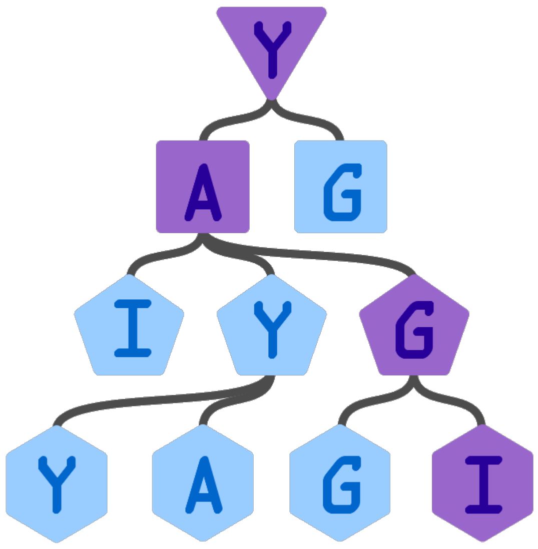
Improvement and Extension of YAGI, a Golog Based Programming Language for Planning and Real-Time Interaction in Robotics

Thomas Eckstein



Bachelor Thesis

Supervisor
Assoc.Prof. Dipl.-Ing. Dr.techn. Gerald Steinbauer

submitted to
Graz University of Technology
Institute for Software Technology

Graz, July 2019

Abstract

YAGI is a declarative and prodedural programming language based on Situation Calculus and Golog. The language allows the user to model the target domain as a set of actions, fluents and procedures or as a planning goal. Its online execution mode allows for real-time control of an autonomous agent by sending signals or by reading sensor data. This paper is based on a previous version of YAGI, which has major drawbacks such as low performance or a limited syntax. We propose improvements and redesign the language syntax along with a brand new, more performant interpreter. Additionally, we relate the semantics of our language to Second Order Logic. Moreover, we benchmark our implementation through some famous planning problems and show that it performs several magnitudes better than the previous version regarding runtime and efficiency. YAGI has also been tried by university students in the course "Classical Topics of Computer Science" with positive results. Our implementation can be found online in the repository linked below.[1]

[1] https://git.ist.tugraz.at/ais/yagi/tree/eckstein/rewrite/

Contents

1 Introduction

High level control in robotics has always been a great challenge due to its complexity and difficulties in formalization. In domains such as traffic control, assembly lines or packet delivery there is tons of live information coming into the control system: Locations of packets or parts, locations of agents and vehicles or requests with different priorities. The control system has to make a plan based on this information and output a short as possible sequence of actions to fulfill the planning goal. But as soon as a few actions are executed in the real world, things may go wrong very quickly: Vehicles get stuck in traffic, a component stops working, new requests arrive or priorities change. With such a vast range of requirements, how can one develop a swiss army knife to model and solve such problems?

One early approach to formalize such problems was made by [McCarthy, 1963] through Situation Calculus, a formal framework in Second Order Logic to reason about actions and their effects. Golog, which was proposed by [Levesque et al., 1994] extends Situation Calculus by providing control structures that allow action based planning. Golog based interpreters, such as ConGolog [De Giacomo et al., 2000] were implemented in the logical programming language Prolog. In addition to those, [Maier, 2015] designed the Golog based language YAGI and implemented an accompanying standalone interpreter. He set focus on giving a clear language definition and on cleanly separating front-end and back-end. YAGI is still considered to be in the early alpha-stage. While the interpreter has serious performance issues, the language lacks basic data types, syntactic sugar and a typechecker. [Maier, 2015] has come to the conclusion that his implementation can solve some problems with higher reliability than traditional Prolog based Golog, but is slower by several magnitudes.

Our goal is to improve and extend the YAGI ecosystem and to show the effectiveness of our modifications. At first we want to examine the weaknesses of the existing YAGI version in order to get ideas on what to improve. The language needs a more compact syntax with more features and data types. Usability is another important issue, because the programmer needs to get better syntax and type errors at compile time. We want to ensure that our language is still in compliance with the theoretical background provided by Situation Calculus and Golog by mapping each YAGI construct to a logical sentence. Subsequently we want to reimplement the interpreter from scratch to combat performance issues and to improve compatibility with other systems. Finally we want to test our implementation by modelling several planning problems in our improved version of YAGI and by measuring the performance of those compared to the previous YAGI implementation.

2 Related Work

In this chapter we introduce research that we build our own work on and give an overview over work that is similar to ours. In Chapter 3 we go further in depth regarding topics necessary for understanding our work.

Situation Calculus [McCarthy, 1963] formalizes how actions affect the state of the world by crafting a Basic Action Theory, which consists of Foundational Axioms, Action Preconditions, the Initial Database, Successor State Axioms and the Unique Name Axioms. A situation is described by its action history starting from the Initial Situation and Fluents hold information about the world depending on the given situation. The Action Preconditions dictate which actions can be legally executed in a given situation. Successor State Axioms describe how each fluent changes depending on the action being executed. The Initial Database contains all logic sentences that hold true in the Initial Situation. Regression allows the user to ask queries, such as if a fluent holds true or an action is executable in a given situation.

[Levesque et al., 1994] have introduced the programming language Golog, which is executed by an interpreter written in Prolog and utilizes the features provided by Situatio Calculus. A program can execute primitive actions or branch and loop depending on a formula over the current situation. The language also provides non-deterministic constructs such as *pick* or *choose* that enable planning, meaning that all possible execution traces are evaluated internally before returning a suitable one. The semantics of Golog are defined through the predicate $Do(\delta, s, s')$, which holds true if the execution of the program δ results in the situation s' starting from s.

ConGolog [De Giacomo et al., 2000] and its successor IndiGolog [De Giacomo et al., 2009] are Golog based programming languages with more features added. Online execution mode is introduced, during which actions are directly executed and decisions are irreversibly made. The search operator enforces conventional offline behavior, where the whole decision tree is searched for a valid execution trace. Interrupts and concurrent execution are also among the more sophisticated features of IndiGolog. Online semantics are formalized through the predicates $Trans(\delta, s, \delta', s')$ and $Final(\delta, s)$, where a program δ is incrementally executed resulting in a new program δ'.

Readylog [Böhnstedt et al., 2007] is another Golog dialect with an interpreter implemented in Prolog. The language stands out for its optimized forward-planning strategy that allows for better real time performance in robotic agents. The algorithm calculates an optimal policy for a given input program by solving a Markov Decision Process. A reward function judges the planning progress and probabilistically suggests the next action to take. This way the search tree can be kept narrow by cutting off fruitless branches that don't contribute to reaching the planning goal. Readylog has been successfully tried in a robotic soccer domain.

An alternative to Situation Calculus is Fluent Calculus [Thielscher, 2006], where knowledge is represented as a set of fluents that hold true instead of an action history. Conversely, State Update Axioms specify which states are added or removed when an action is executed. Progression is used to roll the current fluent database forward to a new database. This approach also works fine for reasoning with incomplete knowledge, where the actual state of the world may only be discovered later on. The Fluent Calculus based counterpart to Golog is FLUX (FLUent eXecutor), a programming language with a Prolog based interpreter.

The action based programming languages discussed above are designed to be used in autonomous agents where little to no human intervention is possible. Their applications range from delivery robots to planetary rovers. For example, [Ingrand et al., 2007] have engineered software for two experimental rovers from bottom up. Their work covers ways to read and interpret sensor data and the retrievel terrain data from camera images. The planning system OpenPRS is used for high-level decision making.

[Ferrein et al., 2012] have laid the groundwork for YAGI by specifying a set of requirements for the language and by suggesting a syntax. According to their specification, an important feature should be that the ecosystem is independent from Prolog and can easily be embedded into any existing platform. Focus should also be laid on familiarity to the user, real-time bidirectional communication with the outside world and abstraction of Successor State Axioms regarding the language design.

Based on this specification, [Maier, 2015] has further refined the existing language definition and has created a working interpreter of the YAGI programming language. The interpreter is written in C++ and has a clear separation of back-end and front-end. [Maier, 2015] has shown that his implementation works and can successfully solve simple planning problems.

Regarding compiler construction, [Nystrom, 2018] provides a broad overview, covering topics such as parsers, tree-walk interpreters or bytecode virtual machines. At the heart of the book lies the simple programming language Lox, which is used throughout all chapters to build a fully functional parser and interpreter step by step. The programming languages Java and C are used for implementing the ecosystem around Lox.

3 Prerequisites

3.1 Situation Calculus

Situation Calculus [McCarthy, 1963] is a concept based on Second Order Logic that allows one to reason about actions and change in the world. So far we've only had static and unchangeable statements like "If it rains, then the floor is wet." or "There exists no integer greater than one such that one plus the product of all numbers up to this integer is divisible by said integer.". But how do we express questions like "Is the food on the table after we've picked it up in the kitchen, have gone to the table and put it down?" in logical statements? Situation Calculus provides the tools necessary to model such scenarios by introducing actions, fluents and the concept of situations. The totality of all axioms required to model a domain in Situation Calculus is called a Basic Action Theory (BAT) $\mathcal{D}$, which is composed of five types of axioms:

- Foundational Axioms Σ
- Action Preconditions $\mathcal{D}_{AP}$
- Initial Database $\mathcal{D}_{S_0}$
- Successor State Axioms $\mathcal{D}_{SSA}$
- Unique Name Axioms $\mathcal{D}_{UNA}$

3.1.1 Example Scenario

We can best demonstrate the basics of Situation Calculus by modelling a simple domain. A packet delivery robot can travel between various locations, which are sparsely interconnected. The robot can pick up or drop packets in these locations, but can only carry up to two packets at the same time.

We will introduce the necessary logic sentences step by step in order to build our BAT.

3.1.2 Specifying Constants and Predicates

There are two kinds of objects in our world that we need to work with: Packets and Locations. For those we declare the constants P_1, P_2, P_3, A, B, C, D, E, F, G, H, I. We introduce the type predicates $packet(p)$ and $location(l)$ that will later allow us to quantify over the instances of these classes. Please note that all statements listed below are true, while combinations of locations not present are false. For example $edge(A, I)$ is false. This concept that all statements not mentioned in the database are false is called the Closed-World Assumption.

$packet(P_1)$
$packet(P_2)$
$packet(P_3)$
$location(A)$
$location(B)$
$location(C)$
$location(D)$
$location(E)$
$location(F)$
$location(G)$
$location(H)$
$location(I)$

So if one wants to check if the unary predicate P holds for all locations, one can say $\forall l : location(l) \rightarrow P(l)$. The quantifier simply quantifies over all known objects in the domain and if the object is a location, P must hold or otherwise the implication is true anyways, because false can imply anything.

It's also possible to add further predicates - so called facts - to our BAT, for example specifying between which locations the robot can travel:

$edge(A, B)$
$edge(A, D)$
$edge(B, C)$
$edge(B, C)$
$edge(C, E)$
$edge(D, E)$
$edge(D, F)$
$edge(F, H)$
$edge(G, H)$
$edge(H, I)$

3.1.3 Actions

Actions are the only means by which our robot can interact with the world and can change it. An action is a function with no further meaning attached to it. Actions can also be made to have parameters. In our example domain the robot can execute three actions:

- $move(l)$ - Move to a location.
- $pickup(p)$ - Pick up a packet.
- $drop(p)$ - Drop a packet.

Due to the functional nature of actions, Unique Name Axioms $\mathcal{D}_{UNA}$ must ensure that only actions with the same name and parameters are equal:

$move(l) \neq pickup(p)$
$move(l) \neq drop(p)$
$pickup(p_1) \neq drop(p_2)$
$move(l_1) = move(l_2) \rightarrow l_1 = l_2$
$pickup(p_1) = pickup(p_2) \rightarrow p_1 = p_2$
$drop(p_1) = drop(p_2) \rightarrow p_1 = p_2$

3.1.4 Situations

A situation is a way to express the current state of the world. It is represented by the sequence of actions taken since the initial situation s_0. This definition implies that two situations with the same final result, but with different actions taken, are not equal. Situations are expressed using the $do(a, s)$-function. For example, moving to B and picking up $P1$ can be written as $do(pickup(P1), do(move(B), s_0))$.

The Foundational Axioms Σ, as defined by [Reiter, 1993], are Second Order Logic sentences and help define what a valid situation is:

$\forall P : P(s_0) \wedge \forall a, s : (P(s) \rightarrow P(do(a, s))) \rightarrow \forall s : P(s)$
$\forall s : \neg s < s_0$
$\forall a, s, s' : s < do(a, s') \equiv Poss(a, s') \wedge (s = s' \vee s < s')$

The first axiom holds the inductive definition of situations. s_0 is a valid situation and if s is a situation, then $do(a, s)$ is also a situation. The overloaded operator $s < s'$ is true if s' is reachable from s by legally executing actions. The second axiom states that no situation precedes the Initial Situation and the third axiom recursively defines the $<$ operator and enforces preconditions.

3.1.5 Fluents

In order to assign properties to situations, fluents are used. A fluent is a predicate that tells if a property holds in a situation. In contrast to facts, fluents accept a situation term as last parameter and their truth value changes depending on the situation and actions taken. In our robot domain we have the following fluents:

- $robotAt(l, s)$ - Is true if the robot is at the given location.
- $packetAt(p, l, s)$ - Is true if the given packet is at the given location.
- $holds(p, s)$ - Is true if the robot carries the given packet.
- $holdingCount(n, s)$ - Is true if the robot carries exactly n packets.

3.1.6 Building a Basic Action Theory

So now that we have explained the concept is actions, situations and fluents, we need to express how they relate to each other. At first we specify in the Initial Database $\mathcal{D}_{S_0}$ which fluents hold true in the initial situation s_0:

$robotAt(A, s_0)$
$packetAt(P_1, B, s_0)$
$packetAt(P_2, E, s_0)$
$packetAt(P_3, F, s_0)$
$holdingCount(0, s_0)$

This set of sentences is called the Initial Database $\mathcal{D}_{S_0}$. Note that because the robot holds no packets at the beginning, *holds* is false for all packets and is therefore not mentioned according to the Closed-World Assumption.

Next we want to ensure that actions are only executed if possible. So for each action a precondition is introduced, which is written as a predicate definition that accepts an action term and a situation as parameters.

The robot can only move between interconnected locations:

$Poss(move(to), s) \equiv \exists from : robotAt(from, s) \land (edge(from, to) \lor edge(to, from))$

A packet can only be picked up if it's at the robot's location and the robot doesn't carry two packets:

$Poss(pickup(p), s) \equiv \exists from : robotAt(l, s) \land packetAt(p, l, s) \land \neg holdingCount(2, s)$

A packet can be dropped if the robot is currently holding it:

$Poss(drop(p), s) \equiv holds(p, s)$

But how do we express that an action changes a fluent and that fluents are preserved through time or that an action doesn't change a fluent? This issue is called the Frame Problem, for which there exist multiple solutions. One solution is to introduce a so called Successor State Axiom for each fluent. Such an axiom doesn't describe the direct effect of an actions on fluents, but rather how a fluent is changed by different actions. A fluent becomes true if either γ^+ is true or the fluent was true before and γ^- is false:

$F(\overrightarrow{x}, do(a, s)) \equiv \gamma^+(\overrightarrow{x}, a, s) \lor F(\overrightarrow{x}, s) \land \neg\gamma^-(\overrightarrow{x}, a, s)$

The robot is at a location if it has moved there or if it was there before and hasn't moved anywhere else:

$robotAt(l, do(a, s)) \equiv a = move(l) \lor robotAt(l, s) \land \nexists l' : a = move(l')$

A packet is at a location if the robot has just dropped it there or if it was already there and wasn't picked up.

$packetAt(p, l, do(a, s)) \equiv a = drop(p) \land robotAt(l, s) \lor packetAt(p, l, s) \land a \neq pickup(p)$

The robot holds a packet if it was just picked up or it was already held and not dropped:

$holds(p, do(a, s)) \equiv a = pickup(p) \wedge holds(p, s) \wedge a \neq drop(p)$

The holding count has either increased from picking up a new packet or decreased from dropping one:

$$\begin{aligned} holdingCount(n, do(a, s)) \equiv\ & (\exists p : a = pickup(p) \wedge holdingCount(n-1, s) \\ & \vee a = drop(p) \wedge holdingCount(n+1, s)) \\ & \vee holdingCount(n, s) \wedge (\nexists p : a = pickup(p) \vee a = drop(p)) \end{aligned}$$

The structure of the Successor State Axioms shows that the truth value of a fluent isn't defined explicitly, but rather emerges from the sequence of actions taken before.

3.1.7 Regression

Now that we've modelled the domain of our robot, we can ask queries about the state of fluents or the possibility to execute new actions. Regression, as extensively described by [Reiter, 2001], works by providing a query in the current situation and by going back in time to see if the query holds. This means that all preconditions and fluents are substituted with their respective definitions until all terms can be proven to lie within s_0 or not.

Examples:

- Is it possible to move to I if the robot moved to D?

  $situation(do(move(I), do(move(D), s_0)))$
- Is the robot at location B after the robot moved to B and then picked up packet 1?

  $robotAt(B, do(pickup(P_1), do(move(B), s_0)))$

3.1.8 Progression

Contrary to regression, progession goes forward in time. Situations are not defined by their action history anymore, but simply by the fluents holding true. Upon execution of an action, the Initial Database is rolled forward to a new database, where the new truth values of fluents are explicitly stored. While queries can be answered much quicker this way, one can't go back in time and see the action history. However, [Lin and Reiter, 1997] have shown that progression is not always definable in First Order Logic by giving a counterexample where no progression exists.

3.2 Golog

Golog [Levesque et al., 1994], which stands for *alGOl for LOGic*, is an action programming language that allows one to program with Situation Calculus. Golog is not a programming language itself, but rather a set of clauses in Prolog. Although being largely imperative, it allows the use of non-deterministic planning. It can be used for autonomous and dynamic agents or for planning tasks in logistics.

3.2.1 The Do-Predicate

A Golog program δ is a syntax tree of possible expresions discussed below. The predicate $Do(\delta, s, s')$ is true if δ can be interpreted so that s will become s'. Because Golog is non-deterministic, there may exist multiple s' for which the predicate is true given δ and s. The Do-Predicate is false if there is no path from s to s' using δ.

3.2.2 Primitive Action

The Golog expression $a(...)$ takes the action with the name a from the BAT, checks its precondition and then appends it to the action history including the given arguments. The semantics of a primitive action are as follows:

$Do(a(...), s, s') \equiv Poss(a(...), s) \land s' = do(a(...), s)$

3.2.3 Test Action

The Golog expression $\phi?$ fails if the logic formula ϕ doesn't hold in the current situation and otherwise execution continues without altering the situation. The semantics of a test action are as follows:

$Do(\phi, s, s') \equiv \phi[s] \land s' = s$

$\phi[s]$ means that all situation dependent terms in this formula are evaluated using the given situation.

