TU Clausthal

Clausthal University of Technology

Putting APL Platforms to the Test: Agent Similarity and Execution Performance

Tristan Behrens, Koen Hindriks, Jomi Hübner, Mehdi Dastani

IfI Technical Report Series IfI-10-09

Department of Informatics
Clausthal University of Technology

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Federico Schlesinger
Contact: federico.schlesinger@tu-clausthal.de
URL: http://www.in.tu-clausthal.de/forschung/technical-reports/
ISSN: 1860-8477

The IfI Review Board

Putting APL Platforms to the Test: Agent Similarity and Execution Performance

Tristan Behrens, Koen Hindriks, Jomi Hübner, Mehdi Dastani

Abstract

It is our goal to compare agent programs, implemented by means of different agent programming languages, by firstly reasoning about their similarity and secondly by measuring the time it takes them to reach specific states. In this paper, we will establish a *similarity notion* based on comparing *agent states* and *agent runs*. An agent state is usually defined as a snapshot of the agent's internals during execution, an agent run is usually defined as a sequence of such agent states. Examining the different notions of agent states specific to different agent programming languages, we will define generic agent states/runs and establish a similarity notion on top of that. After defining a set of mappings, that map specific agent states/runs to generic ones in order to facilitate a common starting point, we will apply this very notion in a case study. This case study is rendered possible by a toolkit, whose description is part of this paper and that allows for automatically executing and examining agents executed by different agent interpreters.

1 Introduction

In this paper, which is settled in the area of agent-oriented programming, especially in the field of BDI-based software development, we will work towards the goal of a detailed comparison of agent programs and APL platforms. Specifically we want to find out whether a dedicated knowledge representation implementation should be preferred over using a reusable one or not.

We will perform a simple case study with a very simple scenario: calculating Fibonacci-numbers using 2APL [Dastani, 2008], GOAL [Hindriks, 2009] and *Jason* [Bordini et al., 2007]. We will introduce generic notions of agent states and agent runs, which will function as the formal framework of our research. In order to execute the case study, we have developed a toolkit based on a couple of principles that we have to define first.

In summary the contributions of this paper are:

- the definition of an *agent program similarity notion*,

- the specification of an agent programming toolkit called XYZ[1], and
- the execution and results of a case-study.

In the second section, we will establish a notion of agent program similarity based on generic agent runs. After that, in the third section, we will provide a brief description of XYZ. The fourth section consists of our experiment, that is its description, its execution and the results. Finally, we conclude the paper with related work, a conclusion and a hint on future work.

2 Agent Program Similarity

It is our designated goal to compare agent programs that are implemented by means of different agent programming languages. To reach that goal it is necessary to define a *similarity notion*, which in turn requires that we firstly examine a set of programming platforms and find out what they have in common, in order to establish a basis for the similarity notion. We will examine and compare 2APL, GOAL and *Jason* and specify the fundament for similarity by means of generic agent states. Although the considered platforms are expected to differ significantly when it comes to mental attitudes, it is also expected that a definition of generic agent states can be established without much effort. On top of these generic agent states we will define generic agent runs (sequences of agent states). After that, we will define a notion of agent program similarity that is based on these generic runs. We will also provide mappings from platform-specific agent runs to generic agent runs (as depicted in Figure 1).

2.1 Agent Programs and Agent States

The first required notion is the notion of an *agent state*. We will now introduce the different definitions of agent states for 2APL, GOAL and *Jason*. Additionally, we will give a hint on how agent states change over time. Note, however, that it does not lie in the scope of this paper to give a full description of agent syntax and semantics. Please refer to the literature [Dastani, 2008, Hindriks, 2009, Bordini et al., 2007] for complete descriptions. We will conclude this part of the paper with a brief comparison.

The state of a 2APL agent consists of a *belief base* that is expressed by means of a Prolog program, a list of declarative goals, that constitute the *goal base*, a set of plan entries, that constitute the *plan base*, and the *event base*, that consists of external events received from the environment, failed plans, and received messages. Formally:

[1] For the sake of anonymity the actual name is omitted.

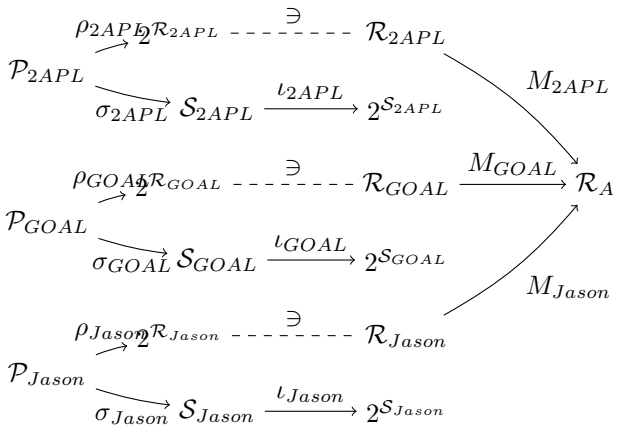

Figure 1: The different data-structures that we introduce in order to compare agent-programs, and the mappings between them.

