Auton Agent Multi-Agent Syst (2026) 40:18
https://doi.org/10.1007/s10458-026-09744-w

Testing BDI-based multi-agent systems using discrete event simulation

Martina Baiardi[1] · Samuele Burattini[1] · Giovanni Ciatto[1] · Danilo Pianini[1]

Received: 28 February 2025 / Accepted: 4 March 2026
© The Author(s) 2026

Abstract

Multi-agent systems are designed to deal with open, distributed systems with unpredictable dynamics, which makes them inherently hard to test. The value of using simulation for this purpose is recognized in the literature, although achieving sufficient fidelity (i.e., the degree of similarity between the simulation and the real-world system) remains a challenging task. This is exacerbated when dealing with cognitive agent models, such as the Belief Desire Intention (BDI) model, where the agent codebase is not suitable to run unchanged in simulation environments, thus increasing the reality gap between the deployed and simulated systems. We argue that BDI developers should be able to test in simulation *the same* specification that will be later deployed, with no surrogate representations. Thus, in this paper, we discuss how the control flow of BDI agents can be mapped onto a Discrete Event Simulation (DES), showing that such integration is possible at different degrees of granularity. We substantiate our claims by producing an open-source prototype integration between two pre-existing tools (JaKtA and Alchemist), showing that it is possible to produce a simulation-based testing environment for distributed BDI agents, and that different granularities in mapping BDI agents over DESs may lead to different degrees of fidelity.

Keywords BDI agents · BDI models · Beliefs-desires-intentions · Multi-agent systems engineering · Discrete event simulation · Software testing

Martina Baiardi and Samuele Burattini contributed equally to this work.

✉ Martina Baiardi
m.baiardi@unibo.it

✉ Giovanni Ciatto
giovanni.ciatto@unibo.it

Samuele Burattini
samuele.burattini@unibo.it

Danilo Pianini
danilo.pianini@unibo.it

[1] Department of Computer Science and Engineering, Alma Mater Studiorum—Università di Bologna, Via dell'Università, 50, Cesena (FC) 47522, Emilia–Romagna, Italy

Published online: 02 April 2026

1 Introduction

Modern software systems are increasingly complex, and novel programming paradigms are emerging to tackle application domains such as Socio-Technical Systems (STSs) [1], Cyber-Physical Systems (CPSs [2], or Pervasive Systems (PSs) [3], where people interact with many computational entities situated in a physical environment that feature automation, autonomy, and intelligence. In these contexts, Multi-Agent Systems (MASs) have been widely adopted to tame complexity [4], as they model software as a collection of interacting autonomous entities.

Depending on whether MASs were exploited to *study* systems' complexity, or to *tame* it, many solutions have been proposed to either use MAS abstractions to *simulate* system behaviour [5] or for designing and *deploying* real-world systems [6].

The difference is subtle but crucial. When simulated, agents operate in a virtual sandbox environment designed to imitate the dynamics of the target real-world deployment in which perceptions (e.g., data from sensors, the flow of time) and actions (e.g., agents' movements in the environment) are controlled by the simulator, and can be reproduced deterministically. When deployed, agents are attached to specific hardware and software runtimes, communicate through networks, and are tied to the flow of real time.

Agents could either run in virtual environments as concurrent (and often distributed) processes interacting with shared resources (e.g., files, databases, network sockets) hence perceiving the state of such resources in real time and acting on them. In some cases, agents may even be embodied in physical entities (e.g., robots, drones, embedded devices) hence perceiving and acting through physical sensors and actuators.

Any of these elements may introduce non-determinism and unpredictability in the system, to which the MAS must be ready to react.

Accordingly, in this paper, when referring to *real-world deployment*, we intend software agents running on their final "production" platform rather than within a simulation, regardless of whether they are situated in a physical environment or not. With *real world* we therefore mean the target execution context for which the agents are designed and implemented.

Aside from being useful in in-silico studies, *simulation* may also aid the *development* and *validation* [7] of MASs intended for real-world deployment. In fact, simulation enables developers to test the behaviour of agents and their dynamics in complex environments ahead of deployment, while postponing and (ideally) reducing costs and efforts, while mitigating risks, associated with real-world execution. The price for having the possibility to test a MAS specification in a simulated environment is paid in additional development effort. First and foremost, developers must model the *target* deployment context as a simulation environment. Moreover, developers must develop the behavior of the agents within the simulation framework, which may differ from the one used for real-world deployment, making sure to maintain alignment between the two versions of the MAS codebase.

While the first challenge is unavoidable for any kind of simulation, we argue that the second one could be mitigated in case that the same MAS specification can execute *with no changes* on both real hardware and a simulator of choice.

The situation is particularly challenging when dealing with MASs featuring cognitive models of agency, such as the Belief Desire Intention (BDI) model [8]. BDI agents

have been proposed for complex scenarios, including healthcare [9], multi-UAV coordination [10], social simulations [11], and cyber-security [12]. The abstraction gap between the BDI programming model and most simulators [13], along with the minimal support of BDI technologies for producing *reproducible* simulations of articulated scenarios [14] lead developers to resort to one of the following approaches:

1. extension of the BDI platform with a dedicated simulation engine, which requires additional development effort;
2. construction and maintenance of two parallel codebases (one for simulation and one for the actual system), leading to consistency issues; or
3. integration of the BDI platform with a general-purpose simulation engine, typically by synchronising their execution through some form of middleware [13].

All such approaches come with their own drawbacks, and, above all, they are symptomatic of the lack of a general solution. In fact, while the effort of modelling the environment is unavoidable for any simulation – and challenging per se [15] –, porting agents' behaviour back and forth between simulation and real systems should be as simple as possible, ideally requiring no changes to the code.

Problem statement In our view, the state of the practice of BDI systems engineering lacks a general-purpose, practical solution for testing BDI systems using simulation, before deployment. Put differently, switching between simulation and real-world deployment is currently cumbersome and impractical. Ideally, starting from the same codebase expressing a BDI system specification, developers should be able to test it in a simulated environment, or run it on real hardware, with no changes to the codebase, while expecting the system to behave consistently in both scenarios.

The problem may seem merely technical, but indeed reaching this goal requires redesigning how BDI specifications are written and interpreted. They should abstract away the implementation details of their execution, while multiple execution engines should support both simulations and real-world deployments. In the particular case of simulations, the environment as well should be programmable,other than only agents.

In practice, we focus on a particular class of BDI programming frameworks: namely, the ones implementing the AgentSpeak(L) language [8] —which gained widespread adoption among BDI programmers according to some recent surveys [6, 16]. In the remainder of this work, we will write "BDI framework" to refer to AgentSpeak(L)-based BDI frameworks, unless explicitly stated otherwise.

Contribution In this paper, we address the long-standing problem of **enabling BDI-based MASs to be executed in a simulation environment with no (or minimal) changes to the original specification** [7]. We do so by tackling the following sub-objectives:

O1 Investigate the practical trade-offs of the methods that have been proposed to develop and test BDI agents in simulation.

O2 Understand which abstractions a BDI framework should provide to integrate seamlessly with a simulation engine, such that the same specification can be executed in both simulation and real-world deployment.
O3 Analyse how different mappings between BDI agents' execution model and Discrete Event Simulation (DES) can be designed, and how they impact the simulation fidelity.
O4 Integrate a BDI framework and a DES simulation engine in such a way that the same specification can be executed in simulation and in real-world deployment, proving feasibility.

Strategy Core to our contribution is the identification of DES as a key enabler for this goal, as well as an underexplored approach for BDI systems simulation. Along this line, the first contribution of this paper is the identification of critical challenges that emerge when attempting to map a BDI MAS on a DES simulator. We analyze first the conceptual implications of such mapping, focusing on the granularity of how BDI agents' control loops can be mapped onto DES events, highlighting the trade-offs of different design choices. Then, we discuss the technical implications of such mapping, which impact the design of the agent programming framework, to make it easier to integrate seamlessly with a simulator.

To substantiate our claims and to demonstrate the feasibility of our approach, we also implement a prototype solution based on the integration of two existing tools, namely: JaKtA [17] – a BDI programming framework –and Alchemist [18] general-purpose DES simulator. The key novelty here is that our prototype focuses on making BDI simulations reproducible and transferable, while also supporting different granularity levels and, therefore, different trade-offs in terms of complexity and fidelity. Aside from providing a practical tool for MAS developers, our goal is to document the experience gained during this integration.

Finally, we exercise the prototype in a (simulated) multi-drone coordination scenario, to exemplify how it can be exploited as means for the *early* testing of BDI systems. More specifically, we show how simulating the same BDI specification with different complexity/fidelity trade-offs may help in spotting issues that would have been otherwise overlooked.

Structure of the paper In Section 2, we recall most relevant concepts behind BDI agents and simulation, proposing DES as a natural and robust way to simulate BDI systems. Then, in Section 3, we discuss existing approaches to simulate BDI-programmed software in complex environments, commenting on their limitations. In Section 4, we show how to bridge the gap between BDI and DES, commenting on the technical implications simulating MASs, and the granularity at which the BDI reasoning cycle can be mapped into simulation events to increase simulation fidelity. We then present in Section 5 an open-source prototype that allows a BDI agent specification written in JaKtA to be executed on the Alchemist simulator. By leveraging the modularity and extensibility features of the selected technologies we are able to achieve a fully integrated solution that does *not* rely on ad-hoc synchronisation mechanisms, while still benefitting from the robustness of a state-of-the-art simulation engine. Finally, in Section 6, we exercise the prototype in a multi-drone coordination scenario, discussing how choosing different mappings between BDI and DES events can lead to misleading results, before concluding and discussing future works in Section 8.

2 Background

Here, we recall the main notions behind BDI agents and computer simulation, under a unifying perspective rooted in the notion of *event*. Accordingly, in Section 2.1 we first summarise the state of the art of BDI agents architectures, then we analyse which events are involved in their lifecycle. We also report in Section 2.2 major definitions such as discrete-event, -time, and multi-agent-based simulation, to ground our selection for the mapping of BDI in simulation.

2.1 The belief desire intention model

The BDI model, rooted in human psychology, is based on the explicit representation of the cognitive process of agents' decision-making. Originally a philosophical concept [19], it is now used to program [20] and simulate [21] agents, and model social behaviour [11]. BDI systems are therefore particular cases of MASs, where agents are endowed with *cognitive* abstractions.

More precisely, the model in [8] prescribes that every agent: *(i)* memorises (possibly partial, or wrong) *beliefs* about itself and its surrounding environment, that it can update by perceiving the environment, by reasoning, or by agent-to-agent communication; *(ii)* is driven by internal *desires* (also known as "goals") it is willing to accomplish, the desires may change over time or lead to new desires when pursued; *(iii)* may commit to several concurrent *intentions* as an attempt to pursue its desires; *(iv)* applies *plans*, i.e., recipes of *actions* that (the agent expects) will lead to the satisfaction of its desires.

BDI architectures Many architectures have been proposed in the literature to implement BDI agents in software, including AgentSpeak(L) [8], Procedural Reasoning System (PRS) [22], and distributed Multi-Agent Reasoning System (dMARS) [23]. In this work, we focus on the former, as it is one of the most widely adopted in the literature [6, 16].

