Constraint-based Plan Transformation in a Safe and Usable GOLOG Language*

Victor Mataré[1] Stefan Schiffer[1] Alexander Ferrein[1] Tarik Viehmann[2] Till Hofmann[2] Gerhard Lakemeyer[2]

ABSTRACT

We present a temporal constraint language to help developers maintain a defined separation between platform-specific behavior contingencies and the domain logic in high-level agent programs. It is implemented as an optional feature in the agent programming and interfacing framework `golog++`.

I. THE PROBLEM

GOLOG is a family of languages for the specification of high-level strategic behavior that allows the programmer to freely mix planning with scripting. Despite their inherent flexibility and expressivity, GOLOG-based languages have yet to make the leap from academic and laboratory use to production applications in an industrial context. Looking back on many years of using various GOLOG dialects in diverse scenarios [1]–[7], we can conclude that the fundamental idea of the language is viable for a wide range of domains. However, when it comes to long-term use of a high-level behavior control language in a production environment, a language must satisfy requirements that go far beyond flexibility and expressivity.

Requirements change and robotic hardware platforms evolve constantly. Advances in sensory devices and effectors may open new behavioral options (through increased payload, range, speed, etc.), but may also bring new restrictions (e.g. through increased power consumption, thermal constraints, etc.).

A high-level agent language must support short development cycles in a field where the classical separation between strategic decision-making and a reactive behavioral layer is difficult to maintain. Consider an RGB-D camera used for object recognition (cf. Fig. 2). Such cameras typically need to be switched off when they are not being used. Switching on usually takes a few seconds, so we want to make sure that it is ready *just before* it will be used. When we hide such maintenance actions within the reactive layer, we lose the ability to perform them in a clever, circumspect manner. When we model them within the strategic layer, we can make smart decisions, but we are also breaking separation of concerns and we are compounding the problem complexity.

Practical experience also shows that seemingly simple tasks can turn out to be quite complex when we factor in runtime robustness against both internal errors and external disturbances. We cannot ignore engineering issues, so clearly defined, rigid interfaces are essential, and a language must be able to enforce them and check their consistency *before* anything is executed.

Despite making important progress in runtime semantics, the classical Prolog-based GOLOG implementations (cf. [8]–[10] among others) all suffer from blurry language boundaries, a lack of both consistency checking and error handling, as well as largely undefined runtime platform interfaces. As such, these implementations are not suitable for maintaining larger code bases within a production environment.

II. THE SOLUTION

We present the GOLOG-based development and interfacing framework `golog++` [11] that addresses the problems outlined above. It is inspired by previous work on more practically oriented GOLOG implementations like YAGI [12] and `golog.lua` [13]. A built-in temporal constraint language allows the programmer to construct an explicit model of runtime contingencies (a *platform model* for short), the fundamental ideas of which have been explored in [14].

At the core of `golog++` is an event-based execution *Controller* (cf. Fig. 1) that coordinates program interpretation (the *Transition Function*) and plan *Transformation* with endogenous dispatch and exogenous event handling (to/from the *Platform Backend* respectively).

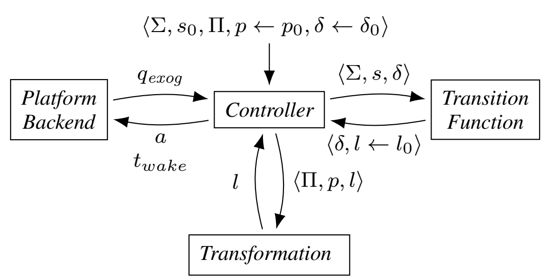

Fig. 1. Flow of events and information between the major components involved in program execution. Σ is the domain model (basic action theory in GOLOG jargon), s_0 is the initial domain situation and s is the current domain situation. Likewise, Π is the platform model, p_0 is the initial platform state and p is the current platform state. δ_0 is the initial program and δ is the remaining program given the actions executed so far. l_0 is the abstract plan (without maintenance actions and time windows), and l is the current transformed plan.

A platform model Π is composed of a set of *component models* and a set of *constraint formulas*. A component model is a timed automaton [15] that describes the runtime behavior of a hardware component or a lower-level software

* This work was supported by the German National Science Foundation (DFG) under grant number FE 1077/4-1

[1] Institute for Mobile Autonomous Systems and Cognitive Robotics, FH Aachen University of Applied Sciences, 52066 Aachen, Germany

[2] Knowledge-based Systems Group, RWTH Aachen University, 52056 Aachen, Germany

component. Transitions between the individual states of a component can be either endogenous (i.e. controlled by the agent, like switching a camera on or off) or exogenous (i.e. uncontrolled, like a shutdown due to overheating or a battery failure). Such a component model is then linked to the structure of a plan by temporal constraint formulas [16] based on MITL semantics [17]. Simply put, a constraint formula is an implication $\phi \supset \psi$, where ϕ is a temporal formula that refers only to action terms from the domain theory Σ, and ψ is a temporal formula that refers only to states from the platform model Π.