Definition 1 (2APL agent state)
The tuple $A_\iota := \langle \iota, \sigma, \gamma, \Pi, \Theta, \xi \rangle$ is an 2APL agent state with:

- ι *a string representing the agent's identifier,*
- σ *a set of belief expressions constituting the belief base,*
- γ *a list of goal expressions constituting the goal base,*
- Π *a set of plan entries,*
- Θ *a ground substitution that binds domain variables to ground terms, and*
- $\xi := \langle E, I, M \rangle$ *an event-base with E the set of events received from external environments, I set of plan identifiers denoting failed plans, and M the set of messages sent to an agent.*

The state of a GOAL agent on the other hand consists of a knowledge base and a *belief base*, both expressed by means of a knowledge representation (KR) language[2], and a *goal base*, again expressed by the same language. Additionally it contains the current percepts, rules for action selection, received messages and actions to be performed. Formally:

Definition 2 (GOAL agent state)
A GOAL *agent state consists of*

[2]Although GOAL does not restrict the agent developer to using a specific KR language, we will stick to using Prolog.

- *a mental state $\langle \mathcal{D}, \Sigma, \Gamma \rangle$, where $\mathcal{D}$ is called a knowledge base, Σ is a belief base, and Γ is a goal base,*
- *a set AR of* action rules *that facilitate the action selection mechanism,*
- *a set $P$of percepts representing the percepts received from the environment,*
- *a set of M messages received from other agents, and*
- *a set of A actions to be executed by the environment.*

The state of a *Jason* agent, however, consists of a belief base expressed by means of a Prolog-like KR language, a plan base, a set of intentions consisting of partially instantiated plans, an event list, a set of actions to be performed, a message box for communicating, and a set of suspended intentions. Formally:

Definition 3 (*Jason* agent state)
The tuple $\langle ag, C, M, T, s \rangle$ is a Jason agent state with:

- *ag is an agent program, which is specified by a set of beliefs and a set of plans,*
- *C is the circumstance, that is a tuple $\langle I, E, A \rangle$ where I is a set of intentions, each one is a stack of partially instantiated plans, E is a set of events, and A is a set of actions to be performed,*
- *$M := \langle In, Out, SI \rangle$ is a tuple where In is the mail inbox, Out is the mail outbox, and SI is the set of suspended intentions,*
- *$T := \langle R, Ap, \iota, \epsilon, \rho \rangle$ stores temporary information, R is the set of relevant plans, Ap is the set of applicable plans, ι, ϵ, and ρ keep record of a particular intention, event and applicable plan being considered during the executions, and*
- *s indicates the current step of the agent's deliberation cycle, that is processing a message, selecting an event, retrieving all relevant plans, retrieving all applicable plans, selecting one applicable plan, adding the new intended means to the set of intentions, selecting an intention, executing the selected intention or clearing an intention*

For the sake of convenience, we will use the following definitions for the rest of our paper:

Definition 4 (agent programs, agent states)

- *$\mathcal{P}_{2APL}$ is the set of 2APL agent-programs,*
- *$\mathcal{P}_{GOAL}$ is the set of GOAL agent-programs,*

- $\mathcal{P}_{Jason}$ *is the set of Jason agent-programs,*
- $\mathcal{S}_{2APL}$ *is the set of 2APL agent-states,*
- $\mathcal{S}_{GOAL}$ *is the set of GOAL agent-states, and*
- $\mathcal{S}_{Jason}$ *is the set of Jason agent-states.*

Now, we have to briefly compare the different notions of agent states in order to define a notion of generic ones. The belief bases in 2APL and GOAL are full Prolog, the belief-base in *Jason* is logic-programming-like, consisting of facts, rules and strong negation. On the other hand, the goal-bases in 2APL and GOAL are declarative. The goal base in 2APL, is an ordered collection of goal-expressions, where every goal-expression is a conjunction of ground atoms. In GOAL the goal base is a set of goals, where each goal is either a literal or a conjunction of literals. In *Jason* there is no explicit declarative goal base. Goals, that is achievement- and test-goals, are either stored in the event base together with other events, or they are stored in the triggering-events of the instantiated plans in the agent's set of intentions. As we will show later, *Jason* goals can be made explicit. We cannot compare the plan-libraries straight away, because in 2APL and *Jason* the semantics is different, and because GOAL lacks plans. For the same reasons, we do not compare intentions. Also, we do not use events, since the notion of events is different in 2APL and *Jason*, and because this notion is absent from GOAL. In summary, we restrict ourselves to using only goals and beliefs, because this is something that all three platforms have in common. Furthermore we have to make an assumption about the belief base. That is, that we are going to use only the facts from the belief-bases, ignoring rules and strong negation. When it comes to goals we will restrict ourselves to goal-bases that consist of a set of goals, where each goal is an atom.

2.2 Agent Runs

An *agent run* is usually defined as a sequence of agent states. We have already defined states of 2APL, GOAL and *Jason* agents. On top of that we will now elaborate on how such states are transformed by the respective interpreter. Again we would like to note that a full definition of agent program syntax and semantics for all three APL platforms, is provided by the literature [Dastani, 2008, Hindriks, 2009, Bordini et al., 2007]. Usually an agent has an initial state, which is determined by the respective agent program. For each individual agent its agent state is transformed by applying the agent interpreter function which is implemented by the respective APL platform. We will just give a brief but sufficient definition of the semantics. Formally:

Definition 5 (initial agent states)

- $\sigma_{2APL} : \mathcal{P}_{2APL} \rightarrow \mathcal{S}_{2APL}$ *maps all 2APL agent programs to their respective initial agent-states,*
- $\sigma_{GOAL} : \mathcal{P}_{GOAL} \rightarrow \mathcal{S}_{GOAL}$ *maps all GOAL agent programs to their respective initial agent-states,*
- $\sigma_{Jason} : \mathcal{P}_{Jason} \rightarrow \mathcal{S}_{Jason}$ *maps all Jason agent programs to their respective initial agent-states.*