AgentSpeak(L) [8] agents are grounded on the notion of *events*, to which agents react by selecting plans driving the execution of the agent. Differently from purely reactive systems, in which events are external stimuli directly mapped to actions, BDI agents are capable of reasoning about such events based on the context of their runtime state and select plans accordingly. Plan execution can produce additional internal events (e.g., sub-goals) that will drive subsequent iterations of the agent's reasoning cycle. Hence, for the remainder of this paper, when we refer to BDI agents as *event-driven*, we aim to highlight that, as we recall in the remainder of this section, the entire lifecycle of a BDI agent is driven by events, either received as environmental perceptions, or generated by the internal processes of the agent. Such event-driven nature of BDI agents is a key feature, especially when it comes to simulate them, as *in-silico* simulations essentially deal with the generation and processing of events in a controlled and predictable way. For these reasons, in the remainder of this work, we abuse the notation by referring to BDI agents and AgentSpeak(L) agents interchangeably.

Events All of the main abstractions of the BDI model can be linked to events that drive the agent's behaviour. Beliefs – encoding information currently stored in the agent memory and considered to be true – are updated in response to *events* from the environment (perception), other agents (messages), or by the agent itself during the execution of plans. When the agent adopts a new goal, a new intention is created to choose a plan to pursue such new goal. Plan executions may cause new events (e.g., sub-goals, belief updates, actions on the environment) to be generated, and so on.

We can distinguish two main categories of events: *internal* and *external*. Such a distinction is not present in the AgentSpeak(L) model, nor necessarily implemented by BDI frameworks, but it is useful in the remainder of this work to discuss the mapping of BDI agents onto DES.

One the one side, *internal* events are generated by each agent during the execution of its own control loop, and they are meant to make it progress. These include:

- belief additions, removals, updates;
- the addition or failure of novel (sub-)goals.

On the other side, *external* events are related to the environment and the changes therein occurring, and, in particular, how these changes are perceived or provoked or reacted to by the agents. External events include – but are not limited to – an agent:

- entering or exiting the environment (i.e., terminating);
- perceiving or receiving messages from the environment;
- sending a message to some recipient;
- executing an action that mutates the environment.

In the current practice, the specific way external and internal events are handled may vary across BDI implementations and be fairly different between in-silico simulations and real-world executions, indicating a substantial abstraction gap.

Additionally, it is important to notice that further *external* events may occur independently of agent actions and provoke changes in the environment. These may or may not be explicitly modelled, but for the sake of BDI agents programming, it is sufficient to model the *perception* of these changes. For instance, should a BDI system be affected by the environmental temperature, a real-world deployment would require temperature sensors (or an external temperature monitoring service) to be attached to the agents, to perceive a *spontaneously*-changing temperature from the environment. Should the same BDI system be simulated, the simulator would be in charge of generating events related to temperature variations, so that the agents can observe them and react accordingly.

Control loop Each AgentSpeak(L) agent is animated by a control-loop, i.e., a sequence of operations (grouped into three major phases) that are repeated indefinitely until the agent terminates. Technically speaking, the control-loop has been implemented in so many ways – each one coming with a different impact on the external concurrency of the agents [24] –, but the core abstraction is always the same.

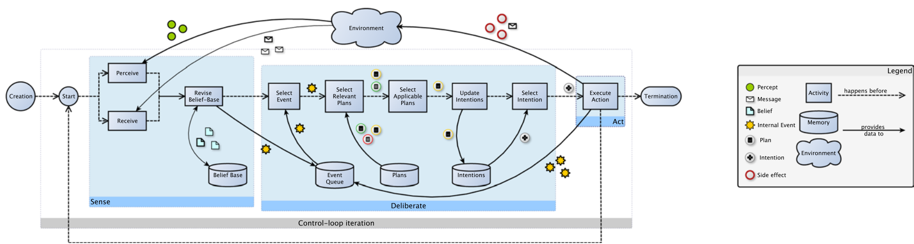

Fig. 1 Graphical representation of the AgentSpeak(L) architecture for BDI agents

Figure 1 shows a graphical representation of overall architecture of an AgentSpeak(L) agent, with a focus on the major phases and operations composing the agent's control loop, and the internal/external events therein involved. Roughly speaking, the control-loop assumes that each agent has a queue of internal events manipulated as follows. Each iteration of the control-loop starts with the *sense* phase, where the agent collects *(i)* percepts and *(ii)* incoming messages, from the environment (i.e. external events), *(iii)* updates the beliefs accordingly (enqueuing events from the updated beliefs). Next, the agent executes the *deliberation* phase, where it *(a)* picks one event from the queue; *(b)* selects a plan for handling the event, assigning it to some pre-existing or novel intention; and, consequently, *(c)* selects an intention to advance. Finally, the agent executes the *act* phase, where it executes one action from the intention selected in step *(c)*. The action execution may provoke further internal events, which are then enqueued (and therefore processed in the next iterations of the control-loop), as well as external events, which are delivered to the environment or other agents.

Execution The execution of a BDI agent boils essentially down to scheduling control-loop phases and handling events. Practically, this may vary across implementations and be fairly different between simulations and real-world executions. Specifically, defining how the control loop is scheduled may lead to different emerging behaviours of the overall MAS due to which interleaving of agent's control loops is allowed by the implementation. On the one side, in simulations, the simulator is in charge of (reproducibly) virtualizing the flow of time. In this case, 'executing the BDI system' means to advance the simulation time while deciding which agent(s) shall execute their control-loop phases next. The goal here is emulate concurrency, while keeping the execution of the agents deterministic and reproducible. On the other side, in real-world deployments, the agents are executed by the operating system, the flow of time corresponds to the real-world time, and the agents are scheduled by the operating system's scheduler, which may introduce non-determinism in the execution of the agents. In this case, the goal is to balance agents execution onto the available computational resources, and to exploit parallelism accordingly.

2.2 Simulation concepts: DES, DTS, MABS

We call simulation the process of replicating a real-world process in a simplified and controlled fashion to gather insights about its behaviour. Simulation can be performed by different means, in this paper we refer to *in-silico* simulation (where the system model is described in software and executed on a computer) of systems that feature temporal evolution.

A general problem in simulation is how to ensure that the simulated system has sufficient similarity with the real-world system it aims to replicate. The degree of similarity is often referred to as the *fidelity* of the simulation [25]. A related concept often used in software engineering is the *reality gap* [26], which refers to the discrepancies that may arise when transitioning from a design to its real-world implementation.

When using simulation as a validation tool, the reality gap can be used to refer to the differences between the behaviour of the simulated system and of the real-world system once deployed. The concepts are obviously related: increasing the fidelity of the simulation reduces the reality gap, and a wide reality gap is often a symptom of low fidelity. In this paper we will use both concepts – which are valid for all kinds of simulated systems – to discuss the trade-offs of different design choices when simulating BDI MASs.

Main kinds of simulation Temporal evolution can be driven by events that happen and cause time to advance, in case of Discrete Event Simulation (DES); or by "forced" leaps forward (typically of a constant amount of time) that cause the re-evaluation of the system state, in case of Discrete Time Simulation (DTS). Different models of temporal evolution are suitable for different scenarios, with DESs being generally capable of capturing more fine-grained dynamics (for instance, stochastic chemistry [27]), while DTSs being better at scaling and parallelisation when the phenomenon under study has a continuous nature in time (for instance, n-body simulations [28]).

Simulation frameworks that rely on the notion of agent to model their domain are called Multi-Agent-Based Simulation (MABS) [5, 29], of which some prominent examples in the literature include NetLogo [30], Mason[31], RePast [32], GAMA [33]. and Alchemist [18]. MABS are designed to support the modelling of complex systems in terms of independent agents, with the goal to replicate complex phenomena for which other models (e.g., Ordinary Differential Equations (ODEs)) are unsuitable [34, 35].

As such, MABS are *not* primarily designed to support *testing of agent-based software*. Indeed, they expose their own notion of agent, typically simpler than agent abstractions devoted to programming distributed systems, such as BDI agents. Using a MABS framework to simulate a BDI MAS requires a significant effort to either write a simplified version of the system to be tested (at the price of losing fidelity) or build an adaptation layer mapping the original BDI program onto the MABS' abstractions.

Simulation for BDI MASs testing In this work, we are interested in leveraging simulation as part of the development toolkit of BDI MASs, specifically to test the system's behaviour in a controlled environment ahead of time, to discover bugs, stress-test the system in corner cases, measure performance, investigate transients, and so on. To do so, it is paramount that the BDI specification (i.e., the program code) exercised in testing via simulation is *the same* that will be later deployed in the real-world system, or the price paid in fidelity will hardly be acceptable (and, as a consequence, testing via simulation will lose its potential value). Although not designed for this purpose, we argue that MABS are the tool that most closely resembles the needs of BDI developers. As previously introduced, however, there is a substantial abstraction gap between the two models, filling which is part of the contribu-

tion of this work. From the point of view of the time model, DESs appear the most natural choice to simulate BDI agents, as they are event-driven in nature as discussed in Section 2.1. One of the contributions of this work is showing how the control-flow events of BDI agents can be mapped onto DES engine events, and what are the implications of alternative design choices.

In exploring the trade-offs, we also consider practical considerations that affect how simulation supports software validation in early development stages. In many scenarios, simulation could be used to explore a cheap virtual prototype of the system, even before a real counterpart exists [36, 37]. As development progresses, the expected fidelity of the simulation should naturally increase, improving its usefulness as a validation tool. Achieving higher fidelity, however, often requires more fine-grained models, which come with greater computational cost. This leads to an inevitable trade-off between accuracy, development time, and simulation duration. Different phases of the engineering process may therefore call for different fidelity levels: fast, low-detail simulations to rapidly test ideas early on, and slower, high-detail simulations later when behaviour must be assessed more thoroughly.

3 BDI agents simulation methods

To achieve O1, in this section, we investigate the state of the art of BDI agents simulation.

The idea of testing MASs dynamics via simulation dates back to the early 2000s [7]. However, despite recognizing the relevance and complexity inherent in testing MASs [38, 39], and the existence of methodologies that include simulation for MAS development [40], there are not many tools that effectively allow one *"to execute agents as they are and to switch arbitrarily between execution in the real environment and the virtual test environment"* [7]. The lack of dedicated simulation tools is especially relevant for BDI agents, whose agents' lifecycle is richer compared to other models.

This seamless switching is the primary goal for our work [41]. In fact, keeping the *same* agent specification ensures that simulation results are not artefacts induced by the software translation into a simulation-compatible model. Additionally, if the specification is the same, issues discovered at deployment time that were not evident during testing will be at most imputable to the abstraction gap of the environment model. A recent work has tackled this problem for non-BDI agents [42], by developing a simulation environment for the JADE MAS platform [43]. The work focuses on achieving portability of the same agent behaviour between simulation and real-world execution, by implementing a simulation environment that mimics the JADE run-time environment. We exclude this work from our comparison, as JADE agents are not BDI-based.

Several attempts to use simulation as a testing tool for BDI MASs have been done in the past, falling into the three main categories [13], described below.

(1) Adding simulation features in BDI frameworks Agent frameworks can, in principle, be modified so that the MAS environment is replaced by a simulated one. In practice, this operation may be difficult because it depends on the *concurrency model* [44] adopted by the

framework and how customizable the framework is. Nevertheless, some BDI programming frameworks include simulated execution. For instance, Jason [21] allows a DTS execution mode with steps that execute one reasoning cycle for each agent, and in [45] is present a proposal to apply DES to JaCaMo.

