Portable High-level Agent Programming with golog++

Victor Mataré[1][a], Tarik Viehmann[2][b], Till Hofmann[2][c], Gerhard Lakemeyer[2][d], Alexander Ferrein[1][e] and Stefan Schiffer[1][f]

[1]*Institute for Mobile Autonomous Systems and Cognitive Robotics, FH Aachen University of Applied Sciences, Aachen, Germany*

[2]*Knowledge-based Systems Group, RWTH Aachen University, Aachen, Germany*

{matare, ferrein, s.schiffer}@fh-aachen.de, {hofmann, viehmann, gerhard}@kbsg.rwth-aachen.de

Keywords: Robotics, Agents, Planning, Constraints, Abstraction, Golog, Action Logic.

Abstract: We present golog++, a high-level agent programming and interfacing framework that offers a temporal constraint language to explicitly model layer-penetrating contingencies in low-level platform behavior. It can be used to maintain a clear separation between an agent's domain model and certain quirks of its execution platform that affect problem solving behavior. Our system reasons about the execution of an abstract (i.e. exclusively domain-bound) plan on a particular execution platform. This way, we avoid compounding the complexity of the planning problem while improving the modularity of both golog++ and the user code. On a run-through example from the well-known blocksworld domain, we demonstrate the entire process from domain modeling and platform modeling to plan transformation and platform-specific plan execution.

1 INTRODUCTION

Conceptually, a high-level control program makes a robot execute a sequence of actions that achieves a certain goal, where each action is some elementary step that manipulates the environment in a predictable manner. A (formal) description of the preconditions and effects of such actions is often called a *domain model* (or *domain theory*), and it is supposed to cover the robot's environment with its specific tasks and challenges. In particular, a description of the *domain* is not supposed to be tied (i.e., refer in any way) to concepts that are specific to the robot *platform* (i.e., its hardware, software and their limitations).

In practice however, it turns out that a perfect separation between an agent's environment and its platform is unattainable. Every hardware platform has specific properties and limitations, which cannot simply be abstracted by a reactive layer below the agent level.

As an example, consider an RGB-D camera used for object recognition. Such cameras often actively project an infrared pattern from which depth information is reconstructed, so they are notoriously power-hungry and may also cause disturbance in other IR-sensitive devices in the environment. As a result, such cameras need to be switched off when they are not being used. When switching them on, they typically take a few seconds to initialize, so in order to not cause delays, we want to switch it on shortly *before* it will be used.

For this reason, switching the camera on and off is a platform-specific behavior that can only be sensibly integrated at the strategic agent level. There are approaches that model everything from the domain theory down to its mapping onto a particular platform as one unified planning problem. However, this can result in a significant increase in the domain size and thus the search space size, thereby impairing planner performance.

As an alternative approach, (Hofmann et al., 2018) proposed to separate the platform model from the domain theory such that the domain theory is abstract and independent of the underlying platform. The platform model describes the platform components with timed automata. Temporal constraints describe the relationship between domain theory and platform model, e.g., by stating that two seconds before doing a `scan` action, the camera should be switched on.

golog++ is a GOLOG development framework

[a] https://orcid.org/0000-0003-4606-4758
[b] https://orcid.org/0000-0003-0264-0055
[c] https://orcid.org/0000-0002-8621-5939
[d] https://orcid.org/0000-0002-7363-7593
[e] https://orcid.org/0000-0002-0643-5422
[f] https://orcid.org/0000-0003-1343-7140

that decouples the language syntax, the runtime semantics and the platform interfacing. In this paper, we describe the runtime semantics of the temporal constraint language we offer as a means to explicitly model how platform-specific quirks and contingencies interact with a platform-independent domain theory.

We begin with some background on the GOLOG language family and other related work in Section 2. Section 3 is the main part of this paper, where we build up an application example based on the well-known Blocksworld domain. We start by introducing a simple Basic Action Theory (BAT) in Section 3.1, and continue in Section 3.2 with a `golog++` main procedure that solves a problem within this domain. Section 3.3 then introduces our platform modeling language by linking the above-mentioned RGB-D camera problem to the Blocksworld BAT. Section 3.4 roughly describes one particular method of transforming a platform-independent plan into a platform-specific schedule given the domain theory and the platform model shown before. The example run-through is finished in Section 3.5 with more detailed account of how a transformed schedule is executed and how platform consistency is maintained at runtime.

2 BACKGROUND & RELATED WORK

As mentioned before, advances in hybrid planning systems (Halsey et al., 2004; Shu et al., 2005; Eyerich et al., 2012, etc.) make it possible to treat a domain theory along with temporal platform contingenciens as a single, integrated planning problem (Stock et al., 2015; Dvorak et al., 2014). There is however no way around the fact that this approach can compound the problem problem complexity to the point that an otherwise solvable problem becomes intractable.