2APL agent states evolve as follows: 1. instantiating plans while taking into account the goal base and the belief base, 2. executing the first action of all instantiated plans, and 3. processing internal/external events and messages, which yields new instantiated plans. The evolution of GOAL agents, on the other hand, is facilitated by a simple instance of a sense-plan-act-cycle: 1. storing percepts and incoming messages in the belief base, and 2. randomly selecting an applicable rule and executing it, which will yield actions to be executed in the environment, and 3. execute the actions. *Jason* agents are executed as follows: 1. processing percepts and incoming messages, 2. selecting an event and instantiating a plan from that event, and 3. selecting an instantiated plan and executing its first action. Formally the agents evolve by applying the respective interpreter functions:

Definition 6 (agent interpreter functions)

- $\iota_{2APL} : \mathcal{S}_{2APL} \rightarrow 2^{\mathcal{S}_{2APL}}$ *is the 2APL interpreter-function,*
- $\iota_{GOAL} : \mathcal{S}_{GOAL} \rightarrow 2^{\mathcal{S}_{GOAL}}$ *is the GOAL interpreter-function, and*
- $\iota_{Jason} : \mathcal{S}_{Jason} \rightarrow 2^{\mathcal{S}_{Jason}}$ *is the Jason interpreter-function.*

Agent runs are generated by repeatedly applying interpreter-functions:

Definition 7 (agent runs)

- $\mathcal{R}_{2APL} := (S_{2APL})^+$ *is the set of 2APL-runs,*
- $\mathcal{R}_{GOAL} := (S_{GOAL})^+$ *is the set of GOAL-runs,*
- $\mathcal{R}_{Jason} := (S_{Jason})^+$ *is the set of Jason-runs,*
- $\rho_{2APL} : \mathcal{P}_{2APL} \rightarrow 2^{\mathcal{R}_{2APL}}$ *is the function that computes all 2APL-runs of a 2APL agent-program, using* ι_{2APL} *and the initial agent-state derived from a given agent-program,*
- $\rho_{GOAL} : \mathcal{P}_{GOAL} \rightarrow 2^{\mathcal{R}_{GOAL}}$ *is the function that computes all GOAL-runs of a GOAL agent-program, using* ι_{GOAL} *and the initial agent-state derived from a given agent-program, and*

- $\rho_{Jason} : \mathcal{P}_{Jason} \rightarrow 2^{\mathcal{R}_{Jason}}$ *is the function that computes all Jason-runs of a Jason agent-program, using* ι_{Jason} *and the initial agent-state derived from a given agent-program.*

Note, that for the sake of abstraction we will assume that the interpreter functions are *deterministic*. Non-determinism would be absolutely feasible for the framework that we are about to introduce, but for supporting the readability of this paper, we will refrain from coping with that specific issue now.

2.3 Generic Agent States and Agent Runs

In the previous subsections, we have shown how 2APL-, GOAL- and *Jason*-agents are defined and how they evolve during runtime. We have also calculated that, if we chose to define generic agent states and runs, judging from the notions that all three platforms have in common, it would make sense to use primitive (that is consisting of ground literals) beliefs and goals as the main building block for defining generic agent states:

Definition 8 (generic agent state)

- $B_A := \{b_1, b_2, \ldots\}$ *is a set of generic beliefs,*
- $G_A := \{g_1, g_2, \ldots\}$ *is a set of generic goals, and*
- $\mathcal{S}_A$ *is the set of generic agent states where each* $s_i \in \mathcal{S}$ *is a tuple* $\langle B_i, G_i \rangle$*, where* $B_i \subseteq B_A$ *is a set of beliefs and* $G_i \subseteq G_A$ *is a set of generic goals.*

Of course, a generic agent run is a sequence of generic agent states:

Definition 9 (generic agent run)

- *Every sequence* $\langle B_0, G_0 \rangle \rightarrow \langle B_1, G_1 \rangle \ldots$ *is an abstract agent run, and*
- $\mathcal{R}_A := (S_A)^+$ *is the set of all abstract runs.*

Now we have to define a set of generalization mappings, that map specific agent states and runs to generic ones:

Definition 10 (generalization mappings)

- $\mu_{2APL} : \mathcal{S}_{2APL} \rightarrow \mathcal{S}_A$ *maps each 2APL agent-state to an abstract agent-state,*
- $\mu_{GOAL} : \mathcal{S}_{GOAL} \rightarrow \mathcal{S}_A$ *maps each GOAL agent-state to an abstract agent-state,*
- $\mu_{Jason} : \mathcal{S}_{Jason} \rightarrow \mathcal{S}_A$ *maps each Jason agent-state to an abstract agent-state,*

- $M_{2APL} : \mathcal{R}_{2APL} \rightarrow \mathcal{R}_A$ *with*

$$M_{2APL} : (s_1, \ldots, s_n) \mapsto (\mu_{2APL}(s_1), \ldots, \mu_{2APL}(s_n))$$

maps each 2APL agent run to an abstract agent run,

- $M_{GOAL} : \mathcal{R}_{GOAL} \rightarrow \mathcal{R}_A$ *with*

$$M_{GOAL} : (s_1, \ldots, s_n) \mapsto (\mu_{GOAL}(s_1), \ldots, \mu_{GOAL}(s_n))$$

maps each GOAL agent run to an abstract agent run, and

- $M_{Jason} : \mathcal{R}_{Jason} \rightarrow \mathcal{R}_A$ *with*

$$M_{Jason} : (s_1, \ldots, s_n) \mapsto (\mu_{Jason}(s_1), \ldots, \mu_{Jason}(s_n))$$

maps each Jason agent run to an abstract agent run.