(2) Implementing BDI features in simulators As discussed in Section 2.2, the agent model in MABS simulators are simpler than BDI agents, thus, their execution requires a reimplementation of the agent interpreter on top of the simulation engine. BDI extensions exist for several popular MABS. Some basic BDI and FIPA [46] communication libraries have been implemented in NetLogo [30] for educational purposes [47], and a BDI layer has been implemented on top of the GAMA [48] platform to support social simulations with cognitive models [49]. Other approaches rely on the Discrete Event System Specification (DEVS) formalism to represent BDI reasoning, such as JAMES [50].

(3) Integration through synchronisation Finally, an approach is the integration of an existing BDI framework with an off-the-shelf simulator. This is typically achieved through some form of synchronisation, either through inter-process communication or shared memory. A generic framework for such integration is proposed in [13] suggesting the adoption of a standard interface to map BDI agents with MABS agents; and showing validation through different combinations of BDI and MABS platforms. In [51], instead, a shared memory approach is used to make Jason agents interact with the simulated environment asynchronously, reducing the time the simulation engine spends *waiting* for the agent reasoning.

Remarks Categories (1) and (3) would preserve the same code for the agent behaviour. Category (1), besides requiring a considerable effort to build a simulation engine from scratch, would hardly match existing state-of-the-art simulators in terms of performance, environment modelling, visualization, and data export tools. Using strategy (3), as we discuss in more detail in Section 4, can have important consequences on the reality gap with respect to the real-world execution. Depending on how synchronisation is achieved, assumptions on agent executions may be introduced implicitly, possibly leading to unreliable results. Simulating the behaviour of an existing MAS following strategy (2) requires to rewrite the agents' behaviour using the simulator's Application Programming Interface (API). This is critical, as it is hard to guarantee that the software will be the same, and costly to realize and maintain.

In this paper, we take an approach that sits in between these categories. We do start from an existing BDI framework and an existing simulator, but instead of relying on some form of synchronisation, we make the simulator run BDI agents directly. This is similar to the approach used to support social simulations by integrating MASON [31] with Jason [20] by incorporating the Jason's interpreter within MASON to directly execute agents written in AgentSpeak(L) [52]. The main difference of our proposal is in the scope: we do not mean – as [52] – to build better social simulations using BDI agent modeling, rather, we want to provide a toolkit to test the behaviour of deployable BDI software systems. This difference in scope makes the execution of a complete reasoning cycle at each simulation step as done in [52] not viable under the fidelity point of view, as we will discuss in Section 4 and demonstrate in Section 6.

4 Integrating BDI and DES

In this section (and, more specifically, in Section 4.1), we discuss how to allow for testing the behaviour of BDI agents systems in a simulation environment, without changes to the codebase of the MAS specification (O2). In particular, as mentioned in Section 2.2, we are interested in studying the exploitation of DES to serve this purpose (O3). At the intuition level, this should be possible, as the events generated by the BDI agent's control loop can be scheduled in a DES engine and interleave with all other simulation events.

In principle, given a MAS made up of *(i)* agents' behavioural specifications, *(ii)* an environment abstraction to handle perceptions, actions, and communication, and *(iii)* an execution platform to run the system, one could retain the agents specification and swap the remainder with a simulator to achieve the desired objective. However, this task is *deceptively simple*: in the following subsection we will analyse how each of these elements needs to be carefully designed to make such swap possible.

Before proceeding, let us clarify that, in the remainder of this paper, we specifically focus on agents whose behaviour is *manually* specified beforehand by human developers – such as BDI agents – rather than *learned* autonomously —e.g., via reinforcement learning. In principle, though, similar considerations apply for learning agents, provided that learning occurs *online*[1] – meaning *during* the agent's interaction with the environment – as opposed to *offline* training —which occurs *before* agents' execution.

4.1 Portable agent specifications

We have established that the main goal of this work is to allow the same BDI agent specification to be executed both in simulation and in real-world deployments.

The technical implication of this goal is to ensure that the agent code is *portable* i.e., it depends solely on platform-agnostic APIs, which can be implemented differently depending on the execution context (simulation or real-world).

Platform-specific implementations – at least, one for real-world deployments and another for simulation – must be provided for such a platform-agnostic API. We refer to these implementations as the *interpreters* of the BDI agent specification, which are responsible for executing the agent code by relying on the underlying platform functionalities. The two interpreters may of course differ in their implementations, yet both should guarantee the AgentSpeak(L) semantics is preserved. In this regard, a good practice is to let the two interpreters share as much code as possible: ideally implementing the agent's control loop in a shared library, which allows for plugging in different platform-specific functionalities.

Given the role of environment abstractions for MAS programming [55] that can be used to abstract agents from the execution context – ranging from distributed systems [56] to

[1] If an agent's behavioural specification includes some form of *online* learning capabilities, (e.g., by implementing Q-learning [53]), then the agent would be able to learn from its experience in the simulated environment as it would do in the real-world deployment. However, we acknowledge that modern approaches to reinforcement-learning (e.g., deep reinforcement learning [54]) commonly rely on an *offline* training phase, which leverages the simulation environment to generate training data and update the agent behaviour (i.e., the learned *policy*) accordingly. This training phase is outside the scope of this work, but given the final specification of the agent behaviour (e.g., a trained neural network implementing a policy), our assumptions discussed at the beginning of Section 4 still apply. In fact, the learned policy would simply work in the target environment (no matter whether it is simulated or real-world), provided that the agents' perceptions and actions are properly abstracted and wired to the neural network's input and output layers.

robotics [57, 58] – we discuss what are the fundamental requirements to design an environment API that allows for portable BDI agent specifications that can seamlessly run on a simulation platform. Namely, the environment API must be designed to abstract the following functionalities.

Time Agent behaviour may depend on the passage of time. Hence, agents may need to observe the current time, usually through the system's clock in a real-world deployment. In simulation, however, the flow of time is virtualized and controlled by the simulator, and it usually does not correspond to real-world time. Thus, access to *time-dependent* functionalities (e.g., timestamps, timeouts, delays) must not be implemented by accessing system-level time primitives directly. Accessing the system's clock, or using sleep functions must be avoided in the agent code, and instead the environment API must provide the necessary functionalities to access the current time or to schedule timeouts and delays. It will then be the responsibility of the integration with the simulation platform to provide a proper implementation of such functionalities upon the simulator's notion of time, guaranteeing semantic correspondence with the real-world counterpart.

Randomness Sometimes, the agents' behaviour may require a degree of *non-determinism* —e.g., using random number generators. In simulation, to ensure reproducibility of simulation, all non-determinism must be controlled by the simulator. Thus, similar to time-related primitives, random number generation must be provided by the environment API. Reproducibility is especially significant when simulations are a mean to debug anomalies. When debugging, any two executions with the same random seed *must* produce the same sequence of events, otherwise, the original issue may not re-occur, or, worse, may manifest differently, making understanding and fixing the problem much harder— an issue known in the software engineering community as *flaky behaviour* [59].

Perceptions and actions Agents' interaction with the environment happens through perceptions and actions. On the one side, the environment API must provide functionalities to perceive the state of the environment entities and deliver perception *events* to the agents. On the other side, the environment API must encapsulate all the actions that agents can take to influence the environment state. This level of separation is the most common in MAS programming frameworks, but it is worth stressing its importance when the goal is to run the same agent specification both in simulation and in real-world deployments. In (Section 4.2) we will discuss in depth the implications of this abstraction when discussing how to bind the BDI environment API to a DES simulation platform.

Communication Agents in a MAS often need to communicate with each other. To support inter-agent communication, the environment API must provide functionalities to send and receive messages. Real-world (distributed) deployments rely on network protocols (e.g., HTTP, MQTT, XMPP, etc.), and although in principle they can be used in simulation as well, a typical choice driven by performance and improved experimental control is to abstract them away and provide direct control over relevant control variables (latency, message/package loss, retransmissions). Typically, in MAS the environment acts as a *relay* for messages, and messages might be subject to some sort of *propagation* mechanism. The

way messages are propagated may depend on the scenario, for instance, taking into account the environment topology by limiting the delivery of messages to *reachable* destinations, or introducing some form of *delay* or *loss* in the delivery of messages to better match with the expected real-world environment dynamics and stress the properties of the system under different conditions. Again, failing to abstract communication functionalities would make impossible to simulate the system, as the simulator would need to rely on the underlying network stack to deliver messages between agents, which would make the simulation results unreliable and non-reproducible.

4.2 Design of the simulated environment

Having defined the requirements for portable BDI agent specifications (Section 4.1) by encapsulating all platform-specific functionalities into a generic environment API, we now focus on the implications of implementing such API on top of a DES simulation platform. These considerations are generally valid for any simulation environment that aims to support the typical application scenarios of (BDI) MASs, i.e., distributed systems where agents interact with each other and with a dynamic environment, possibly featuring entities that can move in a physical space. Although not specific to BDI agents, we still discuss these aspects, as they set requirements and constraints in the choice of the simulation platform and in the design of the simulation environment that will host the BDI MAS under test.

The most important difference between real-world deployments and simulations concerns the way *external* events are delivered to the agents. In this section we focus on the modeling of such events, leaving the discussion on the scheduling of *internal* events to Section 4.3. The BDI model already assumes agents to be reactive to the external events they perceive. In real-world deployments, developers can wire BDI agents' perceptions and actions to physical sensors and actuators or connect the agents with I/O peripherals and external services. External events just occur, and agents either perceive or cause them. Instead, in simulation, external events are delivered to the agents by the simulator, and therefore require additional modelling and programming efforts.

Topology and situatedness BDI MASs can be deployed either in physical or virtual environments. Depending on the nature of the target scenario, to accurately replicate the real-world mechanisms, the simulation environment must support the notion of *situatedness* (cf. Wooldridge [60]), that is crucial for all kinds of agents and determines the range of observability and influence of the agents. In some cases, for instance when the target scenario features robots or mobile devices, agents need to be further *embodied* [61], making them directly part of the environment, and observable to other agents.

While in real-world deployments the space and topology of a MAS is a *constraint*, in simulation they are aspects to be modelled and programmed; and the programming efforts might vary significantly depending on the target application. When agents are expected to be deployed on mobile devices, the simulation can use manifolds (e.g., 3D spaces or city maps) where location and distance can be defined. In other cases, for instance in *networked* systems, manifolds are not relevant, but the *network topology* is. These are *logical* environments, in which entities are located into nodes of a graph, and distances are computed

as functions of the paths connecting them. In some applications, these aspects mix: for instance, a wireless sensor network may feature geographically *situated* devices for which a logical topology can model the communication channels (e.g., based on the relative distance of the devices). In any case, a notion of *closeness* is necessary to limit the perception range and influence of agents, whose specific reification should be left to the simulation implementation, as it is scenario-dependent.

These aspects are important to consider when implementing the environment API on top of a DES platform. For instance, the perception range influences which events should deliver perceptions to the agents, and the topology may affect how messages are propagated in the environment. These parameters should be carefully modelled in the simulation and can be used as variables to explore different simulated scenarios.

Environment dynamics In any non-trivial scenario, there will be events generated externally to the BDI agents, as real-world environments are typically dynamic (i.e., they change over time) or feature some form of stochastic behaviour (e.g., package loss in networked systems). Movement, communication, and, more generally, any phenomena that span over time affecting the agents' perceptions, must be modelled into events with appropriate time distributions. These events must be modelled in simulation and properly interleave with the agents' reasoning cycle. The interleaving between agents' internals and other events essentially determines how agents are executed in the simulation, as we discuss in the next section. The design of the environment dynamics allows one to implement the API used by the agents to perform actions that have effects on the environment state, but also to model external sources of change completely independent of the agent behaviour, that may challenge the MAS execution, thus stressing its properties in different scenarios. As for any simulation, the more accurate the model of the environment dynamics, the higher the fidelity of the simulation will be.