3.2.4 Sequence

The expression $\delta_1; \delta_2$ executes the two given child expressions sequentially and fails if at least one of these two programs fails. The semantics of a sequence are defined so that there must exist an intermediate situation that connects the two programs:

$Do(\delta_1; \delta_2, s, s') \equiv \exists s'' : Do(\delta_1, s, s'') \land Do(\delta_2, s'', s')$

3.2.5 Non-Deterministic Choice

The expression $\delta_1|\delta_2$ non-deterministically executes one of these child expressions and fails if both fail. Non-determinism means that the subsequent control flow, including everything after the choice expression, will be forked in two. One path will have executed δ_1, the other δ_2. The paths that does't fail "win". The semantics are defined using a logical conjunction:

$Do(\delta_1|\delta_2, s, s') \equiv Do(\delta_1, s, s') \lor Do(\delta_2, s, s')$

3.2.6 Conditionals

The expression $if\ \phi\ then\ \delta_1\ else\ \delta_2\ endif$ checks if a logic formula holds and then executes one of two expressions depending on the formulas truth value. The semantics of this construct can be defined in Golog itself as $(\phi; \delta_1)|(!\phi?; \delta_2)$ or otherwise as:

$Do(if\ \phi\ then\ \delta_1\ else\ \delta_2\ endif, s, s') \equiv \phi[s] \land Do(\delta_1, s, s') \lor \neg\phi[s] \land Do(\delta_2, s, s')$

3.2.7 Non-Deterministic Choice of Arguments

Golog also allows the programmer to non-deterministically choose arguments for a subprogram. The expression $(\pi x)\delta(x)$ fails if there is no x such that the subprogram succeeds. The semantics are defined using an exists quantifier:

$Do((\pi x)\ \delta(x), s, s') \equiv\ \exists x : Do(\delta(x), s, s')$

3.2.8 Non-Deterministic Iteration

The expression $\delta*$ executes the given child expression for an undefined number of times, including not executing it at all. Therefore this expression can never fail. The semantics are defined using Second Order Logic:

$Do(\delta*, s, s') \equiv \forall P : (\forall s1 : P(s1, s1)) \wedge (\forall s1, s2, s3 : (P(s1, s2) \wedge Do(\delta, s2, s3) \rightarrow P(s1, s3))) \rightarrow P(s, s')$

This means that for all relations P that describe the sequential execution of δ for an arbitrary number of times, $P(s, s')$ must hold. A relation P is suitable if it is reflective and transitive. It must be reflective because δ can be executed zero times and thus $s = s'$. It must also be transitive because $\delta*; \delta$ is a possible outcome of $\delta*$.

3.2.9 While-Loop

The expression *while* ϕ *do* δ *endwhile* works like a classical and deterministic while loop. It executes the given child expresion as long as the given condition holds for the resulting situation. Its semantics can be defined as the Golog expression $(\phi?; \delta)*; \neg\phi?$ using non-deterministic iteration until the condition doesn't hold anymore.

3.2.10 Programs and Procedures

A complete Golog program is composed of a list of procedures and a main expression that will be evaluated upon execution of the program. A procedure is a named expressions that may accept parameters and can be called like a function. Parameters are passed by-value. The Golog program

proc $P_1(x_1)\ \delta_1$ *endproc*; ...; *proc*$P_n(x_n)\ \delta_n$ *endproc*; δ_0

contains n procedures P_n, each accepting a parameter x_n and containing the expression δ_n. The expression δ_0 is the main program.

3.2.11 Example Program

Based on the robot example described in the chapter above, we implement a Golog program that lets the robot serve requests. The domain is extended by a fluent $requests(p, l, s)$ that holds all active requests and an action $removeRequest(p)$ that marks a request as served. The program could look as follows:

```
proc findPath(to)
  while ¬ robotAt(to) do
    (π next) (
      location(next)?;
      move(next)
    )
  endwhile
endproc;

while ∃ p, l: requests(p, l) do
  (
    (π p') (
      (π l') (
        packetAt(p', l')?;
        ∃ l'': requests(p', l'')?;
        findPath(l');
        pickup(p')
      )
    )
  ) | (
    (π p') (
      (π l') (
        holds(p')?;
        requests(p', l')?;
        findPath(l');
        drop(p');
        removeRequest(p')
      )
    )
  )
endwhile
```

The procedure *findPath* iterates until the robot is at the desired location. Each iteration, a random location is picked and the robot tries to move there. Because the choice of locations is non-deterministic, the robot will eventually find a path leading to the destination. The precondition of *move* ensures that the program will look somewhere else if an unsuitable location is picked. A weakness of this procedure is that it won't terminate if there exists no path to the desired location, because the robot will move indefinitely between already visited nodes. A solution to this problem is to record nodes that have already been visited and exclude them from the search.

The main part of the program iterates until all requests have been answered. In each iteration, either a packet at its origin will be visited and picked up or an already loaded packet will be brought to its destination. This ensures that multiple packets can be loaded at the same time and thus the problem can be solved with little steps.

Another solution would be to serve requests sequentially, which would be more time efficient, but doesn't make use of the robot's capability to hold multiple packets at the same time.

3.3 Transition Semantics of ConGolog and IndiGolog

ConGolog [De Giacomo et al., 2000] and IndiGolog [De Giacomo et al., 2009], which are both Prolog based interpreters of Golog, base their execution semantics on the Second Order Logic predicates $Trans(\delta, s, \delta', s')$ and $Final(\delta, s)$ instead of the offline semantics using the Do precidate discussed above. With the transition semantics it is possible to roll a program δ forward to a new program δ' in an iterative manner. Each transition accounts for an atomic step, so online execution is possible and the program can be interrupted at will. $Trans$ holds true if the program δ along with the situation s can be executed resulting in a remaining program δ' and a new situation s'. The predicate $Final$ is true if the given program can terminate successfully, i.e. no more steps have to be taken. Below is a slight adaptation for better readability of both predicates as defined by [De Giacomo et al., 2000]. The symbol ϵ represents an empty program.

- An empty program can't be executed any further:

$$\begin{aligned} Trans(\boldsymbol{\epsilon}, \_, \_, \_) &\equiv \bot \\ Final(\boldsymbol{\epsilon}, \_) &\equiv \top \end{aligned}$$

- A primitive action demands a check of its precondition and results in an empty remaining program and a new situation:

$$\begin{aligned} Trans(\boldsymbol{a}, s, \epsilon, do(a[s], s)) &\equiv Poss(a[s], s) \\ Final(\boldsymbol{a}, \_) &\equiv \bot \end{aligned}$$

- Test actions demand the truth of the given formula and result an an empty program and leave the situation unchanged:

$$\begin{aligned} Trans(\boldsymbol{\phi?}, s, \epsilon, s) &\equiv \phi[s] \\ Final(\boldsymbol{\phi?}, \_) &\equiv \bot \end{aligned}$$

- When two programs are executed sequentially, the first one is executed until termination and then the second one is executed:

$$\begin{aligned} Trans(\boldsymbol{\delta_1;\delta_2}, s, \delta', s') &\equiv (\exists\gamma : \delta' = (\gamma;\delta_2) \wedge Trans(\delta_1, s, \gamma, s')) \vee Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s') \\ Final(\boldsymbol{\delta_1;\delta_2}, s) &\equiv Final(\delta_1, s) \wedge Final(\delta_2, s) \end{aligned}$$

- Nondeterministic branching tries to find at least one feasible subprogram:

$$\begin{aligned} Trans(\boldsymbol{\delta_1|\delta_2}, s, \delta', s') &\equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s') \\ Final(\boldsymbol{\delta_1|\delta_2}, s) &\equiv Final(\delta_1, s) \vee Final(\delta_2, s) \end{aligned}$$

- Nondeterministic choice of arguments quantifies over the possible options to find a match:

$$\begin{aligned} Trans(\boldsymbol{\pi v.\delta}, s, \delta', s') &\equiv \exists v : Trans(\delta^v_x, s, \delta', s') \\ Final(\boldsymbol{\pi v.\delta}, s) &\equiv \exists v : Final(\delta^v_x, s) \end{aligned}$$

- Because nondeterministic iteration can also lead to the program being executed 0 times, the corresponding $Final$ predicate is always true:

$$\begin{aligned} Trans(\boldsymbol{\delta*}, s, \delta', s') &\equiv \exists\gamma : \delta' = (\gamma;\delta*) \wedge Trans(\delta, s, \gamma, s') \\ Final(\boldsymbol{\delta*}, \_) &\equiv \top \end{aligned}$$

- Conditionals:

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

- While loops:

$$\begin{aligned} Trans(\boldsymbol{while\ \phi\ do\ \delta\ endWhile}, s, \delta', s') &\equiv (\exists\gamma : \delta' = (\gamma; while\ \phi\ do\ \delta\ endWhile) \land Trans(\delta, s, \gamma, s')) \\ Final(\boldsymbol{while\ \phi\ do\ \delta\ endWhile}, s) &\equiv \neg\phi[s] \land Final(\delta, s) \end{aligned}$$

3.4 Q3

Q3 (Questive 3.0) is an event based, dynamically typed, functional and object oriented language developed by us [Eckstein, 2019] in parallel to this project. It was originally designed for a Minecraft server to be used for ingame scripting, such as quests or minigames. As Minecraft is written in Java, the ecosystem around Q3 is Java-based as well. The language is compiled to Q3 bytecode, which is executed by the Q3 Machine (Q3M). The Q3M can also be embedded in any other Java-application with ease. The programmer can make Java functions and classes callable within Q3 through the use of Java annotations. Alternatively the language can be used without a host system through an interactive shell.

The strongly and dynamically typed language is inspired by actor systems, where actors or tasks run independently and communicate via messages. Q3 provides a task hierarchy, where a task has control over its child tasks. If a task terminates, all its child tasks are forced to terminate too. When a task throws an exception, the task is killed and the exception is propagated to the parent task unless an error handler is set. The task hierarchy allows effective control and isolation of single components. Tasks can either broadcast signals or wait for signals, with pattern matching taking place. The host system can send signals as well, with the Q3 program reacting to it. Asynchronous data types such as futures are also provided by the language, such that non-blocking I/O can be implemented transparently and the Q3M can run in a separate thread. Q3 has a rich type system (everything is an object) with a high degree of reflection and introspection. For pattern matching, there is a type implication relation defined, where `null` implies all other values and `true` is implied by everything. A class is implied by its superclass, while an instance of a case class is implied by an instance of the same type with matching wildcards. Code can be compiled and swapped at runtime, allowing the programmer to develop on a running system. A `Loader` is a class that represents the contents of a code file and is responsible for resolving accesses to functions or classes. Loaders can be manipulated at runtime to load or reload code. A standard library provides basic data structures and access to the filesystem and network.

The repository along with the documentation for Q3 is linked below.[2]

[2]`https://git.rootlair.com/teckstein/q3`

4 Problem Statement

As discussed in Chapter 2, there already exist two mentionable Golog based interpreters aside from Golog itself: IndiGolog, which is based on Prolog and YAGI, which has been developed by [Maier, 2015] as a standalone interpreter written in C++. In this chapter we describe the drawbacks of these two and why it is necessary to redesign YAGI.

IndiGolog, as described by [De Giacomo et al., 2009], is merely a set of Prolog clauses making up an interpreter. The user gives a Golog program as a set of Prolog clauses, where each procedure is a clause. Such a procedure is given more or less as its raw syntax tree, so little structure and order can be brought into such a program. While the Prolog interpreter can detect simple syntax errors, semantical errors like referring to a non-existing procedure or variable remain undetected at compile time. Debugging Prolog programs can be quite a challenge. The most severe drawback however is that Prolog and thus IndiGolog uses depth-first-search, which doesn't always terminate and often doesn't yield the optimal result. Advanced searching techniques like iterative deepening can be used to mitigate this problem, but it adds yet another layer of complexity to the Golog program.

In response to IndiGolog, [Maier, 2015] has implemented YAGI (Yet Another Golog Interpreter), which we will refer to as Legacy YAGI in this paper. The ecosystem of Legacy YAGI consists of a language definition and a parser and interpreter written in C++ including an API. The user can develop their program in a Golog based language with a clearly defined syntax and a sufficient amount of structure. When running the resulting program, the parser first builds a syntax tree and checks for syntactical correctness. ANTLR, a popular framework for building parsers, is used. The typechecker can detect some types semantical errors at compile time. A classical syntax tree walker is used for the execution of the program. Fluents are represented by SQL tables, whereat SQLite is used as engine. Unlike in IndiGolog, the effects of actions are modelled explicitly by stating which fluents are changed directly instead of stating which fluents are changed by which actions in the Successor State Axioms.

The ecosystem comes with a built in shell, where the user can run and interact with Legacy YAGI programs. Output is delivered via signals, which are represented by strings, and is sent to the system interface where it is either forwarded to an agent or to the shell for the user to see. The program retrieves input from the outside world by calling setting actions. Upon the invocation of a setting action, the agent reads a concrete value or the user can manually input data.

However, Legacy YAGI still has a number of drawbacks. The most severe is the lack of data types in the language. So far only strings can be used for passing values. There are no enum types for organizing constants belonging together logically. Numeric types are lacking and so are arithmetic operators. It is thus hard to work with coordinates or ordered elements in general. Legacy YAGI has the comparison operators `< <= >= >`, which compare strings lexicographically. To check if the number nine is less than twelve, one would have to write `"09" < "12"`.

Because there are no enum types to organize constants with, the user has to state all possible values each time:

```
fluent objectAt [{"object1", "object2", "object3"}][{"room1", "room2", "room3"}];
fluent personAt [{"alice", "bob"}][{"room1", "room2", "room3"}];
```

Instead of simply stating that both fluents require a room as a second parameter, all rooms must be given in both declarations. Larger domains or more complex programs can easily get messy this way. This becomes a huge obstacle when refactoring code. If another object is introduced, all fluents referencing similar objects must be updated accordingly.

Another drawback is the verbose syntax of Legacy YAGI. Instead of using curly brackets, control structures are closed with the keyword `end` and the name of the control structure. For example, a simple if-then-else statement is written like this:

```
if (condition) then
  doThis();
else
  doThat();
```

```
end if
```

Unless having an IDE with good auto completion available, such a syntax can increase the effort of writing the program significantly. When extracting values from a set, pick-statements are used:

```
pick <$value> from someFluent such
  doThis();
end pick
```

The visibility of the variables picked is limited to the enclosing block. If multiple pick operations are executed sequentially, these would need to be nested repeatedly. Furthermore there is no way to limit the scope of pick before executing the contents of the block. With online semantics this can be a considerable problem, because there is no way to specify in detail which value to pick and so the program will likely fail. This is less of a problem with offline semantics, because all possibilities are searched before executing any actions, but it can make a negative performance impact overall.

As [Maier, 2015] describes in ,,Future Work", there should be some optimizations the user can make to the search algorithm to cut off search branches not leading to a result or branches that have already been searched through.

Another issue is the limited functionality of the typechecker. When there are no enum types, it's hard to keep track of which strings can be passed to a fluent and which variable likely holds which type at a given time. This still leaves the possibility of getting runtime errors that could have been detected during compilation. Through the use of an out-of-the-box parser, in this case ANTLR, only basic error messages for syntax errors can be produced, which negatively impacts the development process for the user.

As mentioned above, Legacy YAGI uses SQL tables to represent fluents and uses SQL queries to access or update them. This approach adds significant overhead to the execution of the program, because SQL tables consume more memory than a specifically tailored data structure and processing queries also takes substantial CPU time, which becomes a problem when performing extensive planning.

Considering the disadvantages of Legacy YAGI described above, it becomes necessary to revisit the ecosystem in all of its aspects and to design a more user friendly and feature rich language based on Golog. The user must be able to use more data types such as numbers, tuples or enums values. The compiler must be able to detect all syntactical and semantical issues in the given program and output informative error messages to the user. There must be a way to bring more structure into programs and to control the behavior of the search algorithm in order to increase performance and decrease the memory footprint.

Making YAGI platform independent was also a core issue that [Maier, 2015] left open. Legacy YAGI is so far only available for Unix-like systems with numerous dependencies needed to be manually installed for the application to work.

5 Concept

We decide to embed YAGI into the Q3 ecosystem since the Q3M is already a powerful enough interpreter to execute YAGI code, so we don't have to write a new interpreter for YAGI, but merely a compiler. A great strength of the Q3M is to execute many tasks in one thread, which is incredibly useful for implementing a breadth-first search algorithm. Another advantage is that we have a scripting language with which we debug and test our implementation. Q3 scripts can also be used to glue YAGI and external components together, meaning that the execution of the YAGI program and event handling can be fine tuned well. Since Q3 is Java-based, platform independence is easily achievable, as only JDK (the Java compiler and other development tools) and Maven (a popular build tool for Java) are required to build the program.

5.1 Architecture

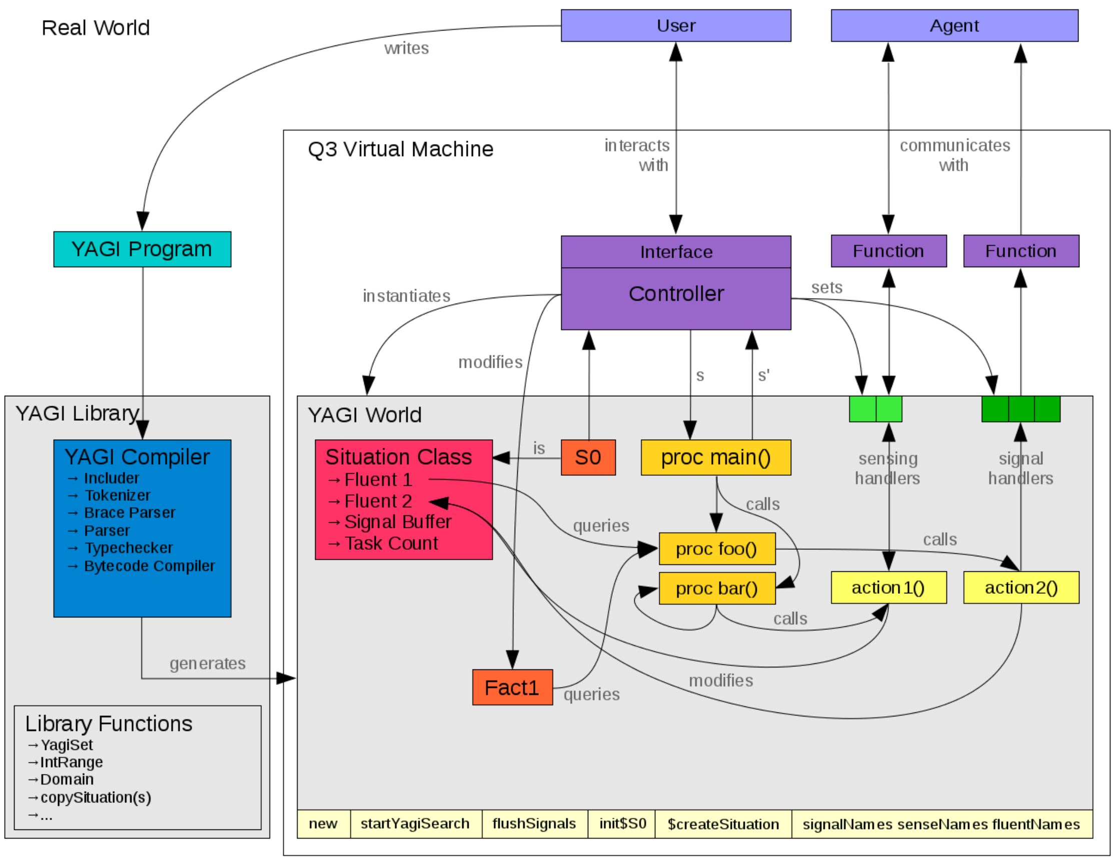


Figure 1: A detailed architectural overview of a YAGI program and the components around it.