We do not have to elaborate the mappings from specific agent runs to generic ones, since these are inductive in nature. The mappings from specific agent states to generic ones, however, are more interesting. For 2APL we consider the belief base and copy every fact contained therein to the generic belief-base. An equivalent procedure is applied to all atomic goals from the goal base. The procedure for mapping GOAL's mental attitudes is the same. Again we keep all facts from the belief base, while ignoring the rules, and copy all atomic goals from the goal base to the generic one. For *Jason* mapping the beliefs is almost the same, except for the facts being stripped off their annotations. Because *Jason* does not hold a notion of declarative beliefs, we have to apply a special treatment to the mental attitudes. Goals can be extracted 1. from the event-base, and 2. from the triggering-events of instantiated plans.

2.4 Agent State and Agent Run Similarity

Now, with the definitions of generic agent states and generic agent runs in place, we can concentrate on *agent state* and *agent run similarity*. We expect repeating states for two reasons: 1. the mapping from specific agent states to generic ones, which acts as a kind of filtering-function, might yield for two different specific agent states the same generic one, and 2. agent states in general might repeat under certain conditions (e.g. when nothing happens). In order not to burden ourselves with repeating states we introduce a compression function:

Definition 11 (compression function)
$\delta : (e_1, e_2, \ldots, e_n) \mapsto (e'_1, e'_2, \ldots, e'_m)$ *is the compression function that maps each sequence* $s := (a_1, \ldots, a_n)$ *defined over an arbitrary set* A *to a second one* $s' := (a'_1, a'_2, \ldots, a'_m)$ *where* s' *is* s *without repeated entries* [3].

[3] Example: $\delta(1, 2, 2, 3, 1, 1) = (1, 2, 3, 1)$

In order to reason about generic agent runs effectively, we define a couple of filtering projections that allow us to restrict the considered beliefs and goals to subsets. Sometimes it might not be necessary to take the full belief and goal bases into account:

Definition 12 (filtering projections)
[4]

- $\pi_B : \mathcal{S}_A \to B$ *with* $\pi_B : \langle B, G\rangle \mapsto B$ *projects an abstract agent-state to the respective beliefs,*
- $\pi_G : \mathcal{S}_A \to G$ *with* $\pi_B : \langle B, G\rangle \mapsto G$ *projects an abstract agent-state to the respective goals,*
- $\Pi_B : \mathcal{R}_A \to B^*$ *with*

  $$\Pi_B : (s_1, s_2, \ldots) \mapsto (\pi_B(s_1), \pi_B(s_2), \ldots)$$

  projects all abstract agent-runs to sequences of beliefs, and
- $\Pi_G : \mathcal{R}_A \to G^*$ *with*

  $$\Pi_G : (s_1, s_2, \ldots) \mapsto (\pi_G(s_1), \pi_G(s_2), \ldots)$$

  projects all abstract agent-runs to sequences of goals.

It is about time to put things together and define the desired notion of similarity:

Definition 13 (n-B-/n-G-/n-BG-similar)

- *two agent-programs* p_{l_1}, p_{l_2} *with* $l_1, l_2 \in \{\texttt{2APL}, \texttt{GOAL}, \texttt{Jason}\}$ *and* $p_1 \in \mathcal{P}_{l_1}$, $p_2 \in \mathcal{P}_{l_2}$ *are* n-B-similar *if*

  $$r_1 := \delta(\pi_B(\mu_{l_1}\rho(p_1))) = \delta(\pi_B(\mu_{l_2}\rho(p_2))) =: r_2 \ \wedge \\ \mid r_1 \mid = n = \mid r_2 \mid$$

- *two agent-programs* p_{l_1}, p_{l_2} *with* $l_1, l_2 \in \{\texttt{2APL}, \texttt{GOAL}, \texttt{Jason}\}$ *and* $p_1 \in \mathcal{P}_{l_1}$, $p_2 \in \mathcal{P}_{l_2}$ *are* n-G-similar *if*

  $$r_1 := \delta(\pi_G(\mu_{l_1}\rho(p_1))) = \delta(\pi_G(\mu_{l_2}\rho(p_2))) =: r_2 \ \wedge \\ \mid r_1 \mid = n = \mid r_2 \mid$$

- *two agent-programs* p_{l_1}, p_{l_2} *with* $l_1, l_2 \in \{\texttt{2APL}, \texttt{GOAL}, \texttt{Jason}\}$ *and* $p_1 \in \mathcal{P}_{l_1}$, $p_2 \in \mathcal{P}_{l_2}$ *are* n-BG-similar *if*

  $$r_1 := \delta(\mu_{l_1}\rho(p_1)) = \delta(\mu_{l_2}\rho(p_2)) =: r_2 \wedge \ \mid r_1 \mid = n = \mid r_2 \mid$$

We will apply this similarity notion later, when we elaborate on our case study.

[4] Example: $\Pi_{\{fib(1,1)\}}(\langle\{fib(1,1)., fib(2,1).\}, \emptyset\rangle)$ is $(\langle\{fib(1,1).\}, \emptyset\rangle)$.

3 XYZ– A Toolkit for Putting APLs to the Test

This section deals with the software that helps us to compare agent programs implemented in 2APL, GOAL and *Jason*. We will firstly lay down a couple of principles which will then lead to an infrastructure.