So introducing some kind of separation between domain and platform is a natural choice, and the relation between the two is being investigated with diverse goals and requirements. (Konečnỳ et al., 2014) introduce a concept they call *execution semantics* which is used for consistency monitoring during execution. It is based on Allen's Interval Algebra (Allen, 1983), much like the groundwork for our endeavour (Schiffer et al., 2010). Their goals are however different from ours in that they aim to increase plan robustness with regard to changes in the world state. (Kunze et al., 2011) leverage the Web Ontology Language (Bechhofer et al., 2004) to define a language called Semantics Robot Description Language (SRDL) to map a task-level strategy onto kinematic description languages.

The roots of our work can be traced back to the Situation Calculus (McCarthy, 1963; McCarthy and Hayes, 1969), SitCalc for short. The SitCalc is a second-order logical language with equality that is designed to reason about actions and their effects. A situation in the world is the result of a sequence of actions given an initial situation s_0. The preconditions of actions are described in terms of so-called *fluents*: Predicates that carry a situation term as their last argument. Successor state axioms specify which fluents hold true in a given situation s. This idea spawned the first GOLOG language (Levesque et al., 1997), where a nondeterministic imperative program is combined with a so-called *Basic Action Theory* that specifies a SitCalc-like model of the problem domain. This allows an interpreter to ground the nondeterministic choices and obtain a grounded action sequence that can be executed.

Since then, the GOLOG language family has grown significantly, with newer dialects increasing the capabilities of the original language in different directions. CONGOLOG (De Giacomo et al., 2000) introduced concurrent execution and exogenous actions, allowing an agent to react to events not triggered by itself. INDIGOLOG (De Giacomo et al., 2009) made the resolution of nondeterminisms (i.e. planning) more explicit and thereby increased capabilities for incremental planning. DTGOLOG (Boutilier et al., 2000) introduced the capability to optimize planned choices against a reward function, thus introducing decision-theoretic planning. READYLOG (Ferrein and Lakemeyer, 2008) integrated features from DTGOLOG, PGOLOG and CCGOLOG (Grosskreutz and Lakemeyer, 2000; Grosskreutz and Lakemeyer, 2003) to combine decision-theoretic planning with uncertain action outcomes and fluents that change continuously over time. The problems associated with active sensing, resource management and execution monitoring have also been touched early in the GOLOG evolution (Hähnel et al., 1998; Lakemeyer, 1999).

Despite all the progress in GOLOG language semantics, there has been surprisingly little effort put into the more engineering-related issues like language usability, platform interfacing and maintainability of both user code and interpreter implementations. This effectively puts the GOLOG world at a disadvantage towards better integrated agent languages like PDDL or PRS. Important progress towards language usability and integration has been made by golog.lua (Ferrein, 2010) and the YAGI dialect (Ferrein et al., 2012), the latter of which has inspired the the `golog++` archi-

tecture (Mataré et al., 2018) and its action execution interface (Kirsch et al., 2020).

While earlier explorations of the way we handle temporal platform contingencies (Schiffer et al., 2010) were still based on Allen's Interval Algebra (Allen, 1983), this work is based on a more expressive temporal language called *t-ESG* (Hofmann and Lakemeyer, 2018; Hofmann et al., 2018), which embeds the Metric Temporal Logic (Koymans, 1990, MTL for short) into *ESG* (Claßen and Lakemeyer, 2008).

3 PLATFORM ABSTRACTION IN golog++

In golog++ (same as in many other GOLOG dialects), programmers have the freedom to interleave planning with imperative code. Planning is done over regular imperative code where some parameter choices are made nondeterministically. Wrapping an imperative code block in a `solve(f, h){...}` statement lets golog++ resolve all nondeterminisms in such a way that the cumulative reward function `f` is maximized given the maximum search depth (horizon) `h`. This gives programmers the freedom to, for instance, generate many short plans that solve a problem step by step, to use a fixed plan library, or even to rely only on scripted behaviour and not use planning at all.

In classical GOLOG, actions are completed *instantaneously*. In contrast, all domain actions defined in golog++ (as shown in the next section) are implicitly *durative*. That means they have a *begin*, a *duration* and an *end*, similar to the concept of durative actions introduced by (Reiter, 1996). So when a user defines an `action a()`, this durative definition spawns the instantaneous actions `start(a())` and `end(a())`. The precondition of `a()` becomes the precondition of `start(a())`, and the precondition of `end(a())` depends on a fluent that can only be set by an exogenous action triggered by the *platform backend* (cf. fig. 2).

This reflects the fact that an agent generally cannot control if and when a robot platform is finished executing a certain task. For example, an agent may of course cancel a robot's movement while it is on its way to a certain target, but it cannot simply stipulate that it has now arrived. So when a programmer calls `a();`, that call is internally expanded into `{start(a()); end(a());}`, although these instantaneous actions can also be called explicitly.

If the precondition of any action is not satisfied, execution will block at that point and wait for an exogenous event to change the world state before testing the precondition again.