4.3 Mapping the BDI execution on a simulator

Mapping BDI events in a DES to achieve O3 requires an understanding of which events are *atomic* in a BDI system execution. Then, any collection of these atomic events can be mapped to a single DES event, defining the mapping *granularity*, and thus defining which BDI groups of events can interleave with other simulation events. The granularity influences the *fidelity* of the simulation, and is similar to choosing the external concurrency model [44] of a BDI system upon deployment. The granularity does not affect the individual agent program semantics (the specification of behaviour is the same), but may influence the collective dynamics of the MAS due to the interleaving of individual agent actions. Coarse-grained mappings (i.e. multiple agents events mapped to one simulation event) introduce implicit synchronisation that suppresses possible interleavings, thus potentially hiding timing-dependent faults, race conditions, message races, and sensor (or actuators) delays that would happen in a real deployment. Finer-grained (i.e. one agent event mapped to one simulation event) mappings expose asynchrony and delays, increasing fidelity and reducing the reality gap, but at the cost of a more difficult integration mapping. Figure 2 summarises the options that we discuss below, showing the trade-offs for each choice.

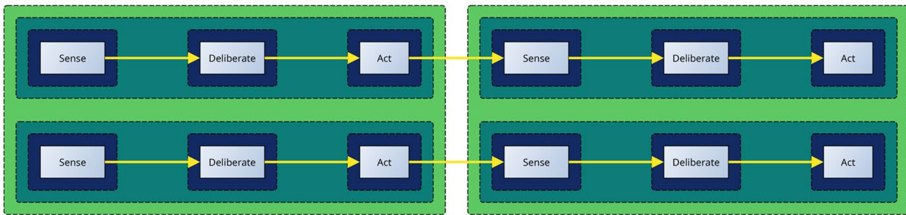

Fig. 2 Granularities for atomic BDI events. Yellow arrows are causal links. In AMA (light green) an event is a run of all control loops of the whole MAS; it implicitly synchronises all agents. In ACLI (emerald) an event is a single control loop iteration of a single agent, preventing phase interleaving. In ACLP (dark purple), single control loop phases of every agent can interleave. ABE is not shown as it is indistinguishable from ACLP at the agent level

4.3.1 Atomic MAS advancements (AMA)

Each DES event corresponds to a *full* control-loop iteration of *all* agents in the system, making it the most coarse-grained option. This approach is adopted in [21] and [52], which use DTS, and it is similar to the one proposed in [13], which advances the simulation only after all agents are *idle*. One obvious consequence of this choice is that all agents in the MAS run at the same *frequency*, thus inducing implicit synchronisation, and discarding all the effects caused by possible *interleaving* agents' actions.

4.3.2 Atomic control-loop iterations (ACLI)

Each DES event corresponds to a *full* control-loop iteration of a *single* agent, making this option slightly more fine-grained than AMA. In this case, arbitrary interleaves among agents' *actions* are possible, but not among different agents' control-loop *phases*. This option does not allow modelling different durations for different phases of the control loop, for instance, it cannot be captured a situation in which deliberation takes too long and makes the sensing phase outdated before an action is taken.

4.3.3 Atomic control-loop phase (ACLP)

Every single *atomic phase* (sense, deliberate and act) is represented as a DES event. This option preserves the behaviour of concurrent or distributed BDI agents, allowing phases to interleave across different agents, thus capturing complex inter-agent concurrency patterns.

4.3.4 Atomic BDI event (ABE)

Each BDI event is mapped to a single DES event. This is the finest-grained option, first proposed in [45]. The approach has value for investigating the internals of the agent execution platform implementation, e.g., to verify the correctness of the BDI interpreter implementation and its degree of parallelism, but, from the point of view of an observer of the MAS, this model and ACLP are indistinguishable, as the phases of the control loop of each agent are atomic. For this reason, in the remainder of the paper, we will consider ABE to be subsumed by ACLP.

4.4 Tool requirements for the integration

Following the key features outlined in the sections above, the tools to be adopted for the integration must provide:

- neat *modularisation* of the BDI execution engine, to simplify its replacement with the simulation engine;
- abstract *"external concurrency"* model [44], to allow for different granularities of the BDI events – and therefore different fidelity levels – in simulations;
- *extensibility* of the DES framework, which must accommodate the BDI framework's abstractions with no major changes;
- *simulation model* featuring locality, position, and communication channels, to support simulating agents situated in physical or logical spaces;
- native *integration* (i.e., same runtime) between the BDI framework and the simulator, to avoid the additional burden of inter-process communication mechanisms.

5 Prototype: JaKtA over Alchemist

To demonstrate the feasibility of the ideas discussed in the previous sections, and in order to achieve O4, here we present a proof of concept implementation of a BDI interpreter supporting both real-world and simulation execution.

In particular, our discussion revolves around the JaKtA BDI framework [17], which *already* supports real-world execution of MAS as concurrent applications, and it allows for plugging in different *concurrency models* [24]. Put simply, JaKtA is a BDI interpreter which lets developers customise the way agents' control loop iterations – as well as agents' perceptions and actions – are executed. For the sake of conciseness, in this section, we only show how JaKtA can be extended to support one more way to run agents, namely: in a simulated world, as created by the Alchemist simulator [18]. The interested reader can refer to [62] for a more detailed discussion on JaKtA, and its support to BDI agents in real-world deployments.

Further motivations exist for the technological choice of JaKtA and Alchemist, as we discuss in Section 4.4 and in Section 5.1. As the reader will notice, our prototype is tailored on these two technologies, and the discussion in this section is specific to them. However, we believe that our experience has a general value: not only it proves that BDI systems' execution can be swapped between real-world deployments and DES simulation without changing the MAS specification, but it also allows us to study the impact of the many simulation granularity levels (see Section 4.3) on the validation of a BDI system. This is indeed the purpose of Section 6.

Technically speaking, our prototype is implemented as a stand-alone module in the JaKtA's codebase, named `alchemist-jakta-incarnation`, which imports simulator's API and allows developers to plug their BDI specification to be executed in a simulated environment. The source code is publicly available[2] under a permissive licence and archived on Zenodo [63] for future reference.

[2] https://github.com/jakta-bdi/jakta

5.1 Design and technological choices

In line with the requirements in Section 4.4, we selected the Alchemist simulator and the JaKtA BDI framework also because they both run on the Java Virtual Machine (JVM) and are therefore interoperable. Alchemist is designed with extensible base abstractions, thus allowing for reasonably straightforward integration with other stand-alone tools; indeed, it has been used in the past to simulate programmable tuple spaces [64], biochemical systems [65], and aggregate programming languages [66, 67]. Complementarily, JaKtA has been designed to be easily adaptable to different execution engines. So, bridging the two technologies is a matter of implementing a new *incarnation* in Alchemist that can run JaKtA agents.

Alchemist Alchemist [18] is a general-purpose DES originally conceived to simulate bio-inspired systems, from which it inherits the metaphors and terminology in use to this day. In the remainder of this discussion, terms in `typewriter font` represent names of existing software entities, whose name has been used verbatim, as found in the API.

In Alchemist, the simulated model entry point is the `Environment`, which represents a Riemannian manifold.[3] The `Environment` is populated by `Nodes`, which *(i)* can be equipped with arbitrary `NodePropertys` *(ii)* have a `Position` in space, *(iii)* can be programmed with `Reactions`, *(iv)* can be connected to other `Nodes` via a `LinkingRule` to form `Neighborhoods`, *(v)* can contain `Molecules` (data identificators), each with an associated `Concentration` (data value).

Events in Alchemist are called `Reactions`, as they were originally conceived to simulate chemical reactions. `Reactions` in Alchemist extend the concept of reaction from chemistry.

Instead of being defined by a set of reactants that transform into products at a rate defined by the *law of mass action*, `Reactions` in Alchemist are more generally defined as collections of conditions, that, when satisfied, trigger a set of actions according to an arbitrary *rate equation* which depends on the observable state of the `Environment` and a provided `TimeDistribution`.

The concept of `Concentration` in Alchemist can be reified in arbitrary data types. Different types of `Concentration` may lead to a different semantics for `Molecules`, `Conditions`, `Actions`, `Reactions`, and any other abstraction depending on them, thus allowing for different *data models*, and providing a flexible and straightforward way to bridge Alchemist with other tools. Every set of abstractions defining a coherent semantics for these entities is called an `Incarnation` in the simulator jargon.

At the time of writing, Alchemist provides several incarnations for different application domains, including bio-chemical systems [65], networked tuple spaces [64], and aggregate programming via Protelis [66] and Scafi [67].

[3] A Riemannian manifold is a generalised Euclidean space that can be curved (useful to support geospatial data), discontinuous (useful to model inaccessible areas), and for which the distance between two points $\overline{AB}$ may depend on the direction: $\overline{AB} \neq \overline{BA}$ (useful to implement asymmetric navigation, e.g., on street maps with one-way roads).

An Alchemist simulation is configured using a YAML file, which specifies the `Environment` type, the `Nodes` to populate it with, their initial `Molecule` concentrations, the `LinkingRule` that connects them, and any `Reactions` to be executed.

When the simulation starts, Alchemist loads the configuration file and instantiates the specified `Environment` and its contents.

JaKtA JaKtA [62] is a Kotlin-based modular BDI framework. Each agent in a JaKtA MAS is equipped with an `AgentLifecycle` which implements the BDI control loop. JaKtA cleanly separates the definition of the agents' behaviour from the execution platform in charge of executing the agents' control loop phases, thus allowing different concurrency models for agents to be plugged in. JaKtA's environments (not to be confused with Alchemist's `Environments`) are lightweight abstractions that handle perceptions, communications, and interaction with the external world. MASs in JaKtA are programmed through a Kotlin Domain Specific Language (DSL), which provides an expressive, type-safe, and Integrated Development Environment (IDE)-friendly way to define agents' beliefs, goals, and plans. Indeed, by leveraging Kotlin as the host language, JaKtA programs are regular Kotlin applications, which may lower the entry barrier for developers familiar with mainstream programming languages (Kotlin is the reference language for Android development[4]) and are thus natively supported by any IDE supporting Kotlin, significantly lowering the maintenance cost.

Multiple granularity support As discussed in Section 4.3, the mapping between the BDI framework and the DES simulator can be realised with different granularity. We implemented multiple granularities in our prototype to show how each of them impacts on the tested behaviour of a BDI software. We implemented all those introduced in Section 4.3 but ABE because, as previously discussed, its behaviour is indistinguishable from ACLP and supporting it while retaining real-world deployment capability for JaKtA specifications would have been highly impractical.[5]

Driven by these choices, in the next subsection, we proceed to identify the modelling abstractions of the two technologies and we describe how they have been mapped in order to merge the JaKtA framework into the Alchemist simulator.

5.2 BDI MASs in Alchemist

Integrating the two frameworks has been first an exercise of model-to-model mapping. The mapping of the frameworks' abstractions on each other is summarised in Fig. 3.

Each agent's lifecycle phases (i.e., sense, deliberate, and act) are mapped onto separate DES events (Alchemist's `Reactions`). As introduced in Section 5.1, each such event has a duration modelled through a `TimeDistribution`. To guarantee that each agent's

[4] "Android's Kotlin-first approach", archived 2024-02-15 https://archive.is/EtWqn

[5] To support ABE, the JaKtA interpreter should be modified to expose each atomic event as a schedulable unit of execution. While this is feasible in principle, the effort required is not justified by the benefits, as ABE does not provide any additional value with respect to ACLP. One may argue that "mocking" is, in a broad sense, a form of simulation. However, as we discuss extensively in Section 2.2, we consider as "simulation" only the execution of a system in a *virtualised* execution environment, where both time and space are virtual and detached from the real world, and where the whole dynamic of the system is controllable. This is not the case for mocked components.

