$n> from fon such
      serve($n);
    end pick
end proc

proc control()
  search
    while exists <$n> in fon do
      serveafloor();
    end while

    park();
  end search
end proc

control();
```

C.1.4. YAGI (conditional, full planning)

```
@include "fluents.y";

action turnoff($n)
precondition:
  <$n> in fon;
effect:
  fon -= {<$n>};
signal:
  "Turn-off button at floor " + $n;
end action

action open()
signal:
  "Open door";
end action

action close()
signal:
  "Close door";
end action

action up($n)
precondition:
  exists <$x> in floors
  such (currFloor == {<$x>} and $x < $n);
effect:
  currFloor = {<$n>};
signal:
  "Move up to floor " + $n;
end action

action down($n)
precondition:
  exists <$x> in floors
  such (currFloor == {<$x>} and $x > $n);
effect:
  currFloor = {<$n>};
signal:
  "Move down to floor " + $n;
end action

proc park()
  if (currFloor == {<"0">}) then
    open();
  else
    down("0");
    open();
```

```
  end if
end proc

proc goto($n)
  if (exists <$x> in currFloor such $x < $n) then
    up($n);
  else
    down($n);
  end if
end proc

proc serve($n)
  goto($n);
  turnoff($n);
  open();
  close();
end proc

proc serveafloor()
    pick <$n> from fon such
      serve($n);
    end pick
end proc

proc control()
  search
    while exists <$n> in fon do
      serveafloor();
    end while

    park();
  end search
end proc

control();
```

C.1.5. Golog

```
            %  A SIMPLE ELEVATOR CONTROLLER IN GOLOG (for SWI Prolog)

                        % written by Ray Reiter

% To run:
%        1) start SWI prolog, i.e. run "pl"
%        2) load the Golog interpreter, i.e. ?- [golog_swi].
%        3) load this file, i.e. ?- [simple_elevator].
%        4) run the main procedure, i.e. ?- do(control,s0,S).
%
% To see the whole sequence of actions in the situation S, use
%   ?- do(control,s0,S), show_act_seq(S).

% Primitive control actions


:- [golog_swi].

primitive_action(turnoff(N)).  % Turn off call button N.
primitive_action(open).        % Open elevator door.
primitive_action(close).       % Close elevator door.
primitive_action(up(N)).       % Move elevator up to floor N.
primitive_action(down(N)).     % Move elevator down to floor N.

% Definitions of Complex Control Actions

proc(goFloor(N), ?(currentFloor(N)) # up(N) # down(N)).
proc(serve(N), goFloor(N) : turnoff(N) : open : close).
```

```
proc(serveAfloor, pi(n, ?(nextFloor(n)) : serve(n))).
proc(park, if(currentFloor(0), open, down(0) : open)).

/* control is the main loop. So long as there is an active call
   button, it serves one floor. When all buttons are off, it
   parks the elevator.   */

proc(control, while(some(n, on(n)), serveAfloor) : park).

% Preconditions for Primitive Actions.

poss(up(N),S) :- currentFloor(M,S), M < N.
poss(down(N),S) :- currentFloor(M,S), M > N.
poss(open,S).
poss(close,S).
poss(turnoff(N),S) :- on(N,S).

% Successor State Axioms for Primitive Fluents.

currentFloor(M,do(A,S)) :- A = up(M) ; A = down(M) ;
              not(A = up(N)), not(A = down(N)), currentFloor(M,S).

on(M,do(A,S)) :- on(M,S), not(A = turnoff(M)).

% Initial Situation. Call buttons: 3 and 5. The elevator is at floor 4.

/* nextFloor(N,S) is an abbreviation that determines which of the
   active call buttons should be served next. Here, we simply
   choose an arbitrary active call button.   */

nextFloor(N,S) :- on(N,S).

% Restore suppressed situation arguments.

restoreSitArg(on(N),S,on(N,S)).
restoreSitArg(nextFloor(N),S,nextFloor(N,S)).
restoreSitArg(currentFloor(M),S,currentFloor(M,S)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% added by Yves Lesperance

show_act_seq(s0).
show_act_seq(do(A,S)):- show_act_seq(S), write(A), nl.

run:- do(control,s0,S), show_act_seq(S).