3.1 Principles and Infrastructure

In the following, we will define five principles that are supposed to be the fundament for the infrastructure of XYZ:

- **Plug-in architecture:** the overall-infrastructure consists of a *core*, and of *components*, that can be plugged in. Components are 1. *interpreters*, that load, manage and execute agents, 2. *environments* to which agents are connected, which provide them with percepts and in which agents can act, and 3. *tools* that evaluate the execution of multi-agent systems.
- **Minimal assumptions about components:** we assume almost nothing about agents, except that they conform to the agent-definition by Russel-Norvig [Russell and Norvig, 2003], and that the mental-attitudes can be accessed from the outside and are mappable to a common representation. We assume almost nothing about the interpreters, except for the requirement that each interpreter executes the MAS in a stepwise manner and that each interpreter adheres to a specific interface definition. We assume almost nothing about the tools, except for the requirement that each tool needs to adhere to a specific interface definition.
- **Execution fairness:** we assume that the executions of the different components are strictly separated and interleaved. That is, that interpreters are executed first and in a single-threaded manner. After that the tools are executed. Tools and interpreters should not interfere with the execution of other components. This is undesired because it will lead to disturbances when it comes to performance measurements.
- **Areas of application:** we intend to use the toolkit for *profiling*, *automated testing*, *debugging*, and *gathering statistics*.

Figure 2 shows the infrastructure of XYZ. The components consist of interpreters, tools, an environments. There can be multiple instances of each component-class. Interpreters execute agents, environments process actions and generate percepts, and tool query both. The core is a glue-component. It is static and facilitates loading, executing and releasing components, which can be dynamically linked during runtime. The core also delivers step-results (amongst other things: execution time, changed mental attitudes) to the

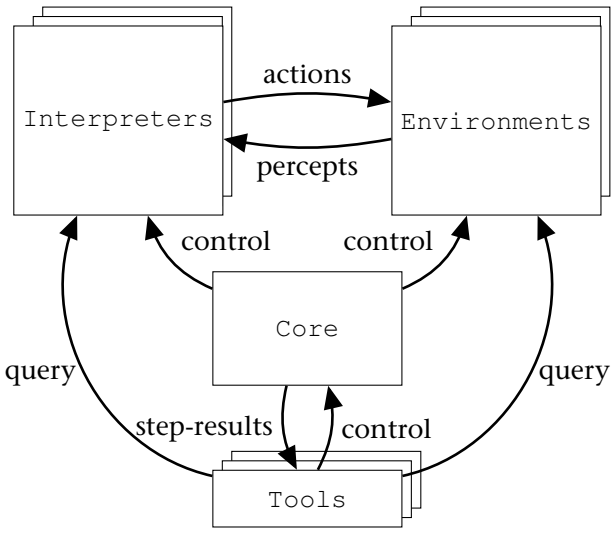

Figure 2: The XYZ infrastructure. Tools are active components, when it comes to querying, but it is the interpreters and environments that provide the data.

tools. Note that the core, can be controlled, that is that the overall execution can be paused, resumed or finished. The execution model is as follows: 1. execute interpreter(s) one step, 2. deliver step-results to the tools and let them evaluate, 3. repeat.

3.2 Implemented Components

For our experiments we have implemented a couple of components for XYZ, including the standalone interpreters for 2APL, GOAL and *Jason*, and a set of tools for monitoring/examining the execution of agents. Standalone means that the interpreters are absolutely decoupled from any IDE, thus ruling out the IDE as an entity that decreases the performance[5].

The `2APLInterpreter` uses the 2APL code-base without adaptions. Initialization parameters are 1. a MAS-file to be loaded, 2. the executor to be used, either single-threaded or multi-threaded execution, 3. if Jade (agent middleware) is to be used and if so, 4. the Jade parameters host and port. The `GOALInterpreter` extends the GOAL code-base by a custom made stepping-scheduler, that executes MASs in a step-wise fashion. Initialization parameters are 1. the MAS-file to be loaded, and 2. the selection of the middle-

[5] In our first experiments it became clear that the IDE can be a significant factor when it comes to performance.

ware (local, RMI, Jade) to be used. The `JasonInterpreter` makes use of a custom-made agent-architecture, that, based on available *Jason* source-codes, loads and executes agents. The only parameter is the MAS-file. Note, that other parameters to the execution of a MAS, e.g. the agent middle-ware, are specified in the MAS-file.

The `BreakpointDebugger`-tool signals when a specified condition is satisfied. Such a condition could be for example an agent holding a belief or goal. The signal is issued either if the condition is satisfied for the first time, or every time it is satisfied. Possible reactions to a signal can be stopping, pausing or finishing the execution of the MAS. The output of a signal contains the current step of the respective interpreter and the current execution time. The `AgentInspector` on the other hand is a convenience tool for inspecting the mental states of the agents, which are represented in terms of generic agent states. The `SimilarityChecker` is more sophisticated. It has two main functions: 1. storing the evolution of each individual agent as generic agent runs, and 2. reasoning about the similarity of these agent runs. The output will be a set of full state-traces and a similarity result.

4 Case Study and Experiments

In this section we will elaborate on a case study that is based on a very simple task: computing Fibonacci-numbers [of Pisa, 1202]. As a reminder, here is the definition of the Fibonacci sequence:

$$F_1 := 1,\ F_2 := 1,\ F_{n+2} := F_{n+1} + F_n$$

Although this scenario is fairly simple, it will turn out later that results derived from it are relatively insightful. In this section, we will develop agent programs for 2APL, GOAL and *Jason*, that compute the first one thousand Fibonacci-numbers and compare how much time each program consumes to reach that goal. The figures 3, 4, and 5 show the source-codes. The functionality of the agents is as follows: (1) each agent knows the first two numbers right from the beginning, (2) each agent has the initial goal of computing the first one thousand numbers, starting with the third, and (3) if an agent has the goal of computing a specific number, it computes it, stores the result in the belief base, drops the respective goal, and adopts the goal of computing the next one, all until the final number is computed.