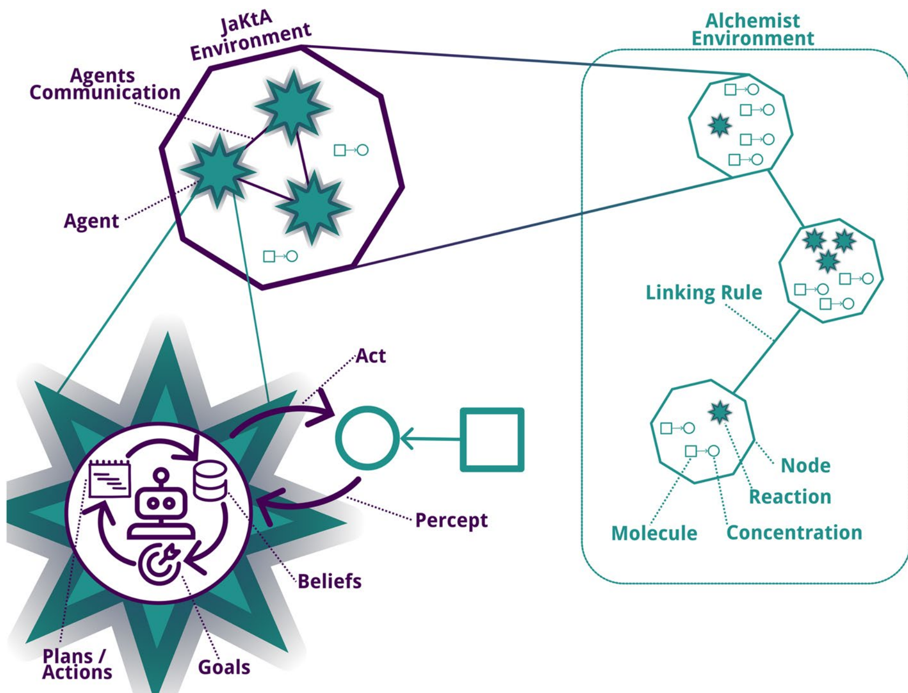

Fig. 3 Mapping JaKtA BDI abstractions onto Alchemist ones. Each Alchemist Node represents a physical *device* running a centralised JaKtA MAS. Simulation of distributed MAS is achieved having multiple Nodes in the same environment that communicate via Alchemist's Linking Rules. Each agent's control-loop is implemented as an Alchemist `Reaction`, with a custom `TimeDistribution` for each phase. Agents can act on and perceive Alchemist's molecules in the environment

control loop phases are executed in order, we implemented a custom `TimeDistribution` (`JaktaTimeDistribution`) composed of three sub-`TimeDistributions`, one for each phase of the control loop. `JaktaTimeDistribution` maintains an internal state which tracks the current phase of the agent, and executes the next phase according to its configured `TimeDistribution`. This design allows us to flexibly support different granularities:

- ACLP is immediately and natively supported, as any three distributions can be picked for the three phases.
- To support ACLI, the `TimeDistribution` of the *deliberate* and *act* phases must be an as-soon-as-possible distribution (for instance, a negative exponential distribution with rate $\lambda = \infty$), thus allowing the next agent's control loop to be scheduled immediately after the previous one.
- To support AMA, in addition to the constraint used for ACLI, the `TimeDistribution` of the *sense* phases across all agents must be scheduled to occur exactly once before any agent can execute a new control loop iteration: we do so by using, as *sense* `TimeDistribution` for any agent, a `DiracComb` with a period *T*, starting at a random time $\tau_0 < T$. This strategy guarantees that every *T* units of simulated time, all agents will have completed exactly one reasoning cycle, executed in some arbitrary (but

deterministic and equal for each T interval) order.

While the `TimeDistribution` for internal events depends on the chosen granularity, external events must be explicitly modelled in the simulation configuration. For example, the movement of a `Node` in the environment is an external `Event`, thus it must be specified with its own `TimeDistribution` in the simulation.

`Nodes` in Alchemist naturally map onto situated devices, as they possess a defined location in space. Consequently, we could not map JaKtA's `Agents` directly onto Alchemist's `Nodes`, as in a distributed BDI MAS multiple agents can be hosted on the same device. Rather, we mapped Alchemist's `Nodes` into instances of devices executing the JaKtA platform, thus capable of hosting multiple agents at once.

Mapping `Nodes` to platform instances (or devices) enables direct inheritance of the Alchemist's `Linking Rule` abstraction to model the communication channels, and in principle allows for the implementation of mobile agents. The Alchemist-compatible platform instance (named `JaktaEnvironmentForAlchemist`) is a JaKtA-specific `NodeProperty`, adapting JaKtA to run inside every Alchemist `Node`, driven by the simulator's control flow.

Agents can reason on the current device status, as information stored inside `Nodes`' is reified as perceived knowledge from JaKtA agents. Information inside nodes is stored as key-value pairs (`Molecules`) and associated `Concentrations` (c.f. Section 5.1), that agents can manipulate through actions. Changes are then treated by the simulator as "regular" environment modifications, consequent to simulated events.

Communication among agents is modeled via an Alchemist `NodeProperty`; each `Node` (device) is equipped with a message broker that collects messages for the agents on that device. When an agent sends a message to another agent on the same `Node`, the message is delivered directly to the recipient's message queue. When the recipient is on a different `Node`, the message is delivered if the recipient's `Node` is in the neighborhood of the sender's `Node`, according to the `LinkingRule` defined in the simulation configuration.

6 Evaluation

In this section, we exercise the prototype introduced in Section 5 to demonstrate *feasibility* of the proposed mapping of BDI agents over DES and the *transparency* of execution with respect to the agent specification. Additionally, we measure the impact granularity has on the *reality gap* of the simulation (O3), showing that the same MAS logic can produce different emerging behaviours when executed with different granularities.

Our evaluation mimicks a situation where multiple Unmanned Aerial Vehicles (UAVs), each controlled by a BDI agent, must cooperate to maintain a circular formation. To demonstrate the portability of the MAS codebase between simulation and real-world, we show that the same code can be executed inside the simulator or directly as a regular multithreaded application. In the former case, the UAV controls are simulated through the tools provided by the DES API, while, in the latter case, the UAV controls are encapsulated in a different API.

It is worth highlighting that our experimental design (specifically: the first part, where the MAS is validated via DES) is particularly useful to show the impact of simulation granu-

larity onto validation. In fact, as we discuss in the next subsections, choosing a coarser granularity for the simulation may hide potential issues in the agent programs, which would instead emerge when a finer granularity is used.

6.1 Use case description

We use a UAVs coordination scenario as our reference. We use JaKtA to instruct a flock of UAVs, each controlled by a BDI agent, to build and maintain a circular formation while following a moving leader UAV. The *leader* UAV moves in a circular path (radius $r_l = 5m$), while follower UAVs must coordinate to build a circular formation around the leader so that every follower is at distance $r_f = 2.5m$ from the leader, and all followers are equally distanced to each other. This formation must be maintained while the leader moves along its path.

We assume direct Line of Sight (LoS) UAV-to-UAV communication within a short range of $r_c = 5m$. UAVs can navigate the space at a maximum speed of $1\frac{m}{s}$. If no signal is received from the leader, followers hover at their current location. In this scenario, we assume UAVs are equipped with some localisation system that provides them with exact coordinates of their position in a given shared reference system. Initially, follower UAVs are randomly displaced into a circular arena of radius $10m$, while the leader is located at the arena centre.

Each UAV runs the BDI control loop in rounds at a certain maximum execution frequency $f = \frac{1}{T}$, cf. Section 5.2. Each control-loop phase would take some time to complete, depending on the complexity of the computations involved. For instance, a frequency of $f = 1Hz$ implies the execution of one agent control loop per second. Tuning this frequency is a way to balance the responsiveness of the agent to changes in the environment with resource utilization (e.g., battery consumption), and must be tuned to stay within the physical limits of the UAV's hardware.

Additionally, devices are not started simultaneously, each of them experiences a random start delay $\tau_0 \in \left[0, \frac{1}{f}\right]$ (or, equivalently, $\tau_0 \in [0, T]$ see Section 5.2).

6.2 Modeling UAV behaviour in JaKtA

During the experiment we deliberately used simplified BDI programs that rely on tight synchronisation of agents' control loops. We model two agent roles: a single *leader* and multiple *followers*. The leader continuously traverses a circular trajectory at constant speed. Its behaviour is expressed as a single BDI goal named `move` (i.e., following the circular path) and an associated plan that repeatedly:

1. moves the leader to the next waypoint along the circle, and
2. broadcasts a message to nearby agents (within communication radius r_c) inviting them to join the formation.

Followers do not possess a proactive goal; they are reactive agents driven by messages from the leader. Upon receiving a "join formation" message, a follower executes a plan that computes and moves to its assigned position in the circular formation. The target position

```
fun AgentScope.leader() {
  goals { achieve("move") }
  plans {
    +achieve("move") then {
      execute("circleMovementStep")
      execute("notifyAgent")
      achieve("move")
    }
  }
}

fun AgentScope.follower() {
  plans {
    val shape = "joinCircle"(C, R, N)
    +achieve(shape) then {
      execute("follow"(C, R, N))
    }
  }
}
```

```
agent("leader") {
  // Platform code
  ...
  // Shared logic
  leader()
}

agent("follower") {
  // Platform code
  ...
  // Shared logic
  follower()
}
```

Fig. 4 Code extracts from the companion artifact. The agent specification (left) is completely platform-agnostic and reusable, some glue code (right) wires the logics with the underlying execution platform

is determined from the current number of followers already in the formation. The BDI program that expresses this behaviour is shown in Fig. 4.

6.3 Experimental setup

Given the shared behavior implementation described above, in this section we describe how we experimented with the developed prototype to demonstrate that:

- different granularities in the simulation execution may lead to different emergent behaviours of the MAS,
- the same MAS codebase can be executed both in simulation and on a real-world deployment without modification.

UAVs in a real system would run independently. This makes the ACLP granularity the most realistic one, as it allows for control-loop phase interleaving. Simulating at a coarser granularity may lead to incorrect forecasts about the correctness of the agent program.

To demonstrate the portability of the MAS codebase to a real-world deployment, we execute the UAV control MAS as a regular process where all agents run concurrently as threads. This is, of course, a simplified approximation of a deployment on real UAVs, but it suffices to show that the same codebase can be executed without modification in both simulation and real-world settings.

We ensured code portability by enforcing that all interactions with the environment are done through the JaKtA environment API, which we implemented both for the real-world deployment and for the Alchemist simulator.