% definition of executable (legal) situation

executable(s0).
executable(do(A,S)) :- poss(A,S), executable(S).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- use_module(library(statistics)).
:-include('fluents.pl').
:- current_prolog_flag(argv, Argv), concat_atom(Argv,' ', SingleArg),
open(SingleArg,append,OS),
write(OS,'Running testcase...'),
nl(OS),
statistics(cputime, T0),
run,
statistics(cputime, T1),
T is T1-T0,
write(OS,T),nl(OS),close(OS).
:- halt.
```

C.1.6. Golog (conditional)

```
            %  A SIMPLE ELEVATOR CONTROLLER IN GOLOG (for SWI Prolog)

                        % written by Ray Reiter

% To run:
%        1) start SWI prolog, i.e. run "pl"
%        2) load the Golog interpreter, i.e. ?- [golog_swi].
%        3) load this file, i.e. ?- [simple_elevator].
%        4) run the main procedure, i.e. ?- do(control,s0,S).
%
% To see the whole sequence of actions in the situation S, use
%   ?- do(control,s0,S), show_act_seq(S).

% Primitive control actions

:- [golog_swi].

primitive_action(turnoff(N)).  % Turn off call button N.
primitive_action(open).        % Open elevator door.
primitive_action(close).       % Close elevator door.
primitive_action(up(N)).       % Move elevator up to floor N.
primitive_action(down(N)).     % Move elevator down to floor N.

% Definitions of Complex Control Actions

proc(goFloor(N), if(some(x,currentFloor(x) & x<N), up(N), down(N))).
proc(serve(N), goFloor(N) : turnoff(N) : open : close).
proc(serveAfloor, pi(n, ?(nextFloor(n)) : serve(n))).
proc(park, if(currentFloor(0), open, down(0) : open)).

/* control is the main loop. So long as there is an active call
   button, it serves one floor. When all buttons are off, it
   parks the elevator.   */

proc(control, while(some(n, on(n)), serveAfloor) : park).

% Preconditions for Primitive Actions.

poss(up(N),S) :- currentFloor(M,S), M < N.
poss(down(N),S) :- currentFloor(M,S), M > N.
poss(open,S).
poss(close,S).
poss(turnoff(N),S) :- on(N,S).

% Successor State Axioms for Primitive Fluents.

currentFloor(M,do(A,S)) :- A = up(M) ; A = down(M) ;
              not(A = up(N)), not(A = down(N)), currentFloor(M,S).

on(M,do(A,S)) :- on(M,S), not(A = turnoff(M)).

% Initial Situation. Call buttons: 3 and 5. The elevator is at floor 4.

/* nextFloor(N,S) is an abbreviation that determines which of the
   active call buttons should be served next. Here, we simply
   choose an arbitrary active call button.   */

nextFloor(N,S) :- on(N,S).

% Restore suppressed situation arguments.
restoreSitArg(on(N),S,on(N,S)).
restoreSitArg(nextFloor(N),S,nextFloor(N,S)).
restoreSitArg(currentFloor(M),S,currentFloor(M,S)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% added by Yves Lesperance
```

```
show_act_seq(s0).
show_act_seq(do(A,S)):- show_act_seq(S), write(A), nl.

run:- do(control,s0,S), show_act_seq(S).

% definition of executable (legal) situation

executable(s0).
executable(do(A,S)) :- poss(A,S), executable(S).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- use_module(library(statistics)).
:-include('fluents.pl').
:- current_prolog_flag(argv, Argv), concat_atom(Argv,' ', SingleArg),
open(SingleArg,append,OS),
write(OS,'Running testcase...'),
nl(OS),
statistics(cputime, T0),
run,
statistics(cputime, T1),
T is T1-T0,
write(OS,T),nl(OS),close(OS).
:- halt.
```

C.2. Blocks World

C.2.1. YAGI (non-deterministic, no planning)

