LightJason
A BDI Framework Inspired by Jason

Malte Aschermann*, Philipp Kraus, and Jörg P. Müller

Department of Informatics, Clausthal University of Technology, Julius-Albert-Str. 4
D-38678 Clausthal-Zellerfeld,Germany
{malte,philipp}@lightjason.org,joerg.mueller@tu-clausthal.de
http://lightjason.org

Abstract. Current BDI agent frameworks often lack necessary modularity, scalability and are hard to integrate with non-agent applications. This paper reports ongoing research on *LightJason*, a multi-agent BDI framework based on *AgentSpeak(L)*, fine-tuned to concurrent plan execution in a distributed framework; *LightJason* aims at efficient and scalable integration with existing platforms. We state requirements for BDI agent languages and corresponding runtime systems, and present the key concepts and initial implementation of *LightJason* in the light of these requirements. Based on a set of requirements derived for scalable, modular BDI frameworks, the core contribution of this paper is the definition of a formal modular grammar for *AgentSpeak(L++)*, a modular extension of *AgentSpeak(L)*, and its underlying scalable runtime system. A preliminary validation of *LightJason* is given by means of an example evacuation scenario, an experimental analysis of the runtime performance, and a qualitative comparison with the Jason platform.

Keywords: agent programming language, scalability, multiagent-based simulation

1 Introduction

Agent-oriented programming (AgOP) [18] is about building systems consisting of *software agents* maintaining mental states, based on declarative logical languages. The Belief-Desire-Intention (BDI) paradigm [16] has become the prevalent approach to AgOP and multi-agent systems (MAS). Such agent programs consist of statements in first-order logic, allowing agents to deduce new facts, commit to plans and eventually execute actions. A very popular language for programming BDI agents is *AgentSpeak(L)* [15]. *Jason* [4] has been instrumental to the popularity of *AgentSpeak(L)* by providing a BDI agent framework that combines an extension of *AgentSpeak(L)* with an interpreter and provides integrated development environment (IDE) plugins for JEdit and Eclipse. However,

* Parts of this work were supported by the German Research Foundation (DFG) through the Research Training Group *SocialCars: Cooperative (De-)centralized Traffic Management (GRK 1931)*.

analysing the level of usage of BDI agent frameworks in software engineering practice reveals a sobering picture. A look at the major programming indices Tiobe [20], Redmonk [17] and PopularitY [13], which measure the popularity of programming languages, shows that the world of practice is still dominated by imperative and object-oriented languages. Only Tiobe lists any logic-based languages: The major proponent *Prolog* is ranked 33rd. AgOP languages are not represented at all. Furthermore, in their study of MAS application impact, [12] show that among the agent languages, the only 'true' BDI language with some application impact is Jack, a proprietary language, while the use of languages like *Jason* or *GOAL* is restricted to academic prototypes. The hypothesis underlying our research is that part of the reasons for this dire state are elementary shortcomings of AgOP languages regarding modularity, maintainability, software architecture interoperability, performance, and scalability. This paper reports ongoing research on a multi-agent framework based on *AgentSpeak(L)* which aims at an efficient and scalable integration into existing platforms, enabling non-agent-aware systems to incorporate agent-based optimisation techniques to solve distributed problems. We present the initial version of *LightJason*, a BDI agent framework fine-tuned to concurrent plan execution in a distributed environment.[1]

2 Requirements and State of the Art

Requirements Over the past years, we have gained experience in modelling and engineering multi-agent applications based on the BDI paradigm (most notably in domains of traffic and industrial business processes), but also with developing agent programming languages and runtime platforms. While we consider the BDI abstraction appealing and intuitive for modelling sociotechnical systems, we were often confronted with the limitations of today's agent platforms. From these limitations, we derived a number of requirements for BDI agent platforms, which extend the list of general requirements from [4, p. 7]) and are summarised as follows: 1) Integrability in existing software architectures. 2) Modularisation of agents and underlying data structures. 3) Agent scripting language with strict language syntax. 4) Action checking during parsing time, not during run time. 5) Avoid *action-centric* reasoning cycle as argued by [1]. 6) Parallel execution of plans in separated execution tasks. 7) Agent generation mechanism for easy instantiation of large numbers of agent. 8) Hierarchically structured belief bases and actions in semantic groups.

Discussion of state of the art The main concepts of BDI frameworks are mostly based on the Procedural Reasoning System (PRS) [8,7] and the first robust implementations such as dMARS [6]. As [10] and subsequent surveys point out, virtually all existing multi-agent frameworks are not designed for productive use (performance, scalability) and easy integration with specific domains. The design of agent-based scripting languages leads to challenges in maintainability;

[1] For a much more comprehensive version of this short paper, we refer to [2].

e.g. Bordini et al. [3, p. 1300] state that: *"[T]he AgentSpeak(L) code is not elegant at all. The resulting code is extremely clumsy because of the use of many belief addition, deletion, and checking (for controlling intention selection) [...] [and] thus a type of code that is very difficult to implement and maintain."* Though this is a paper from 2002, the situation has not changed much. MAS platforms like *Jason* provide a separate runtime system, these approaches raise issues regarding scalability and consistency, especially when combining existing systems with MAS. In the case of *Jason*, this also can lead to ill-defined execution behaviour of agents, especially regarding clarity when an iteration of the agent control cycle has ended (see requirement 2 above).

In this paper, we focus on the comparison with *AgentSpeak(L)*/*Jason* as the most prominent (open-source) representative of BDI languages/platform. We compared the legacy *Jason* 1.4 branch, which is still in use in our research group for small-scale agent-based traffic simulation (e.g. [5]), and the quite recently published *Jason* 2.0 branch with our requirements. *Jason* 1.4 lacks support for all the above-mentioned requirements except a partially support for modular agents (requirement 2), due to its `include` functionality. *Jason* 2.0 additionally supports a hierarchical structuring of agents (requirement 2), but this feature is limited to beliefs and plans[2]. Also, one new feature of *Jason* 2.0 is parallel execution of plans [22], which addresses requirement 2. However, like *Jason* 1.4, *Jason* 2.0 still heavily relies on synchronised data structures in their architecture design, implying slow-downs due to locking and CPU context switches during each agent cycle. In their approach adding concurrency to the reasoning cycles in *Jason*, [22] provided benchmark results regarding scalability; their test setup with only two CPU cores and synthetic benchmarks (e.g. nested for-loops and Fibonacci sequence) resulted in a linear increase in execution time for up to 500 agents, which would also be expected for single-thread applications.

In order to tackle the above requirements, we start from the architecture design of Multi-Agent Scalable Runtime platform for Simulation (*MASeRaTi*) [1], as an attempt to tackle the scalability issues in modern MAS. We created a modified, light version of *AgentSpeak(L)* (named *AgentSpeak(L++)*) and build a Java-based implementation of the *MASeRaTi* architecture.

3 *LightJason* Architecture and Data Model

There is broad agreement in the AgOP literature that *"[a] multi-agent system is inherently multithreaded, in that each agent is assumed to have at least one thread of control [21, p. 30]"* meaning that agents should be able to pursue more than one objective at the same time. To implement this conceptual notion of concurrency at the technical level, we refer to the basic notion of a thread [19] as a *"lightweight process"*, and that all threads are running within the same process. Thus, in *LightJason*, an agent is be controlled by a thread during the reasoning process and stores all data for the reasoning internally, by following the thread-local-storage model. Our general approach in *LightJason* is to conceive *AgOP* as

[2] `https://git.io/vXqup`

a combination of *Imperative*, *Object-Oriented* and *Logic Programming*, see [2, p. 6]. To get into a more detailed view, an agent is not one single software component but it is split up into two different elements, i.e. *agent-mind* and *agent-body*. This approach is a reverence to the Mind-Head-Body model proposed by Steiner in [9]. The symbolic representation of an agent's mind is stored as *logic literals*, as in *Prolog* or *AgentSpeak(L)*. All literals of *LightJason*'s agents are stored within a belief base for getting access during runtime. During execution the agent asks for particular literals, initiating a *unification* process. As this process is run many times, we optimised the internal data structure representing the logic elements for parallel execution and avoiding cost-intensive back-tracking. The *Imperative Programming* paradigm is used to describe the execution behaviour of agents in *LightJason* (similar to the Patterns of Behaviour (PoBs) in the InteRRaP architecture [11]). In contrast, to InteRRaP, we provide for parallel execution of PoBs, so that actions, assignments or expressions can be run or evaluated in parallel. Finally, *LightJason* is Java-based; the internal representation of agents is written in an Object-Oriented Programming (OOP) style with *concurrent data structures*, allowing us to create inheritable agent objects running in a multi-threading context and easier integration with domain-specific software systems. To further parallelise execution and gain more scalability, we made extensive use of state-of-the-art Java techniques, such as lambda-expressions[3] and streams[4].

4 *AgentSpeak(L++)* Language Definition

We regard an agent as a hybrid system, which combines different programming language paradigms, allowing programmers to describe complex behaviour. This abstract point of view allows a flexible structure – also for non-computer scientists – to parameterise or specify a software system. The whole syntax was designed as a *logic programming language*, by which all elements could be reduced to *terms*[5] and *literals*[6], defining a symbolic representation of behaviour and (environment) data. This allows modelling a generalised multi-agent system, which can later be concretised for different applications, i.e. scenarios and supports the agent programmer to design the behaviour by scripting beliefs, rules, plans and actions. Our first contribution is the definition of a *scripting language* based on a modified and extended *AgentSpeak(L)* grammar. We modularised the grammar into subgrammars to obtain a more abstract structure of the agent programming language. The main grammar definition of *LightJason* is hierarchically structured into the modules depicted in Figure 1 and is explained in detail in [2, p. 7ff.]. This allowed us to get a more flexible parsing component, which could be split up into a layer-based structure.

Built-in Actions The language structure and the underlying architecture of our implementation allows to create a flexible action interface. In comparison to

[3] `http://www.webcitation.org/6lfbGeOlc`
[4] `http://www.webcitation.org/6lfbNP7nX`
[5] Term: `https://git.io/viKWQ`, EBNF: `https://git.io/viKWx`
[6] Literal: `https://git.io/viKlt`, EBNF: `https://git.io/viKlI`

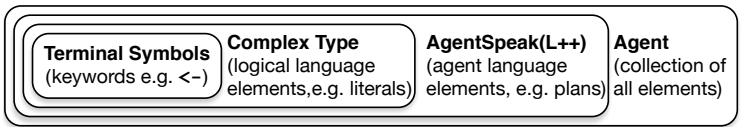


Fig. 1. Modular Grammar Structure

Jason, we can detect the agent is running, if the agent source code is syntactically correct and all actions can be executed. If the action does not exist, the parsing process will fail. The built-in actions are organised in packages. In our framework we support actions related to various types of computation. For a complete overview of these actions and how they can be implemented, we refer to [2, p. 14] and our unit test agent `complete.asl`[7], published in the appendix of [2].

5 Evaluation and Discussion

Evacuation scenario In this section we illustrate the capabilities of *LightJason* on a grid-based evacuation scenario, where agents needed to reach an exit to leave the grid. The *AgentSpeak(L++)* code for the corresponding walking agent is displayed in the listing below. For finding a route to the exit the agent used the *Jump Point Search (JPS+) with Goal Bounding* [14] algorithm, which, after an initial $O(n^2)$ preprocessing of the grid, outperforms A^* by two to three orders of magnitude in speed. To demonstrate the clarity of *LightJason*'s grammar, we grouped all *plans* and respectively *actions* describing a `moving` behaviour, e.g.

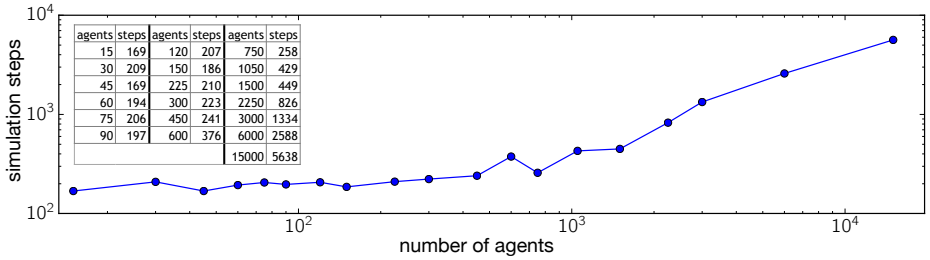
```
+!movement/walk/forward <-
  move/forward();
  !movement/walk/forward.
```

```
+!movement/walk/right <-
  move/right();
  !movement/walk/forward.
```

The concrete agent with its source code is available in [2].

Preliminary validation To validate our results, we conducted first tests with *LightJason* implementation of the evacuation scenario. The goal was to investigate whether the design and implementation of *LightJason* leads to good scalability and cycle consistency regarding the routing model, and number of concurrent running agents. We chose a grid-based scenario with 250×250 cells on an iMac with an Intel® Core™ i7-3770 and 16 GB RAM. Each agent received the same exit destination $(140, 140)$; it disappeared once it reached the approximate destination (± 10 cells). Figure 2 illustrates the run-time behaviour of the agents. It (not surprisingly) shows that with an increasing number of agents, each agent needs more cycles to complete its task. This can be attributed to additional invocations of repair plans when an agent's path got obstructed by other agents. This scales sub-linearly up to roughly 1000 agents. After that point, the mainly egoistic approach of each agent prevents them to find a free path to the exit, resulting in plan-failures and necessary re-routing. From a technical perspective we also observed that the CPU utilisation is constantly at around 70% for 15000

[7] `https://git.io/vi67u`



agents	steps	agents	steps	agents	steps
15	169	120	207	750	258
30	209	150	186	1050	429
45	169	225	210	1500	449
60	194	300	223	2250	826
75	206	450	241	3000	1334
90	197	600	376	6000	2588
				15000	5638

Fig. 2. Number of agents plotted against cycles until all agents left the scenario.

agents (for detailed plots we refer to [2]). The constant CPU load shows that the workload induced by agents is distributed fairly and evenly, avoiding spikes and idle times. Furthermore we observed a low utilisation of the JVM's *survivor space* (roughly 3.5 MB after the initialisation spike), reflecting the design in relying on *lazy bindings* and *LightJason*'s ability to share references to concurrently used data structures, e.g. plans, which only differ in their context and parameters.

Discussion In this paper we presented our design and implementation of an agent framework, introducing *LightJason*, an *AgentSpeak(L)* variant. The key aspects we focused on were modularity, flexibility, scalability and deterministic execution behaviour. The *AgentSpeak(L++)* language supported by *LightJason* reflects *AgentSpeak(L)* as implemented by [4], we differ on a number of aspects, in terms of the language features and – to a larger extent – in the software architecture underlying the implementation. Among others the most notable additions to *AgentSpeak(L)* are lambda-expressions, multi-plan definitions, explicit repair-planning, multi-variable assignments, parallel execution and thread-safe variables. When considering to port an existing *Jason* code to *LightJason* it is important to understand, that by design in *LightJason* all plans which conditions evaluate to *true* get instantiated. Here we argue, that in comparison to *Jason*, a non-synchronised system's behaviour results in a considerably more plausible multi-agent system, considering the requirements formulated by [21].

Additional Features Most of the *AgentSpeak(L)* expressions find their equivalents in *LightJason*'s *AgentSpeak(L++)*. Major additions are expressions for parallel execution and unification (`@`). As it is in general possible to design an agent to run plans sequentially, we argue, that for performance reasons it is sensible to make use of parallel execution whenever possible.

Jason 2.0 With the quite recent release of *Jason* 2.0, there now exist new features[8] in *Jason* which are similar, but independently developed, to some of our own. *Jason* 2.0 introduces *modules and namespaces* to modularise beliefs, goals and plans. In our approach we go even further by integrating those concepts deeply into the fundamental agent grammar. Thus it is possible for us to, for example, modularise actions, functions or beliefs by building hierarchical structures in arbitrary depth allowing greater flexibility than in *Jason*. Another new

[8] `https://git.io/vXmeK`

feature of *Jason* 2.0 are *concurrent courses of actions* [22]. As parallel execution is a fundamental aspect of scalability, we made this an integral part of *LightJason*'s architecture by mainly using state-of-the-art Java 1.8 developing techniques and features to enable concurrency at a very fine granularity.

6 Conclusion & Outlook

The contribution of this paper is a flexible agent programming framework *LightJason*, which can be easily integrated into existing systems. The key features of *LightJason* are the simplification of the agent's reasoning cycle and the support of some important requirements including modularity, maintainability, and scalability, combined with state-of-the-art techniques in software development. At the core of *LightJason* is *AgentSpeak(L++)*, a declarative agent scripting language extending *Jason*. We provide a formal grammar definition describing the features of *AgentSpeak(L++)*. For the sake of usability, *LightJason* supports many built-in actions and a structure to load actions in a pre-processing step of the parser. Thus, by parsing the agent's source code it is possible to check that the agent is syntactically correct and can be executed. We further provide generator structures that enable automated creation of large numbers of agents which can be further customised by the user. We also support a fully concurrent and parallel agent execution model of an agent. This paper describes ongoing work. Our next steps will involve a formal definition of the semantics of *AgentSpeak(L++)*. The reader will have noticed that *AgentSpeak(L++)* does not contain language elements for communication. This is intentional, because in our view, communication is a matter of the runtime system rather than of the compilation mechanism. Yet, agent communication is one of the next features to be added to *LightJason*. Also, while we performed an initial qualitative comparison with *Jason*, a thorough experimental benchmarking remains to be performed. Our project can be found under `http://lightjason.org` providing further documentation[9] and source code[10].

References

1. Ahlbrecht, T., Dix, J., Köster, M., Kraus, P., Müller, J.P.: An architecture for scalable simulation of systems of cognitive agents. International Journal of Agent-Oriented Software Engineering (2016), to appear.
2. Aschermann, M., Kraus, P., Müller, J.P.: LightJason: A BDI Framework Inspired by Jason. IfI Technical Report Series IfI-16-04, Department of Informatics, Clausthal University of Technology (2016)
3. Bordini, R.H., Bazzan, A.L., de O Jannone, R., Basso, D.M., Vicari, R.M., Lesser, V.R.: AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling. In: Proc. 1st Int. Joint Conf. on Autonomous Agents and Multiagent Systems: part 3. pp. 1294–1302. ACM (2002)

[9] `http://lightjason.github.io/AgentSpeak/index.html`
[10] `https://github.com/LightJason/AgentSpeak`