On the external concurrency of current BDI frameworks for MAS

Martina Baiardi[1 [0009-0001-0799-9166]], Samuele Burattini[1 [0009-0009-4853-7783]], Giovanni Ciatto[1 [0000-0002-1841-8996]], Danilo Pianini[1 [0000-0002-8392-5409]], Alessandro Ricci[1 [0000-0002-9222-5092]], and Andrea Omicini[1 [0000-0002-6655-3869]]

Department of Computer Science and Engineering (DISI)
Alma Mater Studiorum—Univerisità di Bologna
Via dell'Università 50, 47522 Cesena (FC), Italy

{*m.baiardi*, *samuele.burattini*, *giovanni.ciatto*, *danilo.pianini*, *a.ricci*, *andrea.omicini*}`@unibo.it`,
`https://www.unibo.it/sitoweb/`{*m.baiardi*, *samuele.burattini*, *giovanni.ciatto*, *danilo.pianini*, *a.ricci*, *andrea.omicini*}`/en`

Abstract. The execution of Belief-Desire-Intention (BDI) agents in a Multi-Agent System (MAS) can be practically implemented on top of low-level concurrency mechanisms that impact on efficiency, determinism, and reproducibility. We argue that developers should specify the MAS behaviour independently of the execution model, and choose or configure the concurrency model later on, according to their target domain's specific needs, leaving the MAS specification unaffected. We identify patterns for mapping the agent execution over the underlying concurrency abstractions, and investigate which concurrency models are supported by some of the most commonly used BDI platforms. Although most frameworks support multiple concurrency models, we find that they tend to hide them under the hood, making them opaque to the developer, and effectively limiting the possibility of fine-tuning the MAS.

Keywords: Agent-Oriented Programming · Concurrency · BDI Agents · Threading · Parallelism

1 Introduction

The Agent-Oriented Programming (AOP) paradigm was introduced almost thirty years ago [35] as a way to model software in terms of autonomous computational entities capable of carrying on several courses of action *simultaneously*—there including, interacting with their environment and among each other. Since its conception, AOP has been strictly linked with the *strong* notion of agency [38], where agents are assumed to be aware of their own goals and able to reason about if, when, and how to pursue them—not necessarily in a predefined order. Along this line, AOP frameworks evolved to embrace the Belief-Desire-Intention (BDI) model [10], where agents are modelled and implemented by means of abstractions mimicking typically human-level notions. By construction, BDI agents are able to carry on multiple intentions at any given time [31], and many research and software-development efforts have been devoted to the definition of BDI architectures and programming languages giving precise semantics to the *concurrent* execution of such intentions [9].

As computational entities, agents are autonomous as they encapsulate their own *control flow* [27]. Control-flow encapsulation is commonly referred to as *computational* autonomy [28], and it is considered a necessary – yet not sufficient – pre-requisite for autonomy in software agents.

On mainstream programming platforms, (such as the Java Virtual Machine (JVM) [12], used for the implementation of several BDI frameworks), computational autonomy may be achieved by mapping each agent onto a control-flow-related primitive: a thread, a processes, or an event loop. This, in turn, enables and constraints the ways by which multiple agents may be *concurrently* executed. In this paper, we refer to the mapping between BDI abstractions and the underlying concurrency primitive as the *concurrency model* of the framework.

The selection of an appropriate concurrency model deeply impacts several aspects of the agent programming framework: efficiency, determinism, and reproducibility. In particular, the concurrency model determines whether, and to what extent *(i)* multiple agents can run in parallel, and *(ii)* one agent can carry on parallel activities. Parallelism, in turn, affects the efficiency of MAS execution (particularly on hardware supporting true parallel execution) and the determinism of the overall MAS dynamics. In fact, parallelism introduces non-deterministic interleaving of the agent's actions, undermining predictability and reproducibility, which may be a strict requirement in some applications, such as multi-agent based simulation [5]. Finely capturing and controlling concurrency is crucial in modern software engineering, even beyond MASs: consider, for instance, trends such as event-driven [14] and reactive [4] programming.

Unfortunately, dealing with concurrency is commonly acknowledged as error-prone and challenging. Thus, mainstream programming languages and platforms are featuring more and more constructs helping developers to leverage concurrency through better abstractions (e.g., Javascript's async/await [25], Akka's reactive streams [15], and Kotlin's coroutines [18]), hiding part of the subtle intricacies under the hood. AOP tools and frameworks are no exception to this trend: they come with one or more concurrency models, often (in compliance with the information hiding principle) hidden under the hood to let programmers focus on the agents' behaviour.

In this work we argue that, although the separation of concurrency models and MAS specifications is paramount, removing control from the developers' hand is not the best solution: they should be aware of the available possibilities and related trade-offs, and select (and, possibly, swap) them depending on the specific needs of their application and execution environment. This is particularly true for BDI agent technologies, where the semantics of intention scheduling may be realised in many different ways.

Contribution. In this work, we introduce the notions of *internal* and *external* concurrency, capturing, respectivelly, the concurrency among agents' activities and the concurrency induced by the selection of the mapping of multiple agents onto the underlying concurrency abstractions. These two abstractions influence each other: enforcing either one restricts the range of possibilities of the other, impacting performance, determinism, and reproducibility. Despite that, the previous literature focuses on internal concurrency, leaving the external one as an implicit consequence of the choices made to support internal behaviour. Thus, in this paper we provide a taxonomy of the concurrency models that may be adopted by BDI frameworks, and we classify several notable BDI agent technologies accordingly. Finally, we draw practical engineering recommendations for the development of BDI agent technologies, suggesting to take into account the control of *external* concurrency at design time.