```
@include "fluents.y";
@include "fluentsNoSearch.y";

action move($b1, $b2)
precondition:
  (<$b1> in fclear and
  (<$b2> in fclear and
  $b1 != $b2));
effect:
  onTable -= {<$b1>};
  fclear -= {<$b2>};

  foreach <$x1, $y1> in bon do
    if ($x1 == $b1) then
      fclear += {<$y1>};
    end if
  end for

  bon -= {<$b1, _>};
  bon += {<$b1, $b2>};

signal:
  "Move " + $b1 + " on top of " + $b2;
end action

action moveToTable($b)
precondition:
  (<$b> in fclear and not(<$b> in onTable));
effect:
  onTable += {<$b>};

  foreach <$xx, $yy> in bon do
    if ($xx == $b) then
      fclear += {<$yy>};
    end if
  end for
```

```
  bon -= {<$b, _>};

signal:
  "Move " + $b + " to table";
end action

proc doMove()
  pick <$x> from fclear such
    pick <$y> from fclear such
      doExec($x,$y);
    end pick
  end pick
end proc

proc doExec($first, $second)
  choose
    moveToTable($first);
  or
    move($first,$second);
  end choose
end proc

control();
```

C.2.2. YAGI (conditional, no planning)

```
@include "fluents.y";
@include "fluentsNoSearch.y";

action move($b1, $b2)
precondition:
  (<$b1> in fclear and
  (<$b2> in fclear and
  $b1 != $b2));
effect:
  onTable -= {<$b1>};
  fclear -= {<$b2>};

  foreach <$x1, $y1> in bon do
    if ($x1 == $b1) then
      fclear += {<$y1>};
    end if
  end for

  bon -= {<$b1, _>};
  bon += {<$b1, $b2>};

signal:
  "Move " + $b1 + " on top of " + $b2;
end action

action moveToTable($b)
precondition:
  (<$b> in fclear and not(<$b> in onTable));
effect:
  onTable += {<$b>};

  foreach <$xx, $yy> in bon do
    if ($xx == $b) then
      fclear += {<$yy>};
    end if
  end for

  bon -= {<$b, _>};

signal:
```

```
  "Move " + $b + " to table";
end action

proc doMove()
  pick <$x> from fclear such
    choose
      if (not(<$x> in onTable)) then
        moveToTable($x);
     end if
    or
      pick <$y> from fclear such
        if ($x != $y) then
          move($x, $y);
        end if
      end pick
    end choose
  end pick
end proc

control();
```

C.2.3. YAGI (non-deterministic, full planning)

```
@include "fluents.y";
@include "fluentsSearch.y";

action move($b1, $b2)
precondition:
  (<$b1> in fclear and
  (<$b2> in fclear and
  $b1 != $b2));
effect:
  onTable -= {<$b1>};
  fclear -= {<$b2>};

  foreach <$x1, $y1> in bon do
    if ($x1 == $b1) then
      fclear += {<$y1>};
    end if
  end for

  bon -= {<$b1, _>};
  bon += {<$b1, $b2>};

signal:
  "Move " + $b1 + " on top of " + $b2;
end action

action moveToTable($b)
precondition:
  (<$b> in fclear and not(<$b> in onTable));
effect:
  onTable += {<$b>};

  foreach <$xx, $yy> in bon do
    if ($xx == $b) then
      fclear += {<$yy>};
    end if
  end for

  bon -= {<$b, _>};

signal:
  "Move " + $b + " to table";
end action

proc doMove()
```

```
  pick <$x> from fclear such
    pick <$y> from fclear such
      doExec($x,$y);
    end pick
  end pick
end proc

proc doExec($first, $second)
  choose
    moveToTable($first);
  or
    move($first,$second);
  end choose
end proc

control();
```

C.2.4. YAGI (conditional, full planning)

```
@include "fluents.y";
@include "fluentsSearch.y";

action move($b1, $b2)
precondition:
  (<$b1> in fclear and
  (<$b2> in fclear and
  $b1 != $b2));
effect:
  onTable -= {<$b1>};
  fclear -= {<$b2>};

  foreach <$x1, $y1> in bon do
    if ($x1 == $b1) then
      fclear += {<$y1>};
    end if
  end for

  bon -= {<$b1, _>};
  bon += {<$b1, $b2>};

signal:
  "Move " + $b1 + " on top of " + $b2;
end action

action moveToTable($b)
precondition:
  (<$b> in fclear and not(<$b> in onTable));
effect:
  onTable += {<$b>};

  foreach <$xx, $yy> in bon do
    if ($xx == $b) then
      fclear += {<$yy>};
    end if
  end for

  bon -= {<$b, _>};

signal:
  "Move " + $b + " to table";
end action

proc doMove()
  pick <$x> from fclear such
    choose
      if (not(<$x> in onTable)) then
        moveToTable($x);
```

```
      end if
    or
      pick <$y> from fclear such
        if ($x != $y) then
          move($x, $y);
        end if
      end pick
    end choose
  end pick
end proc

control();
```

C.2.5. Golog

```
:- [golog_swi].