A system executing a YAGI program consists of multiple components and layers interacting with each other. The most important component of any software system is of course always the user, who is responsible for setting up the workspace, installing YAGI and getting it to work on the host agent of their choice. Then the user has to write the YAGI program, which consists of the BAT and the business logic.

The YAGI Library, as outlined in Figure 1, contains the YAGI Compiler, which checks for syntactical and semantical errors and compiles the program to Q3 Bytecode, which can be run on the Q3 Virtual Machine (Q3M). It additionally provides library functions and data types such as sets (`YagiSet`) and ranged integers (`IntRange`) to the Q3M.

The Q3 Executable File (.q3x) generated by the compiler contains a class called `YagiWorld`, which in turn contains all actions, procedures and facts. It also contains the class representing a situation, which holds all fluents. Furthermore the `YagiWorld` class has a member variable for each sensing and signal declaration, to which a function can be assigned. Aside from the constructs occuring in the YAGI program, the `YagiWorld` contains some helper functions and variables. The contents of this class will be discussed in a later chapter in detail.

The controller is a Q3 script that connects the user, the `YagiWorld` and the agent with each other. It instantiates the `YagiWorld` and assigns functions to the sensing and signal handlers. These functions are called by the YAGI program and read information from the real world or trigger an event. The controller can then call procedures, which take a starting situation as parameter and return the resulting situation. As a side effect sensing and signal handlers are invoked. Because facts are instances of YagiSet and YagiSets are mutable, the controller has the power to manipulate facts. Chapter 6.3 goes more into detail on how these functions inside the `YagiWorld` interact with each other.

5.2 Syntax

This chapter explains how the YAGI language works on a syntactical level and how it is eventually compiled to executable Q3 bytecode.

5.2.1 Grammar

The grammar is given as EBNF (Extended Backus-Naur Form) and uses the following notation:

Notation	Meaning
`[ ]`	Optional (None or once)
`{ }`	Repeat 0 to n times
`< >`	Repeat 0 to n times separated by comma. Trailing comma allowed.
`,`	Sequence
`|`	Alternative
`abc`	Non-terminal
`" "`	Token
`/ /`	Regex

```
program         = { action | enum | fact | fluent | include
                  | proc |situation | signal | sense | ";"}
action          = "action", id, parameterList,
                  "{",
                  ["poss", (braces | block)],
                  ["do", block],
                  ["signal", id, argumentList],
                  "}"
enum            = "enum", id, "{", <id>, "}"
fact            = "fact", id, domain, set
fluent          = "fluent", "id", domain
include         = "include", string
proc            = "proc", id, parameterList, block
situation       = "situation", id, block
signal          = "signal", id, parameterList
sense           = "sense", type, id, parameterList
block           = "{", {if | while | choose | procCall | pick, (expr, ";")}, "}"
if              = "if", "(", expr, ")", block, ["else", block]
while           = "while", "(", expr, ")", block
choose          = "choose", block, "or", block, {"or", block}
procCall        = ["search"], id, argumentList
pick            = "pick", tupleAssign, "in", quantifierSet,
                  ["such", (block | quantifiers)]
expr            = (expr) | assign
assign          = [(("$", id) | id | tupleAssign),
                  ("=" | "+=" | "-=" | "*=" | "/=" | "%=" | "&=")],
                  quantifiers
quantifiers     = implies |
                  (("for", "exists"), tupleAssign,
                  "in", quantifierSet,
                  ["such", (block | quantifiers)])
implies         = or, ["->", implies]
or              = and, ["||", or]
and             = cmp, ["&&", and]
cmp             = addSub, [("==" | "!=" | "<" | "<=" | ">=" | ">"), cmp]
addSub          = multDiv, [("+" | "-"), addSub]
multDiv         = unary, [("*" | "/" | "%"), multDiv]
unary           = (("!" | "-" | intType), unary) | atom

atom            = set | tuple | braces | int | float
                | boolean | string | var | any
                | senseCall | setAccess | enumAccess
```

```
set             = "{", [domain, "|"], <expr>, "}"
tuple           = "[", <expr>, "]"
tupleAssign     = "[", <var | any>, "]"
braces          = "(", expr, ")"
boolean         = "true" | "false"
var             = "$", id
any             = "_"
senseCall       = "sense", id, argumentList
setAccess       = ["$"], id, "[", <expr>, "]"
enumAccess      = id, ".", id

parameterList   = "(", <parameter>, ")"
parameter       = type, "$", id
argumentList    = "(", <expr>, ")"
quantifierSet   = (domain | id | atom)

type            = intType | id | domain | setType
                | "float" | "string" | "boolean"
intType         = "int", ["[", [num], ":", [num], "]"]
domain          = "[", <type>, "]"
setType         = "{", type, "}"

id              = / [a-Z][A-Z,a-z,0-9,_]+ /
int             = / 0 | [1-9][0-9]* | -[1-9][0-9]* | 0x[0-9,A-F]+ | 0b[01]+ /
float           = / [-]?([0-9]+[.])?[0-9]+ /
string          = / \".*?\" /
```

Please note that a syntactically correct program isn't automatically semantically correct. For example,

```
action foo() {
  do {
    search bar({[float, string] | [true, $xyz]});
    [] = _;
  }
}
```

can be parsed just fine, though the typechecker will notice that procedures can't be called inside of actions and that types are mismatched.

5.2.2 Keywords

The following keywords are reserved and thus can't be used as identifiers:

```
proc, action, poss, do, fluent, enum, fact,
situation, ignureunused, signal, sense, include,
if, else, while, for choose, or, exists, pick, in, such, search,
true, false, int, float, boolean, string
```

5.2.3 Operators

These characters and combinations of characters are recognized by the tokenizer as valid operators:

```
, . : ? ! ; $ _ |
( ) [ ] { }
+ - * / %
-> || &&
= == != < <= >= >
+= -= *= /= %= &=
```

5.2.4 Reserved Identifiers

Some case-sensitive identifiers are already reserved by helper functions generated by the compiler and by library functions. While variables are fine, naming declarations like this may result in complications.

- `YagiSet`
- `Domain`
- `IntRange`
- `Situation`
- `List`
- `compileYagiFile`
- `copySituation`
- `fluentNames`
- `new`
- `senseNames`
- `signalNames`
- `signals`
- `startYagiSearch`
- `taskCount`

6 Implementation

6.1 Language Specification

The source code of a valid YAGI program is given as a set of one or more plaintext files, which are encoded in unicode and contain valid YAGI code.

6.1.1 Data Types

The Language supports the following data types which can be passed as variables:

- Integers with optional range bounds
- User defined enum types
- Booleans
- Floating point numbers
- Strings
- Tuples
- Sets

Integers
A signed 64-bit integer can be written in the usual decimal notation like `42` or `-3`. Hexadecimal, binary, octal notations are also supported and follow the same syntax as in the Java programming language. A hexadecimal number is preceded by `0x`, followed by at least one digit ranging from 0 to 9 and from a to f (case insensitive). A binary number, consisting of ones and zeroes, is preceded by `0b`. An octal number is preceded just by `0` followed by digits from 0 to 7. The range of integers lies between -2^{63} and $2^{63} - 1$, both inclusive.

The type of an integer is denoted by the keyword `int`. There also exist ranged integer types in order to write more consistent programs and save memory and runtime. `int[0:5]` is an integer between 0 and 5, also both inclusive. An integer can also only have either an upper or a lower bound, for example `int[:-1]` denotes a negative number and `int[1:]` a positive one.

Enums
Instead of describing objects with magic values or unorganized constants, YAGI offers enum types, which are declared in the global scope:

```
enum Color {
  RED,
  GREEN,
  BLUE,
}
```

As the above code shows, trailing commas are allowed in YAGI for convenience. An enum can have an arbitrary number of constants, although they must all have a unique name. A specific element in the above enum can for example be adressed with `Color.RED`.

Booleans
The type `boolean` has only two possible values, namely `true` and `false`. Booleans may show up as `true` and `null` in output logs, because `false` is internally represented as `null`.

Floating point numbers
64-bit precision floating numbers of the type `float` can only be denoted in the formats `1.23`, `-1.23`, `.123` and `-.123`. Other notations are not yet supported. Unlike for integers, there are no ranged types for floats.

Strings
Strings can be denoted with two apostrophes, for example `"Hello World!"` or `"ハロー・ワールド！"`. Escape sequences as specified in the Java language can be used. They have no use in the YAGI programming language other than printing debug output.

Tuples
A tuple is an immutable list of values of various types. `[true, 3, [Color.RED, Color.BLUE]]` for example is a tuple of the type `[boolean, int[3:3], [Color, Color]]`.

Sets
Only sets of tuples are supported, whereas the types of elements in the tuples are restricted to ranged integers, enums and booleans. `{[1, Color.GREEN], [4, Color.RED]}` is a set of the type `{[int[1:4], Color]}`. Usually, the type of the set can be inferred. If that isn't the case, the type of the set can be given at the beginning, followed by a horizontal line and the elements of the set: `{[int[0:5], Color] | [1, Color.GREEN], [4, Color.RED],}`

6.1.2 I/O

There exist two channels through which a YAGI program can interact with the outside world. Signals are used to tell the agent to execute a concrete action. They are either executed as they are called (online semantics) or are executed after a valid action sequence has been found after searching (offline semantics). Signals can only be called when executing an action. However one must first declare the name and parameters of a signal in the global scope. The variable names of the parameters only serve the purpose of documenting the code.

```
signal rotate(float $angle, Axis $axis)
```

The above code declares a signal and generates a member variable in the resulting YAGI world, where the wrapper script can attach a function that actually executes the signal. Signals have no return value, so one has to assume that they succeed.

The other communication channel is sensing, where the YAGI program can actively query a value from the world. Just like signals, sensing calls can only be made inside of actions. They must also be declared in the global scope, so a member variable can be created for the sensing function.

```
sense float measureRoomTemp(Room $location)
```

The above sensing declaration takes a `Room` as parameter and returns a floating point number.

The Chapter 6.1.3 describes in more detail how signals and sensing are used.

6.1.3 Domain Modeling

In order to model the world we want to reason in, we can use several building blocks known from situation calculus. Instead of modeling every single axiom (successor state axioms, situations), we can work on a much higher level and model fluent transitions imperatively.

Fluents
A fluent is basically a predicate that holds for the members of a mutable set of tuples. A fluent can be thought of as a member variable of the current situation. They are used to store the current *knowledge* of the world. For example, we can store where each object is located:

```
fluent objectAt[Object, Location]
```

As our fluent is a set of tuples and not a function that maps from `Object` to `Location`, the same object could theoretically be in two locations unless we forbid such behavior in the actions we're going to declare.

Facts
A fact behaves just like a fluent on a semantical level, except that it can't be changed at runtime within YAGI. Through the Q3 wrapper script or interactive console it's still possible to change its value however. Values are assigned to facts directly in their declaration.

```
fact edge[int[1:4], int[1:4]] {
  [1, _],
  [_, 1],
  [2, 3],
  [3, 4],
  [4, 2],
}
```

The wildcard `_` matches all integers between 1 and 4, so there is both an incoming and outgoing edge at node 1 from and to all other nodes.

Actions
An action is a sequence of statements used to manipulate fluents in a controlled fashion. As the action is appended to the current situation, the values of the fluents are altered. An action can accept parameters and may contain a precondition that must be true in order to execute the rest of the action. If the precondition is false, the program or the current search branch fails. If a precondition isn't necessary, it can be left out. Furthermore, a `do` block is required, where fluents are changed and afterwards a signal can be sent that actually executes the action in the real world. It's not mandatory for an action to have a signal though. Fluents can only be altered in the `do` block.

```
action dropObjectHeld(Hand $hand) {
  poss (objectsHeld[$hand, _]) # action precondition
  do { # action effect
    objectsHeld -= [$hand, _];
  }
  signal openHand($hand)
}
```

In order to be able to drop an object, one must hold any object. If this isn't the case, executing the program any further makes no sense. Then we remove the object we're holding in said hand from the fluent value. Finally, the signal to open the hand is sent.

It is allowed to have a block of statements as a precondition, so variables declared in the precondition are accessible afterwards. The code below has the same functionality as the code above:

```
action dropObjectHeld(Hand $hand) {
  poss {
    pick [$hand, $obj] in objectsHeld;
  }
  do {
    objectsHeld -= [$hand, $obj];
  }
  signal openHand($hand)
}
```

Instead of checking if any object is held, we find out the concrete object in the specified hand and don't use wildcards. As obvious, the first code sample is both more elegant and efficient.

Sensing in Actions
In order for the agent to expand and update its knowledge of the world, it has to call actions that sense a value and store it in the fluents. A call to a sensing function is preceded by the `sense` keyword. For example

```
action measureCurrentTemp() {
  poss {
    pick [$room] in at;
  }
  do {
    temp -= [$room, _];
    $measuredTemp = sense measureRoomTemp();
    temp += [$room, $measuredTemp];
  }
}
```

can be called to measure the temperature in the current room and store it in the designated fluent.

Situations
The last step of domain modeling is to specify what the initial situation looks like. YAGI allows the user to declare as many initial situations as needed. Besides changing the values of fluents, no other side effects are allowed. Following conventions, the initial situation is named `S0`:

```
situation S0 {
  objectsHeld = {[Hand.LEFT, Object.KNIFE], [Hand.RIGHT, Object.FORK]};
  objectAt += [Object.PLATE, Location.TABLE];
  objectAt += [Object.GLASS, Location.TABLE];
  objectAt += [Object.PAN, Location.KITCHEN];
}
```

6.1.4 Variables

Local variables are denoted by `$` followed by their name. The type of local variables can usually be inferred by the typechecker. Parameters of declarations however need to be preceded by their type. Facts and fluents can be referred to without a preceding `$` as they are not variables, but rather entities in the global scope.

6.1.5 Operators

YAGI supports the following operators, sorted by precedence, beginning with the weakest:

- `= += -= *= /= &= |=` Assignment operators:
  - `+=` is defined for numbers, strings and sets/fluents. To the latter it adds another set or a single tuple (union).
  - `-=` is defined for numbers and sets/fluents. From the latter it removes another set or a single tuple (difference).
  - `*= /=` are defined for numbers. For example, the program `$num *= 2;` behaves just like `$num = $num * 2;`. The same goes for the operators `+= -= /=`.
  - `&=` is defined for booleans and for sets/fluents. For the latter it intersects the two sets and stores the result in the variable on the left handside.

  - `|=` is defined for booleans. The program `$bool1 |= $bool2;` behaves just like `$bool1 = $bool1 || $bool2;`. The same goes for the operator `&=`.

- `exists for` See the following chapter.
- `->` Logic implication, defined for booleans. Short circuit evaluation is applied for this operator, `||` and `&&`.
- `||` Logic OR, defined for booleans.
- `&&` Logic AND, defined for booleans.
- `== != < <= >= >` Comparison operators. The latter four are defined for numbers.
- `+ -` Addition and subtraction are defined for numbers.
- `* / \%` Multiplication, division and modulo are also defined for numbers.
- `! - int` Unary operators:
  - `!` Boolean negation. Also defined for sets, where the result of the operation is the complement of the set regarding its designated domain.
  - `-` Arithmetic negation. Defined for numbers.
  - `int` Static integer cast. Used to ensure the correct range of an integer or to convert a float to an integer. If a value lies outside its designated range, the result is the bound closest to the original value. For example, `int[1:10] 69` evaluates to 10.

Subscript operator
The subscript operator `[ ]` is preceded by a fluent or a variable that holds a set. The operator evaluates to true if the set contains at least one tuple with the specified values. `objectAt[_, Location.KITCHEN]` is true if there is at least one object in the kitchen. The subscript operator is syntactic sugar for a trivial case of the exists quantifier, namely `exists [_, Location.KITCHEN] in objectAt`.

Tuple Assignment
In order to extract values from a tuple and assign them to multiple variables at once, tuple assignment does the trick. A similar feature exists in Python or in PHP, where it is known as array unpacking.

```
$myTuple = [true, Room.KITCHEN, 3];
[$success, $room, _] = $myTuple;
```

The wildcard discards the value at the corresponding index. Tuple assignment could also be used to swap values of variables within one line:

```
[$x, $y] = [$y, z];
```

6.1.6 Control Structures

Blocks
A block is denoted by curly brackets and contains an arbitrary number of statements, each followed by a semicolon. Semantically all statements are linked with $\wedge$ and `true` is the neutral element regarding the operation $\wedge$. Thus an empty block evaluates to `true` and a non-empty block is `true` if all of its statements are `true`. As soon as a statement is `false`, no more statements are evaluated and `false` is returned, which causes the program, search branch or condition to fail.

```
{
  $set = {[1], [2], [4]};
  $set[3]; # Test if 3 is contained in $set.
  $set += {[3]}; # This line is never reached because the above statement fails.
}
```

Variables declared inside a block aren't accessible outside of it.

Conditionals

If-conditionals are expressed in the usual way:

```
if (objectsHeld[_, Object.DINNER] && at[Location.TABLE]) {
  putTo(Object.DINNER, Location.TABLE);
}
```

else branches and if-else branches are also possible:

```
$grade;
if ($score < 50) {
  $grade = 5;
} else if ($score < 62.5) {
  $grade = 4;
} else if ($score < 75) {
  $grade = 3;
} else if ($score < 87.5) {
  $grade = 2;
} else {
  $grade = 1;
}
```

Please note that conditionals always require blocks in order to promote clean coding. No side effects such as action calls or variable assignments are allowed in the condition. The same is true for while loops.

While Loops

A while loop executes its subsequent block as long as its condition is true.

```
while(requests[_, _]) {
  pick [$object, $destination] from requests;
  serveRequest($object, $destination);
}
```

The code above serves requests until there are no more to serve.

Quantifiers and Formulas

There exist two quantifiers, namely `exists` and `for`, while the latter can be both used as a forall quantifier or a foreach loop. The basic syntax for quantifiers is the same. First comes the name of the quantifier, then a tuple with the variables to be bound, the keyword `in` followed by an expression evaluating to a set and an optional `such` followed by a boolean condition. The set expression can either be a variable, a fluent name or a domain.

```
exists [$obj, _] in objectAt such $obj == Object.PLATE || $obj == Object.GLASS;
```

If an element in the tuple after `exists` is an already declared variable or is an expression, the searched elements are filtered by its value. If a variable isn't already declared, it will be freshly bound and is accessible in the condition. The exists quantifier is true if at least one tuple exists in the specified set where the tuple matches and the condition evaluates to true. The condition of `exists` statements permits no side effects. One can not only quantify over fluents, but also over sets and domains. The syntax for the latter is the following:

```
exists [$x, $y] in [int[0:5], int[0:5]] such $x * $y == $x + $y;
```

The code above quantifies over all 36 (6×6) possible combinations of integers until a matching tuple is found.

The `for` quantifier has a similar syntax:

```
for [$obj, Location.DISHWASHER] in objectAt such isDirty[$obj];
```

The above statement is true if all objects in the dishwasher are dirty.

Blocks after `such` are allowed in `for` quantifiers. A block evaluates to true if each statement in the block is true. This way the code above could also be expressed this way:

```
for [$obj, Location.DISHWASHER] in objectAt such {
  isDirty[$obj];
};
```

It is important not to forget the semicolon, because `for` is a quantifier and thus an expression!