Structure. The remainder of this paper is structured as follows. In Section 2 we define internal and external concurrency in BDI agents and how they have been considered in related works in the AOP community. We then analyse concurrency models commonly adopted in modern software development, and we discuss in Section 3 how agents (and their internal components) can be mapped onto them, evaluating the pros and cons. We then evaluate in Section 4 several BDI technologies from the AOP community from a concurrency-related perspective, eliciting the available concurrency models and their degree of configurability. Finally, in Section 5 we elaborate on the importance of configurabile concurrency models well-separated from the agent's behaviour specification.

2 Background

In this section, we first frame the concepts of *internal* and *external* concurrency, then look at the existing work specifically addressing concurrency in the context of BDI AOP, thus framing our contribution to the state of the art. Then, we discuss the lower-level concurrency abstractions required to understand the remainder of the paper.

2.1 Internal vs. External Concurrency

A multi-BDI-agent system can be modelled in Calculus of Communicating Systems (CCS) [26] as a set of agents running in parallel. Each agent is essentially an infinite loop where, at each iteration step, the three main stages of the agent's control loop are executed—sensing, deliberating, and acting. More formally:

$$\begin{aligned} Mas &::= Agent_1 \parallel \ldots \parallel Agent_N \\ Agent &::= \texttt{sense} \cdot \texttt{deliberate} \cdot \texttt{act} \cdot Agent \end{aligned} \tag{1}$$

where *(i)* operation `sense` is responsible for handling new percepts and incoming messages, generating update events accordingly, *(ii)* operation `deliberate` is responsible for choosing how to handle those events and picking the next action to be executed and *(iii)* operation `act` is responsible for executing the selected action—e.g. sending a message, affecting the environment, or changing the agent's internal state.

This simple modelling focuses on the control loop of agents, while hiding another key aspect of MAS: interaction among agents—i.e. how each agent's actions may influence other agents. Interaction may consist of either communication (e.g. direct message passing) or stigmergy (e.g. indirectly altering the environment to affect other agents). In both cases, interaction implies one agent acting and another agent perceiving the effects of that action, so, as far as concurrency and control-loops are concerned, the modelling above is sufficient.

Internal Concurrency. We call *internal concurrency* how these operations are modelled, there including whether they are further decomposable or not, their degree of concurrency, and their interleaving. For instance, in [39], two major patterns are identified: the *synchronous* one where *all* percepts and messages are *sequentially* handled in the sensing stage, and only one action is selected by the deliberation stage, and therefore only one action is executed by the action stage:

$$\begin{aligned} \mathit{Agent} &::= \mathit{Sense} \cdot \mathit{Deliberate} \cdot \mathit{Act} \cdot \mathit{Agent} \\ \mathit{Sense} &::= \texttt{sense}_1 \cdot \ldots \cdot \texttt{sense}_M \\ \mathit{Deliberate} &::= \texttt{deliberate} \\ \mathit{Act} &::= \texttt{act} \end{aligned} \tag{2}$$

and the *asynchronous* one where *multiple* percepts and messages are *concurrently* handled in the sensing stage, and deliberation and action stages are executed concurrently as well:

$$\begin{aligned} \mathit{Agent} &::= \mathit{Sense} \parallel \mathit{Deliberate} \parallel \mathit{Act} \\ \mathit{Sense} &::= (\texttt{sense}_1 \parallel \ldots \parallel \texttt{sense}_M) \cdot \mathit{Sense} \\ \mathit{Deliberate} &::= (\texttt{deliberate}_1 \parallel \ldots \parallel \texttt{deliberate}_L) \cdot \mathit{Deliberate} \\ \mathit{Act} &::= (\texttt{act}_1 \parallel \ldots \parallel \texttt{act}_K) \cdot \mathit{Act} \end{aligned} \tag{3}$$

Other patterns may be defined in this framework; for instance, the single step of the control-loop can be modelled as a fork/join, where all percepts are handled concurrently, then all deliberations are handled concurrently, and then all actions are executed concurrently. The key point, however, is that all such models focus on how the control loop of each agent is executed, and, by extension, on how the intentions of each agent interleave. For instance, a system modelled as in Equation (2) would only support *simulated* parallelism—e.g., a very common implementation is: each cycle of the control-loop executes a single action from a single intention. Conversely, a system modelled as in Equation (3) would support *true* parallelism—so, in principle, two or more action could be executed in the same moment.

External Concurrency. Conversely, in this paper, we focus on *external* concurrency, i.e., the way the control loops of multiple agents are mapped onto the underlying concurrency abstractions (Section 2.3). In other words, we are interested in understanding how Equation (1) can be – and commonly is – implemented in practice. Arguably, understanding and explicitly modelling external concurrency is crucial, as the external concurrency model constrains and supports the admissible internal concurrency models: the relationship between the two is bi-directional. Additionally, we argue that the external concurrency model has the most impact on the overall properties of the system. For instance, even though the internal concurrency of an agent may be massively parallel, there will be no speedup compared to a sequential execution if the external concurrency model enforces execution in a single control flow. At the same time, even if agents are internally sequential and predictable, an external model mapping them on multiple threads may lead to unpredictable interleaving of actions, thus having an impact on the predictability of the whole MAS.

We further elaborate on this in Section 3, where we present different models of external concurrency that are at the core of this contribution.