4.1 Agent Programs

Since it would not be feasible to provide a full definition of all three considered agent programming languages, we will only elaborate on syntactical and semantical notions that are required for understanding the examples.

```
1  Beliefs: fib(1,1). fib(2,1).
2  Goals: calcFib(3,1000)
3  BeliefUpdates: { true } Fib(N,F) { fib(N,F) }
4  PG-rules:
5    calcFib(N,Max) <-
6      N < Max and fib(N-1,Z1)
7      and fib(N-2,Z2) and is(Z,Z1+Z2)  |
8      {[
9        Fib(N,Z);
10       adopta(calcFib(N+1,Max));
11       dropgoal(calcFib(N,Max))
12     ]}
13   calcFib(N,Max) <-
14     N = Max and fib(N-1,Z1)
15     and fib(N-2,Z2) and is(Z,Z1+Z2)  |
16     {[
17       Fib(N,Z);
18       dropgoal(calcFib(N,Max))
19     ]}
```

Figure 3: The Fibonacci 2APL agent program.

The 2APL agent program in Figure 3 shows the basic syntactical components of an 2APL agent. The initial belief base consists of a full Prolog program and thus is usually a list of facts and rules. The belief base is facilitated by JIProlog[6]. In our case it contains only facts about the first two Fibonacci-numbers. The initial goal base consists of a single goal, that is calculating the first one thousand numbers, starting with the third. In general, the belief-updates section contains actions that update the belief base by adding/removing facts if certain preconditions hold. We use a single belief-update to insert computed numbers. PG-rules trigger the instantiation of plans in order to reach goals. We have two rules. The first rule computes a number and triggers the computation of the next one, whereas the second rule computes a Fibonacci-number and ceases computation afterwards. 2APL programs can also contain rules for handling events or repairing plans, but we do not need such rules in this paper.

The GOAL agent program in Figure 4 consists of a belief base, a goal base and a program section. Again, the initial belief base is a full Prolog program. GOAL uses SWIProlog[7] as a knowledge representation language. Note, however, that other languages can also be used for knowledge representation in GOAL. The initial belief base contains two facts representing the first two Fibonacci-numbers and the goal base consists of the single goal to com-

[6] http://www.ugosweb.com/jiprolog/
[7] http://www.swi-prolog.org/

pute the first one thousand Fibonacci-numbers, beginning with the third. In GOAL the program-section specifies how the state of the agent changes over time. It contains three rules. The first one drops the goal of calculating a specific number if it is believed by the agent, the second one calculates the nth number if it is not believed, and the third one raises the goal of computing the next number.

```
1  main: fibonacci {
2    beliefs { fib(1,1). fib(2,1). }
3    goals { calcFib(3,1000). }
4    program[order=linear] {
5      if
6        goal(calcFib(N,Max)),
7        bel(Prev is N-1),
8        goal(calcFib(Prev,Max))
9      then
10       drop(calcFib(Prev,Max)).
11     if
12       goal(calcFib(N,Max)),
13       bel(
14         not(fib(N,F)),Prev is N-1,
15         PrevPrev is Prev-1, fib(Prev,FPrev),
16         fib(PrevPrev,FPrevPrev),
17         FN is FPrev + FPrevPrev
18       )
19     then
20       insert(fib(N,FN)).
21     if
22       goal(calcFib(N,Max)),
23       bel(fib(N,F),Next is N+1)
24     then
25       adopt(calcFib(Next,Max)).
26   }
27 }
```

Figure 4: The Fibonacci GOAL agent program.

The *Jason* agent program in Figure 5 consists of a belief base, a goal base and two rules. The belief base of a *Jason* agent is expressed by a logic-programming-like language, that incorporates facts, rules and strong negation. *Jason* uses a knowledge representation language that has been tailored specifically for *Jason*, instead of encapsulating an already existing one like 2APL and GOAL do. Like before, the belief base consists of two facts and the goal base contains a single goal. The first rule calculates the next Fibonacci-number and triggers the computation of the successive one. The second rule calculates the last number.

```
1  fib(1,1). fib(2,1).
2  !calcFib(3,1000).
3
4  +!calcFib(N,Max) :
5    N < Max & fib(N-1, Z1) & fib(N-2,Z2) & Z = Z1+Z2 <-
6    +fib(N,Z);
7    !!calcFib(N+1,Max).
8
9  +!calcFib(N,Max) :
10   N == Max & fib(N-1, Z1) & fib(N-2,Z2) & Z = Z1+Z2 <-
11   +fib(N,Z).
```

Figure 5: The Fibonacci *Jason* agent program.

We have implemented all three agent programs in accordance with two criteria: 1. all agent programs should yield agent runs that are similar (more on that later), and 2. the programs should execute as fast as possible. Note, that of course there are agent programs that perform faster, but would not yield the desired agent run. For example, a GOAL program that should calculate the Fibonacci-numbers, while the agent run is ignored, would look more elegant and would be faster.

4.2 Agent Runs and Similarity

In this part of the paper, we will show and elaborate on the similarity results gained when comparing the three agent programs directly and automatically using XYZ. We will inspect the agent runs, not in its entirety, but to an extent that makes our point clear.