Apart from the platform modeling features described in sections 3.3 to 3.5, what sets golog++ apart from earlier implementations is its modular architecture and vastly improved language usability. All structural assumptions are explicitly represented in abstract interfaces, and behavioral assumptions between major components are kept to an absolute minimum. This means for example that development on language semantics can be done independently from all of the other concerns.

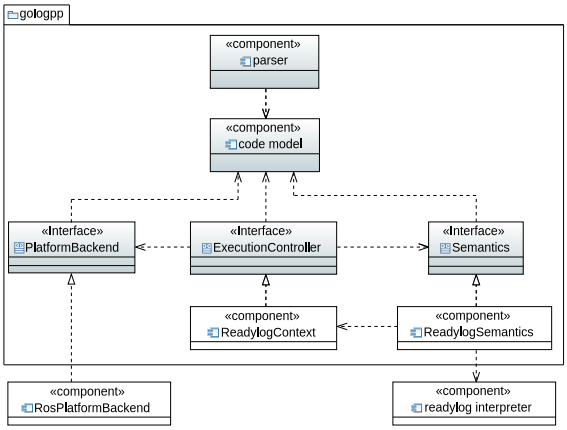

Figure 1: An overiew of the golog++ component architecture. Shaded boxes represent the core architecture, while the plain boxes are exchangeable. For more background and motivation on this architecture, see (Mataré et al., 2018).

Only the shaded boxes in fig. 1 make up the golog++ core architecture. Everything within the "gologpp" package container is shipped with the source distribution, but the white boxes represent components that are intended to coexist with alternative implementations.

The coexistence with alternatives is especially relevant for the *platform backend*: golog++'s interface to a real execution platform (i.e., a robot, a simulation, etc). It typically leverages the functionality of some robotics middleware framework like ROS (Quigley et al., 2009) or Fawkes (Niemueller et al., 2010) to dispatch and monitor action execution and to asynchronously notify golog++ of any relevant exogenous events (Kirsch et al., 2020). Additionally, it supplies a clock source with a timed wakeup mechanism and manages low-level components according to the platform models shown later in section 3.3.

Regarding language usability, golog++ improves on the state of the art by (1) defining an intuitive yet strict language syntax and implementing it in a parser that gives precise error messages, and by (2) making the language typesafe. In combination, these two features allow us to statically verify referential integrity

and thereby catch the most common programming errors before any code is executed.

3.1 The Basic Action Theory Σ

Fundamental to every golog++ program (as in all GOLOG dialects) is the Basic Action Theory (BAT). A BAT Σ describes the domain, mainly in terms of what actions can be performed, what preconditions they have and how they will affect the state of the world.

golog++ uses a notation that is inspired by the C++ syntax. Being a syntax that is optimized for readability (as opposed to grammar size), the full grammar of the language is too large to be reproduced here. That is why we limit ourselves to giving code examples which should suffice to convey an intuition about the language syntax.

Since golog++ is an explicitly, statically type-safe language, all expressions have a fixed type. A **compound** type definition specifies a key-value structure, and can be freely defined and nested by the user. **list** types are also user-defined and are likewise arbitrarily nestable both in each other and in compound types. A **domain** type defines a named subset of values from some other type, typically to enumerate relevant domain elements. In the *blocksworld* scenario, we can use domains of symbols to represent blocks and locations:

```
symbol domain Block = { A, B, C }
symbol domain Location =
    Block | { Table, Home, Unknown }
```

An expression of the **symbol** type can be thought of as an unquoted string that has to be defined as a member of some domain before it can be referenced. So any expression of type `Block` can take the values `A`, `B` or `C`, and any expression of type `Location` can take the values `A`, `B`, `C`, `Table`, `Home` or `Unknown`. With these restrictions, we can define a functional fluent that evaluates to the current location of a given `Block`:

```
Location fluent loc(Block x) {
initially:
    loc(A) = Table;
    loc(C) = Table;
    loc(B) = C;
}
```

In addition to defining the fluent's type signature, this statement also defines the *initial situation* s_0 with regard to the fluent `loc(Block)`. Using anything other than a member of the `Block` domain as an argument or assigning anything other than a `Location` as a value will lead to a static error being raised.

This is already a highly specific description of the world state for our simplified blocksworld scenario. To complete our BAT, all we need is a description of how our agent can affect the world state. That is, we need an action that moves a `Block x` from its current location to another `Location y`:

```
action put_block(Block x, Location y) {
effect:
    loc(x) = y;
precondition:
    x != y & loc(x) != y
    & (!exists(Block z) loc(z) == x)
    & (y == Table
       | !exists(Block z) loc(z) == y)
duration:
    [4, 10]
}
```

Note how the current location is absent from the action's signature and how we are able to encode the precondition without resorting to additional helper predicates like *Free(x)* that are required e.g. in a PDDL encoding of the Blocksworld domain. So in detail, the action's precondition says that we cannot stack a `Block x` on itself, that its current location must be different from `y`, and that there cannot be another `Block z` on top of either `x` or `y`, except if `y` is the `Table` (we can put all we want on the `Table`).