6.3.1 Simulated system deployment on the Alchemist simulator

We use the simulator to compare the execution of the same MAS logic with different granularities, namely, we consider AMA, ACLI, and ACLP (cf. Section 4.3). We expect that

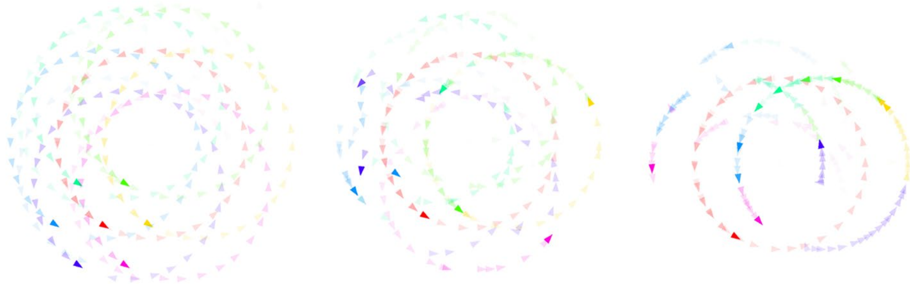

Fig. 5 Simulation snapshots for the three cases: AMA (left), ACLI (centre), and ACLP right for $\tau = 0.6$. Every UAV is depicted with a different colour, the leader is the red one. The follower UAV count has been reduced to six for better visualisation. Color intensity captures the distance in time, more intense colors are closer to the current time. In AMA, all UAVs follow a circular trajectory intersecting the leader's one. The formation is much less regular in ACLI, and it is completely lost in ACLP, showing that failing to capture the model nuances correctly may produce systems that work *because* of the abstraction gap

coarser-grained granularities show better system performance than more finer-grained (in our case, better formation maintenance). The experiment is designed to show that such effect is a consequence of the over-simplification of the BDI program and of its execution using coarse-grained simulation.

Modeling UAVs UAVs are modelled as Alchemist `Nodes`. Each `Node` is mapped to a JaKtA platform instance that hosts a single JaKtA agent controlling the UAV as described in Section 5.2.

Nodes can move in a 2D continuous space using the Alchemist's built-in `MoveToTarget` action. Agents compute the desired destination based on the perceived positions of the other UAVs and the leader, and write it to a dedicated `Molecule` in the environment, which is then read by the `MoveToTarget` action during its execution which applies the movement considering a speed parameter. This mechanism is used to make movement realistic and not instantaneous in the simulation, as UAVs can only move at a maximum speed of $1\frac{m}{s}$.

A pictorial representation of the experiment in form of simulation snapshots is shown in Fig. 5. For the sake of conciseness, we do not provide the complete MAS code here, but we highlight in Fig. 4 the platform-agnostic specification and how it is bound with the underlying execution platform. We refer the interested reader to the companion artefact [63] available online[6] for the complete codebase of the experiment.

Configuring the agent execution and simulation parameters We investigate how the mapping granularity (cf. Section 4.3) impacts the system's behaviour. To do so, we use AMA as baseline, letting the entire MAS run a full cycle every simulated second (Dirac Comb distribution with frequency $f = 1Hz \rightarrow T = 1s$, cf. Section 5.2), thus replicating the behaviour of most current agent-based simulation frameworks, which are time-driven. We compare the baseline with ACLI and ACLP, for which, however, we model each agent's control-loop frequency following a Weibull distribution with mean f and deviation $f \cdot \tau$ (drift). Intuitively,

[6] https://github.com/anitvam/uav-circle-2024-jakta-alchemist

this means that most loops will be scheduled around the mean frequency f, but some loops will be scheduled earlier or later, thus introducing a drift τ in the agents' execution.

Additionally, for the ACLP granularity we model deliberation and action delays, associating each phase with an exponential distribution with rate $\lambda = f$: faster agents (larger f values) have less delay. This is needed to emulate a real-world concurrent deployment, in which different phases of the control loop may take different time to complete and may interleave.

Every experiment is repeated 100 times with a different random seed, changing the initial positions of the followers and the distribution in time of the events for the ACLI and ACLP. In each experiment, the leader follows a circular trajectory of radius $r_l = 5m$ and is set to complete a full circle in 600s. We let the system execute for 1500 simulated seconds.

6.3.2 Real-world system deployment as a concurrent system

To demonstrate the seamless execution of the same MAS specification in both simulated and real systems, we run the JaKtA MAS natively on a stand-alone desktop computer. The execution model in JaKtA runs each agent in a separate thread, thereby emulating a concurrent deployment on independent UAVs.

In this setup, UAVs run at the maximum allowed frequency f, determined by the host machine's CPU capabilities. Any relative drift τ between different UAVs is naturally introduced by the operating system's scheduler. The duration of each phase of the control loop depends on the actual computation time of the agent program, which in turn is determined by the host machine's CPU performance.

Agents interact with a local instance of the environment implemented as a specialization of the generic JaKtA `Environment` (cf. Section 5.1). The environment is responsible for emulating the UAVs' physical aspects, such as position and communication.

Concerning communication between UAVs, agents exchange messages via the environment: each agent invokes the broadcast primitive provided by the environment, which delivers the message to all agents whose current positions lie within the sender's communication radius $r_c = 5m$.

Concerning positioning, the environment stores UAVs positions as two-dimensional coordinates. When an agent attempts perceiving its own position, the environment returns the stored coordinates. When an agent attempts to move to a target destination, it sends a request to the environment, which in turns stores that desired destination for that agent. The actual movement is completed asynchronously via a co-routine which runs in background. This co-routine would periodically (as frequently as possible) update the positions of each UAV along the straight line toward its target destination, by a distance non-greater than $1m$ (so that $v_{\max} = 1\frac{m}{s}$).

This setup demonstrates the portability of the MAS codebase to a non-simulated deployment: the same code used in simulation runs unchanged here. The experiment[6] shows the identical codebase executed in both simulated and real-world settings. The experiment was executed on a machine with an Intel i7-10750H CPU (2.60 GHz) and 32 GB RAM.

6.4 Results

In this section we present and discuss the results of our experiments, focusing on the impact of the simulation granularity on the system behaviour. We discuss the results obtained when simulating the system first, then we present the results obtained from the concurrent system execution.

We evaluate the system performance by measuring the deviation from the ideal positions. More precisely, we use an oracle to determine the ideal position of each follower based on the current leader's position, then, for each follower u, we measure the distance error d_u between the ideal and actual position of u. Our error metric is the overall squared distance error: $\sum_{u \in U} d_u^2$.

Results confirm the expectations that when simulating coarse-grained granularity levels (namely, AMA and ACLI) the agents are able to maintain the formation, while the fine-grained ACLP granularity level reveals that this result is an artefact of the over-simplification of the BDI programs. On a real UAV-programming workflow, this would have revealed a flaw in the agent program, or highlighted the need for higher responsiveness (reasoning cycle frequency) to react to the dynamics of the environment.

6.4.1 Analysis of the Alchemist simulated system deployment

Figure 6 shows the evolution in time results for $\tau \in \{0, 0.5, 0.7\}$. With AMA granularity, after a transient phase in which the leader is still contacting the followers, the error stabilises to a very low level, due to the natural delay between the commands issued by the agent execution and their realisation by the UAV. When $\tau = 0$, ACLI shows the same behaviour for both $f = 1Hz$ and $f = 2Hz$: the system can apparently cope with the problem at hand. As expected, given that the simplistic agent logic we implemented relies on implicit synchronisation, when using the most fine-grained model (ACLP), which is capable of modelling delays between one agent phase and another, the simulation reveals that the error compared to the ideal positions is much larger. Indeed, the system seems to be able to cope with the problem at hand only when the agent runs at $2Hz$: we have thus evidence that there are cases in which capturing a finer grained model can provide insights on the system's behaviour that are not visible with a coarser granularity, potentially imposing additional design or deployment constraints (in this case, we must execute at at least $2Hz$ if we want the system to be robust to delays).

This result is important: since the UAVs in the target deployment would run independently, the ACLP granularity is the most realistic one. If the developers had simulated the system with ACLI (or, worse, AMA) granularity, the system would have failed when deployed, despite positive feedback from the simulation.

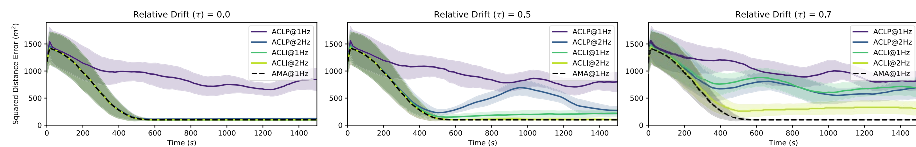

Fig. 6 Error with time for different granularities, with ACLI and ACLP running at $f = 1Hz$ and $f = 2Hz$. Different charts show different values of relative drift τ. Coloured shadows represent $\pm 1\sigma$

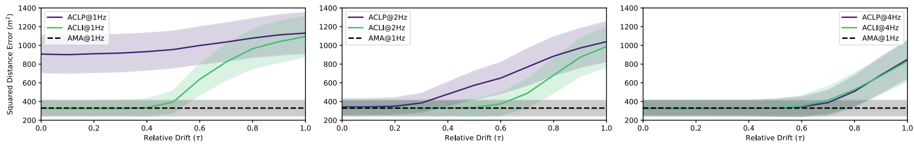

Fig. 7 Mean squared distance error with relative drift (τ), measured for different frequencies for ACLI and ACLP. Coloured shadows represent $\pm 1\sigma$

If we introduce relative drift in the agent's execution frequency – thus improving the realism of the simulation –, we can observe errors even with the ACLI model. With $\tau = 0.5$, both ACLI and ACLP show that the system can work reasonably well, but only with $f = 2Hz$. Raising τ further shows that capturing the agent phases is relevant, as the performance of ACLI and ACLP differs also for $f = 2Hz$.

The impact of τ on the system is better evidenced in Fig. 7, where we sample multiple values of τ for different *f*. We demonstrate that the ACLI model provides unreliable results at lower frequencies, while it is comparable to ACLP as the frequency increases. When the system is challenged, and the frequency is barely insufficient to cope with the problem, it tends to provide overly optimistic results, thus making the system appear as working as intended when, with better detail, we can observe that it is not. Designers can use this information to decide whether their software needs to be amended, or they may clearly state that it is required for the target UAV to be capable to run the agent software at a minimum frequency.

6.4.2 Analysis of the real-world concurrent system deployment

The execution of the same MAS logic on a concurrent system produced results comparable to those obtained in simulation at the ACLP granularity, demonstrating that simulation granularity affects the system's evaluation. The error metric for the concurrent-system execution was measured once per second, and the outcome is shown in Fig. 8.

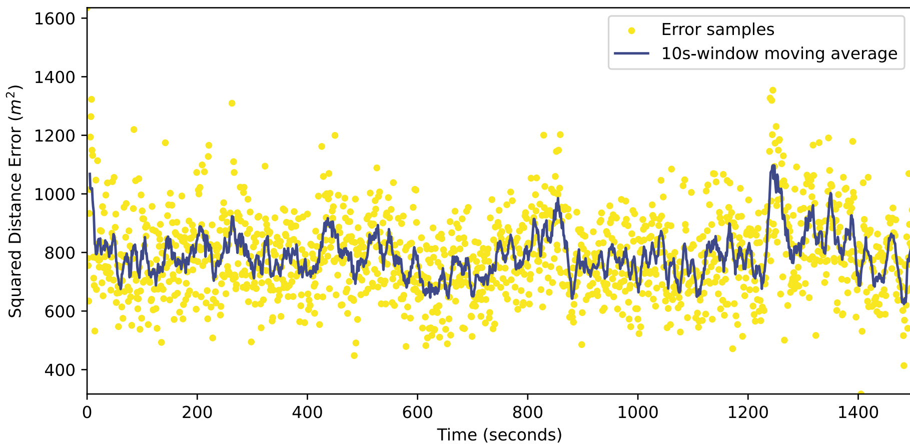

Fig. 8 Error over time for the execution of the BDI MAS on a concurrent system. Yellow line represents the raw error measurements, gathered once every second. Blue line is the rolling average over a window of 10*s*

In line with expectations, the error follows the ACLP simulation results, confirming that coarser-grained event granularities in simulation may obscure subtle concurrency-dependent effects. This is confirmed by results shown in Fig. 8, where the error does not stabilise or decrease towards zero, which is instead the behaviour we would expect from a correct system.

6.4.3 Discussion

With this experiment, we demonstrated that the execution of a BDI MAS in a DES is *feasible* and can be realised at different granularity. We also show that the BDI code can be ported *transparently* into the simulation environment, as it does not directly refer any simulator- or platform-specific entity. Figure 4 shows that the code of BDI specification (left) is completely agnostic of the final destination of the agent and programmed using pure JaKtA syntax, that can run both in a simulation and in a real-world scenario, provided that the necessary interactions with the environment (e.g., the action `circleMovementStep`) are appropriately modelled in both settings. Finally, we show that the impact of the chosen granularity on the *reality gap* is a relevant issue and that executing a MAS in modes that induce *implicit synchronisation* (AMA, and, to a lesser extent, ACLI) can produce unreliable results, especially concerning the system's performance.

The example highlights how simulating the MAS behaviour before deployment can be a valuable tool for software engineers, allowing to identify potential issues. In this case, the simulation provides insights on the relative frequency agents need to run to cope with the problem. Having observed that the system may fail under certain conditions, the developers of the agent specification may decide to introduce additional mechanisms to improve the system's reliability.

7 Threats to validity

Here we briefly comment on a few critical aspects of our study design, which may appear as threats to the general validity of the proposed approach. We motivate design choices, and discuss how specific threats have been mitigated, or why they have been accepted for the current work with the plan of addressing them in future research.

Specificity of technological choices Our prototype is tailored on JaKtA and Alchemist, which are technologies which have been designed and developed by some authors of this paper. While this can be seen as a threat to the generalization of our results, we argue that addressing our research questions is also a technological task, which requires technological commitments. While we try to keep the conceptual discussion about the mapping (Section 4) as general as possible, we preferred to rely on technologies we know well when implementing the mapping.

Furthermore, the technological affinities between JaKtA and Alchemist, were key enablers for the development of this research line, and this is yet another reason why we chose them.

Comparisons with other methods We deliberately avoid comparisons with other approaches for BDI simulation. The reason behind that is that, as discussed in Section 3, we are not aware of any prior attempt to run BDI MASs on DESs with the same level of granularity we propose and comparisons with other approaches would be unfair. As the existing frameworks for BDI simulation are mainly based on DTS, we detailed conceptual differences with DTS frameworks. Yet, we could not see the benefits of benchmarking our approach against other approaches based on DTS. In particular, we expect DTS would serve as a good engineering tool too, despite being less adequate to study the effect of simulation granularity on the BDI engineering process.

Performance analysis We do not study our solution from a computational complexity perspective, and we do not investigate the performance of the simulation.

At the conceptual level, our goal is to study the impact of simulations in the validation of BDI systems, and the role of DES in this process. We take an angle which is rooted in software engineering, so we are more interested in studying what BDI engineers can do with simulations, rather than how fast the simulations run. That said, in our framework, the computational cost of simulations is straightforward to estimate qualitatively: fine-grained granularity levels will be more computationally expensive than coarse-grained ones, as they require more events to be simulated.

Other practical considerations about performance are actually inherited to the technological choices we made. JaKtA is a lightweight BDI framework, still in its infancy, but under active development, and performance analysis are part of the roadmap. Alchemist is a mature DES framework, which has been used in the past to simulate large-scale systems, as proved by the many publications that rely on it for their experiments. So, in this work, performance is not a concern, and not even a controllable variable.

Effort required to design and run a testing scenario The byproduct of our feasibility study is a prototype tool that may be interesting for the community of BDI developers. Despite the usability of the tool is not the focus of this paper, we understand that it might be interesting to evaluate it to understand how easy it is to run a BDI simulation via JaKtA+Alchemist. We consider refining and evaluating the usability of the tool in future works, but we can already anticipate that the effort required to design and run a testing scenario using our approach involves: *(i)* writing agent and, optionally, action specifications, in JaKtA; *(ii)* coding the simulated environment, in Kotlin; *(iii)* configuring simulations using Alchemist, via YAML; *(iv)* run the simulations and analyse simulation traces as text files.

Step *(i)* involves writing Kotlin code using JaKtA's API (as JaKtA technically consists of an internal DSL for Kotlin [62]). Its difficulty depends on the scenario at hand, and on the goal of the programmers. In principle, this step is analogous to writing BDI agents in any other BDI framework, e.g. Jason [20]. Also, this step would be required in any case, even if the MAS is going to be deployed directly with no simulation.

Step *(ii)* is required if the BDI system is going to be tested in a simulated environment. This may require additional design and coding effort, if designers want their simulation to be high-fidelity.

Step *(iii)* is required to configure the simulation. This is as simple as creating a YAML file where the parameters of the simulation are set.

Finally, step *(iv)* is required to analyse the results of the simulation, and this may require additional custom processing, depending on the goals of the users.

Simplified scenario / Scalability The UAV scenario in Section 6 has not been tested on real UAVs. The experiment is not meant to demonstrate the applicability of BDI systems to real-world UAV coordination problems, which may require additional considerations beyond the scope of this work. However, it does demonstrate how simulation positively impacts BDI system validation and how execution granularity affects fidelity.

Additionally, the demonstration of execution of the same MAS codebase as a concurrent system on the host machine through the JaKtA built-in threading model sufficiently shows the portability of the codebase to real-world deployments. Indeed, most of the technical effort in this work has been to make JaKtA agents run in a simulation environment *too*, as JaKtA is designed for programming BDI agents to be deployed in the real world as multi-threaded applications.

The choice of the UAVs scenario was driven by multiple factors. We decided to focus on an example in which the BDI system is complex enough to exhibit different dynamics at different event granularities so that the reader could qualitatively appreciate the impact of the granularity on the system's behaviour. Moreover, we wanted to emulate a real scenario where the effort of validating the MAS via simulation is justified by the cost of deploying the real system, and the UAV scenario perfectly fits this purpose.

Regarding scalability, the chosen scenario involves a limited number of agents (17 in total: 1 leader and 16 followers) to keep the simulation manageable and interpretable for demonstration purposes. This should not be interpreted as a limitation to the scalability of our approach: as we rely on the Alchemist simulator, our approach may easily scale to large-scale systems composed by thousands of agents (or more).

8 Conclusion and future work

In this paper, we analysed the role of simulation in the engineering of distributed BDI MASs, discussing the importance of simulating the same agent specification that will be deployed in the real world, and showing the practical implications of mapping BDI agents on DESs at different granularity. We substantiate our claims by producing an open-source prototype that maps a BDI framework onto a DES simulator, using which we can execute an agent specification in a simulated environment with no changes to the agents' code. We leverage the prototype to show the impact of the granularity on the simulation's reality gap.

The paper shares two main insights for software engineers designing MASs.

1. Simulation is a crucial tool in the development of BDI MASs, and BDI framework designers should consider it upfront. Designing a BDI engine that does not provide appropriate APIs to select fine-enough granularities, in fact, may prevent the system's behaviour and performance from being correctly assessed ahead of deployment.

2. The MAS under development must be validated using the same code used that will be deployed, reducing the time required to test the system and easing software maintenance through simulation-based regression testing.

8.1 Achievement of the work objectives

We here comment regarding the sub-objectives listed in Section 1.

O1: Investigate the practical trade-offs of the methods that have been proposed to develop and test BDI agents in simulation At the design level, two simulation paradigms can be used to develop and test BDI agents: *time-driven* simulation and *event-driven* simulation, the latter commonly implemented as DES. Time-driven simulation advances in fixed time steps, updating all agents at each step, which simplifies implementation and parallel execution (at the price of reducing reproducibility) but can introduce artificial synchronisation and unnecessary computations when no relevant events occur. In contrast, event-driven simulation advances only when meaningful events occur, allowing finer-grained control of agent execution and reducing computational overhead. DES is particularly well-suited for BDI agents, as their reasoning cycle is inherently event-driven, making it a natural fit for mapping the agent's execution flow into a simulation model. However, this approach introduces trade-offs: while DES improves fidelity by accurately modelling asynchronous agent interactions, it can be more complex to implement and requires careful event scheduling to ensure reproducibility and scalability. Our work demonstrates that DES provides an effective balance between implementation complexity and simulation accuracy, making it a promising choice for testing BDI systems before deployment.

At the technical level, simulating BDI agents has been supported in the literature by adding simulation-related features to existing BDI interpreters, by adding BDI-related features to existing simulation platforms, or by synchronising two separate tools (one for BDI and one for simulation). Seeking for some solution which does not require to rewrite the agent specification upon switching from simulation to real-world deployment (or vice versa) while retaining control over the fidelity of the simulation, we identified another viable approach where a new sort of BDI interpreter is natively designed to allow for both in-silico and real-world execution —and this is the approach proposed by our work.

O2: Understand which abstractions a BDI framework should provide to integrate seamlessly with a simulation engine, such that the same specification can be executed in both simulation and real-world deployment A BDI framework must support modular execution models, abstracting platform-specific functionalities and encapsulating them into a deployment-agnostic API, that allows external scheduling of agents' reasoning cycles. The framework should maintain a clean separation between agent logic and execution infrastructure, allowing the same specification to run both in simulation and in the real-world target environment without modifications. Implementations should interpret the same agent logic differently. Core to the *simulated* deployment is the ability to execute BDI agents deterministically, to ensure reproducibility. The same is not true in *real-world deployments* where concurrency and asynchrony are inherent, and sometimes even desirable and exploitable. Therefore, the

deployment-agnostic API should *virtualise* functionalities covering all potential sources of non-determinism, such as random number generation and external events.

O3: Analyse how different mappings between BDI agents' execution model and DES can be designed, and how they impact the simulation fidelity BDI agent execution can be mapped onto DES by treating key stages of the reasoning cycle as discrete events, scheduled within the simulator. We identify four mapping granularities, each with distinct implications:

- **AMA** (Section 4.3.1): The most coarse-grained approach, synchronising all agents at each DES event, simplifying implementation but discarding interleaving effects.
- **ACLI** (Section 4.3.2): A finer-grained model where each agent executes its full control loop independently, allowing action interleaving but not asynchronous phase execution.
- **ACLP** (Section 4.3.3): Each atomic phase (sense, deliberate, act) is a distinct DES event, preserving concurrency and accurately modelling distributed agent behaviour.
- **ABE** (Section 4.3.4): The finest-grained approach, mapping each BDI event to a DES event, useful for debugging but behaviourally equivalent to ACLP.

Our findings demonstrate that BDI execution can be faithfully represented in DES, with granularity selection impacting fidelity, performance, and complexity. ACLP ensures the highest accuracy but at a computational cost, whereas AMA and ACLI simplify execution but risk oversimplifying agent interactions. The appropriate mapping depends on the testing objectives and the trade-offs between efficiency and behavioural fidelity.

O4: Integrate a BDI framework and a DES simulation engine in such a way that the same specification can be executed in simulation and in real-world deployment, proving feasibility Our integration of JaKtA and Alchemist demonstrates that a BDI framework can be embedded within a DES engine by treating agent execution as a series of simulation-driven events. By leveraging modularity in both technologies, we enable BDI agents to operate in a simulated environment without requiring code modifications. The environment is modeled within the simulator, preserving interactions with agents while ensuring portability. The proposed approach ensures that the same BDI specification can be tested in simulation and seamlessly deployed in the real-world environment (e.g., on distributed computing devices) reducing development overhead and minimizing the reality gap.