2.2 Related Work

The existing literature on concurrency in BDI systems mainly focuses on *internal* concurrency. For instance, a recent survey [37] provides an overview of BDI architectures, including considerations on how different platforms deal with the interleaving of agents' intentions. Moreover, when discussing concurrency in BDI systems, the discussion is often about the *interleaving* of sequentially-executed intentions, rarely about their *parallel* execution (also known as *true concurrency* [36]). Interaction among agents that need to share mutable data has also received attention. In particular, the shared data has been modelled with the abstraction of *artifact* [34], capturing *safety* and *synchronisation*; adopting specialised abstractions can in turn impact internal concurrency [33]. The impact of concurrency on the overall performance of the agent has been investigated in [39], where the authors investigate the impact different concurrency configurations mapping the agent control loop can have on the overall performance and properties both of individual agents and the whole MAS.

2.3 Underlying Concurrency Mechanisms

The structured programming theorem [8] states that any computable function can be expressed in terms of selection (executing one of two subprograms depending on a condition), iteration (repeatedly executing a subprogram until a condition is met), and *sequence* (executing a subprogram after another). The latter is the foundation of the so-called *control flow* of a program, and it is rooted in the assumption that instructions are *totally* ordered. In concurrent programs, instead, the execution of instructions is rather *partially* ordered [23]: although subprograms are executed in a given order, instructions of different subprograms may interleave, producing a different total ordering. Concurrent execution can be especially beneficial (and difficult to govern [6]) when the underlying architecture supports multiple control flows (multiple processors, cores, or portions of the execution pipeline).

The realisation of concurrent programs boils down to minimising the amount of ordering constraints imposed on the execution of instructions while guaranteeing correctness, and can be performed through formal or practical tools. Formalism dedicated to concurrent programming include process algebrae [21], CCS [26], Petri-nets [32], and actors [1]. From a practical perspective, some of these formalisms are captured by programming languages, either with a dedicated syntax or libraries, sometimes adopting a custom naming convention, but ultimately preserving the underlying semantics. In the following discussion, we introduce the most common concurrency abstractions available in most modern programming languages.

Threads. Threads are a facility provided by Operating Systems (OSs) to execute sequential programs that share memory; they are considered the basic unit of concurrency [16]. Although the code executed by each thread is sequential, multiple threads run concurrently (scheduled by the OS onto multiple logical cores and/or in a time-sharing fashion), thus the execution of multiple threads may interleave arbitrarily. Since they share memory, threads may easily interact among each other by reading/writing the same memory locations, causing race conditions and other concurrency-related issues. Thus, multithreaded programs commonly require synchronisation, typically achieved by means of arguably low-level primitives such as *locks*, *semaphores*, and *monitors*, enforcing partial ordering among instructions of different threads. Other concurrency abstractions are constructed by coordinating threads by means of these and similar mechanisms.

Processes. Processes are similar to threads, but they do not (normally, in modern OSs) share memory; rather, inter-process communication is possible thorugh the OS mediation via mechanisms such as *pipes*, *sockets*, or the *file system*. Internally, processes can spawn multiple threads, thus, from a concurrency perspective, processes can be intended as containers of threads sharing the process' memory space.

Event Loops. In event-driven programming [14], event loops are abstractions to express concurrent programs while hiding the intricacies of low-level thread synchronisation. An event loop is a single thread executing multiple tasks (subprograms) sequentially from different sources (users, the OS, or other parts of the program). Tasks can be scheduled by registering the corresponding subprogram on the event loop; internally, this operation appends the subprogram to a First-In-First-Out (FIFO) queue internal to the event loop. The event loop's thread executes the tasks in the queue in order, waiting if the queue is empty: any task scheduled on the event loop is *eventually* executed. The perception of parallelism of an event loop come from the fact that new tasks can be scheduled with no need to wait for any previous one to be completed. On the other hand, the sequential nature of event loops becomes evident in case of long-running tasks (e.g., I/O operations), that may lead subsequent ones to starvation. To mitigate this issue, event-loops are commonly coupled with non-blocking I/O [11], where blocking read/write primitives are replaced with asynchronous events.

Notably, event loops are the backbone of many interesting features that are popping up in modern programming languages – there including JavaScript, Python, C#, etc. –, such as `Promise`s, `async`hronous functions, and `await` operators (cf. [25]). In these languages, the event loop is hidden under the hood, and developers are not required to interact with it directly, but rather by means of the aforementioned features. As far as this paper is concerned, we stick to the low-level abstraction of the event loop, as our goal is to make concurrency controllable for AOP developers—rather than hiding it via syntactic sugar.

Executors. Borrowing from the Java's nomenclature[1], executors generalise event loops by supporting multiple threads. They support tasks to be enqueued as event loops do, yet consumption of tasks from the queue is transparently performed by multiple threads (thus, potentially, in parallel). Executors may be further categorised depending on whether their backing thread count can change at runtime. Fixed-sized executors are created with a specific count number of threads N, which imposes an upper bound on the maximum degree of parallelism, as at most N tasks may be executed in parallel at any given moment. Conversely, variable-sized executors may *dynamically* change the number of threads in response to the runtime conditions. A typical case where variable-sized executors are preferable is in the presence of multiple long-running blocking tasks. For instance, assume N such tasks to be selected for parallel execution: the fixed-sized executor would be blocked, starving the other tasks and leaving resources unused; the variable-sized executor, instead, could spawn new threads to execute the other tasks, and let them terminate once no blocking tasks are being run.

Concurrency Abstractions in Practice. Although all the aforementioned concurrency abstractions are equivalent in terms of expressiveness, there are relevant practical implications associated with any choice.

Consider, for instance, the CCS system $\mathtt{a}\cdot\mathtt{b}\cdot\mathtt{c} \parallel \mathtt{x}\cdot\mathtt{y}\cdot\mathtt{z}$, modelling two parallel suprograms performing a sequence of atomic tasks. Such system, as specified, allows tasks to interleave arbitrarily, as far as their order within the subprogram is respected (for instance, `b` can never happen before `a`, but $\mathtt{a}, \mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{b}, \mathtt{c}$ is a perfectly valid execution). When subprograms are executed by independent threads, this semantics is respected. When using an event loop, instead, some combinations become impossible, as the execution of the next task is scheduled after the previous one's completion; consequently, if both `a` and `x` are enqueued, only two round-robin inter-leavings are possible, depending on which one is on the top of the queue: $\mathtt{a}, \mathtt{x}, \mathtt{b}, \mathtt{y}, \mathtt{c}, \mathtt{z}$ or $\mathtt{x}, \mathtt{a}, \mathtt{y}, \mathtt{b}, \mathtt{z}, \mathtt{c}$. So, we say that implementing the concurrent system on an event loop reduces the non-determinism as well as developers' degrees of control. With an executor, all possible interleavings are still possible, but the degree of parallelism can be selected.

[1] `https://archive.is/zF1FL`

Generalising on this observation, we may state that the choice of concurrency abstraction has an impact on the determinism and controllability of the concurrent system.

3 A Taxonomy of Concurrency Patterns for MAS Execution

In this section, we identify the most relevant *external* concurrency models for MASs, namely, how the atomic parts of the agent' control loop get mapped onto the underlying abstractions described in Section 2.3. Of course, different *internal* concurrency models dictate different levels of granularity of the atomic components of the control loop, thus influencing the *external* concurrency model. Consequently, in the upcoming discussion, we focus on the mapping of the largest possible autonomous unit in AOP, the entire agent, discussing the potential external concurrency models for a MAS.

One-Agent-One-Thread (1A1T). Each agent is mapped onto a single thread, which executes its whole control loop. Hence, the MAS consist of several threads managed by the OS scheduler, and the interleaving among different agents' operations is unpredictable. The controllability of the MAS execution is abysmal, as control is delegated to the OS; for the same reason, determinism is minimal. Additionally, with 1A1T the active thread count of the MAS is unbound, and when such count largely exceeds the logical processors performance degrades [24].

All-Agents-One-Thread (AA1T). The whole MAS is executed on a single thread which internally schedules all agents' execution in a custom way, following some (usually cooperative) scheduling policy—e.g., round-robin. Using a single thread with custom internal scheduling policy renounces parallelism (hence, performance) in favour of controllability, and it is thus a good choice when determinism, reproducibility, and predictability are primary concerns; such as in many simulated or time-critical scenarios. Notably, because of the cooperative nature of the scheduling, sensing, deliberation, and actuation operations should terminate as quickly as possible, to avoid blocking the whole MAS.

All-Agents-One-Event-Loop (AA1EL). The whole MAS is executed on a single event loop, which internally schedules all agents' execution with a FIFO queue of tasks, guaranteeing fairness by design if all new tasks in the event loop are inserted by other tasks of the event loop. At the conceptual level, this is equivalent to a fair AA1T, as such, controllability and performance are akin to AA1T. In practice, however, AA1EL requires explicitly modelling the agent's control loops activities as tasks on the event loop, thus, despite the conceptual equivalence, technical implementations of AA1T and AA1EL may be fairly different.

All-Agents-One-Executor (AA1E). Similarly to AA1EL, each atomic operation is mapped onto a task to be enqueued on a shared executor. This model enables the parallel execution of multiple agents, and, if the internal concurrency model supports it, the parallel execution of the same agent's activities. In case each agent enqueues at most one task at a time (a solution often used to enforce consistency), AA1E is conceptually equivalent to 1A1T. From a technological perspective, however, AA1E is preferable, as the agent (and agent's actions) count is decoupled from the thread count, and thus there is finer control on the degree of parallelism (by governing the amount of threads in the executor), and therefore a better exploitation of the underlying resources. Two further specialisations of this model are possible, depending on whether the executor is fixed- or variable-size. In the former case, there is an upper bound on the amount of threads the MAS can leverage. Although helpful to limit the resource exploitation in constrained environments, it may introduce subtle interdependencies among agents. For instance, when there are M agents and $N < M$ threads, if N agents are performing blocking operations, then the other $M - N$ agents must wait. Of course, the fixed-sized executor with $N = 1$ is equivalent to AA1EL. If the executor is variable-sized, then the number of threads is adjusted dynamically, upon need—i.e., by trying to match the count of active threads and logical processors. AA1E is generally preferable over 1A1T, as the total thread count is controllable.

3.1 Concurrency at different granularity levels

Concurrency abstractions may be combined to form more complex ones. For instance, both processes and executors are composed by threads. Threads in a process may be part of the same executor, or multiple ones. In a distributed setting, a system may be composed by many processes spread across a several machines, losing shared memory and thus requiring serialisation to communicate. In orchestration frameworks, the same service may consist of multiple containers, distributed on different machines, each one running multiple processes.

In principle, when implementing a MAS, agents may be mapped onto any of these concurrency abstractions with different trade-offs between flexibility and controllability. For instance, when One-Agent-One-Process (1A1P) is adopted, the agent's internal control loop may be implemented with multiple threads, but communication among agents will require (de)serialisation, as agents will not share memory with each other. For BDI agents, threads may be used to model intentions, paying a price in terms of implementation complexity (as agent-specific synchronisation mechanisms would be required) to obtain an extremely fine-grained degree of control over the execution.

Combinations (and complexity) can scale arbitrarily, as in principle any AOP abstraction can be mapped on any lower-level concurrency abstraction (thus allowing uncommon combinations such as One-Agent-One-Container or One-Intent-One-Process). For the sake of simplicity, in this paper we focus on the cases listed in Section 3, which we show suffice to capture the behaviour of all the selected BDI technologies. However, we discuss the implications of more nuanced concurrency models briefly in Section 5.

Listing 1.1. ASL description for *pinger* agent.

```
!ping.

+!ping <-
    .revealCurrentThread("intention 1");
    .send(pong, tell, ball);
    !!showThread(2);
    .revealCurrentThread("intention 1").

+ball <-
    !!showThread(4);
    .revealCurrentThread("intention 3").

+!showThread(X) <- .revealCurrentThread("intention " + X).
```

4 Analysis of BDI Technologies and Concurrency Models

In this section, we inspect the external concurrency models supported by a selection of actively-maintained open source BDI programming frameworks. In particular, we focus on Astra [13], Goal [20], Jadex [30], JaKtA [2], Jason [9], Phidias [17], and Spade-BDI [29]. We do not claim this selection to be exhaustive, so we leave a more complete analysis for future works.

4.1 Methodology

We performed our analysis in three steps:

1. *empirical evaluation* through a synthetic benchmark designed to reveal how many threads are involved in the execution of a MAS and how they interleave;
2. *documentation and source code inspection* to understand implementation details and customisability.
3. *direct contact* with the current maintainers, asking for confirmation of our findings and for further details, including a subjective evaluation of the feasibility of supporting additional external concurrency models.

Empirical Evaluation. We created a benchmark [3] to reveal how threads are leveraged in a BDI MAS. The benchmark consists of a simple MAS, composed by two agents enacting one round a ping–pong protocol: the pinger agent initiates the protocol by sending a message to the ponger agent, which replies by sending the a message back to the pinger. To reveal how threads are used, we make agents execute a custom action – `revealCurrentThread` – before and after each message sending and reception. As a reference, we show our Jason implementation for pinger (Listing 1.1) and ponger (Listing 1.2). To maximise the likelihood of intercepting all threads, when supported we force agents to pursue different intentions simultaneously (in the reference specification, this is done through the `!!` Jason operator). We then analyse the trace obtained by multiple executions of the benchmark to extract the underlying external concurrency model.

Listing 1.2. ASL description for *ponger* agent.

```
+ball[source(X)] <-
    .revealCurrentThread("intention 5");
    .send(X, tell, ball);
    !!showThread(6);
    .revealCurrentThread("intention 5").

+!showThread(X) <- .revealCurrentThread("intention " + X).
```

Documentation and Source Code Inspection. In general, the empirical evaluation can let *some* external concurrency model emerge, but it cannot be exhaustive: as discussed in Section 2.3, some abstraction may not show all their possible behaviours even with repeated executions, and some may produce the same outputs. Additionally, for some platforms, the empirical evaluation is more difficult to implement and less revealing. For instance, the thread inspection primitives of the JVM (with which Jason can interact) are much more expressive than those of SWI-Prolog, (with which GOAL interacts). We thus inspected the source code and the official documentation of the surveyed frameworks to learn as much as possible.

Direct Contact. Once we gathered the results from the previous steps, we contacted maintainers of each framework to confirm our assessment and gain additional insights. This operation was useful to get past what is available out-of-the-box, and what could be achieved with reasonably limited extensions. We described the developers the tassonomy of Section 3 and reported our results. We asked them to evaluate on our findings, adding comments about whether the non-supported external concurrency models were available out of the box (thus, missed by the analysis), could be supported with reasonable effort, or required extensive rewriting of the codebase[2]. We received answers from all developers except for Jason and SPADE-BDI; all answers confirmed our initial results.

4.2 Results

Table 1 summarises the results of our analysis. When evaluating 1A1P, we also required agents to be capable of inter-process communication, e.g., by means of protocols such as TCP/IP. In the remainder of this section, we detail how the analysis was performed for each technology, summarising the most prominent findings.

Astra. Astra [13] is a BDI agent technology written in Java designed with a C-family syntax. Astra provides fine-grained control over the execution of the MAS entities, our benchmark indeed revealed that any iteration step of the control loop of the same agent may run on a different thread, suggesting a

[2] A template of the email that we sent to all maintainers can be found in Appendix A

AA1E model. Source code inspection confirmed the analysis and revealed that the executor is *variable-sized*. Since Java executors can be used as event loops, AA1EL is supported, too. Notably, Astra supports user-side customisation of the concurrency model through a `SchedulerStrategy`; thus, 1A1T and AA1T could be implemented reasonably easily.

Goal. GOAL [20] is a Java BDI library distributed as an Eclipse IDE plugin that integrates with SWI Prolog through Java Native Interface (JNI) [22], that does not expose Java primitives. Due to its peculiar integration with SWI Prolog, GOAL is bound to the 1A1T model, and it does not support customisations without major changes to the code base. However, the library comes with an option for emulating AA1T; although internally agents are still executed on different threads, these are executed sequentially.

Jadex. Jadex [30] is a BDI Java library. We analysed the latest version of the library, namely Jadex V, which improved modularization and simplified agents' concurrency management through (Jadex's terminology) `ExecutionFeatures`. Jadex V natively supports variable-sized AA1E as default behaviour, 1A1P and AA1T. Further customisation options could be implemented with custom `ExecutionFeatures` modules.

JaKtA. JaKtA [2] is a Kotlin-based Domain-Specific Language (DSL)[3] for BDI MAS running on the JVM. It exposes the concurrency model as a first-class abstraction in the DSL (see Listing 1.3) supporting custom models through the implementation of the `ExecutionStrategy` interface. By default, JaKtA uses AA1T to support reproducibility while debugging or simulating; but 1A1T, AA1E, and AA1EL are also supported natively. 1A1P is not supported out of the box, but could be implemented through an extension.

[3] `https://archive.is/El3fE`

Table 1. Summary of BDI technologies and their concurrency models. The symbols ✓, ∼, and × indicate, respectively, that the concurrency model is supported, that it could be supported with a custom implementation, and that it is not supported.

Model → Tech. ↓	1A1T	AA1T	AA1EL	AA1E fixed	AA1E variable	1A1P
Astra	∼	∼	✓	✓	✓	∼
Goal	✓	×	×	×	×	×
Jadex	∼	✓	∼	∼	✓	✓
JaKtA	✓	✓	✓	✓	✓	∼
Jason	✓	∼	✓	✓	∼	✓
Phidias	✓	×	×	×	×	✓
Spade-BDI	×	×	✓	×	×	✓

Listing 1.3. Example of MAS configuration with execution strategy customisation in JaKtA.

```
mas {
  environment { /* external actions' definitions here */}
  agent("pinger") { /* pinger specification here */ }
  agent("ponger") { /* ponger specification here */ }
  executionStrategy { oneThreadPerAgent() } // first-class support
}.start()
```

Jason. Jason [9] is a well-known AgentSpeak(L)-compliant BDI agent technology implemented in Java. Jason defaults to a 1A1T model, but the concurrency models are customisable by specifying a different *infrastructure*. Similarly to JaKtA, these can be configured at MAS specification time and customised; implementations available out of the box provide support for AA1T (`Local/threaded` infrastructure), fixed-sized AA1E and AA1EL (`Local/pool` infrastructure); and 1A1P (`Jade` infrastructure).

Phidias. PHIDIAS [17] is a Python internal DSL defaulting to the 1A1T concurrency model. Even though using threads, the execution in most Python interpreters will not be parallel because of the Global Interpreter Lock (GIL)[4]. Inter-agent communication is implemented through HTTP, suggesting that 1A1P is supported too. However, there is no way to customise the concurrency model. In other words, the 1A1T and 1A1P models are hard-coded.

Spade-BDI. SPADE-BDI [29] is a Python library defined on the top of SPADE [19], adding BDI agents on top of SPADE primitives. Agents in both SPADE-BDI and SPADE are implemented via Python's native event loops and coroutines, providing native support for AA1EL. Similarly to PHIDIAS, inter-agent communication is supported through a standard protocol (XMPP), suggesting potential support for 1A1P. Changing concurrency model, however, is not supported.

5 Conclusion, Recommendations, and Future Works

The external concurrency model is a key aspect to be taken into account when designing or using a (BDI) MAS technology. Generally speaking, the more options the better, as applications can be finely tailored to the specific requirements of the problem at hand.

Controllability. Reproducibility and controllability are key aspects of MAS engineering, especially during debug and simulation. When full control is required, AA1T and AA1EL are the best choices, as they enforce a single control flow.

[4] `https://archive.is/5KBqn`

Performance. Better Performance is generally achieved by exploiting parallelism. Although the 1A1T model seems attractive for its simplicity, AA1E is preferable, especially in cases in which the agent count largely exceeds the logical processors.

Design of BDI Technologies. We argue that designers of BDI technologies should provide means to customise the concurrency model with a dedicated and well-documented Application Programming Interface (API). Doing so requires careful consideration of the concurrency model as early as possible. Building a BDI platform around assumptions on the desired concurrency model may simplify the implementation, but it is likely to backfire later on, limiting extensibility and applicability (for instance, preventing the system to scale up and down depending on the available resources). If assumptions must be made due to technical constraints, some choices are more flexible than others; for instance, AA1E can emulate 1A1T and AA1EL, while the opposite does not apply. The key to design BDI platforms capable to adapt to multiple concurrency models is the *complete separation between agents' control loop and their target concurrency abstraction.* Following this principle, it should be possible to write the MAS specification once, then run it on different concurrency models with no changes.

Impact on Internal Concurrency. We discussed how internal concurrency models impact external concurrency models by bounding the maximum *granularity* at which the agent's control loop can be parallelised. However, the influence is bidirectional: enforcing external abstractions binding specific BDI abstractions to one or more control flows (such as 1A1T or AA1T) may hinder further attempts to control the degree of parallelism by exploiting finer-grained internal concurrency models (e.g., parallelise at the level of intentions).

Final Remarks. The external concurrency of BDI agents is a paramount aspect of practical MAS engineering. In this paper, we provided clear terminology and taxonomy to support decision-making about concurrency in MAS, addressing both the construction of MAS and the (re)design of BDI technologies. We analysed the state of the art of several relevant BDI technologies, showing that there is heterogeneity in terms of supported concurrency models and their customisability. We advocate for further research efforts to provide BDI technology designers with clear guidelines and best practices regarding practical external concurrency models, favouring harmonisation and standardisation.

5.1 Future Works

In this paper, we focus on relatively small set of well-established BDI technologies. In the future, we plan to apply our inspection on a wider range of BDI technologies, possibly adopting exhaustiveness as a criterion.

In fact, it is worth mentioning that our inspection methodology could be applicable, in principle, to *any* AOP technology, regardless of whether it is BDI or not: *external* concurrency is a key aspect of AOP technologies in general, whereas *internal* concurrency – as it is defined in this paper – is specific for the BDI paradigm—and, specifically, to the notion of *intention*.

Accordingly, in the future, we plan to extend the analysis to other AOP technologies, possibly beyond the realm of BDI architectures. Along this line, we also plan to widen the definition of *internal* concurrency, to account for other behavioral abstractions, possibly adopted by other AOP architectures. This would be for instance the case of *behaviors* in JADE [7] or SPADE [29].