For demonstration purposes, we also introduce two other actions. The `go_to(...)` action makes the robot drive to a certain location using collision avoidance, while `align_to(...)` will align it precisely so as to be able to perceive and manipulate objects. We're omitting the definitions of these two actions here because for the purpose of this paper we are only concerned about their interaction with the platform model as shown further below.

3.2 The Main Procedure δ_0

As mentioned in the introduction, GOLOG-based languages do not automatically search the entire action space for a solution to a certain goal. Instead, the programmer writes an imperative program δ_0 that *can* include nondeterministic elements. These nondeterministic elements make up the search space for a planning operator that resolves them so that the code block nested into it becomes executable[1]. One example of such a nondeterminism is the `pick(T var) CODE(var);` statement. It tells golog++ to assign a value from the type `T` to the variable `var` for which `CODE(var)` is executable. Combined with classical imperative constructs, this can be used to write a program that solves a Blocksworld problem like a classical planner would:

[1] "Executable" here means that the preconditions of all executed actions hold true.

```
bool function goal() =
  loc(A) == Table & loc(B) == A
  & loc(C) == B

number function reward() =
  if (goal())
    100
  else
    -1
```

The `goal()` function decides whether the desired situation has been achieved, and the `reward()` function gives a large payout in the `goal()` situation while slightly penalizing all other situations.

```
procedure main() {
  solve(16, reward()) {
    go_to(Table);
    align_to(Table);

    while (!goal())
      pick (Block x) pick (Location y)
        put_block(x, y);

    go_to(Home);
  }
}
```

The `main()` procedure shown above combines deterministic action calls (`go_to(Table)`, `align_to(Table)` and `go_to(Home)` with a nondeterministic parameter choice within the `while (...) {...}` loop. The result is a plan that always begins with the two hard-coded actions and ends with the hard-coded `go_to(Home)`, but has a variable sequence of `put_block(x, y)` actions in the middle that realizes the shortest path to a situation that satisfies the `goal()` function. The two `pick(...)` statements are responsible for nondeterministically binding the variables `x` and `y`. Without the enclosing `solve(...)` block, parameter assignments would be picked at random and every action would be dispatched for execution immediately (*online* execution in GOLOG jargon). The `solve(...)` block enables *offline* execution: Actions are not dispatched to the platform backend, only their preconditions are checked and their effects are applied, forming a tree of situations. The entire `while(...)` block is executed this way until it terminates or until the maximum search depth (here `16` actions) is reached. If an action's preconditions are not satisfied, that branch is not expanded further, and penalizing any non-goal situation results in the shortest plan giving the highest cumulative reward.

For illustration's sake, say we want to formulate a different strategy: Maybe we have a lot of space on the table and manipulation works well, so we think it's best to first unstack every block and spread them out on the table, and *only then* stack them back up in the desired configuration. Then we could just "transcribe" that strategy in two nondeterministic loops:

```
procedure main2() {
  solve(22, reward()) {
    go_to(Table);
    align_to(Table);

    while (exists(Block x) loc(x) != Table)
      pick (Block x)
        put_block(x, Table);

    while (!goal())
      pick (Block x) pick (Block y) {
        test(loc(x) == Table);
        put_block(x, y);
      }

    go_to(Home);
  }
}
```

While this solution may look more complicated at first glance, it actually results in a less complex planning problem. It encodes sort of a divide-and-conquer strategy, sacrificing solution optimality for planning speed: In `main2()`, the first loop only uses a branching factor of 1, making that search trivial. Then in the second loop, blocks can only be moved onto other blocks and the `test(...)` statement further restricts the search to blocks which are still on the table, effectively limiting the search depth to the number of blocks in existence. Depending on the initial situation and the goal, the generated plan will likely contain a number of actions that could have been saved (think of stacking A-B-C into C-B-A for example), but the planning will now scale linearly with the number of blocks instead of quadratically.

3.3 The Platform Model Π

The example code shown so far is concerned purely with the domain semantics. To execute it efficiently on a particular robot platform, certain layer-penetrating details of lower-level components may have to be considered. This phenomenon has been observed across many areas of software development (i.e. not just in robotics) and is being discussed under the term *leaky abstraction* (Spolsky, 2002).

In Section 1, we mentioned a depth camera as an example of a hardware component that exhibits behaviour that cannot be completely "abstracted away". Here our platform modeling language can be used to describe precisely what hardware-specific behaviour should be accounted for:

```
component depth_cam {
    clocks: bcl
    states: off, boot (bcl < 4), on, error
    transitions: off => boot resets(bcl),
                 boot ->(bcl > 2) on,
                 on => off,
                 error => off
}
```

The *depth_cam* component is modeled as a timed automaton according to (Alur and Dill, 1994). The distinctive feature of timed automata is that they can have an arbitrary number of clocks that all count upward with a uniform period and can be reset by certain transitions. Both states and transitions can then be guarded by simple conditions over these clocks.