This is an excerpt of the generic agent run generated by the 2APL agent:

1. $B = \{fib(2,1)., fib(1,1).\}$
   $G = \{calcFib(3,1000).\}$

2. $B = \{fib(2,1)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(4,1000).\}$

3. $B = \{fib(2,1)., fib(3,2)., fib(4,3)., fib(1,1).\}$
   $G = \{calcFib(5,1000).\}$

4. $B = \{fib(2,1)., fib(3,2)., fib(4,3)., fib(5,5)., fib(1,1).\}$
   $G = \{calcFib(6,1000).\}$

This is an excerpt of the generic agent run generated by the GOAL agent:

1. $B = \{fib(2,1)., fib(1,1).\}$
   $G = \{calcFib(3,1000).\}$

2. $B = \{fib(2,1)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(3,1000).\}$

3. $B = \{fib(2,1)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(4,1000)., calcFib(3,1000).\}$

4. $B = \{fib(2,1)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(4,1000).\}$

5. $B = \{fib(4,3)., fib(2,1)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(4,1000).\}$

6. $B = \{fib(4,3)., fib(2,1)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(4,1000)., calcFib(5,1000).\}$

7. $B = \{fib(4,3)., fib(2,1)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(5,1000).\}$

8. $B = \{fib(4,3)., fib(2,1)., fib(5,5)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(5,1000).\}$

9. $B = \{fib(4,3)., fib(2,1)., fib(5,5)., fib(3,2)., fib(1,1).\}$
   $G = \{calcFib(6,1000)., calcFib(5,1000).\}$

10. $B = \{fib(4,3)., fib(2,1)., fib(5,5)., fib(3,2)., fib(1,1).\}$
    $G = \{calcFib(6,1000).\}$

This is an excerpt of the generic agent run generated by the *Jason* agent:

1. $B = \{fib(2,1)., fib(1,1).\}$
   $G = \{calcFib(3,1000).\}$

2. $B = \{fib(2,1)., fib(1,1)., fib(3,2).\}$
   $G = \emptyset$

3. $B = \{fib(2,1)., fib(1,1)., fib(3,2).\}$
   $G = \{calcFib(4,1000).\}$

4. $B = \{fib(2,1)., fib(4,3)., fib(1,1)., fib(3,2).\}$
   $G = \emptyset$

5. $B = \{fib(2,1)., fib(4,3)., fib(1,1)., fib(3,2).\}$
   $G = \{calcFib(5,1000).\}$

	2APL		GOAL		*Jason*	
Fib	Step	Time	Step	Time	Step	Time
50	47	752.9	141	968.5	94	27.7
100	97	1226.5	291	2005.6	194	55.1
150	147	1652.7	441	3047.2	294	84.5
200	197	2048.0	591	4080.6	394	111.4
250	247	2457.4	741	5107.0	494	136.4
300	297	2854.0	891	6126.7	594	161.0
350	347	3158.8	1041	7154.2	694	185.3
400	397	3497.4	1191	8171.1	794	208.4
450	447	3885.3	1341	9186.9	894	230.1
500	497	4172.9	1491	10208.5	994	251.2

Table 1: Performance profiles for the three agents, showing the number of steps and the execution time.

6. $B = \{fib(2,1)., fib(5,5)., fib(4,3)., fib(1,1)., fib(3,2).\}$
   $G = \emptyset$

7. $B = \{fib(2,1)., fib(5,5)., fib(4,3)., fib(1,1)., fib(3,2).\}$
   $G = \{calcFib(6,1000).\}$

As you can see, the belief bases evolve in the same way, but the goal bases' evolutions differ greatly. The agents are n-B,G-similar with $n := 999$, $B := B_A$, and $G := \emptyset$. That is, when filtering the generic agent runs down to ones that only respect the belief base then the programs are similar for 999 steps, which is the exact number of different steps it takes to compute the first one thousand Fibonacci-numbers.

4.3 Performance Results

Finally, we will have a look at how fast the agents compute the first one thousand Fibonacci-numbers. Note, at this point and for the sake of comparability, that this task is very trivial. We have computed the number via a Java-program written from scratch. This took about $0.361ms$ on our machine[8].

As the Table 1 and Figure 6 clearly show, the *Jason* agent program performs best, followed by 2APL and GOAL. Now it becomes clear that we need a deeper examination on where the reasons for the differences in performance lie. We suppose that the use of the specific knowledge representation languages play a major role.

[8]MacBook Pro CoreDuo 2GHz with 2GB RAM running MacOSX Snow Leopard.

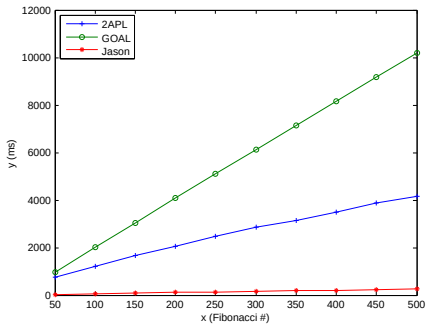

Figure 6: The performance of the three agents. The *Jason* agent is fastest, followed by the 2APL and GOAL agents.

