Time in Multi-Agent Systems

Niklas Fiekas

Department of Informatics, Clausthal University of Technology
Julius-Albert-Str. 4, D-38678 Clausthal
niklas.fiekas@tu-clausthal.de

Abstract. This is a research proposal, aiming to improve tooling and features of agent-oriented programming languages, in particular Jason, to handle virtual time and real-time deadlines. The main idea is to apply known techniques and patterns from asynchronous programming in classical programming languages, and extend Jason as necessary.

Keywords: agent-oriented programming, simulation, soft real-time

1 Introduction

This early stage proposal is motivated by two use cases that were encountered in prior research.

First, the **SimSE** (Simulating Software Evolution) project aims to simulate the behavior of human software developers, in order to make predictions about deadlines and software quality [1]. The environment is a graph created from the real software repository to be simulated. The developers are modelled as BDI agents in a multi-agent system, using Jason/AgentSpeak to describe plans.

This kind of simulation is typically divided into discrete virtual time steps, but by default all actions and reasoning in Jason are performed instantly, with little support to model the time that a simulated human developer would take, much less synchronize a simulated virtual clock across agents.

Second, the yearly **multi-agent programming contest** provides challenging scenarios to benchmark multi-agent programming languages, platforms and tools [2]. Contest games are played in discrete time steps (typically 4 seconds). The participants program agents that receive percepts and submit actions for each step.

This is a soft real-time system. Missing the deadline to submit an action for the current step is not fatal, but it degrades the performance of team. Furthermore, participants report a snowball effect: Missing one deadline leads to a backlog of incoming percepts, and more missed deadlines if the agent cannot catch up. It can be difficult to recover.

It seems likely that the described issues with virtual time, real time, and deadlines are common in multi-agent systems, especially when simulating real-world scenarios or moving closer to the implementation of physical systems, including ambient AI. The proposal is therefore to survey the state of solutions in multi-agent systems, and improve them on the language and platform level.

2 Related Work

On the practical side, the motivating examples come from the SimSE project [1] and the yearly Multi-Agent Programming contest. In the latter, participants describe their experience and practical issues while implementing multi-agent systems [2].

Jason 2.0 recently introduced the fork/join operators, to support structured plan level concurrency [3], and foundations for concurrent programming are laid [4].

On the theoretical side, approaches using LTL, CTL and extensions [5] can show properties like safety and liveness (never something, always eventually again), but are not designed to answer if agents can meet a particular deadline.

Keeping time and achieving consensus in distributed systems comes with interesting algorithmic challenges [6], but this is a not an issue with virtual time in simulated environments, nor when trying to meet deadlines of a centralized game server (intended to simulate deadlines of local sensors and actuators).

Real time agent systems are typically not BDI based, much less using high level agent languages like Jason [7]. Outside of the agent community, there is a wide array of work around asynchronous programming in classical programming languages. Many languages adopted syntax extensions to support writing asynchronous programs in straight-line fashion (e.g., Python, JavaScript, Rust, Kotlin).

All in all, this is an active area of research (even with regard to classical programming languages). This proposal focuses on improvements to practical agent oriented programming.

3 Proposal

The driving conjecture is that many of the techniques and patterns for asynchronous programming in classical languages can also be applied to agent-oriented languages, such as Jason. This is not obvious. Consider for example negative results for the analogous conjecture with regard to testing and fuzzing [8].

Initial techniques to investigate are generators and coroutines. These involve computations that can be interrupted and resumed at predefined points, for example to wait for network I/O. Before using these techniques, they must be unified with the operational semantics of Jason agents. Instead of simply putting them on top, it seems possible to express the existing Jason semantics in these terms. For example, plans can also be interrupted in favor of other plans and resumed later. This is the first work package.

Existing Jason programs should be shown to be equivalent under the new semantics. This is the second work package, although it may be deferred, in order to first gain more confidence in the practical relevance of the proposed semantics, with possible iterations on the design.

Many classical programming languages have added language level features to support the mentioned techniques. The proposal therefore includes extending

the Jason language as necessary and providing a working implementation. This is a third work package, and will provide the means for further evaluation.

Generators and coroutines are frequently combined with timeouts and cancellation tokens, or to build *structured concurrency* abstractions. In new Jason programs, timeouts and cancellations should interact in practically useful ways with long term desires and short term intentions of agents, as well as recovery plans. The final work package is evaluating this based on the use cases from the introduction.