Here, the model says that the component can only remain in the *boot* state as long as the *bcl* clock counts less than **4** (seconds in our case). Switching from *off* to *boot* resets *bcl* to **0**, and the transition from *boot* to *on* can only be taken after at least **2** seconds have passed on *bcl*. The thin arrow (**->**) on the transition from *boot* to *on* also indicates that this is an *exogenous* transition, meaning that it is not triggered by the agent, but by the component itself. The agent cannot influence if and when it happens: it can merely acknowledge the fact that it did. This is the timed automata equivalent of saying "The camera takes 2-4 seconds to boot". Now all we need to say is that we want the *depth_cam* to be in the *on* state during any *put_block(...)* action:

```
constraints {
    during(put_block(*, *)):
        state(depth_cam) = on;
    during(go_to(*)):
        state(depth_cam) = off;
}
```

The set of all component automata combined with the set of all constraint formulas is called the *platform model* Π. Generally speaking, a constraint formula is an implication $\phi \supset \psi$, where ϕ is a formula with *t-ESG* semantics that refers only to action terms from the domain theory Σ, and ψ is formula that refers only to component states from the platform model Π. We call ϕ the *action spec* and ψ the *state spec*. In the golog++ notation shown above, the colon "**:**" represents the implication sign "$\supset$". All temporal operators can optionally be suffixed with temporal bounds **[t,u]**. Other than **during**, golog++ also supports the temporal operators **previous**/**next**, **future**/**past**, and **since**/**until**. The semantics of such temporal formulas and their relation to the situation calculus are defined in (Hofmann and Lakemeyer, 2018).

3.4 Plan Transformation

The plan produced by $\langle \Sigma, \delta_0 \rangle$ is platform-agnostic and doesn't specify temporal boundaries. The plan transformation takes this as input and attempts to satisfy all platform constraints by inserting maintenance actions and assigning a $[t_{min}, t_{max}]$ time window to each action. The transformation specifically is not supposed to alter the initial action sequence in any way, because the platform control is assumed to be completely disjoint from the domain of reasoning, resulting in a clear separation between platform specifics and the agent BAT as in (Hofmann et al., 2018; Hofmann and Lakemeyer, 2018). In combination with the simple structure of platform constraints that demand platform control solely based on the occurrence of certain domain actions, this allows us to treat the plan transformation as an isolated task only being concerned with respecting

1. the order and durations of actions according to the initial plan
2. the temporal features of the platform model (guards and invariants of the modeled TAs without exogenous transitions as the framework has no control over them)
3. the desired platform control according to the platform constraints.

Any transformation procedure capable of dealing with these requirements may be suitable for our application. One particular procedure developed with our use case in mind is presented in (Viehmann, 2019). There the task is tackled by encoding the entire platform model (i.e. both the component automata and the constraints) as a reachability problem over timed automata. Despite being a PSPACE-complete problem (Alur and Dill, 1994), this is often solvable quite efficiently with modern techniques. The rough idea is to construct a single automaton with a designated final state, such that every path to that state corresponds to a possible transformed plan together with a range of possible action time groundings, from which any one particular may be computed.

Generally there is no single unique solution to such a plan transformation. Solutions may differ in the order or types of inserted maintenance actions and (most likely) in the time windows assigned to each action.

3.5 Execution

During execution, the main program δ (initially $\delta = \delta_0$) is traversed by the *Controller* (cf. Figure 2).

The basic idea is that of an event loop, which is realized by a blocking queue q_{exog}. First, all pending exogenous events are processed in a non-blocking manner, i.e. execution continues as soon as q_{exog} is empty.

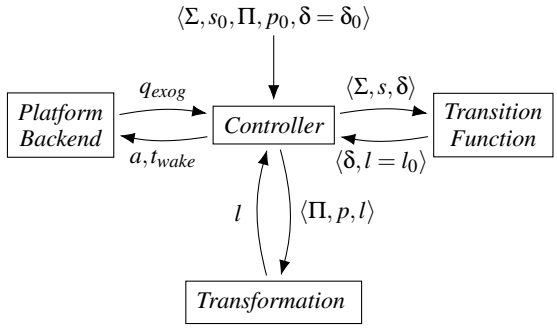

Figure 2: Flow of events and information between the major components involved in program execution.

The *Transition* function is responsible for interpreting program elements up to the point where the next online action must be executed. It returns the remaining program δ and the resulting abstract plan l_0. If δ calls the next action from outside a planning block (*online*), l_0 is a trivial plan that contains just this single action. If a planning block is encountered, the *Transition* function will plan through it and return the resulting plan l_0. If the end of the planning block was reached, l_0 is valid and the planning block is removed from the remaining program δ. Note that a valid plan could also be the empty plan `{}`.

If the transition function does not reach the end of the planning block, that means no valid plan could be found in the current situation s. The remaining program δ is not changed, l_0 is `null` (i.e. invalid) and we must wait for the environment to change (i.e. block on q_{exog}) before trying again.