5 Related Work

A wide variety of specialized *integrated development environments* is available [Pokahr and Braubach, 2009]. We will only consider those who are most interesting for our work. AgentBuilder [AgentBuilder Team,] is an agent platform and toolkit. The software is Java-based, commercial and supports the reticilar agent definition language (RADL). AgentFactory, on the other hand, is a cohesive framework for the development and deployment of multi-agent systems, which require the developer to restrict himself to a single flavor of agent [Muldoon et al., 2009]. That is, that the developer can either use an already existing agent interpreter (e.g. AF-APL) or develop a new one from scratch. We have already introduced the 2APL Platform [Dastani, 2008], which offers development and debugging facilities for agents written in the 2APL language, executed on the 2APL architecture. We have considered the GOAL interpreter [Hindriks, 2009] as well. The IDE allows for creating, editing, running and inspecting MASs and agents. The *Jason* IDE [Bordini et al., 2007] facilitates implementing agents based on an extension of the AgentSpeak(L) language [Rao, 1996] and developing environments in which these agents are situated. JIAC [Albayrak and Wieczorek, 1999] is an intelligent agent componentware, that is a tool suite and an agent platform. It uses the BDI-style language JADL. And finally, SPARK/eclipse [Morley and Myers, 2004] is a successor to PRS[Ingrand et al., 1996]. It runs as an Eclipse-plugin.

The multi-agent programming contest (MAPC) [Behrens et al., 2008] is an example for an infrastructure that allows for comparing the performance of different APL platforms and MASs developed using them. The main focus, however, lies more on comparing the MASs instead of on comparing the APL platforms. The performance of a MAS is determined in a series of games and is usually measured by the score achieved in a single game and the overall score achieved in all games. This score is a very abstract value (e.g. number of gold-nuggets collected, or number of cows herded) and thus does not reflect

that much of the internals of the considered systems. Statistics that would facilitate the comparison on a deeper level, values like response-times, and sizes/dynamics of the mental attitudes, are not gathered. We deem that it would be beneficial to combine our approach with the MAPC-infrastructure in order to establish the gathering of different performance values with similar agent programs.

6 Conclusion and Future Work

In this paper we have discussed a method to compare agent programs based on a notion of agent program similarity. We have also demonstrated a simple case study. The case study was implemented using the toolkit XYZ.

Because we have not fully reached one of our goals, that is the answer to the question whether a dedicated knowledge representation should be preferred over a portable one, we estimate that it would pay off to investigate a finer notion of similarity. For example we should lift the restriction of our approach to examine similar agent runs and replace it with an approach that is based on statistics about the execution of basic actions. That is, for example, the execution of actions that update the mental attitudes, perform a query on them, send messages to other agents or manipulating/querying the environment. We strongly believe that a second notion of similarity based on counting the execution of these basic actions would make sense.

Also, we believe that the environment should definitely be taken into account as well when comparing. An environment per se is a strong source for further data that may help judging the performance of a MAS. The environment interface standard (EIS) [Behrens et al., 2011] is an initiative to making environments portable and distributable. It would be a good idea to establish EIS-compatibility in XYZ to tap that source of further data.

Finally, it would make sense to tackle a more sophisticated case study. We have already mentioned the MAPC, which would be good to act as another case study. This way we could gather statistics about for example response time, that is the time between receiving percepts and acting, and sizes/dynamics of mental attitudes.

References

[AgentBuilder Team,] AgentBuilder Team. Agentbuilder. `http://www.agentbuilder.com/`.

[Albayrak and Wieczorek, 1999] Albayrak, S. and Wieczorek, D. (1999). Jiac - a toolkit for telecommunication applications. In Albayrak, S., editor, *Pro-*

ceedings of the 3rd International Workshop on Intelligent Agents for Telecommunication Applications (IATA 1999), pages 1–18. Springer.

[Behrens et al., 2011] Behrens, T., Dix, J., Koen Hindriks, M. D., Bordini, R., Hübner, J., Pokahr, A., and Braubach, L. (2011). An interface for agent-environment interaction. In *Post-Proceedings of ProMAS*.

[Behrens et al., 2008] Behrens, T. M., Dastani, M., Dix, J., and Novák, P. (2008). Agent contest competition: 4th edition. In *ProMAS*, pages 211–222.

[Bordini et al., 2007] Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons.

[Dastani, 2008] Dastani, M. (2008). 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248.

[Hindriks, 2009] Hindriks, K. V. (2009). Programming rational agents in GOAL. In El Fallah Seghrouchni, A., Dix, J., Dastani, M., and Bordini, R. H., editors, *Multi-Agent Programming:*, pages 119–157. Springer US.

[Ingrand et al., 1996] Ingrand, F., Chatila, R., Alami, R., and Robert, F. (1996). PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 1996)*, pages 43–49.

[Morley and Myers, 2004] Morley, D. and Myers, K. (2004). The spark agent framework. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 714–721, Washington, DC, USA. IEEE Computer Society.

[Muldoon et al., 2009] Muldoon, C., O'Hare, G. M., Collier, R. W., and O'Grady, M. J. (2009). Towards pervasive intelligence: Reflections on the evolution of the agent factory framework. In El Fallah Seghrouchni, A., Dix, J., Dastani, M., and Bordini, R. H., editors, *Multi-Agent Programming:*, pages 187–212. Springer US.

[of Pisa, 1202] of Pisa, L. (1202). *Liber Abaci*.

[Pokahr and Braubach, 2009] Pokahr, A. and Braubach, L. (2009). A survey of agent-oriented development tools. In El Fallah Seghrouchni, A., Dix, J., Dastani, M., and Bordini, R. H., editors, *Multi-Agent Programming:*, pages 289–329. Springer US.

[Rao, 1996] Rao, A. (1996). AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In de Velde, W. V. and Perram, J., editors, *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW 1996)*, pages 42–55. Springer.

[Russell and Norvig, 2003] Russell, S. J. and Norvig (2003). *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall.