Omniscient Debugging for Cognitive Agent Programs

Vincent J. Koeman, Koen V. Hindriks and **Catholijn M. Jonker**
Delft University of Technology, Mekelweg 4, 2628CD, Delft, The Netherlands
{v.j.koeman, k.v.hindriks, c.m.jonker}@tudelft.nl

Abstract

For real-time programs reproducing a bug by rerunning the system is likely to fail, making fault localization a time-consuming process. Omniscient debugging is a technique that stores each run in such a way that it supports going backwards in time. However, the overhead of existing omniscient debugging implementations for languages like Java is so large that it cannot be effectively used in practice.

In this paper, we show that for agent-oriented programming practical omniscient debugging is possible. We design a tracing mechanism for efficiently storing and exploring agent program runs. We are the first to demonstrate that this mechanism does not affect program runs by empirically establishing that the same tests succeed or fail. Usability is supported by a trace visualization method aimed at more effectively locating faults in agent programs.

1 Introduction

For traditional (cyclic) debugging to work, the program under investigation has to be deterministic. Otherwise, reproducing a bug by rerunning the program is likely to fail as it will not hit the same bug again or even hit different bugs [Engblom, 2012]. Real-time programs like multi-agent systems are typically not deterministic. Running the same agent system again more often than not results in a different program run or trace, which complicates the iterative process of debugging. Koeman *et al.* [2016a] also showed that the most frequently occurring type of failure in agent programs (a failure to select the right action) is caused by faults that occurred in a past state far from the point of detection. In other words, the root cause of a failure in an agent program is more often than not both far removed in time and in code (location).

Omniscient debugging is an approach to tackle these issues. Also known as reverse or back-in-time debugging, omnisicient debugging is a technique that originates in the context of object-oriented programming (OOP), allowing a programmer to explore arbitrary moments in a program's run by recording the execution. Such a 'time travelling debugger' is regarded as one of the most powerful debugging tools [Zeller, 2009; Bracha, 2012].

However, omniscient debugging is still not widely adopted. An important reason for this is that existing (OOP) implementations have a significant performance impact, with slowdown factors ranging from 2 to 300 times. Moreover, most existing solutions are heavy on memory or disk space requirements, requiring tens of gigabytes for a single trace.

The fact that the *agent-oriented programming* (AOP) paradigm is based on a higher level of abstraction compared to most other programming languages provides an opportunity to apply omniscient debugging techniques with a significantly lower overhead. The premise here is that tracing for AOP can be based on capturing only high-level decision making events instead of the lower-level computational events of OOP. Tracing techniques for AOP would thus need tracing of significantly fewer events while still being able to reconstruct all program states, making omniscient debugging for AOP more feasible in practice than for e.g. OOP.

The main contribution of this paper is the design of a tracing mechanism for cognitive agent programs that: (i) has a small impact on runtime performance; we show that our technique only has a 10% overhead instead of the much larger factors known from the literature (see Table 1); (ii) has virtually no impact on program behaviour; we empirically establish that the same tests succeed and fail with or without our tracing mechanism; (iii) can be effectively used for debugging, we propose a visualization technique tailored to cognitive agents and illustrate its application for fault localization. The key question we thus address is whether it is feasible and practical to apply omniscient debugging techniques to AOP without affecting testability.

The remainder of this paper is organized as follows. After discussing related work in Section 2, we propose a tracing mechanism in Section 3. In Section 4, we empirically establish that our approach does not affect the detection of failures. Section 5 introduces a trace visualization technique to support effective fault localization. The paper concludes with directions for future work in Section 6.

2 Related Work

Omniscient debugging is based on the idea that a developer explores a run that failed by reversing back into its execution to locate the corresponding fault, rather than trying to reproduce an observed failure in a separate (re)run as traditional (cyclic) debugging requires [Engblom, 2012]. This

Work	Method	Storage req.	Slowdown	Storag loc.
Lewis and Ducasse [2003]	Java bytecode instr.	100 MB/s	7-300x	in-memory[1]
Hofer *et al.* [2006]	Smalltalk bytecode instr.	1-100 MB/s	6-248x	in-memory
Pothier *et al.* [2007]	Java bytecode instr.	15 MB/s	10-115x	files
Lienhard *et al.* [2008]	VM modification	300 MB/s	2-7x	in-memory
Ko and Myers [2010]	Java bytecode instr.	1-10 MB/s	2-15x	files
This paper	Recording events	0.1 MB/s	1.1x	files

Table 1: A comparison of omniscient debugger implementations. Reported numbers have been rounded.

is especially useful for programs that have non-deterministic aspects (e.g., randomness) and/or rely on external resources (e.g., agent environments), as such programs generally do not behave exactly the same way on each run.