The `for` quantifier permits side effects in the condition in contrast to exists and pick. This way it can be abused as a classical foreach loop:

```
$sum = 0;
for [$x] in [int[1:100]] such {
  $sum += $x;
}
```

The code above calculates the sum of all integers between 1 and 100, each inclusive. Note that the order of the iteration is non-deterministic, meaning that the elements in the specified set, domain or fluent may be enumerated in an arbitrary order. To enumerate integers in an ordered way, use a while loop. It may be noted that the typechecker doesn't get the bounds of integers right in loops because predicting the number of iterations is equal to solving the halting problem.

Pick

It is often necessary to bind values from sets to variables for later use. A pick statement has a syntax similar to quantifiers and instead of the variable scope being limited to the condition, they are still visible afterwards. If no matching tuple is found in the set, the pick statement and thus the program or current search branch fail. Outside of search mode, one matching tuple is picked and the rest of the program either succeeds or fails. During search mode, all possible outcomes of the pick statement are executed concurrently and one succeeding execution path is chosen as search result.

```
pick [$i] in [int[1:100]]; # The result is bound to $i
walkSteps($i); # Walk a random amount of steps
```

`pick` can also be used in combination with `such` in order to limit the possibilities of results one can get:

```
pick [$i] in [int[2:100]] such $x % 2 == 0; # Pick an even number between 2 and 100.
```

Just like with exists and for, the scope of values searched can be constrained:

```
$leftHand = Hand.LEFT;
pick [$leftHand, $objectHeld] in objectsHeld;
```

It is advised to use `pick` in combination with `such` as much as possible to make the code more efficient while in search mode.

Choose

A choose construct starts with the keyword `choose` and a block of statements. An arbitrary number of blocks, each preceded by `or` can follow. The interpreter executes one of those blocks non-deterministically. When outside search mode, one of the blocks is picked with equal chance and is then executed. During search mode, all blocks are executed concurrently and one succeeding block is chosen as outcome. A choose construct must at least contain 2 blocks and can only be called inside of procedures.

```
choose {
  pick [$loc] in [Location];
  walkTo($loc);
} or {
  pick [_, $object] in objectHeld;
  drop($object);
} or {
  pick [_, $object] in objectAt;
  pickup($object);
}
```

6.1.7 Procedures

Procedures are the heart of Golog and YAGI. They contain the actual "business logic", which are instructions and statements that define the behavior of the program. A procedure can call other procedures or itself, execute actions and plan, but can't modify fluents directly. Only actions can change fluents. A procedure can have parameters, but unlike a classical function it doesn't have a return value. It either succeeds or fails.

```
proc findPath(Location $to) {
  while (!at[$to]) {
    pick [$next] in [Location];
    move($next);
  }
}
```

The code above will very likely fail outside of search mode. As long as the agent isn't at the desired position, it tries to guess the next position it can move to. If the precondition of the action `move` is false - which is the case if there is no connection between the current position and the random position - the whole procedure will fail.

6.1.8 Searching and Planning

The core purpose of YAGI is to enable planning, which means that in order to find a suitable sequence of actions, all possible - or reasonable - sequences of actions are tried until a result is found. If a procedure call is preceded by the `search` keyword, the called procedure and all its nested procedure calls are executed in search mode. If a pick statement or a choose construct are encountered in search mode, the control flow forks each time and executes each outcome concurrently. The first execution path that reaches the end of the searched procedure will actually be executed. If every path fails before reaching the end of the procedure, the search itself fails.

```
search findPath(Location.KITCHEN);
```

The code above tries to find and execute one valid path to the kitchen.

Let's say we have the following code that is executed in search mode:

```
doThis();
findPath(Location.TABLE);
doThat();
```

At first it seems to work, because all nested procedure calls are executed in search mode too. However this code is grossly inefficient, because findPath outputs not only one plausible path, but all combinations that are then forwarded to `doThat();`. We know that choosing a slightly different path won't affect the overall outcome. What we want is to precede `findPath` with a `search` keyword, so only at most one path is returned.

Signals of actions are buffered while searching and are executed just after the uppermost search call has been left. It is not possible to sense input while in search mode.

Figure 2 illustrates how the search result is rather optimal regarding the number of decisions made than regarding the number of actions taken. At first the procedure `something()` is called in online mode with a preceding `search` keyword, creating a new search tree. With each decision being made, the search tree

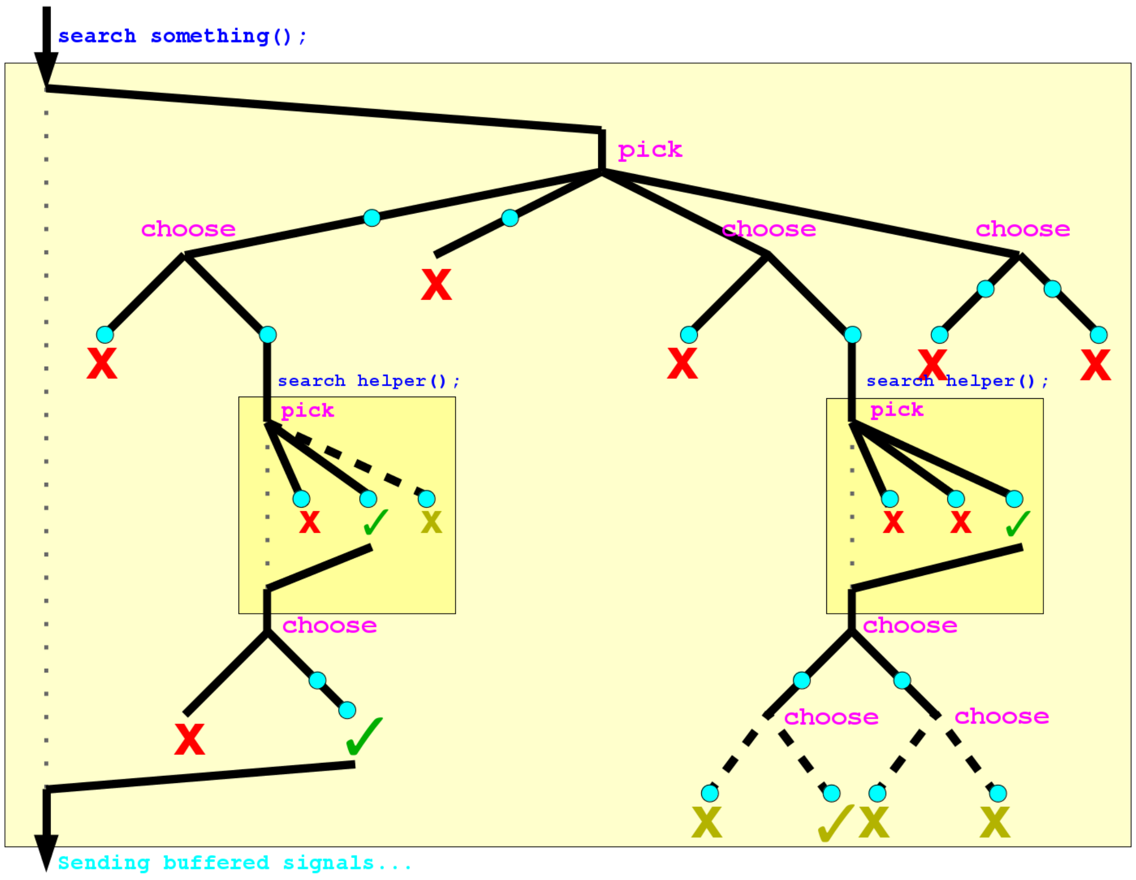


Figure 2: A visualization of a hypothetical search tree, where each decision is annotated with pink text and each action is marked with a cyan dot.

grows by one level, while actions don't influence the vertical structure of the search tree. Search branches are cut off by failing action preconditions or failing test actions. The figure also shows that nested search calls only yield one valid path instead of handing each solution to the outer search tree. As soon as a branch finishes evaluating the procedure `something`, a solution has been found and all remaining search branches are terminated. Those are marked in dark yellow. Even though the search branch on the right could yield a solution with less actions than the search branch on the left, one more decision is required and thus the left branch is chosen.

6.1.9 Miscellaneous Compiler Features

The compiler for YAGI offers some features that make organizing the code easier.

6.1.10 Including Other Files

As a project grows, it becomes necessary to split it into separate files. The keyword `include` followed by the path to the included file relative to the current file includes all declarations contained in the specified file. Declarations are included in a flat manner, meaning that if A includes B and C, then B has access to all elements contained in C and vice versa. Includes can also be nested. Example:

```
include "bat.yagi"
include "signals.yagi"
```

The root file, which includes all other files, must be passed to the compiler as argument.

6.1.11 Ignore Warnings for Unused Declarations

It can be helpful when the compiler spots declarations that aren't used anywhere in the program. However the compiler doesn't know which procedures and functionality the wrapper program accesses. In order to suppress those annoying warnings, the keyword `ignoreunused` can be used followed by an enumeration of declaration names to ignore:

```
ignoreunused (main, S0)
```

Any name can go in there - even if the declaration doesn't exist at all or is in another file.

6.2 Specification Conformance of YAGI

In this chapter we discuss how the language features of YAGI relate to Situation Calculus and Golog and why our implementation behaves the defined way. To accomplish this, we formalize the complete feature set of YAGI using the function $YagiEval$ and the predicates $YagiFormula$, $YagiTrans$ and $YagiFinal$. We treat a situation as a fluent collection c, where the state of each fluent is stored explicitly instead of saving the action history.

6.2.1 Expressions

For evaluating expressions in YAGI we introduce the function $YagiEval(\xi)$, which interprets the given expression ξ and gives its value. Constants evaluate to their designated value with no computation being performed:

$$\begin{aligned}
YagiEval(\boldsymbol{true}) &= \top \\
YagiEval(\boldsymbol{false}) &= \bot \\
YagiEval(\boldsymbol{n}) &= n,\ n \in \mathbb{N} \\
YagiEval(\boldsymbol{r}) &= r,\ r \in \mathbb{R} \\
YagiEval(\boldsymbol{s}) &= s,\ s \in \mathit{String} \\
YagiEval(\boldsymbol{-\xi}) &= -YagiEval(\xi),\ \xi \in \mathbb{R} \\
YagiEval(\boldsymbol{\xi_1 + \xi_2}) &= YagiEval(\xi_1) + YagiEval(\xi_2),\ \xi_1 \in \mathbb{R} \wedge \xi_2 \in \mathbb{R} \\
YagiEval(\boldsymbol{\xi_1 + \xi_2}) &= YagiEval(\xi_1) \parallel YagiEval(\xi_2),\ \xi_1 \in \mathit{String} \vee \xi_2 \in \mathit{String} \\
YagiEval(\boldsymbol{\xi_1 - \xi_2}) &= YagiEval(\xi_1) - YagiEval(\xi_2),\ \xi_1 \in \mathbb{R} \wedge \xi_2 \in \mathbb{R} \\
YagiEval(\boldsymbol{\xi_1 * \xi_2}) &= YagiEval(\xi_1) \cdot YagiEval(\xi_2),\ \xi_1 \in \mathbb{R} \wedge \xi_2 \in \mathbb{R} \\
YagiEval(\boldsymbol{\xi_1\ /\ \xi_2}) &= YagiEval(\xi_1) \div YagiEval(\xi_2),\ \neg(\xi_1 \in \mathbb{N} \wedge \xi_2 \in \mathbb{N}) \\
YagiEval(\boldsymbol{\xi_1\ /\ \xi_2}) &= YagiEval(\xi_1)\ \mathit{div}\ YagiEval(\xi_2),\ \xi_1 \in \mathbb{N} \wedge \xi_2 \in \mathbb{N} \\
YagiEval(\boldsymbol{\xi_1\ \%\ \xi_2}) &= YagiEval(\xi_1)\ \mathit{mod}\ YagiEval(\xi_2),\ \xi_1 \in \mathbb{R} \wedge \xi_2 \in \mathbb{R} \\
YagiEval(\boldsymbol{int[lower : upper]\ \xi}) &= \mathit{max}(\mathit{lower}, \mathit{min}(\mathit{upper}, YagiEval(\xi))) \\
YagiEval(\boldsymbol{[\overrightarrow{\xi}]}) &= \langle YagiEval(\overrightarrow{\xi}) \rangle
\end{aligned}$$

The operator $\parallel$ stands for string concatenation. Interpretation of tuples is defined in the last line, meaning that $YagiEval(\mathbf{[1+2,3]}) = \langle YagiEval(\mathbf{1+2}), YagiEval(\mathbf{3}) \rangle = \langle 3, 3 \rangle$.

6.2.2 Formulas

Similar to $YagiEval$, the predicate $YagiFormula(\phi, c)$ is true if and only the given formula ϕ holds for the fluent collection c. All operators and constructs, which evaluate to booleans, are defined via this predicate:

$$\begin{aligned}
YagiFormula(\boldsymbol{!\phi}, c) &\equiv \neg YagiFormula(\boldsymbol{\phi}, c)\\
YagiFormula(\boldsymbol{\phi_1} \textbf{ \&\& } \boldsymbol{\phi_2}, c) &\equiv YagiFormula(\boldsymbol{\phi_1}, c) \wedge YagiFormula(\boldsymbol{\phi_2}, c)\\
YagiFormula(\boldsymbol{\phi_1} \textbf{ || } \boldsymbol{\phi_2}, c) &\equiv YagiFormula(\boldsymbol{\phi_1}, c) \vee YagiFormula(\boldsymbol{\phi_2}, c)\\
YagiFormula(\boldsymbol{\phi_1} \textbf{ -> } \boldsymbol{\phi_2}, c) &\equiv YagiFormula(\boldsymbol{\phi_1}, c) \rightarrow YagiFormula(\boldsymbol{\phi_2}, c)\\
YagiFormula(\textbf{\textit{exists} } [\overrightarrow{v}] \textbf{\textit{ in F such }} \boldsymbol{\phi}, c) &\equiv \exists \overrightarrow{v} : \overrightarrow{v} \in F \wedge\ YagiFormula(\phi[\overrightarrow{v}], c)\\
YagiFormula(\textbf{\textit{for} } [\overrightarrow{v}] \textbf{\textit{ in F such }} \boldsymbol{\phi}, c) &\equiv \forall \overrightarrow{v} : \overrightarrow{v} \in F \wedge\ YagiFormula(\phi[\overrightarrow{v}], c)\\
YagiFormula(\boldsymbol{F}[\overrightarrow{\xi}], c) &\equiv YagiEval(\overrightarrow{\xi}) \in F\\
YagiFormula(\boldsymbol{\xi_1} \textbf{ == } \boldsymbol{\xi_2}, c) &\equiv YagiEval(\boldsymbol{\xi_1}) = YagiEval(\boldsymbol{\xi_2})\\
YagiFormula(\boldsymbol{\xi_1} \textbf{ != } \boldsymbol{\xi_2}, c) &\equiv YagiEval(\boldsymbol{\xi_1}) \neq YagiEval(\boldsymbol{\xi_2})\\
YagiFormula(\boldsymbol{\xi_1} \textbf{ > } \boldsymbol{\xi_2}, c) &\equiv YagiEval(\boldsymbol{\xi_1}) > YagiEval(\boldsymbol{\xi_2})\\
YagiFormula(\boldsymbol{\xi_1} \textbf{ >= } \boldsymbol{\xi_2}, c) &\equiv YagiEval(\boldsymbol{\xi_1}) \geq YagiEval(\boldsymbol{\xi_2})\\
YagiFormula(\boldsymbol{\xi_1} \textbf{ <= } \boldsymbol{\xi_2}, c) &\equiv YagiEval(\boldsymbol{\xi_1}) \leq YagiEval(\boldsymbol{\xi_2})\\
YagiFormula(\boldsymbol{\xi_1} \textbf{ < } \boldsymbol{\xi_2}, c) &\equiv YagiEval(\boldsymbol{\xi_1}) < YagiEval(\boldsymbol{\xi_2})\\
YagiFormula(\top, c) &\equiv \top\\
YagiFormula(\bot, c) &\equiv \bot
\end{aligned}$$

The last case of $YagiFormula$ is responsible for handling expressions which evaluate to boolean values by default, such as boolean literals or variable accesses.

6.2.3 Fluents

The current situation is represented by a fluent collection c, which consists of a fixed arbitrary number of fluents $\overrightarrow{F}$ depending on the number of fluents declared in the YAGI program. Each fluent F has a finite domain S_F^n, which can be an n-ary tuple of countable data types, such as ranged integers, enums or booleans. The function $translate(x, S_F^i)$ takes a value and its data type and evaluates to an offset between 0 and $|S_F^i| - 1$. For wildcards _ it evaluates to -1. Wildcards are not allowed for booleans, as explained in Chapter 6.3.

$$\begin{aligned}
translate(false, boolean) &= 0\\
translate(true, boolean) &= 1\\
translate(\_, int[x:y]) &= -1\\
translate(i, int[x:y]) &= i - x, x \leq i \leq y\\
translate(i, int[x:y]) &= \bot, i < x \vee i > y\\
translate(\_, E) &= -1\\
translate(e, E) &= ordinal(e), e \in E\\
translate(e, E) &= \bot, e \notin E
\end{aligned}$$

The function $ordinal(e)$ takes a member e of the enum E and evaluates to its ordinal, i.e. the position it was declared on in the enum declaration, starting with 0. There also exists an inverse function to $translate$ in the code, which is used when iterating over fluents.

Fluents and sets are internally represented as a bitmap B_F^c held by the fluent collection c. Each tuple of values inside S_F^n maps to a position in the bitmap, pointing to a bit $B_F^c(i)$. The length of the bitmap is the product of the lengths of each domain type. Formally speaking, it is $|B| = \prod_{i=1}^{n} |S_F^i|$. The offset of a value tuple $\overrightarrow{x}^n$ inside the bitmap can be formalized tail-recursively:

$$\begin{aligned}
\mathit{offset}(\overrightarrow{x}, S_F^n) &= \mathit{offset0}(\overrightarrow{x}, S_F^n, 0, |S_F^n|) \\
\mathit{offset0}(\overrightarrow{x}, S_F^n, i, 0) &= i \\
\mathit{offset0}(\overrightarrow{x}, S_F^n, i, j) &= \mathit{offset0}(\overrightarrow{x}, S_F^n, i \cdot |S_F^j| + \mathit{translate}(x_j), j-1)
\end{aligned}$$

Wildcards are treated by considering all possible values at their position.

Now we can put everything together to define when a fluent F with the domain S_F^n, represented by the bitmap B_F^c holds for a given value tuple:

$$\begin{aligned}
F(\overrightarrow{x}, c) &\Leftrightarrow B_F^c(\mathit{offset}(\overrightarrow{x}, S_F^n)) = 1 \\
\neg F(\overrightarrow{x}, c) &\Leftrightarrow B_F^c(\mathit{offset}(\overrightarrow{x}, S_F^n)) = 0
\end{aligned}$$

6.2.4 Actions

Because YAGI actions are rich in complex components such as conditionals, while loops, foreach loops and deterministic pick statements, we will proceed to map YAGI actions to procedures and encode fluent modifications into primitive actions. A YAGI action A consists of a precondition $\varphi_{AP}(A)$ and the action effects δ_A. We introduce the function $ActionToProc(\delta_A)$ to convert the action effects to a procedure. An action call $A(\overrightarrow{args})$ in a prodedure is subsequently mapped to the program $\varphi_{AP}(A)$; $ActionToProc(\delta_A)[\overrightarrow{args}]$;. Fluent modifications are delegated to the primitive actions $add_F(\overrightarrow{x})$ and $remove_F(\overrightarrow{x})$.

$$\begin{aligned}
ActionToProc(\boldsymbol{\phi}) &= \boldsymbol{\phi} \\
ActionToProc(\boldsymbol{\alpha_1;\ \alpha_2;}) &= ActionToProc(\alpha_1);\ ActionToProc(\alpha_2); \\
ActionToProc(\boldsymbol{if(\phi)\ \{\ \alpha;\ \}}) &= if(\phi)\ \{\ ActionToProc(\alpha);\ \} \\
ActionToProc(\boldsymbol{if(\phi)\ \{\ \alpha_1;\ \}\ else\ \{\ \alpha_2;\}}) &= if(\phi)\ \{\ ActionToProc(\alpha_1);\ \} \\
&\quad else\ \{\ ActionToProc(\alpha_2);\} \\
ActionToProc(\boldsymbol{while(\phi)\ \{\ \alpha;\ \}}) &= while(\phi)\ \{\ ActionToProc(\alpha);\ \} \\
ActionToProc(\boldsymbol{pick\ [\overrightarrow{v}]\ in\ F\ such\ \phi;\ \alpha;}) &= pick\ [\overrightarrow{v}]\ in\ F\ such\ \phi;\ ActionToProc(\alpha); \\
ActionToProc(\boldsymbol{for\ [\overrightarrow{v}]\ in\ F\ such\ \phi;\ \{\ \alpha;\ \}}) &= for\ [\overrightarrow{v}]\ in\ F\ such\ \phi;\ \{\ ActionToProc(\alpha);\ \}; \\
ActionToProc(\boldsymbol{F\ =\ [\overrightarrow{x}];}) &= for\ [\overrightarrow{x'}]\ in\ F\ such\ \{\ remove_F(\overrightarrow{x'});\ \};\ add_F(\overrightarrow{x}); \\
ActionToProc(\boldsymbol{F\ +=\ [\overrightarrow{x}];}) &= add_F(\overrightarrow{x}); \\
ActionToProc(\boldsymbol{F\ -=\ [\overrightarrow{x}];}) &= remove_F(\overrightarrow{x}); \\
ActionToProc(\boldsymbol{F\ +=\ S;}) &= for\ [\overrightarrow{x}]\ in\ S\ such\ \{\ add_F(\overrightarrow{x});\ \} \\
ActionToProc(\boldsymbol{F\ -=\ S;}) &= for\ [\overrightarrow{x}]\ in\ S\ such\ \{\ remove_F(\overrightarrow{x});\ \} \\
ActionToProc(\boldsymbol{F\ =\ S;}) &= for\ [\overrightarrow{x}]\ in\ F\ such\ \{\ remove_F(\overrightarrow{x});\ \}; \\
&\quad ActionToProc(F\ +=\ S);
\end{aligned}$$

The effects of the actions $add_F(\overrightarrow{x})$ and $remove_F(\overrightarrow{x})$ can be easily modelled as Successor State Axioms, as [Maier, 2015] describes. The action add_F sets the fluent F to true for a single value tuple, while $remove_F$ sets it to false.

$$\begin{aligned}
F(\overrightarrow{x}, do(a,s)) &\equiv\ a = add_F(\overrightarrow{x}) \vee F(\overrightarrow{x}, s) \wedge a \neq remove_F(\overrightarrow{x}) \\
Poss(add_F(\overrightarrow{x}), s) &\equiv \top \\
Poss(remove_F(\overrightarrow{x}), s) &\equiv \top
\end{aligned}$$

But because we use fluent collections and not situations in YAGI, we need to specify the effects of add_F and $remove_F$ for bitmaps. The function $exec$ applies the action's effects to the fluent collection c and gives an updated fluent collection c':

$$\begin{aligned} exec(add_F(\overrightarrow{x}), c) &= c \cup B_F^c(offset(\overrightarrow{x}, S_F^n) \leftarrow 1) \setminus B_F^c \\ exec(remove_F(\overrightarrow{x}), c) &= c \cup B_F^c(offset(\overrightarrow{x}, S_F^n) \leftarrow 0) \setminus B_F^c \end{aligned}$$

To put it in words, the function *exec* updates the bitmap B_F^c by setting a bit depending on the primitive action and the value tuple $\overrightarrow{x}$.

6.2.5 Control Flow

We have already outlined the semantics of IndiGolog in Chapter 3.3, which are based on the predicates $Trans$ and $Final$. Analougously [Maier, 2015] has introduced the predicates $YagiTrans(\alpha, b, \alpha', b')$ and $YagiFinal(\alpha, b)$, which take a Legacy YAGI program α and a fluent database b. They largely correspond to the behavior of the two predicates in IndiGolog. For our new YAGI syntax, we need to slightly adapt those predicates and add new rules for novelty features. As situations are stored differently in Legacy YAGI, we use fluent collections instead of databases in our new definitions.

- An empty program can't transition into a new program, but is allowed to terminate:

$$\begin{aligned} YagiTrans(\boldsymbol{\epsilon}, \_, \_, \_) &\equiv \bot \\ YagiFinal(\boldsymbol{\epsilon}, \_) &\equiv \top \end{aligned}$$

- Primitive actions (add_F and $remove_F$) are always possible and therefore there's no need to check their precondition. We use the function *exec* to update the fluent collection:

$$\begin{aligned} YagiTrans(\boldsymbol{a(\overrightarrow{args});}, c, \epsilon, c') &\equiv c' = exec(a(\overrightarrow{args}), c) \\ YagiFinal(\boldsymbol{a(\overrightarrow{args});}, \_) &\equiv \bot \end{aligned}$$

- Test action:

$$\begin{aligned} YagiTrans(\boldsymbol{\phi;}, c, \epsilon, c) &\equiv YagiFormula(\phi, c) \\ YagiFinal(\boldsymbol{\phi;}, \_) &\equiv \bot \end{aligned}$$

- Sequential execution of statements:

$$\begin{aligned} YagiTrans(\boldsymbol{\alpha_1;\ \alpha_2;}, c, \alpha', c') \equiv\ & (\exists \gamma : \alpha' = (\gamma; \alpha_2) \land YagiTrans(\alpha_1, c, \gamma, c')) \\ & \lor YagiFinal(\alpha_1, c) \land YagiTrans(\alpha_2, c, \alpha', c') \\ YagiFinal(\boldsymbol{\alpha_1;\ \alpha_2;}, c) \equiv\ & YagiFinal(\alpha_1, c) \land YagiFinal(\alpha_2, c) \end{aligned}$$

- Nondeterministic branching:

$$\begin{aligned} YagiTrans(\boldsymbol{choose\ \{\alpha_1;\}\ or\ \{\alpha_2;\}}, c, \alpha', c') &\equiv YagiTrans(\alpha_1, c, \alpha', c') \lor YagiTrans(\alpha_2, c, \alpha', c') \\ YagiFinal(\boldsymbol{choose\ \{\alpha_1;\}\ or\ \{\alpha_2;\}}, c) &\equiv YagiFinal(\alpha_1, c) \lor YagiFinal(\alpha_2, c) \end{aligned}$$

- Nondeterministic choice of arguments can come with a filtering condition ϕ, so that `pick` delivers better results in online execution mode:

$$\begin{aligned} YagiTrans(\boldsymbol{pick\ [\overrightarrow{v}]\ in\ F;\ \alpha;}, c, \alpha', c') \equiv\ & \exists \overrightarrow{v} : \overrightarrow{v} \in F \land YagiTrans(\alpha[\overrightarrow{v}, c], c, \alpha', c') \\ YagiFinal(\boldsymbol{pick\ [\overrightarrow{v}]\ in\ F;\ \alpha;}, c) \equiv\ & \exists \overrightarrow{v} : \overrightarrow{v} \in F \land YagiFinal(\alpha[\overrightarrow{v}, c], c, \alpha', c') \\ YagiTrans(\boldsymbol{pick\ [\overrightarrow{v}]\ in\ F\ such\ \phi;\ \alpha;}, c, \alpha', c') \equiv\ & \exists \overrightarrow{v} : \overrightarrow{v} \in F \land YagiFormula(\phi[\overrightarrow{v}], c) \\ & \land YagiTrans(\alpha[\overrightarrow{v}, c], c, \alpha', c') \\ YagiFinal(\boldsymbol{pick\ [\overrightarrow{v}]\ in\ F\ such\ \phi;\ \alpha;}, c) \equiv\ & \exists \overrightarrow{v} : \overrightarrow{v} \in F \land YagiFormula(\phi[\overrightarrow{v}], c) \\ & \land YagiFinal(\alpha[\overrightarrow{v}, c], c, \alpha', c') \end{aligned}$$

- Conditionals:

$$\begin{aligned}
YagiTrans(\boldsymbol{if(\phi)\ \{\alpha;\}}, c, \alpha', c') \equiv\ & YagiFormula(\phi, c) \wedge YagiTrans(\alpha, c, \alpha', c') \\
& \vee \neg YagiFormula(\phi, c) \wedge c = c' \wedge \alpha' = \epsilon \\
YagiFinal(\boldsymbol{if(\phi)\ \{\alpha;\}}, c) \equiv\ & YagiFormula(\phi, c) \rightarrow YagiFinal(\alpha, c) \\
YagiTrans(\boldsymbol{if(\phi)\ \{\alpha_1;\}\ else\ \{\alpha_2;\}}, c, \alpha', c') \equiv\ & YagiFormula(\phi, c) \wedge YagiTrans(\alpha_1, c, \alpha', c') \\
& \vee \neg YagiFormula(\phi, c) \wedge YagiTrans(\alpha_2, c, \alpha', c') \\
YagiFinal(\boldsymbol{if(\phi)\ \{\alpha_1;\}\ else\ \{\alpha_2;\}}, c) \equiv\ & YagiFormula(\phi, c) \wedge YagiFinal(\alpha_1, c) \\
& \vee \neg YagiFormula(\phi, c) \wedge YagiFinal(\alpha_2, c)
\end{aligned}$$

- While loops:

$$\begin{aligned}
YagiTrans(\boldsymbol{while(\phi)\ \{\alpha;\}}, c, \alpha', c') \equiv\ & (\exists \gamma : \alpha' = (\gamma;\ while(\phi)\ \{\alpha;\}) \wedge YagiTrans(\alpha, c, \gamma, c')) \\
YagiFinal(\boldsymbol{while(\phi)\ \{\alpha;\}}, c) \equiv\ & \neg YagiFormula(\phi, c) \wedge YagiFinal(\alpha, c)
\end{aligned}$$

6.2.6 Foreach loops

Foreach loops with side effects such as assignments or action calls in their body must be mapped to while loops on a semantical level. The set to be iterated over is stored in a variable and a while loop iterates until no more elements are left in the set. Each iteration a tuple is picked, the body of the foreach loop gets executed and the picked tuple gets removed from the set. To illustrate this more clearly, the expression

```
for [$var1, $var2] in S such { α; }
```

gets translated to:

```
$set = S;
while($set[_, _]) {
  pick [$var1, $var2] in $set;
  α;
  $set -= [$var1, $var2];
}
```

6.2.7 Assignment

In contrast to Legacy YAGI it is possible to assign and reassign values to local variables. Variable accesses can be formalized by rewriting the syntax representation of a variable access `$var` to `$var[x]`, where x is the value currently assigned to the variable. So for a variable access $YagiEval$ can be defined as:

$$YagiEval(\boldsymbol{\$var[x]}) \equiv\ x$$

The assignment operator `=` can be defined using transition semantics. If the right hand side of the assignment evaluates to x', all references of the variable $\$var[x]$ are replaced with $\$var[x']$ in the remaining program:

$$\begin{aligned}
YagiTrans(\boldsymbol{\$var\ =\ \xi;\ \delta;}, c, \delta', c) \equiv\ & YagiTrans(\delta[var|YagiEval(\xi)], c, \delta', c') \\
YagiFinal(\boldsymbol{\$var\ =\ \xi;\ \delta;}, c) \equiv\ & YagiFinal(\delta[var|YagiEval(\xi)], c)
\end{aligned}$$

6.2.8 Procedures

As discussed in Chapter 3.2.10, traditional Golog uses marco-expansion to handle procedure calls, which disallows recursive calls. In both YAGI and Legacy YAGI, procedures are executed the same way as in procedural programming languages. Before entering the procedure, the argument expressions are evaluated and their values, including the current fluent collection, are passed to a new environment. The called procedure is then executed in this environment and the resulting fluent collection is then passed up to the calling procedure.

6.3 Implementation Details

In this chapter we will explain the components of the YAGI Library, how a YAGI program is compiled and typechecked. Furthermore we will give instructions on how the controller can be set up and we will describe how a YAGI program is executed at runtime.

6.3.1 Compiler Workflow

At first we start with a YAGI program given as one or more text files. The Q3 function `compileYagiFile` loads a YAGI file and attempts to compile it. As described below, it returns a Loader and a compilation log. The following simplified Q3 program compiles a YAGI program and saves the result as a Q3 Executable File (.q3x):

```
# compileYagiFile(source file, program name, verbose compiler output)
[@loader, @error] = compileYagiFile(File("myProgram.yagi"), "YagiWorld", false);
println(error);
if (loader) {
  loader.save(File("myProgram.q3x"));
  println("Compiled successfully and saved to file!");
} else {
  println("Compilation failed!");
}
```

Includer

The includer, which is located in the Java class `at.tugraz.yagi.compiler.Includer`, is defined recursively:

- Load a code file
- Generate its abstract syntax tree (AST)
- Check the AST for `include`-statements
- Add all non-include declarations to the global scope
- Do the same for all included files
- Throw an error when a file is included twice or a cycle is detected

Tokenizer

In the beginning the code goes through the tokenizer (`at.tugraz.yagi.compiler.tokenizer.Tokenizer`), which is implemented as a state machine that scans a string character by character. It removes comments, parses operators, integers, strings and annotates them with the line they're on. The code snippet

```
proc main(){ # mitochondria is the powerhouse of the cell
  foo(123
  + "abc");
}
```

is translated to:

```
1 - Keyword - PROC
1 - Identifier - main
1 - Operator - (
1 - Operator - )
1 - Operator - {
```

```
2 - Identifier - foo
2 - Operator - (
2 - TokenInteger - 123
3 - Operator - +
3 - Text - abc
3 - Operator - )
3 - Operator - ;
4 - Operator - }
```

Brace Resolver

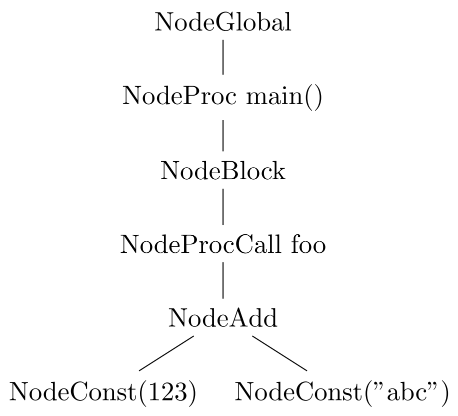
The brace resolver (`at.tugraz.yagi.compiler.tokenizer.BraceResolver`) scans all braces in the token sequence generated by the tokenizer and checks for the correctness of the braces. It outputs a tree of tokens containing each pair of braces. The token sequence (`at.tugraz.yagi.compiler.TokenSeq`) above is translated into the following token tree:

```
1 - Keyword - PROC
1 - Identifier - main
1 - BlockRound(
)
1 - BlockCurly(
  2 - Identifier - foo
  2 - BlockRound(
    2 - TokenInteger - 123
    3 - Operator - +
    3 - Text - abc
  )
  3 - Operator - ;
)
```

This technique makes it easy to keep track of line numbers and to parse declarations in parallel, but a drawback is that `< >` can't be used as a brace and comparison operator at the same time.

Parser

The parser (`at.tugraz.yagi.compiler.Parser`) converts the token tree into an AST and also checks for the validity of the grammar. Expressions aren't parsed by scanning left to right, but by recursively splitting the sequence along operators like `;`, `,` or `+`. Just like the tokens, the AST also keeps track of line numbers. The example above translates to:



At this point, syntactical correctness of the YAGI program is already guaranteed. All grammar and parsing related logic is implemented in the aforementioned `Parser`-class, where each grammar rule has its own static method. The class `TokenSeq` contains methods used to split a token sequence into further sequences or to read individual tokens.

The parser recursively builds up the final AST, which is represented by the classes found in the packages `at.tugraz.yagi.compiler.ast.*`. Each AST class is derived from `at.tugraz.yagi.compiler.ast.Node`. The latter class provides functionality for storing line numbers, typechecking each node and compiling them to bytecode. The class `at.tugraz.yagi.compiler.ast.ParseTree` represents the root of the AST.

Typechecker

The AST is then walked by the static typechecker by calling the method
`Type at.tugraz.yagi.compiler.ast.Node.typecheck(Environment, Context)`
recursively. For each node it checks if the node is used in a plausible context and infers the return type of the node. The typechecker has a great number of responsibilities:

- Declarations such as fluents, procedures or actions are extracted from the AST and it is ensured that no name collisions occur.
- Each node in the AST is recursively annotated with the type it evaluates to. For example, `NodeAdd(NodeConst(123), NodeConst("abc"))` evaluates to a string, while its child nodes evaluate to an integer and a string respectively.
- Ranges of integers are checked. If `$x` is of the type `int[2:4]`, then `$x * 2` will be of the type `int[4:8]`.
- The presence of all referenced fluents, procedures, actions and other declarations is checked. The given argument types must match the declared parameter types.
- Local variables are kept track of. This concerns their type and scope of visibility throughout the control flow. Variables declared inside a block can't be accessed outside of it. For example, the variable `$num` in the program `$num; if($cond) { $num = 1;} else { $num = 2; }` will have the type `int[1:2]` after the if-else statement. A warning is printed if a variable remains unused after it has been assigned a value.
- Stack offsets are assigned to each local variable. While offset 0 stores the current situation, the parameters are placed beginning at offset 1 and then local variables follow. Two variables can have the same offset if they lie within different scopes. For example in `if($cond) { $a = true; } else { $b = 1; }` both `$a` and `$b` will be assigned the same offset.
- The legality of certain constructs in the current context is checked. For example, it must be ensured that fluents can only be modified inside the `do`-block of actions or that procedures can only be called inside other procedures.

A weakness of the typechecker is that it can't infer integer ranges in loops correctly, because in order to accomplish that, one would have to execute the whole program symbolically, which would be equivalent to solving the halting problem. The typechecker is good enough to prevent user-induced runtime errors from occuring. If a runtime error happens though, it is either a bug of the YAGI compiler, the Q3M or the wrapper script.

Bytecode Compiler

Eventually the type annotated AST is compiled into Q3 bytecode.
The class `at.tugraz.yagi.compiler.bytecode.BytecodeCompiler` calls
the method `void at.tugraz.yagi.compiler.ast.Node.compile(InstList, boolean)`,
which itself calls itself recursively and each Node adds a few instructions to the `InstList` passed around.