/*  Action Precondition Axioms  */
poss(move(X,Y),S) :- clear(X,S), clear(Y,S), not(X = Y).
poss(moveToTable(X),S) :- clear(X,S), not(ontable(X,S)).

/*  Successor State Axioms  */
clear(X,do(A,S)) :-( A = move(Y,Z) ; A = moveToTable(Y)),on(Y,X,S)
; clear(X,S), not(A = move(Y,X)).

on(X,Y,do(A,S)) :- A = move(X,Y) ;
                   on(X,Y,S), not(A = moveToTable(X)), not(A = move(X,Z)).
ontable(X,do(A,S)) :- A = moveToTable(X) ;
                      ontable(X,S), not(A = move(X,Y))        .

% Primitive control actions
primitive_action(move(N,M)).
primitive_action(moveToTable(B)).


proc(doMove, pi(n, pi(y, moveToTable(n) # move(n,y)))).

% Restore suppressed situation arguments.
restoreSitArg(clear(N),S,clear(N,S)).
restoreSitArg(on(N,M),S,on(N,M,S)).
restoreSitArg(ontable(M),S,ontable(M,S)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% added by Yves Lesperance

show_act_seq(s0).
show_act_seq(do(A,S)):- show_act_seq(S), write(A), nl.

run:- do(control,s0,S), show_act_seq(S).

% definition of executable (legal) situation

executable(s0).
executable(do(A,S)) :- poss(A,S), executable(S).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- use_module(library(statistics)).
:-include('fluents.pl').
:- current_prolog_flag(argv, Argv), concat_atom(Argv,' ', SingleArg), open(SingleArg,append,OS),
write(OS,'Running testcase...'),
nl(OS),
statistics(cputime, T0),
run,
statistics(cputime, T1),
T is T1-T0,
write(OS,T),nl(OS),close(OS).
:- halt.
```

C.2.6. Golog (reordered)

```
:- [golog_swi].

/*  Action Precondition Axioms  */
poss(move(X,Y),S) :- clear(X,S), clear(Y,S), not(X = Y).
poss(moveToTable(X),S) :- clear(X,S), not(ontable(X,S)).

/*  Successor State Axioms  */
clear(X,do(A,S)) :-( A = move(Y,Z) ; A = moveToTable(Y)),on(Y,X,S)
; clear(X,S), not(A = move(Y,X)).

on(X,Y,do(A,S)) :- A = move(X,Y) ;
                   on(X,Y,S), not(A = moveToTable(X)), not(A = move(X,Z)).
ontable(X,do(A,S)) :- A = moveToTable(X) ;
                      ontable(X,S), not(A = move(X,Y))        .

% Primitive control actions
primitive_action(move(N,M)).
primitive_action(moveToTable(B)).

proc(doMove, pi(n, pi(y, move(n,y) # moveToTable(n)))).

% Restore suppressed situation arguments.
restoreSitArg(clear(N),S,clear(N,S)).
restoreSitArg(on(N,M),S,on(N,M,S)).
restoreSitArg(ontable(M),S,ontable(M,S)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% added by Yves Lesperance

show_act_seq(s0).
show_act_seq(do(A,S)):- show_act_seq(S), write(A), nl.

run:- do(control,s0,S), show_act_seq(S).

% definition of executable (legal) situation

executable(s0).
executable(do(A,S)) :- poss(A,S), executable(S).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- use_module(library(statistics)).
:-include('fluents.pl').
:- current_prolog_flag(argv, Argv), concat_atom(Argv,' ', SingleArg), open(SingleArg,append,OS),
write(OS,'Running testcase...'),
nl(OS),
statistics(cputime, T0),
run,
statistics(cputime, T1),
T is T1-T0,
write(OS,T),nl(OS),close(OS).
:- halt.
```