Fundamentally, it is impossible to reverse the execution of a program because for many operations there is no way to take the state after the operation and infer the state before the operation [Engblom, 2012]. *Omniscient debugging* facilitates 'going back in time' by recording an entire run of a program in a log, also called a *trace* of the run. Such a trace should allow any state of a program's execution to be correctly reconstructed. An intermediate form between cyclic and omniscient debugging is *record-replay debugging*, in which only the aspects of a run that cannot be reconstructed (i.e., by rerunning) are recorded. With this method, going back to a previous point in the execution requires restarting the run, and then feeding the recorded aspects back in at exactly the right times. Record-replay debugging is easier to implement, but also more time-consuming for developers, as a complete restart and re-run is needed to go back only one state in an execution. Moreover, it is not always possible to record all required (non-deterministic) aspects of a program, especially when a program relies on external resources such as external environments. Finally, we note that tracing a program for debugging purposes is different from manual instrumentation like in Lam and Barber [2005] or collecting (performance) measurements like in Helsinger *et al.* [2003].

Although omniscient debugging has been a research topic since the 1970s, one of the first influential attempts to apply this debugging technique to a modern programming language (Java) was performed by Lewis and Ducasse [2003]. In this paper, a proof-of-concept omniscient debugger is presented with the intention of demonstrating an upper bound for the costs of collection and display. Every change to every accessible object or local variable is recorded in memory separately for each thread by adding instrumentation code before every assignment and around every method call. The author claims that this proof of concept is effective for many kinds of bugs. A similar effort for Smalltalk was performed by Hofer *et al.* [2006], which also provides support for searching traces by queries (boolean expressions) specified in the language itself.

The work of Pothier *et al.* [2007] builds upon the work of Lewis, focusing on efficiency and usability. Events that are generated by (Java) bytecode instrumentation are stored in an on-disk database rather than in the program's memory space. Although this increases the capture cost, this provides better scalability as usually more disk space than memory is available, and reduces interference with the program memory itself. Moreover, this allows for post-mortem debugging by using previously recorded files. According to the authors, the benefits of omniscient debugging in quickly pinpointing hard-to-find bugs far outweigh any performance impact. The work of Lienhard *et al.* [2008] aims to further address performance issues by tracing at the virtual machine level.

A related tool created by Ko and Myers [2010] is *WHYLINE*, which allows developers to pose "why did" or "why didn't" questions about the output of Java programs. A trace is generated in memory through bytecode instrumentation, containing everything necessary for reproducing a specific execution. From this trace, a set of questions and according answers is generated. The authors note that their approach is not suited for executions that span more than a few minutes or executions that process or produce substantial amounts of data. However, their results do show that the approach enables developers to debug failures substantially faster.

Key aspects of the omniscient debuggers discussed in this section are compared with our mechanism in Table 1. We note that it is not possible to precisely compare the storage requirements (per second) and slowdown factors, as each work uses different programs (with varying amounts of activity in a run) in their evaluations, but the reported numbers do give a general indication of the various performance impacts.

A review of current state-of-the-art agent programming platforms shows that only three support something similar to a tracing mechanism[2]. 2APL [Dastani, 2008] provides an event-based mechanism that captures so-called reasoning steps in-memory. Jason [Bordini *et al.*, 2007] provides a similar mechanism, but, as far as we can tell, only captures (snapshots of) the full state of an agent after each of its decision cycles. AFAPL [Collier, 2007] also captures the full state of an agent after each decision cycle. It is not clear if these tracing mechanisms provide sufficient support for implementing an omniscient debugging technique. 2APL and Jason's mechanisms do not scale well as they show fast growing memory usage, and, as a consequence, will quickly cause a significant impact on an agent's execution. The mechanism store the snapshots in files, but it is unclear what the associated performance impact is. From the three platforms discussed only AFAPL supports searching in a trace for the occurrence of specific beliefs, but none of the platforms support more advanced navigation, querying or filtering of a trace. None of these platforms is able to relate agent states that are stored to the agent program's source code, which is another feature that a developer needs for effectively locating faults.

[1] There is a cut-off after 10.000 events on 32-bit systems.

[2] No work has been published on the 2APL and Jason mechanisms; conclusions were drawn from own observations.

3 Agent Trace Design

A tracing mechanism for cognitive agent programs should facilitate *reverting* an agent to any previous state by recording its execution. However, there are many ways to record a program's execution. Different solutions provide support for different techniques, ranging from record-replay debugging to support for full inspection which requires storing a *full trace*, i.e., all events, states, and actions performed in the run. Such a *full trace* that captures each individual state completely in practice is not feasible, as it takes 5-20x more time to execute an agent system and requires more than 50-200x the storage space needed for other mechanisms (see row *Full* in Table 2).

In this section, we take an incremental approach to the design of a tracing mechanism. We begin with an initial mechanism that stores a trace that captures as little information as possible but still provides sufficient information for record-replay. In this first step, a minimal trace is constructed based on *events* that, however, limits the options for a developer to (rapidly) locate points of interest in a trace and establish meaningful relations between these points. Step two extends the trace with information about *state changes* that allows efficient reconstruction of a previous state. In the third step, we add *source code information* associated with points in a trace to enable a debugger to more effectively explore the trace. At each step, we try to minimize the additional time and space resources needed and evaluate the impact of the tracing mechanism on the agent system's (runtime) performance. All evaluations were performed on a Linux server with a quad-core Intel i7 processor and 6GB of RAM. Finally, we discuss how to store traces for later use. In Section 4, we show that the behaviour of different sets of agents in different environments is not significantly affected.

3.1 Tracing Events (Record-Replay)

In order to facilitate record-replay debugging, we need to determine which aspects *must* be stored in order to reconstruct any previous state of an agent program's execution by replaying. Assuming for the moment that agents themselves are deterministic (we will relax this assumption later), events such as percepts from an environment or messages from other agents would be the only items that need to be recorded in the trace. This is true because re-running an agent program does not guarantee the same events to be produced, as the environment is external and asynchronous and because multiple agents generally run concurrently in separate threads without a strict scheduling mechanism. By re-running the agent program with an initially empty set of events and by feeding the right events to the agent program at the right time to 'imitate' the environment and/or other agents, the run can be reconstructed from this *event trace* and the agent can be replayed.

An event trace can be implemented by storing *a snapshot of all events* that happened after each change. However, two optimizations can be applied to agent systems. First, events typically need to be stored only once per agent cycle. Second, only changes in consecutive snapshots need to be stored. It is more efficient to only store events that have been added and deleted compared to the last snapshot as the rate of environment change typically is slow compared to the execution time of a single agent cycle and on consecutive cycles e.g. only a few changes to percepts are received. Storing the changes to events (e.g., with a listener pattern) thus only requires a simple comparison check with previous snapshots, and the required storage is linear in terms of this.

As a method to determine the performance impact of the tracing mechanism, we compare the average amount of cycles that agents performed in one minute. We used a randomly selected program from a pool of GOAL [Hindriks, 2009] multi-agent systems with four (different) agents that control bots in the highly dynamic UT3 environment [Hindriks *et al.*, 2011]. The system first was ran ten times without any tracing enabled, and then ten times with the record-replay mechanism. Although runs are different due to the dynamics of the environment, the run settings (e.g., the map, the number of computer opponents, etc.) were identical in each case. We note that if a GOAL agent receives the same events as in a previous cycle and performs no new action in the environment, it (but not the environment entity) 'sleeps' until a new event occurs; we therefore actually report the number of 'effective cycles' an agent performed in one minute.

The results of these runs are summarized in the rows labelled *None* and *Events* in Table 2; columns match with each of the four agents with an additional column for totals. The results indicate that there is no significant difference in cycle numbers when the event tracing mechanism is enabled or not. Less than 2MB of storage space is needed per agent per minute, with about 100 events generated on average per second.We also established that space requirements grow linearly over time, i.e., no more than 20MB is required when agents are executed for ten minutes.

An important usability metric is how long it takes to reconstruct a program state. In a record-replay mode, it is clear that stepping from the final to the initial state takes no (significant) time at all, as this simply means restarting the agent. Moving forward in time to a next state is also fast as the agent does not need to be restarted. However, going just a single step backwards in time, i.e., to a previous state compared to the current state, an agent will need to be re-started and almost re-run completely to reconstruct that state.

To obtain an indication of our usability metric, we first established that re-playing our example agent programs to obtain the final state starting from the initial state using the event trace takes about 2.5 seconds. Given this measurement, navigating to an arbitrary state in a run in order to inspect that state will take about 1.25 second on average. Users, moreover, will want to evaluate queries to identify states they need to inspect, and in our test cases this will take more than 2.5 seconds as a query will need to be evaluated on each state that is reconstructed as well. For an agent that has run for just one minute (even though in a highly dynamic environment), this means a waiting time of more than 4% relative to execution time, which in practice is quite high.

3.2 Tracing State Changes

We have assumed that the execution of agent programs is deterministic, but this assumption does not hold for multi-agent systems as, e.g., the scheduling of execution steps of agent programs is non-deterministic. Moreover, a single agent can contain non-deterministic choice points like selecting a ran-

Tracing	Cycles 1	Cycles 2	Cycles 3	Cycles 4	Total C.	Space 1	Space 2	Space 3	Space 4	Total S.
None	496	558	1453	749	3256					
Events	506	548	1432	780	3266	2MB	2MB	2MB	2MB	8MB
	+2%	-2%	-1%	+4%	+0%					
Changes	480	517	1281	696	2974	4MB	4MB	4MB	4MB	16MB
	-5%	-6%	-11%	-11%	-9%	+100%	+100%	+100%	+100%	+100%
Changes	469	501	1293	674	2937	5MB	5MB	6MB	5MB	21MB
+ sources	-2%	-3%	+1%	-3%	-1%	+25%	+25%	+50%	+25%	+31%
Full	77	78	83	79	316	394MB	385MB	391MB	390MB	1560MB
	-84%	-84%	-94%	-88%	-89%	+7780%	+7600%	+6417%	+7700%	+7329%

Table 2: An evaluation of different tracing methods by comparing the average amount of cycles over ten 1-minute runs of a system with 4 agents operating in the UT3 environment and the corresponding average amount of storage space that is required. The percentages that are given for a method are relative to the method directly above in the table.

dom element from a list or evaluating rules in a random order that will cause a different trace to be generated even with identical input. It is generally not possible to account for all such points, especially if they are at the knowledge representation level (and thus not explicitly represented in the agent programming language). The substantial waiting times are thus not the only reason why a record-replay approach for agent systems will not be useful in practice for agent systems.

In order to reduce the amount of time a navigation step in the trace takes on average and to facilitate non-deterministic agents, we need to make sure that an agent's state can be reconstructed without requiring re-execution. As storing each state in full is infeasible (see Table 2), we propose a mechanism that in addition to the changes to events also records all changes to an agent's cognitive state. The idea is that by recording all event and state changes, a navigation step can be performed by reconstructing a state by applying all changes between the current state and that target state.

The changes that need to be recorded differ per programming language. For the GOAL language, each change to an agent's beliefs or goals needs to be stored (besides the event changes related to percept and messages). For other agent programming languages that include notions like plans for example, a new intention that is scheduled or a change that pushes a new plan on an intention also needs to be recorded.

Note that it is not sufficient to store the actions performed by an agent program. For example, the actions of inserting a belief that the agent already has or dropping a goal the agent does not have, do not change the agent's cognitive state. In order to be able to navigate back in time, we need to know how we can 'roll back' each action to reconstruct a previous state. For each action performed by an agent that can change the agent's state, therefore, the real change brought about by that action given the agent's current state needs to be computed and stored in the trace. In other words, while executing an agent program, the mechanism needs to store aggregations of items that have been added to and/or removed from a state.

It is also not sufficient to 'instrument' program code to record state changes. Although most state changes correspond to an action that is performed as part of an agent program, they do not always originate directly from program code. For example, accomplishing a goal results in removing that goal automatically from an agent's goal base in GOAL. This means that the tracing mechanism has to be integrated into the virtual machine or interpreter of an agent platform.

As before, we analysed the performance of the *state change* tracing mechanism discussed in this section. The main results are summarized in the row labelled *Changes* in Table 2. Compared to event traces, we now see that on average the number of cycles has decreased by almost 10 percent. Even though this is still much better than the overhead introduced for traditional languages (see Table 1), further evaluation is required, and in Section 4 we will determine whether agent behaviour has been changed to a point where it affects debugging or not. The space requirements have doubled, but on average still less than 4MB per agent per minute is required. The gain we achieve by increasing space usage is that our metric of navigation speed has been much improved: fully reversing an agent's state (either from first to last state but now also the other way around) takes only 0.5 seconds on average, less than 1% of the original execution time, and a substantial speedup compared to record-replay. As the time needed to evaluate queries remains the same, the speed-up factor for search queries on a trace will be lower, but only slightly so, as the time needed for evaluating a query on a state compared to reconstructing a state is almost negligible.