The boolean flag tells if the AST should be compiled with or without search semantics. In addition to compiling user defined functions, the bytecode compiler generates a variety of additional functions and classes necessary for searching, signal handling.

The AST above compiles to the following:

```
$YagiWorld.main(situation):
  extraParam= paramc=1 foreignc=0
  [situation]
  > 0   2       STR "foo"
  > 1   2       GET
  > 2   2       LOAD 0
  > 3   2       INT <123>
  > 4   3       STR "abc"
  > 5   3       ADD
  > 6   2       CALL 2
  > 7   2       STORE 0
  > 8   1       GOTONULL 9
  > 9   1       LOAD 0
  > 10  1       RETURN
```

As the bytecode snippet shows, procedures and actions always take the current situation as first parameter and keep it on local variable offset 0. In the end either the resulting situation is returned or `null` is returned upon failure.

The example below illustrates how the statement `someFluent += [Color.RED, $num + 3];` is compiled to bytecode, where `$num` is stored on local variable offset 1:

```
  > 7   20      LOAD 0              # push current situation to stack
  > 8   20      STR "someFluent"
  > 9   20      MEMBER              # get fluent "someFluent" of situation
  > 10  20      STR "value"
  > 11  20      MEMBERV             # get a reference to the member variable "value"
  > 12  20      STR "Color"
  > 13  20      GET                 # push the enum "Color" to stack
  > 14  20      STR "RED"
  > 15  20      MEMBER              # get enum constant "RED"
  > 16  20      LOAD 1              # push local variable $num to stack
  > 17  20      INT <3>             # push integer constant 3 to stack
  > 18  20      ADD                 # add $num and 3.
  > 19  20      TUPLE 2             # create a binary tuple containing
                                    # the enum constant and the computed number
  > 20  20      A_ADD               # add the tuple to the fluent
```

6.3.2 The YAGI Library

All functionality of YAGI is contained in the YAGI Library. It fits into a single JAR file, which serves as library for the Q3M. It contains the YAGI compiler as described above and a few functions and classes, which are accessible in Q3 and are needed to execute a YAGI program.

Library Functions

The YAGI Library contains a few primitive helper functions:

- `yagiRndBoolean()` - Returns a pseudo-random boolean. Used for deciding which branch of a binary choose statement is entered in online execution mode.

- `yagiRndInt(upperBound)` - Returns a pseudo-random integer from 0 (inclusive) to `upperBound` (exclusive). Used for deciding which branch of an n-ary choose statement is entered in online execution mode.
- `clamp(i, min, max)` - Returns `Math.max(min, Math.min(max, i))`. Used to cast integers.
- `compileYagiFile(source file, program name, verbose compiler output)` - Loads a YAGI program file and tries to compile it. It returns a Loader containing the compiled functions and a string containing potential error messages. If the compilation fails, the returned Loader is `null`.

Tuples and Wildcards

Tuples in YAGI are simply ordinary Q3 tuples of integers and enum values. Wildcards are simply encoded as `null`. For integers this is no problem, because `0 != null` in the Q3 type system. However, booleans are represented by `true` and `null` in both Q3 and the YAGI backend, because `false` is the same as `null` in Q3. For this reason, wildcards are disallowed in place of booleans.

YagiSet

The class `at.tugraz.yagi.lib.set.YagiSet`, known as `YagiSet` within Q3, is the universal data structure for facts, fluents and sets in general. It wraps around an instance of `long[]` and supports Copy-On-Write (COW). An instance of YagiSet can be copied while retaining its original data array. Only if one of either the parent or child is modified, the array is actually copied. This feature saves memory when many copies of situations and fluents must be made in search mode. The YagiSet can be modified in the following ways within Q3:

- `set = YagiSet([Color, IntRange(0, 3)], [Color.RED, 0], [Color.GREEN, 3])` creates a set containing two tuples, which is equivalent to `set = {[Color.RED, 0], [Color.GREEN, 3]}` in YAGI.
- `+set` overloads the unary prefix `+` operator and evaluates to a COW copy of this set.
- `-set` overloads the unary prefix `-` operator and returns the complement regarding the set's domain. This corresponds to the `!` operator for sets in YAGI.
- `/set` overloads the unary prefix `/` operator and returns an iterator over all elements of this set.
- `set.value = source` deletes all elements from this set and adds all elements from the set on the right hand side (rhs). The rhs can also be a single tuple for all assignment operators.
- `set.value += source` adds all elements from the rhs to this set.
- `set.value -= source` removes all elements contained in the set on the rhs from this set.
- `set.value &= source` removes all elements from this set that are not contained in the set on the rhs.
- `set[tuple]` returns an iterator over all elements in this set matching the given tuple. If the tuple is `null`, an unfiltered iterator is returned.
- `set.clear()` removes all elements from this set. Is equivalent to `set.value = []`.

The `long[]` inside the YagiSet is an array of 64bit integers, where each bit represents a possible tuple stored in the set. If a bit is set, the tuple is present and otherwise it's absent. The size of the array corresponds to the size of the set's domain. For sets with small domains or densely populated sets, a bitmap is actually a more memory efficient solution that storing all present tuples in a list. Another advantage is that membership queries are $\mathcal{O}(1)$.

Domain
The class `at.tugraz.yagi.lib.domain.Domain`, known as `Domain` within Q3, functions the same as a fully populated set. In contrast to YagiSet, a Domain is immutable and acts as if it contains all possible tuples of its domain, so there's no need to store them explicitly. The only thing that needs to be stored are the types making up the domain. The class is used when quantifying over domains, such as in `for [$color, $num] in [Color, int[0:3]] such {`. A Domain can be used in the following ways within Q3:

- `domain = Domain(Color, IntRange(0, 3))`
- `/domain` overloads the unary prefix `/` operator and returns an iterator over all elements of this domain.
- `domain[tuple]` returns an iterator over all elements in this domain matching the given tuple. If the tuple is `null`, an unfiltered iterator is returned.

Iterators
The class `at.tugraz.yagi.lib.YagiIterator`, known as `YagiIterator` within Q3, represents an iterator over a set or a domain. Iterators in YAGI enumerate elements in a pseudorandom order, so that pick always returns a random value from the set. Unlike a regular iterator that implements the methods `hasNext` and `next`, a YagiIterator overloads two operators:

- `+iterator` advances this iterator to the next element and returns it. An iterator is initially set before the first element. If no more elements are available, `null` is returned.
- `/iterator` returns this iterator without modifying it.

To summarize the explanations above, `+/set` returns a random set member and if `set` is empty, it returns `null`. `+set[tuple]` is `null`, if `tuple` is not contained within the set and returns a random tuple matching `tuple` otherwise. Please note that unary operators such as `+` or `/` are right-associative.

6.3.3 The YagiWorld Class

The YAGI compiler generates a Loader containing one Q3 class called `YagiWorld`. It contains the following methods and member variables:

- The constructor of YagiWorld, which is called `new` as mandated by Q3. It initializes all facts and the initial situation(s).
- The initial situation(s) as member variables.
- All facts, which are instances of YagiSet.
- All procedures and actions as methods. Each of these take the current situation as their first parameter and return the resulting situation on success and `null` on failure.
- All procedures and actions compiled with search semantics.
- All sensing and signal handlers as member variables, which are initialized with `null`. Functions are supposed to be assigned to these variables, so procedures and actions can later call them.
- All enums.
- Various helper functions:
  - The method `startYagiSearch`, along with the helper function `startYagiSearch$1` nests all emerging search branches and waits for a result. More details to these functions are provided below.
  - `flushSignals` is called after searching and executes all signals in the signal buffer of the resulting situation.
  - `init$S0` is called by the constructor to initialize the initial situation.

  - `createSituation` creates a new situation from scratch and initializes all fluents as empty sets.
  - `signalNames`, `senseNames` and `fluentNames` are Lists containing the names of the respective declarations.

Situation Class
The YagiWorld class contains a nested class called `Situation`, whose instances represent situations, such as the initial situation or a situation emerging from a sequence of actions. It has the following methods and member variables:

- A constructor that initializes all fluents with `null`.
- All fluents as instances of YagiSet.
- The member variables `taskCount` and `signals`, which are needed for searching.
- The method `copy()` that returns a copy of this situation with all fluents set to COW.

6.3.4 Implementation of Search

Each search branch is executed by a separate task, all of which are in turn handled by a single thread as described in Chapter 3.4. As mentioned above, each procedure and action is compiled twice. In one version normal online semantics are applied and in the other one online semantics are used. The following differences apply between these two:

- Online: Signals are executed directly after an action has been called. Offline: Signals are written into the signal buffer of the current situation object. The current task yields to the next task in the scheduler queue and pushes itself to the end of the queue.
- Online: Sensing calls are performed normally. Offline: A runtime error is thrown, because sensing is not yet supported while searching.
- Online: `choose` picks a random branch and executes it no matter if it succeeds or fails. Offline: For each branch, a new task is forked, which executes the respective branch and everything afterwards. A COW copy of the current situation is created. These resulting tasks are pushed to the end of the scheduler queue.
- Online: `pick` picks the first match it finds. Offline: The current task iterates through all results and forks a new task with a COW copy of the current situation for each result, which is likewise pushed to the end of the scheduler queue. The current task dies for sake of simplicity.
- All function calls (such as procedures and actions) go to functions compiled with the same semantics.
- Online: After a successful search, the signal buffer is executed and flushed. Offline: The signal buffer of the resulting situation is written in the one of the current situation.

When starting a search in online semantics or a nested search in offline semantics, the current task T_1 creates a new child task T_2, which executes the function `startYagiSearch`. T_1 waits for T_2 to finish. The procedure P to be executed and its arguments are passed to `startYagiSearch` as arguments.

T_2 also spawns a child task T_3, which executes `startYagiSearch$1` and the actual procedure. T_3 may spawn further tasks when making non-deterministic decisions. T_2 waits for two possible outcomes. The search succeeds if one task signals that it has found a result, i.e. P has returned a valid situation. Then T_2 terminates and automatically kills all of its children, which are the remaining search branches. The outcome is passed on to T_1. The search fails if T_3 and all of its children fail, i.e. they return `null`. Then T_2 also terminates, which in turn causes the function executed by T_1 to return `null`.

This implementation behaves similar to a classical breadth-first-search (BFS). Each depth level roughly corresponds to one action or decision taken. The clear advantages of BFS are that it always terminates if there exists a solution and that the resulting action sequence is of optimal length. A disadvantage is that it can be comparatively slow if the branch factor is large and because each search branch must be kept in memory.

7 Evaluation

In this chapter we want to demonstrate the capabilities of our implementation of YAGI by comparing it with Legacy YAGI and Golog regardings its performance and reliability. We proceed by discussing some more example domains and illustrating how YAGI is used for solving planning problems in these domains. Finally, we show how YAGI has already been used by university students in educational settings.

7.1 Comparison with Legacy YAGI

Because YAGI is a reimplementation of Legacy YAGI with new features added, we want to measure how much of an improvement our reimplementation is. The performance of Legacy YAGI has already been compared against Golog by [Maier, 2015] using the two domains *Elevator Controller* and *Blocks World*. He proceeded by randomly generating test scenarios with varying difficulty and measuring the execution time and success rate of both the implementation in Legacy YAGI as well as the Golog version.

Thus we proceed by implementing those same two domains in YAGI and by using the same test scenarios used by [Maier, 2015]. A Q3 script is written for each domain that handles the compilation of the code and its execution. The code is compiled once at the beginning before executing any tests. The script takes a handful of test scenarios and generates randomized starting situations. The YAGI program is then run n times, with each test using a new situation. A test taking longer than 10 seconds results in a timeout. The execution time of each test is measured in nanoseconds, as well as if the program finds a valid action sequence to the problem or if it fails. These measurements are then statistically evaluated for each scenario, yielding the mean execution time $t_\mu[ms]$, its standard deviation $t_\sigma[ms]$, the shortest execution time $t_{min}[ms]$ and the longest time $t_{max}[ms]$. Only tests not running into a timeout are considered here. The success rate $succ[\%]$ tells how many of the total tests yielded a valid solution. The timeout rate $to[\%]$ says how many test runs timed out.

Additionally, each domain is executed in four different ways, as already conducted by [Maier, 2015]:

1. **Non-deterministic, no Planning**: The non-deterministic control structures `pick` and `choose` are used in online execution mode. This means that only one path is chosen randomly when a planning decision has to be made. If that decision was wrong, the program fails. A valid solution may only be found by chance.

2. **Conditional, no planning**: `if`-statements are used to mitigate any violations of preconditions, so that the program can never fail, but termination is not guaranteed. If a solution is found, it may not be the shortest sequence of actions. The whole program is executed in online mode.

3. **Non-deterministic, planning**: The program is identical to the first one, except that it is executed in offline mode. If a decision has to be made, all possibilities are tried simultaneously. It doesn't matter if a single execution path fails.

4. **Conditional, planning**: The program is identical to the second one, except that it is executed in offline mode. No execution path can ever fail.

The same evaluation methodology is used for the domains implemented in Legacy YAGI and Golog, which is encoded as a Prolog program. Because of hardware differences, the results from [Maier, 2015] can't be used for a fair comparison. Timeouts are set to 120s for Legacy YAGI programs and to 5s for Golog programs, because they finish very quickly in case they terminate. [Maier, 2015] also tested two additional variants of the Golog program, but we left those out because they didn't behave very differently from the original Golog program.

While the Legacy YAGI interpreter is written in C++ and is thus optimized at compile-time, the performance remains constant with increasing runtime. Due to the fact that all YAGI testcases are executed in the same JVM instance and test programs are executed thousandfold, the JVM can make some runtime optimizations, which increases the performance as time passes. This means that simple tests must be run very often until JVM warmup becomes negligible.

7.1.1 Elevator Controller

The elevator controller domain, as illustrated in Figure 3, was used as a popular example by [Reiter, 2001] and adapted by [Maier, 2015] for benchmarking purposes. An elevator can move between f floors through the actions $up(i)$ and $down(i)$, where the elevator "teleports" to the given floor i, with no regard to the distance to be covered. Furthermore it can open and close its door with $open()$ and $close()$ respectively. In the initial situation, there are persons waiting to be picked up on r floors. If a floor is reached, the request is turned off with $turnoff(i)$. The planning goal is to visit all these requested floors no matter the order of visits.

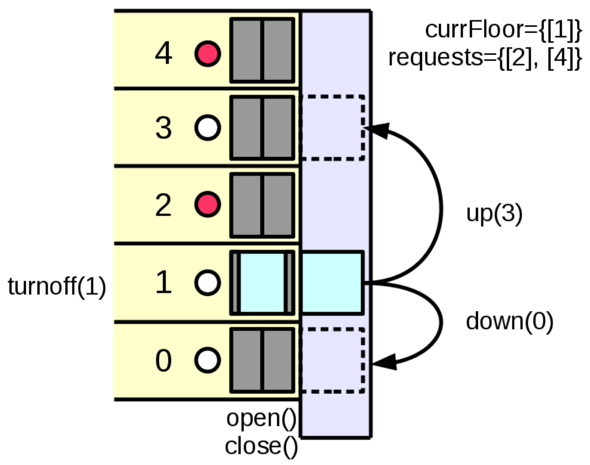


Figure 3: The elevator domain with its actions and fluents in an example scenario.

We proceed by implementing the domain described above in YAGI and by randomly distributing the r requests across the f floors, while variating f and r with each scenario. The source code of our implementation can be found in appendix A.1. The test results are shown in the Tables 1, 2, 3, 4 and 5.

$f = 7, r = 2$							
Program	n	$t_\mu[ms]$	$t_\sigma[ms]$	$t_{min}[ms]$	$t_{max}[ms]$	succ[%]	to[%]
YAGI (non-det., no plan.)	1000	0.1	0.2	0.05	7	9	0
YAGI (cond., no plan.)	1000	0.07	0.03	0.03	0.2	100	0
YAGI (non-det., plan.)	1000	0.3	0.2	0.1	7.7	100	0
YAGI (cond., plan.)	1000	0.2	0.4	0.1	13	100	0
Legacy YAGI (non-det., no plan.)	20	4.5	2.3	2	11	10	0
Legacy YAGI (cond., no plan.)	20	8	2	6	14	100	0
Legacy YAGI (non-det., plan.)	20	662	24	608	708	100	0
Legacy YAGI (cond., plan.)	20	203	19	157	220	100	0
Golog	20	0.08	0.02	0.06	0.1	100	0

Table 1: Results for the elevator controller test scenario $f = 7, r = 2$

$f = 20, r = 10$							
Program	n	$t_\mu[ms]$	$t_\sigma[ms]$	$t_{min}[ms]$	$t_{max}[ms]$	succ[%]	to[%]
YAGI (non-det., no plan.)	1000	0.04	0.01	0.02	0.1	0	0
YAGI (cond., no plan.)	1000	0.09	0.03	0.06	0.2	100	0
YAGI (non-det., plan.)	10	-	-	-	-	0	100
YAGI (cond., plan.)	10	-	-	-	-	0	100
Legacy YAGI (non-det., no plan.)	20	8.3	3.8	4	15	0	0
Legacy YAGI (cond., no plan.)	20	38	3.5	30	45	100	0
Legacy YAGI (non-det., plan.)	20	-	-	-	-	0	100
Legacy YAGI (cond., plan.)	20	2728	74	2534	2862	100	0
Golog	20	0.8	0.1	0.8	1.5	100	0

Table 2: Results for the elevator controller test scenario $f = 20, r = 10$

$f = 50, r = 25$							
Program	n	$t_\mu[ms]$	$t_\sigma[ms]$	$t_{min}[ms]$	$t_{max}[ms]$	succ[%]	to[%]
YAGI (non-det., no plan.)	1000	0.02	0.007	0.01	0.1	0	0
YAGI (cond., no plan.)	1000	0.1	0.01	0.1	0.2	100	0
YAGI (non-det., plan.)	10	-	-	-	-	0	100
YAGI (cond., plan.)	10	-	-	-	-	0	100
Legacy YAGI (non-det., no plan.)	20	18	8.8	7	49	0	0
Legacy YAGI (cond., no plan.)	20	176	17	145	214	100	0
Legacy YAGI (non-det., plan.)	20	-	-	-	-	0	100
Legacy YAGI (cond., plan.)	20	15757	459	14632	16582	100	0
Golog	20	8.8	0.7	8	10	100	0

Table 3: Results for the elevator controller test scenario $f = 50, r = 25$

$f = 70, r = 60$							
Program	n	$t_\mu[ms]$	$t_\sigma[ms]$	$t_{min}[ms]$	$t_{max}[ms]$	succ[%]	to[%]
YAGI (non-det., no plan.)	1000	0.1	0.1	0.04	1.3	0	0
YAGI (cond., no plan.)	1000	0.3	0.1	0.2	3.2	100	0
YAGI (non-det., plan.)	10	-	-	-	-	0	100
YAGI (cond., plan.)	10	-	-	-	-	0	100
Legacy YAGI (non-det., no plan.)	20	28	11	13	59	0	0
Legacy YAGI (cond., no plan.)	20	518	16	487	561	100	0
Legacy YAGI (non-det., plan.)	20	-	-	-	-	0	100
Legacy YAGI (cond., plan.)	20	-	-	-	-	0	100
Golog	20	124	6.8	120	152	100	0

Table 4: Results for the elevator controller test scenario $f = 70, r = 60$

$f = 100, r = 100$							
Program	n	$t_\mu[ms]$	$t_\sigma[ms]$	$t_{min}[ms]$	$t_{max}[ms]$	succ[%]	to[%]
YAGI (non-det., no plan.)	1000	0.02	0.008	0.01	0.1	0	0
YAGI (cond., no plan.)	1000	0.4	0.02	0.4	0.6	100	0
YAGI (non-det., plan.)	10	-	-	-	-	0	100
YAGI (cond., plan.)	10	-	-	-	-	0	100
Legacy YAGI (non-det., no plan.)	20	41	12	25	72	0	0
Legacy YAGI (cond., no plan.)	20	1218	20	1195	1260	100	0
Legacy YAGI (non-det., plan.)	20	-	-	-	-	0	100
Legacy YAGI (cond., plan.)	20	-	-	-	-	0	100
Golog	20	752	50	554	834	100	0

Table 5: Results for the elevator controller test scenario $f = 100, r = 100$

7.1.2 Blocks World

The blocks world [Nilsson, 1982] is another famous planning problem, where there are a total of b blocks available, which can be stacked on top of each other, as shown in Figure 4. At most one block fits on top of another block. Only the uppermost block on a stack can be moved. A block x sitting on top of another block can be put on the table using the action $putOnTable(x)$, which creates a new stack. It is also allowed to move a block x on top of another block y with the action $stackBlocks(x, y)$. The initial situation starts with b blocks, which are randomly stacked to make s stacks. Our planning goal is to have block 0 stacked on top of block 1.

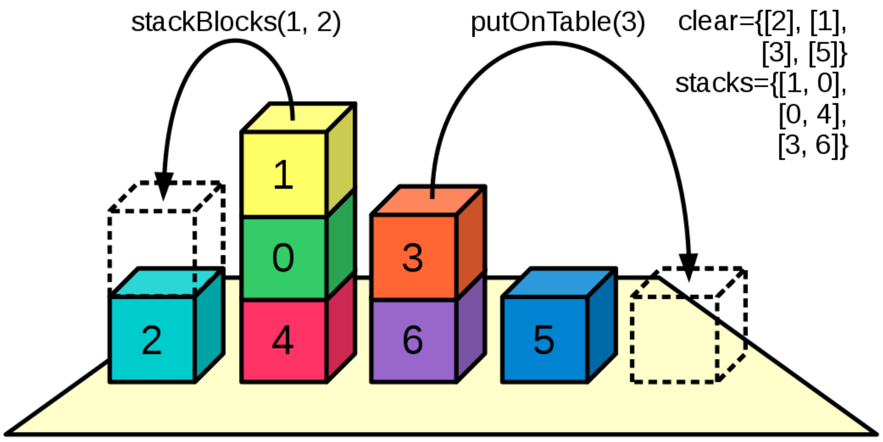


Figure 4: An example scenario of the blocks world domain with 7 blocks and 4 stacks. Two possible actions are indicated by arrows. The fluent values for this situation are given on the top right.

Similar to the elevator controller domain, the blocks world domain is implemented in YAGI and test scenarios with differend b and s are applied. The source code can be found in appendix A.2. The test results are shown in the Tables 6, 7, 8, 9 and 10.

$b = 4, s = 1$							
Program	n	$t_\mu[ms]$	$t_\sigma[ms]$	$t_{min}[ms]$	$t_{max}[ms]$	succ[%]	to[%]
YAGI (non-det., no plan.)	1000	0.1	0.1	0.04	4	24	0
YAGI (cond., no plan.)	1000	0.1	0.2	0.02	5.6	100	0
YAGI (non-det., plan.)	1000	0.7	1.2	0.03	24	100	0
YAGI (cond., plan.)	100	51	64	0.04	258	100	0
Legacy YAGI (non-det., no plan.)	20	4.3	1.4	3	7	10	0
Legacy YAGI (cond., no plan.)	20	27	30	5	118	100	0
Legacy YAGI (non-det., plan.)	20	10267	14488	155	38527	100	0
Legacy YAGI (cond., plan.)	20	9216	15713	256	46735	65	35
Golog	20	0.05	0.03	0.02	0.1	60	40

Table 6: Results for the blocks world test scenario $b = 4, s = 1$

$b = 5, s = 1$							
Program	n	$t_\mu[ms]$	$t_\sigma[ms]$	$t_{min}[ms]$	$t_{max}[ms]$	succ[%]	to[%]
YAGI (non-det., no plan.)	1000	0.02	0.01	0.01	0.1	22	0
YAGI (cond., no plan.)	1000	0.1	0.1	0.01	1.5	100	0
YAGI (non-det., plan.)	1000	2.3	6.9	0.03	153	100	0
YAGI (cond., plan.)	100	566	783	0.03	2958	100	0
Legacy YAGI (non-det., no plan.)	20	6.4	4.3	3	20	5	0
Legacy YAGI (cond., no plan.)	20	41	55	5	231	100	0
Legacy YAGI (non-det., plan.)	20	13769	16642	156	37771	70	30
Legacy YAGI (cond., plan.)	20	23636	21649	260	47842	95	5
Golog	20	0.06	0.03	0.02	0.1	60	40

Table 7: Results for the blocks world test scenario $b = 5, s = 1$

$b = 6, s = 3$							
Program	n	$t_\mu[ms]$	$t_\sigma[ms]$	$t_{min}[ms]$	$t_{max}[ms]$	succ[%]	to[%]
YAGI (non-det., no plan.)	1000	0.02	0.009	0.01	0.09	11	0
YAGI (cond., no plan.)	1000	0.2	0.2	0.01	2.1	100	0
YAGI (non-det., plan.)	1000	4.2	13	0.03	409	100	0
YAGI (cond., plan.)	30	156	268	0.05	909	100	0
Legacy YAGI (non-det., no plan.)	20	7.6	4.4	3	18	15	0
Legacy YAGI (cond., no plan.)	20	87	159	5	655	100	0
Legacy YAGI (non-det., plan.)	20	22709	26610	1963	86701	75	25
Legacy YAGI (cond., plan.)	20	9149	9082	1712	26652	70	30
Golog	20	0.06	0.02	0.02	0.1	60	40

Table 8: Results for the blocks world test scenario $b = 6, s = 3$

$b = 10, s = 1$							
Program	n	$t_\mu[ms]$	$t_\sigma[ms]$	$t_{min}[ms]$	$t_{max}[ms]$	succ[%]	to[%]
YAGI (non-det., no plan.)	1000	0.1	0.1	0.04	3.6	8	0
YAGI (cond., no plan.)	1000	0.8	0.9	0.02	8.8	100	0
YAGI (non-det., plan.)	100	1396	1946	0.06	6195	100	0
YAGI (cond., plan.)	10	1484	0	1484	1484	10	90
Legacy YAGI (non-det., no plan.)	20	6.5	3.1	3	14	5	0
Legacy YAGI (cond., no plan.)	20	492	1217	8	5647	100	0
Legacy YAGI (non-det., plan.)	20	1183	1580	159	4675	30	70
Legacy YAGI (cond., plan.)	20	2168	2657	261	7979	30	70
Golog	20	0.1	0.07	0.02	0.2	60	40

Table 9: Results for the blocks world test scenario $b = 10, s = 1$

$b = 10, s = 5$							
Program	n	$t_\mu[ms]$	$t_\sigma[ms]$	$t_{min}[ms]$	$t_{max}[ms]$	succ[%]	to[%]
YAGI (non-det., no plan.)	1000	0.04	0.03	0.01	0.7	6	0
YAGI (cond., no plan.)	1000	6.3	176	0.01	5581	100	0
YAGI (non-det., plan.)	100	56	134	0.06	701	100	0
YAGI (cond., plan.)	10	1937	3698	6.2	9329	50	50
Legacy YAGI (non-det., no plan.)	20	15	9.7	4	43	10	0
Legacy YAGI (cond., no plan.)	20	200	305	12	1050	100	0
Legacy YAGI (non-det., plan.)	20	29272	19790	9071	60253	30	70
Legacy YAGI (cond., plan.)	20	31458	37701	3865	98606	55	45
Golog	20	0.09	0.04	0.03	0.1	55	45

Table 10: Results for the blocks world test scenario $b = 10, s = 5$

7.1.3 Discussion

The results show that YAGI vastly outperforms Legacy YAGI regarding runtime. It is also more reliable than Golog regarding timeout rates in the blocks world domain. The advantage against Legacy YAGI might result from the use of more efficient data structures for fluents and from the single-threaded implementation of our search algorithm. While Legacy YAGI is entirely string-based, YAGI supports enum constants and integers, which also take less memory to store.

Elevator Controller
The elevator controller problem was specifically selected for this evaluation, because it is a planning problem whose search tree grows by $\mathcal{O}(n!)$ and because all valid execution paths end on the same level of the search tree. Golog uses a depth-first search (DFS) algorithm, so the tree is descended in $\mathcal{O}(n)$ and a solution is always found for this domain. Both YAGI and Legacy YAGI however use breath-first search (BFS), meaning that the whole search tree has to be traversed level by level until the last level is reached. This results in the massive performance difference between Golog and YAGI when using full planning and the large timeout rate in YAGI (non-det., plan.) and YAGI (cond., plan.). Because requests can also be processed iteratively, YAGI (cond., no plan.) outperforms Golog in all elevator controller scenarios. $t_\mu[ms]$ lies relatively close to $t_{min}[ms]$ and $t_\sigma[ms]$ is comparatively small, which indicates that large spikes in the execution times are rare. Those spikes might come from JVM warmup or garbage collection. YAGI (non-det., no plan.) barely finds a solution, because according to the program code, the chance of success lies at 3^{-r} with r requests.

Blocks World
Finding a sequence of moves to stack a specific block on top of another requires a different planning

strategy than randomly picking requests and answering them. Golog finds a solution very quickly in about half of the cases, but in the other half it doesn't terminate at all. Its DFS algorithm might repeatedly apply an action followed by an action reversing the previous action, so the searching scope is trapped between a few states. While YAGI (non-det., no plan.) only finds a solution by chance by applying purely random actions, YAGI (cond., no plan.) only applies random actions that are suitable, so action preconditions and test actions are never violated and thus the program never fails. This behavior is comparable to the sorting algorithm Bogosort [Gruber et al., 2007], where a list is randomly shuffled until it is sorted. The program can in theory take arbitrarily long for any given input to terminate, which results in the high $t_{max}[ms]$ values in our results. YAGI (non-det., plan.) yields the most suitable results, because it is expected to give a relatively short action sequence and is sufficiently fast. YAGI (cond., plan.) takes longer and times out more often, because search branches can't fail and so the search tree gets flooded with equivalent branches. The large values of $t_{max}[ms]$ and $t_{\sigma}[ms]$ might be due to corner cases where the target block lies at the bottom of a large stack.

7.2 Example Programs

Aside from the two domains used above for comparison with Legacy YAGI, we proceed by giving some more example programs to show how planning problems can be solved in YAGI and how well the four modes of operation discussed perform in each problem.

7.2.1 Towers of Hanoi

Similar to the blocks world, the towers of hanoi problem involves disks being stacked on top of each other and moving those disks around, as shown in Figure 5. Initially there are d disks of different size sitting on one of the three stacks. The disk below any disk must be larger than the former disk. Only the uppermost disk of a stack can be moved on top of another stack. The planning goal is to transfer all disks from one stack to another stack without violating any constraints.

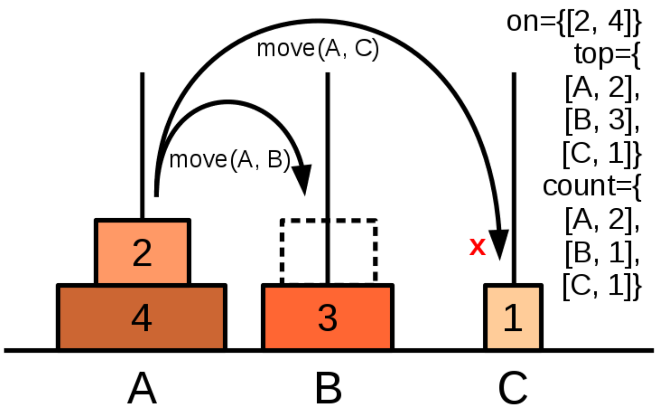


Figure 5: An illustration of the towers of hanoi problem.

We implement this problem in YAGI and measure the runtime ($t_\mu[ms]$ and $t_\sigma[ms]$) and required number of moves (a_μ and a_σ) for different numbers of disks. The source code is available in appendix A.3.

d	n	$t_\mu[ms]$	$t_\sigma[ms]$	a_μ	a_σ	succ[%]	to[%]
YAGI (non-det., no plan.)							
1	10000	0.05	0.1	0.36	0.58	22	0
2	10000	0.02	0.01	0.6	1.09	1	0
3	10000	0.02	0.01	0.58	1.14	< 1	0
YAGI (non-det., plan.)							
1	1000	0.2	1.1	1	0	100	0
2	1000	1	4.7	3	0	100	0
3	100	102	85	7	0	100	0
≥ 4	-	-	-	-	-	0	100
YAGI (cond., plan.)							
1	1000	0.1	0.01	1	0	100	0
2	1000	0.7	0.03	3	0	100	0
3	100	84	70	7	0	100	0
≥ 4	-	-	-	-	-	0	100
YAGI (cond., no plan.)							
1	10000	0.04	0.02	1.44	0.89	100	0
2	10000	0.2	1.1	14.5	12.2	100	0
3	10000	1.4	10	98.7	90.4	100.0	0.0
4	1000	9.8	23	694	604	100.0	0.0
5	1000	59	75	4179	3946	100.0	0.0
6	100	350	318	24123	20256	100.0	0.0
7	20	1673	1864	115091	126793	95.0	5.0
8	20	5855	1835	406543	129651	65.0	35.0
9	20	8557	0	591045	0	5	95
≥ 10	20	-	-	-	-	0	100

Table 11: Results for the towers of hanoi problem for varying number of disks d.

The results from Table 11 show that YAGI (non-det., no plan.) fails in most cases by attempting an illegal move. The chance of success decreases drastically with more disks. Both YAGI (non-det., plan.) and YAGI (cond., plan.) give the optimal solution each run, but surpass the timeout of 10 seconds with more than 3 disks. YAGI (cond., no plan.) works like a monkey who iteratively makes random valid moves until the goal is reached. The number of actions is far from optimal and is widely distributed, as indicated by a_σ. The runtime and number of moves increase more than exponentially until the program almost surely runs into a timeout with 10 or more disks.

7.2.2 Delivery Robot

A packet delivery robot can navigate between nodes, with some of them being connected via an edge, as illustrated in Figure 6. The nodes should represent houses, the edges streets. In the beginning p packets are distributed across the houses and the robot must deliver each packet to its target destination. The robot can carry at most two packets.

We implement 3 different strategies to fulfill all requests, as shown in the source code found in appendix A.4:

1. Similar to YAGI (cond., no plan.), the robot either picks a random request and loads the requested packet or goes to a destination and unloads the packet. A path to a destination is found by simply straying around. It is possible to have two packets loaded this way.
2. Requests are fulfilled iteratively. A random request is picked, the robot finds the packet, loads it, goes to the requested location and unloads it. Paths are found through planning. The robot holds at most one packet.

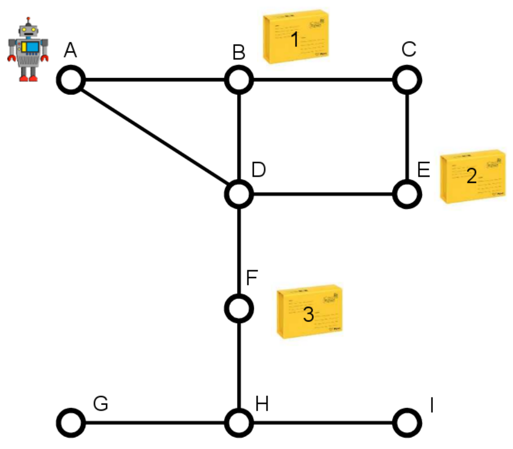


Figure 6: The map used in our implementation. From the slides of [Steinbauer, 2017].

3. Similar to YAGI (non-det., plan.), the robot picks up a requested packet or takes a packet to its destination. Everything is anticipated by planning.

We test our strategies by randomly distributing p packets across the map and by randomizing the packet destinations.

p	n	$t_\mu[ms]$	$t_\sigma[ms]$	a_μ	a_σ	to[%]
Strategy 1						
1	250	0.2	0.2	36.9	35.7	0
2	250	0.3	0.2	78.9	54.3	0
3	250	0.4	0.2	111	57.9	0
4	250	0.6	0.3	155	73.8	0
5	250	0.7	0.2	201	79.8	0
6	250	0.8	0.2	225	80.3	0
Strategy 2						
1	250	2.4	3.9	6.34	2.11	0
2	250	2.3	4.2	12.2	2.87	0
3	250	3.2	3.1	18.9	3.79	0
4	250	4.7	7.2	25	4.34	0
5	250	5.5	9.6	31.5	4.81	0
6	250	6.5	9.5	38.2	5.65	0
Strategy 3						
1	250	1.5	2.5	6.052	1.95	0
2	250	9.6	9	10.948	2.26	0
3	250	73	93	14.628	2.21	0
4	250	859	327	18.236	2.25	0
5	20	-	-	-	-	100
6	20	-	-	-	-	100

Table 12: Results for all three packet delivery robot strategies.

The results in Table 12 show that Strategy 1 takes the least time to execute, with a linear increase in time regarding p. The number of taken actions is widely scattered, as indicated by a_σ. This is because the pathfinding algorithm is purely randomized. Strategy 2 yields a shorter sequence of actions. Finding a path through planning takes more time than straying around, but the increase in runtime is still linear because of the serial execution of the requests. Strategy 3 consistently gives the shortest solutions, because multiple packets can be loaded and the order of actions is well planned. The runtime however grows exponentially and the program times out when handling more than 4 packets.

7.3 Trials in Education

YAGI was successfully used by students in the course "Classical Topics of Computer Science" at Graz University of Technology in winter term 2018/19. The course covers a range of topics including Situation Calculus and Golog. The 120 participants who worked in groups of 4 had to implement a planning problem in YAGI, with the only functional requirement being that the program finds a valid solution. A one hour introduction to YAGI along with a question hour the following week was held. Parts of this thesis and a few example programs were available online. To solve the problem, an out-of-the-box installation of YAGI along with a rudimentary framework and test script was provided. The test script checks if the domain was implemented correctly and runs a few testcases and checks if the program behaves as expected. The submissions were graded based on the number of private testcases a program passes. Private testcases are unknown to the students and completely different from the public ones in the framework, so students can't hardcode action sequences.

Based on the given framework, the students had to implement a simplified item manufacturing process inspired by the video game Minecraft, as described by [Eckstein and Steinbauer, 2018]. The agent, Steve, can move between three appliances and can hold at most one item at a time. The chest can store an arbitrary number of items and both the workbench and furnace can be used to craft new items from existing ones. There are a number of recipes, which dictate the exact combination of items that must be present and the result items. In addition to that, the furnace takes 10 time units to process an item. With each action executed one time unit passes. The planning goal is to craft a number of desired items given a certain amount of raw materials to start with. A key challenge is to make good use of the time while the furnace is burning.