4 Preliminary Results

An extensible Jason interpreter has been developed, https://github.com/niklasf/python-agentspeak. So far, the main novelty is the ability to pause the interpreter at any time, including during agent actions and even Prolog queries, and quickly serialize the state. This is achieved by translating Jason programs to control-flow graphs with high level instructions. The following instructions are sufficient to express all Jason plans:

noop(agent, intention) Does nothing and succeeds always.

push_query(query, agent, intention) Starts a Prolog `query` and adds it to the query stack. This is also used for actions that can yield multiple results.

next_or_fail(agent, intention) Tries to find the next solution for the topmost Prolog query, a substitution of variables.

pop_query(agent, intention) Removes the topmost Prolog query from the query stack.

add_belief(term, agent, intention) Applies the current substitution to `term` and adds it to the belief base. Triggers a belief addition event.

remove_belief(term, agent, intention) Unifies `term` with the first matching belief and removes it from the belief base. Triggers a belief removal event.

test_belief(term, agent, intention) Tries to find a substitution such that `term` is a logical consequence of the belief base. Triggers a belief test event.

call(trigger, goal_type, term, agent, intention) Tries to find a plan matching the `trigger`, `goal_type` and `term` and adds it as a sub-plan to the current intention.

call_delayed(trigger, goal_type, term, agent, intention) Tries to find a matching plan and crates a new intention with it.

Initially this was designed in order to apply data parallelism to multi-agent simulation (e.g., treat agent states as just data and apply techniques like MapReduce to advance the simulation).

However, the same design also allows bringing asynchronous programming to Jason. This includes having agents wait for a synchronized virtual clock without wasting time in the real world, and actual asynchronous communication with the game server of the multi-agent programming contest.

On the other hand the current design is not yet satisfactory for soft real-time systems. While the instructions `call`, `call_delayed`, `push_query`, `pop_query`, and of course trivially `noop`, are constant time, it is hard to predict if queries to

the belief base (`add_belief`, `remove_belief`, `test_belief`) and `next_or_fail` will complete before a given deadline.

Also, importantly, the control flow graph is currently limited to individual plans. This will not suffice, as interactions of errors, timeouts, cancellation and recovery plans appear to be essential.

5 Evaluation Plan

To evaluate new approaches, it seems useful to come back around to the motivating examples. In the *SimSE* project, the goal is to simulate the behavior of human software developers. It will be interesting to see if the existing agents can be simplified, and if more detailed modelling in future iterations can be supported.

For the multi-agent programming contest, participants submit the source code of their solutions, and comment on difficulties that they encountered. Some teams used Jason for their solutions, so it will be possible to apply and evaluate new approaches based on these agent programs.

Finally, language level changes might impact the performance of the Jason interpreter. Its performance can be evaluated in benchmarks and compared with previous versions and the original Jason interpreter.

References

1. T. Ahlbrecht, J. Dix, N. Fiekas, J. Grabowski, V. Herbold, D. Honsel, S. Waack, and M. Welter. Agent-based simulation for software development processes. In Proceedings of the 14th European Conference on Multi-Agent Systems, EUMAS 2016. Springer, December 2016.
2. T. Ahlbrecht, J. Dix, and N. Fiekas: The Multi-Agent Programming Contest 2018 - Agents teaming up in an urban environment. Springer, 2019. https://link.springer.com/book/10.1007/978-3-030-37959-9.
3. Concurrency in Jason. https://github.com/jason-lang/jason/blob/master/doc/tech/concurrency.adoc.
4. A. Muscar, C. Badica. Monadic Foundations for Promises in Jason. Information Technology And Control 43 (1), 65-72. http://www.itc.ktu.lt/index.php/ITC/article/view/4586.
5. R. H. Bordini, M. Fisher, C. Pardavila, M. Wooldridge. Model Checking AgentSpeak. Proceedings of the second international joint conference on Autonomous agents and multiagent systems, 409–416, July 2003. https://dl.acm.org/doi/10.1145/860575.860641.
6. T. Yang, Z. Meng, D. V. Dimarogonas, K. H. Johansson. Global consensus for discrete-time multi-agent systems with input saturation constraints. Automatica (50), 499-506, February 2014. https://doi.org/10.1016/j.automatica.2013.11.008.
7. V. Julián, V. Botti. Developing Real-Time Multi-Agent Systems. Integrated Computer Aided Engineering 11 (2), November 2002. https://content.iospress.com/articles/integrated-computer-aided-engineering/ica00172.
8. M. Winikoff, S. Cranefield. On the testability of BDI agent systems. Journal of Artificial Intelligence Research, 2014. https://www.jair.org/index.php/jair/article/view/10903.