3.3 Tracing Source Code Locations

The state change tracing mechanism supports efficient reconstruction of a program's run. It also facilitates debugging by enabling fast querying of traces to identify unexpected state changes. But it does not yet support fault localization, as it is hard to relate such state changes to program code; the information about the state change itself does not specify where in the agent program it was brought about. For effective fault location, ideally, a tracing mechanism is fully integrated into the existing development facilities of an agent programming platform such as, for example, single-step execution debugging and automated testing. The integration with automated testing in general is relatively straightforward as our tracing mechanism makes a program run available for exploration immediately when a test failure is detected. Test conditions (that fail), moreover, also provide useful clues for executing search queries or applying filters on a trace.

The integration of our tracing mechanism with a source-level debugger, however, is more complicated. Ideally, a developer is able to follow the same stepping flow s/he can create with a source-level debugger but now also in the reverse

direction using the recorded trace. But even if we are given a 'current' source code location and are able to revert to a previous state (given a trace), it is not clear how to 'reverse step' through the source code because there are many paths through a program that can result in the same state. It is not clear whether it is possible to reconstruct a single path from local state change information only, and even less clear how to do that efficiently. Instead, we therefore propose to support 'reverse stepping' by adding *source code location markers*, i.e., the traversed execution events/breakpoints to an agent's trace. This means storing a trace that not only records events and state changes, but also the full path of source code locations that is traversed while executing an agent program. In order to save space, each code location in an agent program is encoded as an integer number and this number instead of the file-based code references are stored together with an inverse mapping to retrieve the locations from these numbers.

To evaluate the impact of also storing code locations, we used the source-level debugging framework proposed in Koeman *et al.* [2016b] to implement such a tracing mechanism and recorded all possible breakpoints, i.e. code locations that are traversed when 'single-stepping' an agent in forward mode. The main results are summarized in the row labelled *Changes + sources* in Table 2. They show that recording source code locations does not have a big impact on the average number of cycles. The additional space needed to store traces was about 25%. It is worth noting though that space needed still only grows linearly over time, e.g., after ten minutes, our traces grew to roughly 50MB per agent (<0.1 MB/s).

3.4 Trace Storage

In order to minimize the impact of the tracing mechanism, the information that needs to be stored can be written to an (in-memory) queue. Due to the possible size of this queue and the memory requirements of agents themselves, this queue will need to be flushed to some more permanent storage in a thread(pool) that is separated from the agent runtime, preferably with a lower priority. One of the most efficient ways to do this is by using memory-mapped files [Roselli *et al.*, 2000], as they facilitate the best I/O performance for large files by mapping between a file and memory space, enabling an application to modify the file by reading and writing directly to the memory. Using files also facilitates debugging a trace at a later point in time (i.e., loaded from the file) and interaction with tools external to the agent runtime itself.

4 Evaluation

The main purpose of a tracing mechanism is to support debugging. It therefore is important to establish that the tracing mechanism does not significantly change the behaviour of an agent program. For debugging purposes, it is most important that the failures that occur in the agent system executed without tracing also occur when runs are being traced. It is also important to establish that the reproduction of a failure while tracing a program will not take many more runs and thus more time. In this section, we empirically investigate and compare the performance and reproduction of a failure in an agent (system) with and without our 'state change and source location' tracing mechanism enabled. Whilst in the previous section we used Unreal Tournament (with 4 agents) for evaluation, in this section we will use the Blocks-World-for-Teams (BW4T; Johnson *et al.* [2009]) environment with varying numbers of agents. Of all EIS-compatible environments [Behrens *et al.*, 2011] available to us ready for testing, UT3 and BW4T are the most dynamic.

4.1 Method

The method we used is to first run a given test set many times in order to identify all failures in an agent system (c.f. Koeman *et al.* [2016a]). We then ran multiple sessions in which those same tests were repeated in order to determine how many repetitions are needed (on average per session) to reproduce all the failures that were initially found. We used a time-out parameter as agent systems that produce many failures may run indefinitely without making any progress (i.e., producing no new results). Failures that are due to a time-out are ignored because these cannot be consistently reproduced. Our method, illustrated in Fig. 1, uses these parameters:

- B: number of initial test runs for each agent system.
- T: number of seconds after which a test is aborted.
- N: number of 'sessions' for each agent system.
- M: maximum number of test runs in a single session.

The aim is to empirically determine the reproduction factor R: the number of times a run needs to be repeated to detect all failures in an agent system. We use R to evaluate our tracing mechanism, but our experiments also contribute useful insights into the testability of agent systems.

We used two sets of agent systems programmed in GOAL that control robots in the BW4T environment. They were created by pairs of first-year Computer Science bachelor students and handed-in with accompanying tests. The first set consists of 84 single-agent systems, and the second set of 42 multi-agent systems (3 agents).