The `realsense` component shown in Fig. 2 remains in the `boot` state for 2 to 4 seconds. The transition from `off` to `boot` can be triggered by inserting the appropriate maintenance action, but the transition from `boot` to `on` is exogenous, so the agent has to wait for it. The transformed schedule will thus always trigger the `off => boot` transition at least 2 seconds before a `scan(*)` action is performed.

```
component realsense {
    clocks: bcl
    states: off, boot (bcl < 4), on, error
    transitions: on => off, off => boot resets(bcl),
                 boot ->(bcl > 2) on, error => off
}
constraints {
    during(scan(*)): state(realsense) = on;
    during(go_to(*)): state(realsense) = off;
}
```

Fig. 2. Exemplary component model with constraints.

Just before an abstract (i.e. platform-agnostic) plan l_0 is executed, the plan *Transformation* turns it into a temporal schedule l that is guaranteed to satisfy the platform model. In l, each action a has a time window $[t_{min}(a), t_{max}(a)]$ and likewise timed maintenance actions are inserted to satisfy the platform model Π, cf. [18]. If $t_{min}(a)$ has not yet arrived for $a = head(l)$, the *Controller* schedules an exogenous wakeup at $t_{wake} = t_{min}(a)$. If a is not (yet) executable (for any reason), the *Controller* blocks on q_{exog}. When the *Platform Backend* (asynchronously) enqueues an exogenous event to q_{exog}, the controller checks a's preconditions (including the time window) again. In case a missed t_{max} or a component has changed state exogenously, it triggers a re-transformation of l to (hopefully) find another conformant schedule. If this fails, it means that platform-conformant plan execution has failed unrecoverably and control is escalated up to the parent of the planner call.

To help usability, robustness and collaboration, `golog++` is also typesafe, which allows the construction of an expressive static code model that guarantees referential consistency across the entire codebase. Blurry language bounds also not an issue since it comes with a parser for a well-readable, C++-like notation that eases the initiation of new developers.

III. Conclusion

Since our constraint logic is independent from the Golog basic action theory yet compatible with its online execution semantics, our domain reasoning scales with the domain complexity alone, while we're still able to cope with platform-specific contingencies at the knowledge level. The flexibility of our Golog language allows much more freedom in defining execution strategies than purely planning-based approaches. We can control when and how much is planned, and this behavior in turn can be made dependent on the outcomes of earlier plan executions. In particular, we can react freely to failures of the composite plan-and-transform system. This, and the rigid syntax with strict and precise static error handling make it both a robust and sustainable platform and a helpful tool to study dynamic failure recovery strategies.

References

[1] W. Burgard, A. B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun, "Experiences with an interactive museum tour-guide robot," *Artificial Intelligence*, vol. 114, no. 1-2, pp. 3–55, 1999.

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

[3] A. Ferrein and G. Lakemeyer, "Logic-based robot control in highly dynamic domains," *Robotics and Autonomous Systems*, vol. 56, no. 11, pp. 980–991, 2008.

[4] A. Ferrein, C. Fritz, and G. Lakemeyer, "On-line decision-theoretic golog for unpredictable domains," in *Annual Conference on Artificial Intelligence*. Springer, 2004, pp. 322–336.

[5] S. Jacobs, A. Ferrein, and G. Lakemeyer, "Unreal golog bots," in *Proceedings of the 2005 IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*, 2005, pp. 31–36.

[6] A. Ferrein, C. Fritz, and G. Lakemeyer, "Using golog for deliberation and team coordination in robotic soccer," *KI - Künstliche Intelligenz*, 2005.

[7] A. Ferrein, C. Maier, C. Mühlbacher, T. Niemueller, G. Steinbauer, and S. Vassos, "Controlling logistics robots with the action-based language YAGI," in *International Conference on Intelligent Robotics and Applications*. Springer, 2016, pp. 525–537.

[8] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl, "GOLOG: A logic programming language for dynamic domains," *Journal of Logic Programming*, vol. 31, no. 1-3, pp. 59–84, 1997.

[9] G. De Giacomo, Y. Lespérance, and H. J. Levesque, "ConGolog, a concurrent programming language based on the situation calculus," *Artificial Intelligence*, vol. 121, 2000.

[10] G. De Giacomo, Y. Lespérance, H. J. Levesque, and S. Sardina, "IndiGolog: A high-level programming language for embedded reasoning agents," in *Multi-Agent Programming*. Springer, 2009, pp. 31–72.

[11] V. Mataré, S. Schiffer, and A. Ferrein, "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*, Oct. 2018, pp. 29–36.

[12] A. Ferrein, G. Steinbauer, and S. Vassos, "Action-based imperative programming with YAGI," in *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[13] A. Ferrein, "Golog.lua: Towards a non-prolog implementation of GOLOG for embedded systems," in *Proceedings of the AAAI Spring Symposium on Embedded Reasoning*, ser. (SS-10-04), G. Hoffmann, Ed. AAAI Press, 2010, pp. 20–28.

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

[15] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[16] T. Hofmann and G. Lakemeyer, "A Logic for Specifying Metric Temporal Constraints for Golog Programs," 2018, pp. 36–46.

[17] R. Alur, T. Feder, and T. A. Henzinger, "The benefits of relaxing punctuality," *Journal of the ACM (JACM)*, vol. 43, no. 1, pp. 116–146, 1996.

[18] T. Viehmann, "Transforming robotic plans with timed automata to solve temporal platform constraints," Master's Thesis, RWTH Aachen University, Dec. 2019.