Acknowledgements

This work has been partially supported by: *(i)* "WOOD4.0 - Woodworking Machines for Industry 4.0", Emilia-Romagna CUP E69J22007520009; *(ii)* "FAIR—Future Artificial Intelligence Research", Spoke 8 "Pervasive AI" (PNRR, M4C2, Investimento 1.3, Partenariato Esteso PE00000013), funded by the EC under the NextGenerationEU programme; *(iii)* "ENGINES — ENGineering INtElligent Systems around intelligent agent technologies" project funded by the Italian MUR program "PRIN 2022" (G.A. 20229ZXBZM), and *(iv)* 2023 PhD scholarship (PNRR M4C2, Investimento 3.3 DM 352/2022), co-funded by the European Commission and AUSL della Romagna. Also, the authors would like to thank all the researchers and developers who answered our request for comments for their invaluable help.

A Appendix: Framework Maintainers Interview

```
Dear <Maintainer>,
we are reaching out to you to ask information about <X>.

Our research group is conducting research on how MAS platforms deal with the underlying
     concurrency mechanisms.
We are surveying several technologies to understand how they map the agents' lifecycle on
     the underlying mechanisms:

  1. One-Agent-One-Thread: Each agent is mapped into a single thread.
  2. All-Agents-One-Thread: The whole MAS is executed on a single thread, following a
       scheduling policy (i.e. Round-Robin).
  3. All-Agents-One-Event-Loop: The MAS is executed over an event-loop.
  4. All-Agents-One-Executor: Similar to case 3, but it uses threads to allocate agents,
       resulting in an effectively parallel execution. We distinguish two sub-cases:
     a. fixed-sized executors (static thread count)
     b. variable-sized (dynamically changing thread count).
  5. One-Agent-One-Process: which, internally, could exploit all the above taxonomies to
       execute its control loop.

We inspected your code source and identified that <X> currently supports <list of supported>,
however, we were not able to infer if it can supports <list of not supported>
Would you agree with the previous assertion?

Would it be possible to write custom extensions to implement <list of not supported> with no
     changes to the current code base of <X>?
If not, what about implementing the missing mechanisms directly?
Would it be feasible, in your opinion?
And if so, would you consider it easy, moderate, hard, or very hard?
```

References

1. Agha, G.A.: ACTORS - a model of concurrent computation in distributed systems. MIT Press series in artificial intelligence, MIT Press (1990). https://doi.org/10.7551/mitpress/1086.001.0001
2. Baiardi, M., Burattini, S., Ciatto, G., Pianini, D.: JaKtA: BDI agent-oriented programming in pure Kotlin. In: Multi-Agent Systems - 20th European Conference, EUMAS 2023, Naples, Italy, September 14-15, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14282, pp. 49–65. Springer (2023). https://doi.org/10.1007/978-3-031-43264-4_4
3. Baiardi, M., Ciatto, G.: BDI Languages Concurrency Model Survey Repository: v1.0.0 (Apr 2024). https://doi.org/10.5281/zenodo.10948573
4. Bainomugisha, E., Carreton, A.L., Cutsem, T.V., Mostinckx, S., Meuter, W.D.: A survey on reactive programming. ACM Comput. Surv. **45**(4), 52:1–52:34 (2013). https://doi.org/10.1145/2501654.2501666
5. Bandini, S., Manzoni, S., Vizzari, G.: Agent Based Modeling and Simulation, pp. 184–197. Springer New York (2009). https://doi.org/10.1007/978-0-387-30440-3_12
6. Batty, M.: Compositional relaxed concurrency. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences **375**(2104), 20150406 (Sep 2017). https://doi.org/10.1098/rsta.2015.0406
7. Bellifemine, F., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley (2007). https://doi.org/10.1002/9780470058411
8. Böhm, C., Jacopini, G.: Flow diagrams, Turing machines and languages with only two formation rules. Communications of the ACM **9**(5), 366–371 (May 1966). https://doi.org/10.1145/355592.365646
9. Bordini, R., Hübner, J., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason. Wiley Series in Agent Technologies, Wiley (2007). https://doi.org/10.1002/9780470061848
10. Bratman, M.E., Israel, D.J., Pollack, M.E.: Plans and resource-bounded practical reasoning. Comput. Intell. **4**, 349–355 (1988). https://doi.org/10.1111/J.1467-8640.1988.TB00284.X
11. Büttner, D., Kunkel, J.M., Ludwig, T.: Using non-blocking I/O operations in high performance computing to reduce execution times. In: 16th European PVM/MPI Users' Group Meeting, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5759, pp. 134–142. Springer (2009). https://doi.org/10.1007/978-3-642-03770-2_20
12. Calegari, R., Ciatto, G., Mascardi, V., Omicini, A.: Logic-based technologies for multi-agent systems: A systematic literature review. Autonomous Agents and Multi-Agent Systems **35**(1), 1:1–1:67 (2021). https://doi.org/10.1007/s10458-020-09478-3, collection "Current Trends in Research on Software Agents and Agent-Based Software Development"
13. Collier, R.W., Russell, S.E., Lillis, D.: Reflecting on agent programming with AgentSpeak(L). In: PRIMA 2015: Principles and Practice of Multi-Agent Systems, pp. 351–366. Lecture Notes in Computer Science, Springer International Publishing (2015). https://doi.org/10.1007/978-3-319-25524-8_22
14. Dabek, F., Zeldovich, N., Kaashoek, F., Mazières, D., Morris, R.: Event-driven programming for robust software. In: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop. pp. 186–189. EW 10, ACM, New York, NY, USA (2002). https://doi.org/10.1145/1133373.1133410