4.2 Results

We applied our method to both sets of agents, with parameter B set to 100, T to 60 seconds, N to 10, and M to 1000. These parameters were chosen after an iterative process of running experiments to minimize the runtime whilst making sure that unreproducible nor new failures would be found in the repetition part of our method (see also the step to increase parameter B in Fig. 1). We first ran agents with tracing turned off and then ran the same agents again with tracing turned on but skipped the first step (see Fig. 1) as the goal is to establish whether failures are reproduced also when tracing is turned on. In total, for our final experiment, almost 23,000 runs were performed with a total runtime of about 330 hours.

We found a significantly lower number of failures for the single-agent systems (on average 0.3 failures, with a maximum of 8 failures) than for the multi-agent systems (on average 4.3 failures, with a maximum of 21 failures). In a rather static environment like BW4T with a low number of failures, we found that a single agent's failure set can be reproduced on every single run both with and without tracing enabled, i.e., $R = 1$. This is very different for multi-agent systems where on average $R = 11$ runs were needed to reproduce all failures that were initially found. Most importantly, this estab-

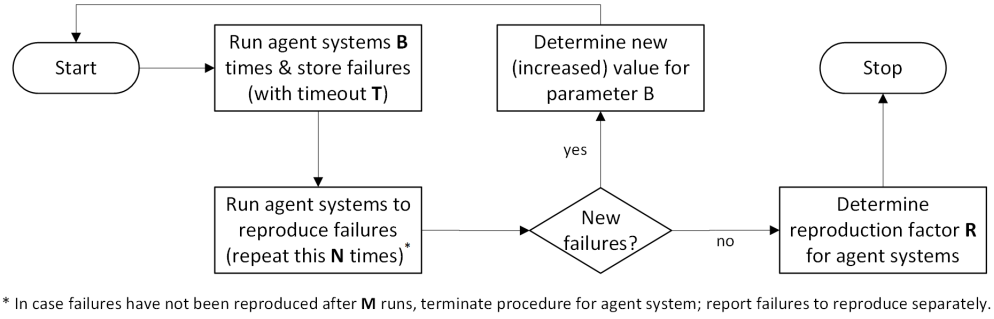

Figure 1: A flowchart of our evaluation method.

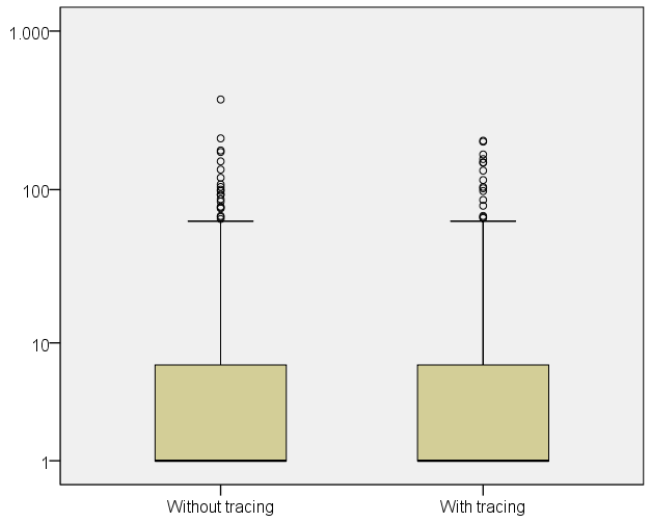

Figure 2: Distribution of R with and without tracing.

lishes that all failures could be reproduced (and no new failures were introduced) when the tracing mechanism is used. The distribution of R for both with and without tracing enabled is shown in Fig. 2. Even though on average only 11 runs were needed, in a few cases up to even 300 repetitions were needed to reproduce all failures. The high number of runs required in these cases provides a strong indication that omniscient debugging is a technique that is needed in practice to be able to debug multi-agent systems. When comparing the distributions for R using a Wilcoxon signed-rank test, no statistically significant difference is found ($Z = -0.79$, $p = 0.43$). This provides additional support for the claim that the tracing mechanism does not impact the agent system's execution. Finally, we note that a few extreme outliers where >500 runs were required were excluded from these results.

5 Visualizing Traces

For efficient fault localization, it needs to be easy for a developer to identify states in a program's execution that are related to the failure under investigation. Moreover, a developer should not get lost in navigating between these states, but always have a sense what point in the execution s/he is evaluating and how the current state affected the execution.