Given the $\langle \Sigma, s_0, \delta_0 \rangle$ described above, the transition function will generate the following abstract plan l_0:

```
{
  start(go_to(Table));
  end(go_to(Table));
  start(align_to(Table));
  end(align_to(Table));
  start(put_block(B, A));
  end(put_block(B, A));
  start(put_block(C, B));
  end(put_block(C, B));
  start(go_to(Home));
  end(go_to(Home));
}
```

The durative domain actions have been expanded into instantaneous `start(...)` and `end(...)` actions and the nondeterministics `pick(...)` statements have been grounded such that the plan produces the maximum cumulative `reward()`.

The abstract plan l_0 is transformed according to the platform model Π given the current platform state p. This turns l_0 into a schedule l that satisfies the platform constraints. In our example, the result will resemble the following:

```
{
  [0,32748] start(go_to(Table))
  [0,32768] end(go_to(Table))
  [1,21] switch_state(depth_cam, off, boot)
  [1,5] start(align_to(Table))
  [3,5] switch_state(depth_cam, boot, on)
  [3,13] end(align_to(Table))
  [13,15] start(put_block(B, A))
  [15,19] end(put_block(B, A))
  [17,19] start(put_block(C, B))
  [19,23] end(put_block(C, B))
  [19,23] switch_state(depth_cam, on, off)
  [21,23] start(go_to(Home))
  [21,43] end(go_to(Home))
}
```

Each action a has been annotated with a time window `[`t_{min}`,`t_{max}`]` for its execution, and platform maintenance actions have been inserted to satisfy platform constraints.

The constraints say that the `depth_cam` must be `off` during any `go_to(...)` action, but they say no such thing about the `align_to(...)` action. Therefore, the transformation can insert the `switch_state(depth_cam, off, boot)` action only *after* the `go_to(...)` has ended. In this case, it coincides with the start of the `align_to(Table)` action, likely giving the device enough time to complete its boot procedure while the robot is aligning to the table. Next up is an exogenous transition from `boot` to `on`, so the action `switch_state(depth_cam, boot, on)` will block until the `depth_cam` component has actually finished its boot procedure. The same is true for `end(align_to(Table))`. Once both actions have been executed, we can start stacking blocks according to the solution found by the `solve(...){...}` block. Afterwards, another `go_to(...)` action is scheduled, so the transformation has arranged for the `depth_cam` to be switched off before it.

Before any action is executed, its time window is checked against the current time: If it is still too early, a timer event at $t_{wake} = t_{min}$ is scheduled on the *Platform Backend*. Then, the *Controller* blocks on q_{exog}. When the timer event fires at the wall time t_{wake}, the *Platform Backend* wakes the *Controller* up by enqueueing an exogenous event `step_context_time(`$t_{wake} + \varepsilon$`)` to q_{exog}. Any exogenous event (including the timer event) will be consumed when it comes up and unblock the *Controller*. Other possible events include exogenous component state changes and any exogenous actions defined by the BAT Σ.

In case the platform state p did change exogenously, we trigger another *Transformation* before attempting to execute the next action a. This re-transformation may insert new maintenance actions, even at the beginning of the plan to maintain or restore platform consistency, effectively changing the next scheduled action a. If and when a finally becomes possible, it is dispatched to the *Platform Backend*, has its effects applied to the current situation and is removed from the remaining schedule l. After this, schedule execution will continue in this way until the schedule is empty, at which point the remaining program δ will continue to be evaluated if it is non-empty.

In the real world, this loop will actually spend most of its wall time being blocked on q_{exog}, waiting for an action to end, waiting until a certain point in time or for any other kind of exogenous event that may make the next action a executable.

4 EVALUATION

In 2018/2019 `golog++` was used by students in a lab course to develop an agent for the RoboCup Logistics League (RCLL Technical Committee, 2018; Niemueller et al., 2015). The students used the Logistics League simulation in a non-adversarial setup (one team of 3 cooperating robots on the playing field). In the Logistics League, robots score points by using the playing field to manufacture and deliver products ordered by a central game controller (the referee computer or *RefBox*). Ordered products vary in their delivery time windows and in their complexity, with the more complex ones requiring forty or more interactions with the playing field.

In the beginning, we could observe the students intuitively employing a trial-and-error behavior while they familiarized themselves with the basic syntax and semantics of the `golog++` language. At this stage, much of the learning process was being driven by the error messages generated by the parser and the type system.[2] Later during the semester, as students' proficiency grew along with their code bases, their focus shifted to experimenting with different team coordination strategies and how to employ incremental planning effectively given the uncertainties inherent in the domain.

[2]Note that this would not have been possible with any of the classical Prolog-based GOLOG dialects because they neither define a GOLOG-specific syntax nor do they check for any level of semantic coherence (static semantics). With these, a self-guided, experimental learning process would have been impossible because many typical beginner's mistakes (syntax errors, parameter mismatches etc.) yield undefined behavior instead of helpful error messages.