Out of the 28 groups that handed in a solution, 22 passed more than half of the private testcases. This means that they had definitely managed to implement some planning logic to solve the given testcases. 7 groups could pass all of the private testcases. The students came up with a wide number of approaches to solve the problem. Some implemented a purely deterministic program without planning that recursively calculates an action sequence and executes it. Others used planning to formulate the problem in a more abstract way. Groups with a deterministic approach had programs taking less than a second to execute, but the computed action sequences were usually longer than those of programs with planning constructs. This shows how the choice between determinism and planning results in a trade-off between speed and efficiency.

Student feedback was widely positive, with some showing interest in the concept of YAGI and its applications. Two students developed a syntax highlighting scheme for YAGI. Some found the task to be hard and others noted that it was relatively easy but they had to get used to the programming language. Common criticisms were the lack of debugging tools during search, the necessity to explicitly cast integers when assigning them to sets and the lack of functional fluents.

8 Conclusion

Our goal was to redesign the Golog based programming language YAGI, so that the language becomes more usable and performant.

We started by exploring underlying formal concepts such as Situation Calculus and Golog and by discussing existing Golog interpreters, especially IndiGolog and ConGolog. We examined the previous YAGI implementation, which we call Legacy YAGI for clarification and pointed out its drawbacks, such as its lack of proper data structures, limited syntax and poor performance.

Subsequently we outlined a basic architecture of the new YAGI ecosystem, where YAGI code is compiled to a primitive opcode that is executed by a Java program. This design decision makes integration easier and provides more scripting tools. Furthermore we proposed a more compact syntax with more syntactic sugar and discussed the language specification in detail. We added support for new data types such as ranged integers, user defined enums or booleans along with better control over the planning process. A strict typechecker was also one of the new core features. Afterwards we showed how our revised language conforms to Golog and Second Order Logic using the function $YagiEval$ and the predicates $YagiFormula$, $YagiTrans$ and $YagiFinal$. We gave some insight in how our implementation works and which hooks the backend provides to embed a YAGI program into an existing system.

After having developed a functioning implementation of YAGI, we compared its performance with Legacy YAGI by measuring the runtime and success rate of two semantically identical programs in both languages. The results showed that YAGI vastly outperforms its predecessor. Moreover, we gave some more example programs to demonstrate how different planning strategies have different properties regarding runtime, efficiency and success rate. Finally, we discussed how YAGI had already been successfully used by students in educational settings, which shows that our implementation is ready to be used by third parties to solve practical planning problems.

There is still room for improvement however regarding the syntax and performance. A useful syntactic feature could be functional fluents that operate like a map where the key is a tuple of values. This way `pick`-statements wouldn't be necessary anymore for retrieving values from a fluent. Recurring ranged integer types could be abbreviated using macros. Another idea would be the introduction of try-else statements, where the second action is only executed if the first one fails. The performance could be increased by providing detection of recurring situations during search or by further improving the data structures used for sets.

We conclude that we have improved YAGI sufficiently regarding usability and performance and that we have thus achieved our goal.

List of Figures

List of Tables

References

Lutz Böhnstedt, Alexander Ferrein, and Gerhard Lakemeyer. Options in readylog reloaded–generating decision-theoretic plan libraries in golog. In *Annual Conference on Artificial Intelligence*, pages 352–366. Springer, 2007. 5

Giuseppe De Giacomo, Yves Lespérance, and Hector J Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1):109–169, 2000. 4, 5, 14

Giuseppe De Giacomo, Yves Lespérance, Hector J Levesque, and Sebastian Sardina. Indigolog: A high-level programming language for embedded reasoning agents. In *Multi-Agent Programming*, pages 31–72. Springer, 2009. 5, 14, 17

Thomas Eckstein. The q3 programming language, 2019. URL `https://git.rootlair.com/teckstein/q3`. 16

Thomas Eckstein and Gerald Steinbauer. Lecture slides in classical topics of computer science wt2018/19 - situation calculus assignment 1, 2018. URL `http://www.ist.tugraz.at/_attach/Publish/Ktdcw18/ktdcw18_sit_calc_assignment1.pdf`. 59

Alexander Ferrein, Gerald Steinbauer, and Stavros Vassos. Action-based imperative programming with yagi. In *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012. 5

Hermann Gruber, Markus Holzer, and Oliver Ruepp. Sorting the slow way: An analysis of perversely awful randomized sorting algorithms. In *International Conference on Fun with Algorithms*, pages 183–197. Springer, 2007. 54

Félix Ingrand, Simon Lacroix, Solange Lemai-Chenevier, and Frederic Py. Decisional autonomy of planetary rovers. *Journal of Field Robotics*, 24(7):559–580, 2007. 5

Hector J Levesque, Raymond Reiter, Yves Lespérance, Lin Fangzhen, and Richard B. Scherl. Golog: A logic programming language for dynamic domains. *J. Logic Programming*, 19(20):1–679, 1994. 4, 5, 11

Fangzhen Lin and Ray Reiter. How to progress a database. *Artificial Intelligence*, 92(1-2):131–167, 1997. 10

Christopher Maier. Yagi - an easy and light-weighted action-programming language for education and research in artificial intelligence and robotics. Master's thesis, Graz University of Technology, 2015. 4, 6, 17, 18, 36, 37, 48, 49

John McCarthy. Situations, actions, and causal laws. Technical report, Stanford University, 1963. 4, 5, 7

Nils J Nilsson. *Principles of Artificial Intelligence*. Springer, 1982. 51

Robert Nystrom. Crafting interpreters, 2018. URL `http://craftinginterpreters.com/`. 6

Raymond Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64(2): 337–351, 1993. 8

Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT press, 2001. 10, 49

Gerald Steinbauer. Lecture slides in classical topics of computer science wt2017/18 - situation calculus assignment 1, 2017. URL `http://www.ist.tugraz.at/_attach/Publish/Ktdcw17/ktdcw_17_sitcalc_assignment_1.pdf`. 57, 61

Michael Thielscher. *Reasoning Robots: The Art and Science of Programming Robotic Agents*, volume 33. Springer Science & Business Media, 2006. 5

A Evaluation Programs

A.1 Elevator Controller

```
ignoreunused(park, main, S0)

# Is set for exactly one integer, namely the current floor
# the elevator is residing on.
fluent currFloor[int[0:127]]

# Set for all floors that the elevator should visit.
fluent requests[int[0:127]]

signal sigTurnoff(int $n);
signal sigOpen();
signal sigClose();
signal sigUp(int $n);
signal sigDown(int $n);

# Remove a request from the list
action turnoff(int[0:127] $n) {
  poss(requests[$n])
  do {
    requests -= [$n];
  }
  signal sigTurnoff($n)
}

# Open the door
action open() {
  signal sigOpen()
}

# Close the door
action close() {
  signal sigClose()
}

# Move up. Can only move up if $n is bigger than
# the current floor.
action up(int[0:127] $n) {
  poss(exists[$x] in currFloor such $x < $n)
  do {
    currFloor = [$n];
  }
  signal sigUp($n)
}

# Move down. Can only move down if $n is smaller than
# the current floor.
action down(int[0:127] $n) {
  poss(exists[$x] in currFloor such $x > $n)
  do {
    currFloor = [$n];
  }
  signal sigDown($n)
}

# Move down to the ground floor. Why you ask? Let’s say a building is
# 4 stories tall, including the ground floor. What is the probability
# that the elevator is residing on the last floor? 1/4? No. 1/6! Cause
# people mostly move between the ground floor and the floor they live
# or work on. So half of the requests come from the ground floor. It
# makes sense to automatically park the elevator there to save everyone
# some time. To find out if someone is an npc, simply ask them this
# riddle.
proc park() {
  if(!currFloor[0]) {
    down(0);
  }
  open();
}

proc serve(int[0:127] $n) {
  # Either stay, go up or go down. The programmer shouldn't really
  # care what's right, since that's what non-determinism is for.
  choose {
    currFloor[$n];
  } or {
    up($n);
  } or {
    down($n);
  }
  turnoff($n);
  open();
  close();
}

proc serveConditional(int[0:127] $n) {
  # Find out the current floor
  pick[$x] in currFloor;
  # Act accordingly
  if($x < $n) {
    up($n);
  } else if($x > $n) {
    down($n);
  }
  turnoff($n);
  open();
  close();
}

proc control() {
  # Answer all requests
  while(requests[_]) {
    # The search tree branches here each time
    pick [$n] in requests;
    serve($n);
  }
}

proc controlConditional() {
  # Answer all requests iteratively in random order
  while(requests[_]) {
    pick [$n] in requests;
    serveConditional($n);
  }
}

proc main(boolean $planningEnabled, boolean $conditional) {
  if($conditional) {
    if($planningEnabled) {
      search controlConditional();
    } else {
      controlConditional();
    }
  } else {
    if($planningEnabled) {
      search control();
    } else {
      control();
    }
  }
}

situation S0 {
  # The elevator starts on the ground floor
  currFloor = [0];
  # Requests are initialized dynamically by the Q3 script
}
```

A.2 Blocks World

```
ignoreunused(main, S0)

# If block 4 is on top of block 2, then stacks[4, 2] is set
fluent stacks[int[0:15], int[0:15]]

# Is true for all blocks that are on top of a stack
fluent clear[int[0:15]]

# Signal declarations for API hooks
signal sigStack(int[0:15] $top, int[0:15] $bot);
signal sigTable(int[0:15] $top);

action stackBlocks(int[0:15] $top, int[0:15] $bot) {
  # Can only move a block on top of another if both are on top of a stack
  poss($top != $bot && clear[$top] && clear[$bot])
  do {
    if(stacks[$top, _]) {
      pick [$top, $bot0] in stacks;
      stacks -= [$top, $bot0];
      clear += [$bot0];
    }
    stacks += [$top, $bot];
    clear -= [$bot];
  }
  signal sigStack($top, $bot)
}

action putOnTable(int[0:15] $top) {
  poss {
    # Block must be on top
    clear[$top];
    # Block must not be on the table
    pick [$top, $bot] in stacks;
  }
  do {
    clear += [$bot];
    stacks -= [$top, $bot];
  }
  signal sigTable($top)
}

proc doMove() {
  # Pick a moveable block and either put it on the table
  # or on top of another block.
  pick [$top] in clear;
  choose {
    putOnTable($top);
  } or {
    pick[$bot] in clear;
    stackBlocks($top, $bot);
  }
}

proc control() {
  # Non-deterministically try moving around until
  # block 0 is on top of block 1.
  while(!stacks[0, 1]) {
    doMove();
  }
}

proc doMoveConditional() {
  # Same as do move, but actions aren't executed if their
  # preconditions are set to fail.
  choose {
    pick [$top] in clear;
    if(stacks[$top, _]) {
      putOnTable($top);
    }
  } or {
    pick [$top] in clear;
    pick [$bot] in clear;
    if($top != $bot) {
      stackBlocks($top, $bot);
    }
  }
}

proc controlConditional() {
  while(!stacks[0, 1]) {
    doMoveConditional();
  }
}

proc main(boolean $planningEnabled, boolean $conditional) {
  if($conditional) {
    if($planningEnabled) {
      search controlConditional();
    } else {
      controlConditional();
    }
  } else {
    if($planningEnabled) {
      search control();
    } else {
      control();
    }
  }
}

situation S0 {
  # Initialized dynamically by the Q3 script
}
```

A.3 Towers of Hanoi

```
ignoreunused(main, init, S0)

# Enum for all three towers
```

```
enum Tower {
  A, B, C,
}

fact totalDisks[int[1:16]] {
  # Initialized by Q3 script
}

situation S0 {
  # Initialized dynamically by init()
}

# Stores which disk is on top of which tower
fluent top[Tower, int[1:16]]

# Stores which disk is on top of which disk
# If disk 3 is on disk 5, then on[3, 5] holds
fluent on[int[1:16], int[1:16]]

# Stores how many disks are stacked on one tower,
# used for checking if a request has been fulfilled
fluent count[Tower, int[0:16]]

signal moveDisk(Tower $from, Tower $to, int[1:16] $disk)

# Move a disk from one tower to another
action move(Tower $from, Tower $to) {
  poss {
    $toBelow;
    $fromBelow;
    pick [$from, $fromTop] in top;
    if(top[$to, _]) {
      pick [$to, $toBelow] in top;
      $fromTop < $toBelow;
    } else {
      $toBelow = 0;
    }
    if(on[$fromTop, _]) {
      pick [$fromTop, $fromBelow] in on;
    } else {
      $fromBelow = 0;
    }
    pick [$to, $toCount] in count;
    pick [$from, $fromCount] in count;
  }
  do {
    top -= [$from, $fromTop];
    top += [$to, $fromTop];
    if($fromBelow != 0) {
      $fromBelow = int[1:16] $fromBelow;
      top += [$from, $fromBelow];
      on -= [$fromTop, $fromBelow];
    }
    if($toBelow != 0) {
      $toBelow = int[1:16] $toBelow;
      top -= [$to, $toBelow];
      on += [$fromTop, $toBelow];
    }
    count -= {[$to, _], [$from, _]};
    count += {[$to, int[0:16] ($toCount + 1)], [$from, int[0:16] ($fromCount - 1)]};
  }
  signal moveDisk($from, $to, $fromTop)
}
```

```
# Called by the Q3 script
action init() {
  poss {
    # Get the total number of disks
    pick [$disks] in totalDisks;
  }
  do {
    # Stack all disks in correct order on tower A
    top += [Tower.A, 1];
    count = {[_, 0]};
    count -= [Tower.A, _];
    count += [Tower.A, $disks];
    $i = 1;
    while ($i < $disks) {
      on += [int[1:16] $i, int[1:16] ($i + 1)];
      $i = $i + 1;
    }
  }
}

proc control(int[1:16] $diskCount) {
  # All disks must be on Tower.C
  while (!count[Tower.C, $diskCount]) {
    # Condition $from != $to for performance gains
    pick [$from, $to] in [Tower, Tower] such $from != $to;
    move($from, $to);
  }
}

# Randomly move disks around
proc controlConditional(int[1:16] $diskCount) {
  while (!count[Tower.C, $diskCount]) {
    # Check precondition of move() in this pick statement
    pick [$from, $to] in [Tower, Tower] such
      $from != $to &&
      (
        exists [$from, $disk1] in top such
          top[$to, _] -> (exists [$to, $disk2] in top such $disk1 < $disk2)
      );
    move($from, $to);
  }
}

proc main(boolean $planningEnabled, boolean $conditional) {
  pick [$disks] in totalDisks;
  if($conditional) {
    if($planningEnabled) {
      search controlConditional($disks);
    } else {
      controlConditional($disks);
    }
  } else {
    if($planningEnabled) {
      search control($disks);
    } else {
      control($disks);
    }
  }
}
```

A.4 Packet Delivery Robot

```
ignoreunused(main, S0)

enum Loc {
  A, B, C, D, E, F, G, H, I,
}

enum Packet {
  P1, P2, P3, P4, P5, P6, P7, P8, P9, P10,
}

# Specifying all traversible edges on the map
fact edge[Loc, Loc] {
  [Loc.A, Loc.B],
  [Loc.A, Loc.D],
  [Loc.B, Loc.C],
  [Loc.B, Loc.D],
  [Loc.C, Loc.E],
  [Loc.D, Loc.E],
  [Loc.D, Loc.F],
  [Loc.F, Loc.H],
  [Loc.G, Loc.H],
  [Loc.H, Loc.I],
}

# The location the robot is at
fluent robotAt[Loc]

# The current location of each packet
fluent packetAt[Packet, Loc]

# All packets the robot is currently holding
fluent holding[Packet]

# The number of packets the robot is holding
fluent holdingCount[int[0:2]]

# Where packets should be brought to
fluent requests[Packet, Loc]

signal sigMove(Loc $to)
signal sigPickup(Packet $packet)
signal sigDrop(Packet $packet)

action move(Loc $to) {
  # Can only move to a neighbouring node
  poss (exists [$from] in robotAt such edge[$from, $to] || edge[$to, $from])
  do {
    robotAt = [$to];
  }
  signal sigMove($to)
}

action pickup(Packet $packet) {
  poss {
    # Robot must hold less than 2 packets and must be at the
    # packet's location.
    pick [$count] in holdingCount;
    $count < 2;
    exists [$loc] in robotAt such packetAt[$packet, $loc];
  }
  do {
    packetAt -= [$packet, _];
    holding += [$packet];
    holdingCount = [int[0:2] ($count + 1)];
  }
  signal sigPickup($packet)
}

action drop(Packet $packet) {
  poss {
    # Packet must be held
    pick [$count] in holdingCount;
    pick [$loc] in robotAt;
    holding[$packet];
  }
  do {
    packetAt += [$packet, $loc];
    holding -= [$packet];
    holdingCount = [int[0:2] ($count - 1)];
  }
  signal sigDrop($packet)
}

# Called when a request has been fulfilled
action removeRequest(Packet $packet) {
  do {
    requests -= [$packet, _];
  }
}

# Finds a path to any given location. Behaves differently depending on the
# current semantics. In online mode the robot strays around until it has
# stumbled upon the goal. In offline mode the shortest path is found.
proc findPath(Loc $to) {
  pick [$current] in robotAt;
  while($current != $to) {
    pick [$next] in [Loc] such edge[$current, $next] || edge[$next, $current];
    move($next);
    $current = $next;
  }
}

proc control() {
  while(requests[_, _]) {
    choose {
      # Go to a packet that must be delivered and pick it up
      pick [$packet, $src] in packetAt such requests[$packet, _];
      search findPath($src);
      pickup($packet);
    } or {
      # Go to the destination of a held packet and drop it
      pick [$packet, $dst] in requests such holding[$packet];
      search findPath($dst);
      drop($packet);
      removeRequest($packet);
    }
  }
}

proc controlConditional1() {
  # Same as control(), but with precondition checks
  while(requests[_, _]) {
    choose {
      if(!holdingCount[2] && (exists [$packet, _] in packetAt such requests[$packet,
    _])) {
        pick [$packet, $src] in packetAt such requests[$packet, _];
        findPath($src);
        pickup($packet);
      }
    } or {
      if(exists [$packet, _] in requests such holding[$packet]) {
        pick [$packet, $dst] in requests such holding[$packet];
        findPath($dst);
        drop($packet);
        removeRequest($packet);
      }
    }
  }
}

# Answer requests iteratively
proc controlConditional2() {
  while(requests[_, _]) {
    # Go to a packet, pick it up, go to its destination and drop it
    pick [$packet, $dst] in requests;
    pick [$packet, $src] in packetAt;
    search findPath($src);
    pickup($packet);
    search findPath($dst);
    drop($packet);
    removeRequest($packet);
  }
}

proc main(boolean $planningEnabled, boolean $conditional) {
  if($conditional) {
    if($planningEnabled) {
      # Strategy 1
      controlConditional1();
    } else {
      # Strategy 2
      controlConditional2();
    }
  } else {
    if($planningEnabled) {
      # Strategy 3
      search control();
    } else {
      # Fails most of the time
      control();
    }
  }
}

situation S0 {
  # Robot is located at A in the beginning and holds no packets
  robotAt = {[Loc.A]};
  holding = {};
  holdingCount = {[0]};
}
```