We adapt the concept of a *space-time view* first developed in Azadmanesh and Hauswirth [2015] in the context of Java programming to cognitive agent programming. A space-time view is a table that is structured along space and time dimensions, where the rows in the table correspond to the space dimension, which is composed of the different elements in a state that are traced. Each cell indicates whether an element was modified by executing an operation or only accessed for inspection at a specific time (the columns in the table).

For cognitive agents, the elements in a space-time view that are traced are the agent's events, beliefs, goals, actions, plans, and/or modules (i.e., sets of decision or plan rules). Assuming a basic representation of a name with associated parameters is used to represent these elements, we use the corresponding *signatures* as the rows in the space dimension. For example, the signature `print/1` in Fig. 3 represents a `print` action with one parameter. Each point (event, state change, source code location) in a trace represents a step (column) in the time dimension. Multiple space elements (signatures) can be used in a single step, e.g., evaluating a query may require accessing several beliefs and goals. The cells in our space-time view contain information about how an element was used at a particular step, which differs per type of element (e.g., a belief can be modified or inspected, an action or plan can be called and performed, a module can be entered or exited). Empty cells indicate the element was not used. An example of a space-time view for a simple agent is shown in Fig. 3.

A developer can use and manipulate a space-time view in several ways. The signatures listed in the space time view can be ordered based on type (beliefs next to beliefs) or alphabetically (using the signature names). A user can also apply queries or filters to a trace both textually as well as through selecting cells of interest or rather cells that should be hidden in the table; see, for example, the bottom table where only the first row is selected by a user in Fig. 3. A user can click on any cell in the table in order to step the agent to the state matching that cell's column (either forwards or backwards through its execution), allowing a developer to use all debugging tools (e.g., inspecting or modifying an agent's beliefs and goals) in that specific historic state.

We illustrate the use of such a space-time view for analysing a failure of the following example test condition associated with a BW4T agent program:

```
goal(holding(B)), bel(atBlock(B))
leadsto done(pickUp(B))
```

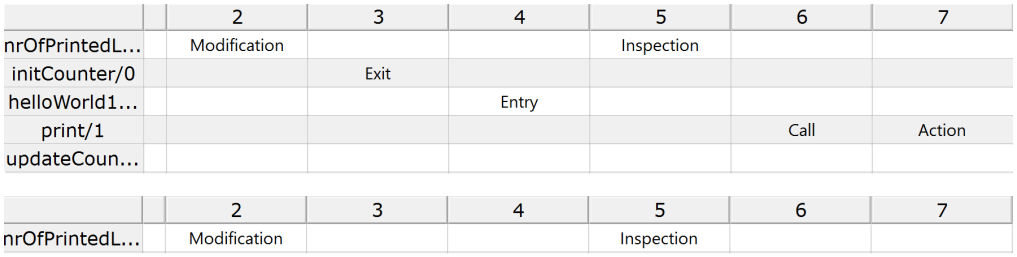

Figure 3: Space-time view (top) and filtered version (bottom)

This condition expresses that if the agent has the goal to hold block `B`, and believes it is at the block, that it should (eventually) pick up `B`. A failure to do so will lead to failure of the test condition (i.e., when the agent is terminated). Without an omniscient debugger, a developer would need to restart the agent, navigate to a point where the goal-believe query holds (*assuming* it will at some point in the *restarted* run), and continue by manually stepping to try to understand why the action is not performed. With an omniscient debugger, we do *not* need to restart the agent, and can use the clues provided by the test condition itself to navigate to the last time that `holding/1` and `atBlock/1` were modified in the space-time view. We can do so either by double-clicking the corresponding cell, or, even faster, by using the query `goal(holding(B)), bel(atBlock(B))` to filter the trace. Note that such a point *must* exist in the run as the test condition failed on the exact same run that was traced. Because our tracing mechanism also traces source code locations and is integrated with a source-level debugger, a developer can now step from that point through the source code *as if* it is executed for the first time (and go backwards whenever needed). In our example, it quickly became clear to the developer that some decision rules were incorrectly ordered, which prevented the `pickUp` action from being executed.

6 Conclusion

We design a tracing mechanism that supports omniscient debugging for cognitive agents, a technique that facilitates debugging by moving backwards in time through a program's execution. We evaluate and demonstrate empirically that the mechanism is efficient and does not substantially affect the runs of program in the sense that the same failures can be reproduced when the mechanism is turned on and off. This essentially shows that our mechanism is fast enough and can be used in practice for debugging failures without a need to rerun a program.

We also introduce a trace visualization method tailored to cognitives agents based on a space-time view of the execution history. A developer can navigate this view, evaluate queries on a trace, and apply filters to it to obtain views of only the relevant parts of a trace. Our approach is integrated with a source-level debugger and traces source code locations, which enables a developer to single-step through a program's execution history and facilitates fault localization.