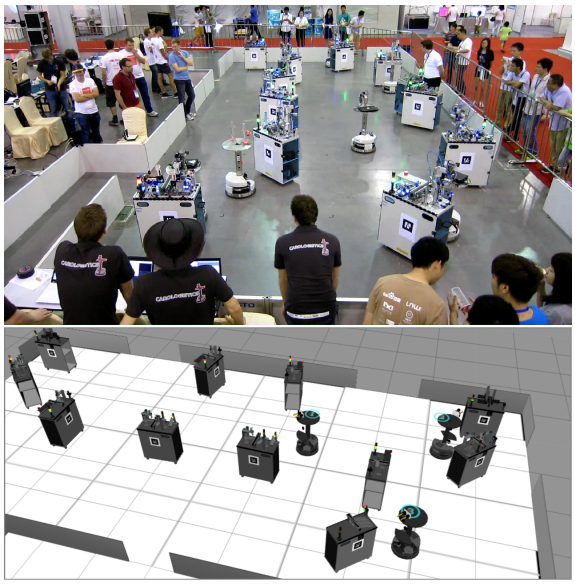

Figure 3: **Top**: Playing field of the RoboCup Logistics League (RCLL), seen here during the RoboCup 2015 in China. The robots (circular bodies) use the Festo MPS machines (rectangular boxes with AR tags) to manufacture and deliver ordered products. The MPS machines are movable and the playing field layout changes with each game. **Bottom**: RCLL simulation. Picture source and further league details see (Niemueller et al., 2016).

The fact that students with little previous programming experience and little to no background in knowledge-based multi-agent development were able to get to this stage shows that `golog++` development can be self-taught sustainably and that it can be a helpful tool in teaching knowledge-based high-level agent programming.

5 CONCLUSION

We have shown how to solve a domain-specific problem in a platform-conformant manner while maintaining a strict separation between domain and platform on all levels. In broader terms, our architecture reflects the fact that while an individual is never entirely separable from its environment, the relationship between both still follows a different logic than that of the environment itself. That is why we think it is both natural and effective to have different formalisms to model a platform (here: timed automata) and a domain (here: GOLOG), and to describe their relationship in a language (here: t-$\mathcal{ESG}$ constraints) whose semantics overlap with both.

We have shown how the execution controller always strives to keep the runtime behavior consistent

with the platform model, even in the presence of unplanned behavior in low-level components. The framework puts particular emphasis on a clean architecture: In user code through type safety and the platform/domain separation, and in its implementation through strict separation of the interfacing, representation, interpretation and execution concerns. First lab tests with uninitiated users give us confidence that both the language design and the system architecture are comprehensible and sustainable.

It will be of particular interest to study how this new architecture helps with the development of error recovery strategies and with porting existing domain models to new robot platforms.

ACKNOWLEDGMENTS

This work was supported by the German National Science Foundation (DFG) under grant numbers GL-747/23-1 and FE 1077/4-1 and by Germany's Excellence Strategy – EXC-2023 Internet of Production – 390621612.

REFERENCES

Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843.

Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical computer science*, 126(2):183–235.

Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. (2004). OWL Web Ontology Language Reference. W3C Recommendation, World Wide Web Consortium. http://www.w3.org/TR/owl-ref/.

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

Claßen, J. and Lakemeyer, G. (2008). A Logic for Non-Terminating Golog Programs. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 589–599.

De Giacomo, G., Lespérance, Y., and Levesque, H. J. (2000). ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121.

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

Dvorak, F., Bit-Monnot, A., Ingrand, F., and Ghallab, M. (2014). A flexible ANML actor and planner in robotics.

Eyerich, P., Mattmüller, R., and Röger, G. (2012). Using the context-enhanced additive heuristic for temporal and numeric planning. In *Towards Service Robots for Everyday Environments*, pages 49–64. Springer.

Ferrein, A. (2010). Golog.lua: Towards a non-prolog implementation of GOLOG for embedded systems. In Hoffmann, G., editor, *Proceedings of the AAAI Spring Symposium on Embedded Reasoning*, (SS-10-04), pages 20–28. AAAI Press.

Ferrein, A. and Lakemeyer, G. (2008). Logic-based robot control in highly dynamic domains. *Robotics and Autonomous Systems*, 56(11):980–991.

Ferrein, A., Steinbauer, G., and Vassos, S. (2012). Action-based imperative programming with YAGI. In *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Grosskreutz, H. and Lakemeyer, G. (2000). Turning high-level plans into robot programs in uncertain domains. In *ECAI*, pages 548–552.

Grosskreutz, H. and Lakemeyer, G. (2003). ccGolog – A Logical Language Dealing with Continuous Change. *Logic Journal of the IGPL*, 11(2):179–221.

Hähnel, D., Burgard, W., and Lakemeyer, G. (1998). GOLEX — Bridging the Gap between Logic (GOLOG) and a Real Robot. In *Annual Conference on Artificial Intelligence*.

Halsey, K., Long, D., and Fox, M. (2004). CRIKEY - a temporal planner looking at the integration of scheduling and planning. In *Workshop on Integrating Planning into Scheduling, ICAPS*, pages 46–52. Citeseer.