15. Dadel, P., Zielinski, K.: Evolution of reactive streams API for context-aware mobile applications. Comput. Informatics **35**(4), 852–869 (2016), http://www.cai.sk/ojs/index.php/cai/article/view/3379
16. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Communications of the ACM **8**(9), 569 (sep 1965). https://doi.org/10.1145/365559.365617
17. D'Urso, F., Longo, C.F., Santoro, C.: Programming intelligent IoT systems with a Python-based declarative tool. In: Proceedings of the 1st Workshop on Artificial Intelligence and Internet of Things co-located with the 18th International Conference of the Italian Association for Artificial Intelligence, 2019. CEUR Workshop Proceedings, vol. 2502, pp. 68–81. CEUR-WS.org (2019), https://ceur-ws.org/Vol-2502/paper5.pdf
18. Elizarov, R., Belyaev, M.A., Akhin, M., Usmanov, I.: Kotlin coroutines: design and implementation. In: Onward! 2021: Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Virtual Event / Chicago, IL, USA, October 20-22, 2021. pp. 68–84. ACM (2021). https://doi.org/10.1145/3486607.3486751
19. Gregori, M.E., Cámara, J.P., Bada, G.A.: A jabber-based multi-agent system platform. In: 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Japan, May 8-12, 2006. pp. 1282–1284. ACM (2006). https://doi.org/10.1145/1160633.1160866
20. Hindriks, K.V.: Programming rational agents in GOAL. In: Multi-Agent Programming: Languages, Tools and Applications, pp. 119–157. Springer, Boston, MA, USA (05 2009). https://doi.org/10.1007/978-0-387-89299-3_4
21. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8), 666–677 (Aug 1978). https://doi.org/10.1145/359576.359585
22. Husaini, S.F.: Using the Java native interface. XRDS: Crossroads, The ACM Magazine for Students **4**(2), 18–23 (1997). https://doi.org/10.1145/332100.332105
23. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (jul 1978). https://doi.org/10.1145/359545.359563
24. Ling, Y., Mullen, T., Lin, X.: Analysis of optimal thread pool size. ACM SIGOPS Oper. Syst. Rev. **34**(2), 42–55 (2000). https://doi.org/10.1145/346152.346320
25. Loring, M.C., Marron, M., Leijen, D.: Semantics of asynchronous JavaScript. In: Ancona, D. (ed.) Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages, Vancouver, BC, Canada, October 23 - 27, 2017. pp. 51–62. ACM (2017). https://doi.org/10.1145/3133841.3133846
26. Milner, R.: A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol. 92. Springer (1980). https://doi.org/10.1007/3-540-10235-3
27. Odell, J.: Objects and agents compared. J. Object Technol. **1**(1), 41–53 (2002). https://doi.org/10.5381/JOT.2002.1.1.C4
28. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the A&A meta-model for multi-agent systems. Autonomous Agents and Multi-Agent Systems **17**(3), 432–456 (Dec 2008). https://doi.org/10.1007/s10458-008-9053-x, Special Issue on Foundations, Advanced Topics and Industrial Perspectives of Multi-Agent Systems
29. Palanca, J., Rincon, J.A., Carrascosa, C., Julián, V., Terrasa, A.: A flexible agent architecture in SPADE. In: PAAMS 2022, Proceedings. Lecture Notes in Computer Science, vol. 13616, pp. 320–331. Springer (2022). https://doi.org/10.1007/978-3-031-18192-4_26

30. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI reasoning engine. In: Multi-Agent Programming: Languages, Platforms and Applications. Springer US, Boston, MA, USA (2005). https://doi.org/10.1007/0-387-26350-0_6
31. Rao, A.S.: Agentspeak(l): BDI agents speak out in a logical computable language. In: Agents Breaking Away. pp. 42–55. Springer Berlin Heidelberg, Berlin, Heidelberg (1996). https://doi.org/10.1007/BFb0031845
32. Reisig, W.: Petri Nets: An Introduction, EATCS Monographs on Theoretical Computer Science, vol. 4. Springer (1985). https://doi.org/10.1007/978-3-642-69968-9
33. Ricci, A., Santi, A.: Concurrent object-oriented programming with agent-oriented abstractions: the ALOO approach. In: Proceedings of AGERE!@SPLASH 2013. pp. 127–138. ACM (2013). https://doi.org/10.1145/2541329.2541333
34. Ricci, A., Santi, A.: From actors and concurrent objects to agent-oriented programming in simpal. In: Concurrent Objects and Beyond – Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 8665, pp. 408–445. Springer (2014). https://doi.org/10.1007/978-3-662-44471-9_17
35. Shoham, Y.: Agent-oriented programming. Artif. Intell. **60**(1), 51–92 (1993). https://doi.org/10.1016/0004-3702(93)90034-9
36. de Silva, L.: An operational semantics for true concurrency in BDI agent systems. In: EAAI 2020. pp. 7119–7126. AAAI Press (2020). https://doi.org/10.1609/AAAI.V34I05.6199
37. de Silva, L., Meneguzzi, F., Logan, B.: BDI agent architectures: A survey. In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020. pp. 4914–4921. ijcai.org (2020). https://doi.org/10.24963/ijcai.2020/684
38. Wooldridge, M.J., Jennings, N.R.: Intelligent agents: theory and practice. The Knowledge Engineering Review **10**(2), 115–152 (1995). https://doi.org/10.1017/S0269888900008122
39. Zatelli, M.R., Ricci, A., Hübner, J.F.: Evaluating different concurrency configurations for executing multi-agent systems. In: Engineering Multi-Agent Systems - Third International Workshop, EMAS 2015, Istanbul, Turkey, May 5, 2015, Revised, Selected, and Invited Papers. Lecture Notes in Computer Science, vol. 9318, pp. 212–230. Springer (2015). https://doi.org/10.1007/978-3-319-26184-3_12