Future work will include a user study to evaluate the usability of our omniscient debugging approach for programmers. This paper's findings that it can be hard to reproduce a failure at least sometimes also prompt the need for further investigation into how failure reproduction for multi-agent systems can be improved. Finally, we believe that our tracing mechanism can provide a starting point for a history-based explanation mechanism that can automatically answer questions such as 'why did this action (not) happen?' [Hindriks, 2012].

References

Mohammad R. Azadmanesh and Matthias Hauswirth. Space-time views for back-in-time debugging. Technical Report 2015/02, University of Lugano, 2015.

Tristan M. Behrens, Koen V. Hindriks, and Jürgen Dix. Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence*, 61(4):261–295, 2011.

Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, Ltd, October 2007.

Gilad Bracha. Debug mode is the only mode. `https://gbracha.blogspot.nl/2012/11/debug-mode-is-only-mode.html`, November 2012. Accessed: 2017-02-19.

Rem Collier. Debugging agents in Agent Factory. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems: 4th International Workshop, ProMAS 2006, Hakodate, Japan, May 9, 2006, Revised and Invited Papers*, pages 229–248. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

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

J. Engblom. A review of reverse debugging. In *System, Software, SoC and Silicon Debug Conference (S4D), 2012*, pages 1–6, September 2012.

Aaron Helsinger, Richard Lazarus, William Wright, and John Zinky. Tools and techniques for performance measurement of large distributed multiagent systems. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '03, pages 843–850, New York, NY, USA, 2003. ACM.

Koen V. Hindriks, Birna van Riemsdijk, Tristan Behrens, Rien Korstanje, Nick Kraayenbrink, Wouter Pasman, and

Lennard de Rijk. Unreal GOAL bots. In Frank Dignum, editor, *Agents for Games and Simulations II: Trends in Techniques, Concepts and Design*, pages 1–18. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

Koen V. Hindriks. Programming rational agents in GOAL. In Amal El Fallah Seghrouchni, Jürgen Dix, Mehdi Dastani, and Rafael H. Bordini, editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 119–157. Springer US, 2009.

Koen V. Hindriks. Debugging is explaining. In Iyad Rahwan, Wayne Wobcke, Sandip Sen, and Toshiharu Sugawara, editors, *PRIMA 2012: Principles and Practice of Multi-Agent Systems*, volume 7455 of *Lecture Notes in Computer Science*, pages 31–45. Springer Berlin Heidelberg, 2012.

Christoph Hofer, Marcus Denker, and Stéphane Ducasse. Design and implementation of a backward-in-time debugger. In *NODe 2006*, volume P-88, pages 17–32, Erfurt, Germany, 2006. GI.

Matthew Johnson, Catholijn Jonker, Birna van Riemsdijk, Paul J. Feltovich, and Jeffrey M. Bradshaw. Joint activity testbed: Blocks World for Teams (BW4T). In Huib Aldewereld, Virginia Dignum, and Gauthier Picard, editors, *Engineering Societies in the Agents World X: 10th International Workshop, ESAW 2009, Utrecht, The Netherlands, November 18-20, 2009. Proceedings*, pages 254–256. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

Andrew J. Ko and Brad A. Myers. Extracting and answering why and why not questions about Java program output. *ACM Transactions on Software Engineering and Methodology*, 20(2):4:1–4:36, September 2010.

Vincent J. Koeman, Koen V. Hindriks, and Catholijn M. Jonker. Automating failure detection in cognitive agent programs. In *Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '16, pages 1237–1246, Richland, SC, 2016. International Foundation for Autonomous Agents and Multiagent Systems.

Vincent J. Koeman, Koen V. Hindriks, and Catholijn M. Jonker. Designing a source-level debugger for cognitive agent programs. *Autonomous Agents and Multi-Agent Systems*, 2016.

D.N. Lam and K.S. Barber. Comprehending agent software. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '05, pages 586–593, New York, NY, USA, 2005. ACM.

Bil Lewis and Mireille Ducasse. Using events to debug Java programs backwards in time. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 96–97, New York, NY, USA, 2003. ACM.

Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings*, pages 592–615. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *SIGPLAN Notices*, 42(10):535–552, October 2007.

Drew S. Roselli, Jacob R. Lorch, Thomas E. Anderson, et al. A comparison of file system workloads. In *USENIX annual technical conference, general track*, pages 41–54, 2000.

Andreas Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.