Hofmann, T. and Lakemeyer, G. (2018). A Logic for Specifying Metric Temporal Constraints for Golog Programs. https://kbsg.rwth-aachen.de/~hofmann/papers/timed-esg-cogrob18.pdf.

Hofmann, T., Mataré, V., Schiffer, S., Ferrein, A., and Lakemeyer, G. (2018). Constraint-Based Online Transformation of Abstract Plans into Executable Robot Actions. In *AAAI Spring Symposium: Integrating Representation, Reasoning, Learning, and Execution for Goal Directed Autonomy*.

Kirsch, M., Mataré, V., Ferrein, A., and Schiffer, S. (2020). Integrating golog++ and ROS for Practical and Portable High-level Control:. In *Proceedings of the 12th International Conference on Agents and Artificial Intelligence*, pages 692–699, Valletta, Malta. SCITEPRESS - Science and Technology Publications.

Konečnỳ, Š., Stock, S., Pecora, F., and Saffiotti, A. (2014). Planning Domain + execution semantics: A way towards robust execution? In *2014 AAAI Spring Symposium Series – Qualitative Representations for Robots*. http://www.aaai.org/ocs/index.php/SSS/SSS14/paper/view/7743.

Koymans, R. (1990). Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299.

Kunze, L., Roehm, T., and Beetz, M. (2011). Towards semantic robot description languages. In *IEEE Int'l Conf. on Robotics and Automation (ICRA 2011)*, pages 5589–5595.

Lakemeyer, G. (1999). On sensing and off-line interpret-

ing in golog. In *Logical Foundations for Cognitive Agents*, pages 173–189. Springer.

Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. B. (1997). GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–84. http://www.sciencedirect.com/science/article/pii/S0743106696001215.

Mataré, V., Schiffer, S., and Ferrein, A. (2018). Golog++: An Integrative System Design. In *Proceedings of the 11th Cognitive Robotics Workshop 2018 (CogRob@KR 2018), Co-Located with 16th International Conference on Principles of Knowledge Representation and Reasoning*, pages 29–36. http://ceur-ws.org/Vol-XXX/#paper-06.

McCarthy, J. (1963). Situations, Actions and Causal Laws. Technical report memo 2, AI Lab, Stanford University, California, USA.

McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B. and Michie, D., editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, New York. http://www-formal.stanford.edu/jmc/mcchay69.html.

Niemueller, T., Ferrein, A., Beck, D., and Lakemeyer, G. (2010). Design Principles of the Component-Based Robot Software Framework Fawkes. In *Proc. of the 2nd Int'l Conf. on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2010)*, volume 6472 of *LNCS*, pages 300–311, Berlin, Heidelberg. Springer Berlin Heidelberg.

Niemueller, T., Karpas, E., Vaquero, T., and Timmons, E. (2016). Planning Competition for Logistics Robots in Simulation. In *WS on Planning and Robotics (PlanRob) at Int. Conf. on Automated Planning and Scheduling (ICAPS)*, London, UK.

Niemueller, T., Lakemeyer, G., and Ferrein, A. (2015). The RoboCup Logistics League as a Benchmark for Planning in Robotics. In Finzi, A., Ingrand, F., and Orlandini, A., editors, *Proceedings of the 3rd Worksh Op on Planning and Robotics (PlanRob-15)*, pages 63–68, Jerusalem.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). ROS: An open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, volume 3, page 5. Kobe, Japan.

RCLL Technical Committee (2018). RoboCup Logistics League – Rules and Regulations 2018. http://www.robocup-logistics.org/rules/rulebook2018.pdf?attredirects=0&d=1.

Reiter, R. (1996). Natural Actions, Concurrency and Continuous Time in the Situation Calculus. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, KR'96, pages 2–13, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. http://dl.acm.org/citation.cfm?id=3087368.3087370.

Schiffer, S., Wortmann, A., and Lakemeyer, G. (2010). Self-maintenance for autonomous robots controlled by readyLog. In *Proceedings of the 7th IARP Workshop on Technical Challenges for Dependable Robots in Human Environments*, pages 101–107.

Shu, I.-h., Effinger, R. T., and Williams, B. C. (2005). Enabling Fast Flexible Planning through Incremental Temporal Reasoning with Conflict Extraction. In *ICAPS*, pages 252–261.

Spolsky, J. (2002). The Law of Leaky Abstractions. https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/. Last accessed 2020-09-25.

Stock, S., Mansouri, M., Pecora, F., and Hertzberg, J. (2015). Online task merging with a hierarchical hybrid task planner for mobile service robots. In *Proc. of the Int'l Conf. on Intelligent Robots and Systems (IROS)*, pages 6459–6464.

Viehmann, T. (2019). *Transforming Robotic Plans with Timed Automata to Solve Temporal Platform Constraints*. Master's Thesis, RWTH Aachen University. https://kbsg.rwth-aachen.de/~hofmann/theses/viehmann.pdf.