8.2 Future research directions

This work opens multiple research directions that we plan to investigate in the future.

Debugging and automated testing Complex software systems are better inspected if the abstractions used to design a system are exposed first-class during inspection, and if the inspection tool supports the injection of dynamic events from the environment. Unfortunately, conventional debug tools fall short in both aspects, as they expose low-level abstractions when debugging BDI programs, they allow solely for the execution of the program in the local device (often not representative of the real-world environment), and do not capture/inject external events easily. Simulation can solve the dynamicity issue by emulating multiple interacting devices at once: it can thus represent the basis towards a debugging

tool for (distributed) BDI MASs. However, the first element remains open to investigation, raising multiple questions: *(i)* how do breakpoint work when the BDI system is being simulated? *(ii)* How does such behaviour is compared to stand-alone execution? *(iii)* Which abstractions should be exposed by the debugger? Moreover, since simulation can be a valid option for testing BDI MAS, we believe it is useful to integrate simulation in automated testing pipelines, to verify MAS compliance with the expected outcomes. Along this line, the very concept of "successful simulation/test" deserves further investigation, especially in the context of stochastic systems.

Static checking of the simulated environment For the simulation to run successfully, the environment must provide the necessary API for the agents to run. If some of them is missing the error will be raised at runtime, potentially even after many successful runs due to the stochastic nature of the simulation. This failure can be prevented with statically verifying the simulation setup, making sure that all the actions required by the BDI MAS are available. We plan to explore how to implement these checks via static analysis or through the type system, to prevent failures at runtime which could make the verification process slower and less reliable.

Adaptive fidelity in simulation-based testing Investigating techniques to dynamically adjust the granularity of DES execution could improve efficiency without compromising fidelity. Adaptive models could vary the simulation resolution based on runtime conditions, prioritising high fidelity only in critical interaction phases.

Hybrid simulation Exploring hybrid approaches that combine simulated and real-world execution could provide a smoother transition from testing to deployment. Methods such as Hardware in the Loop (HiL) simulation [68] may help assess the impact of physical constraints on BDI agent execution, especially when the deployment targets are robots.

Standardisation and Interoperability Developing standardised interfaces for integrating BDI frameworks with simulation engines could enhance interoperability and facilitate adoption in different application domains. Defining common APIs and data exchange formats would improve reusability across tools.

Practical application to robot fleets The construction of a demonstrator with the proposed approach applied to real-world small-scale UAVs or Unmanned Ground Vehicles (UGVs) fleets could validate the practical applicability of the approach. However, this requires addressing challenges related to porting the prototype to low-power devices, network communication, and real-time constraints. Although we have a preliminary demonstrator with UGVs [69], the current prototype needs further refinement to be effectively deployed on real-world robot fleets.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s10458-026-09744-w.

Acknowledgements This work has been partially supported by: "WOOD4.0 - Woodworking Machines for Industry 4.0", (CUP E69J22007520009) Emilia-Romagna regional project, call 2022, art. 6 L.R. N. 14/2014; and by the projects funded by the European Commission under the NextGenerationEU programme: PNRR – M4C2 – Investment 1.3, Partenariato Esteso PE00000013 – "FAIR—Future Artificial Intelligence Research"

– Spoke 8 “Pervasive AI” and PNRR M4C2, Investment 3.3 (DM 352/2022) in collaboration with AUSL Romagna - CUP J33C22001400009 “Digital Twins Ecosystems for the clinical, strategic and process governance in Healthcare”.

Author Contributions M.B. contributed to conceptualization, implementation of the prototype, experimentation and paper writing S.B. contributed to conceptualization, related-work analysis and paper writing G.C. contributed to conceptualization and lead paper writing D.P. contributed to conceptualization, implementation of the prototype and paper writing.

Funding Open access funding provided by Alma Mater Studiorum - Università di Bologna within the CRUI-CARE Agreement.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Baxter, G. D., & Sommerville, I. (2011). Socio-technical systems: From design methods to systems engineering. *Interacting with Computers, 23*(1), 4–17. https://doi.org/10.1016/J.INTCOM.2010.07.003
2. Guan, X., Yang, B., Chen, C., Dai, W., & Wang, Y. (2016). A comprehensive overview of cyber-physical systems: from perspective of feedback system. *IEEE/CAA Journal of Automatica Sinica, 3*(1), 1–14. https://doi.org/10.1109/JAS.2016.7373757
3. Kourouthanassis, P. E., Giaglis, G. M., & Karaiskos, D. C. (2010). Delineating ‘pervasiveness’ in pervasive information systems: a taxonomical framework and design implications. *Journal of Information Technology, 25*(3), 273–287. https://doi.org/10.1057/JIT.2009.6
4. Xie, J., & Liu, C.-C. (2017). Multi-agent systems and their applications. *Journal of International Council on Electrical Engineering., 7*(1), 188–197. https://doi.org/10.1080/22348972.2017.1348890
5. Bandini, S., Manzoni, S., & Vizzari, G. (2009). Agent based modeling and simulation: An informatics perspective. *Journal of Artificial Societies and Social Simulation,12*(4).
6. Calegari, R., Ciatto, G., Mascardi, V., & Omicini, A. (2021). Logic-based technologies for multi-agent systems: a systematic literature review. *Autonomous Agents and Multi-Agent Systems, 35*(1), 1. https://doi.org/10.1007/S10458-020-09478-3
7. Uhrmacher, A. M. (2002). Simulation for agent-oriented software engineering, San Antonio, TX, USA. https://pdfs.semanticscholar.org/b022/be67e1b2ff4f1f482ce3e4352608101ebcd8.pdf. Accessed 28 Mar 2024
8. Rao, A. S., & Georgeff, M. P. (1995). BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multiagent Systems* (pp. 312–319).
9. Croatti, A., Montagna, S., Ricci, A., Gamberini, E., Albarello, V., & Agnoletti, V. (2019). BDI personal medical assistant agents: The case of trauma tracking and alerting. *Artificial Intelligence in Medicine, 96*, 187–197. https://doi.org/10.1016/J.ARTMED.2018.12.002
10. Oliveira Silvestre, I., Lima, B., Dias, P. H., Becker, L. B., Hübner, J. F., & Brito, M. (2023). UAV swarm control and coordination using jason BDI agents on top of ROS. In *PAAMS 2023, Proceedings* (vol. 13955, pp. 225–236). https://doi.org/10.1007/978-3-031-37616-0_19

11. Adam, C., & Gaudou, B. (2016). Bdi agents in social simulations: a survey. *The Knowledge Engineering Review, 31*(3), 207–238.
12. Nunes, I., Schardong, F., & Filho, A. E. S. (2017). Bdi2dos: An application using collaborating BDI agents to combat ddos attacks. *Journal of Network and Computer Applications, 84*, 14–24. https://doi.org/10.1016/J.JNCA.2017.01.035
13. Singh, D., Padgham, L., & Logan, B. (2016). Integrating BDI Agents with Agent-Based Simulation Platforms. *Autonomous Agents and Multi-Agent Systems., 30*(6), 1050–1071. https://doi.org/10.1007/s10458-016-9332-x
14. Kehoe, J. (2016). Robust reproducibility of agent based models. In *The European Simulation and Modelling Conference. Inderscience*.
15. Fujimoto, R., Bock, C., Chen, W., Page, E., & Panchal, J. H. (2017) Research Challenges in Modeling and Simulation for Engineering Complex Systems. Springer, Berlin. https://doi.org/10.1007/978-3-319-58544-4
16. Cardoso, R. C., & Ferrando, A. (2021). A review of agent-based programming for multi-agent systems. *Computers*, *10*(2). https://doi.org/10.3390/computers10020016
17. Baiardi, M., Burattini, S., Ciatto, G., & Pianini, D. (2023). Jakta: BDI agent-oriented programming in pure kotlin. In *EUMAS 2023, Proceedings* (vol. 14282, pp. 49–65). https://doi.org/10.1007/978-3-031-43264-4_4
18. Pianini, D., Montagna, S., & Viroli, M. (2013). Chemical-oriented simulation of computational systems with ALCHEMIST. *Journal of Simulation, 7*(3), 202–215. https://doi.org/10.1057/jos.2012.27
19. Bratman, M., et al. (1987). *Intention, Plans, and Practical Reason* (Vol. 10). Cambridge, MA: Harvard University Press.
20. Bordini, R. H., Hübner, J. F., & Wooldridge, M. (2007). *Programming Multi-agent Systems in AgentSpeak Using Jason*. Oxford, England: John Wiley & Sons.
21. Hübner, J. F., & Bordini, R. H. (2009). Agent-based simulation using BDI programming in jason. In *Multi-Agent Systems - Simulation and Applications* (pp. 451–476). https://doi.org/10.1201/9781420070248.CH15
22. Ingrand, F. F., Georgeff, M. P., & Rao, A. S. (1992). An architecture for real-time reasoning and system control. *IEEE Expert, 7*(6), 34–44. https://doi.org/10.1109/64.180407
23. d'Inverno, M., Luck, M., Georgeff, M. P., Kinny, D., & Wooldridge, M. J. (2004). The dmars architecture: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems, 9*(1–2), 5–53. https://doi.org/10.1023/B:AGNT.0000019688.11109.19
24. Baiardi, M., Burattini, S., Ciatto, G., Pianini, D., Ricci, A., & Omicini, A. (2024). On the external concurrency of current BDI frameworks for MAS. In D. Briola, R.C. Cardoso, B. Logan (Eds.), *Engineering Multi-Agent Systems - 12th International Workshop, EMAS 2024, Revised Selected Papers. Lecture Notes in Computer Science* (vol. 15152, pp. 42–63). Springer, Auckland, New Zealand. https://doi.org/10.1007/978-3-031-71152-7_3.
25. Moon, I., & Hong, J. (2013) Theoretic interplay between abstraction, resolution, and fidelity in model information. In *Winter Simulations Conference: Simulation Making Decisions in a Complex World, WSC 2013* (pp. 1283–1291). IEEE, Washington, DC, USA. https://doi.org/10.1109/WSC.2013.6721515
26. Jakobi, N., Husbands, P., & Harvey, I. (1995). Noise and the reality gap: The use of simulation in evolutionary robotics. In F. Morán, A. Moreno, J. J. M. Guervós, & P. Chacón (Eds.), *Advances in Artificial Life, Third European Conference on Artificial Life. Lecture Notes in Computer Science* (vol. 929, pp. 704–720). Springer, Granada, Spain. https://doi.org/10.1007/3-540-59496-5_337.
27. Cai, X. (2007). Exact stochastic simulation of coupled chemical reactions with delays. *The Journal of Chemical Physics*, *126*(12). https://doi.org/10.1063/1.2710253
28. Richardson, D. (2000). Direct large-scale n-body simulations of planetesimal dynamics. *Icarus, 143*(1), 45–59. https://doi.org/10.1006/icar.1999.6243
29. Drogoul, A., Vanbergue, D., & Meurisse, T. (2002). Multi-agent based simulation: Where are the agents? In *MABS 2002* (vol. 2581, pp. 1–15). https://doi.org/10.1007/3-540-36483-8_1
30. Sklar, E. (2007). Netlogo, a multi-agent simulation environment. *Artificial Life, 13*(3), 303–311. https://doi.org/10.1162/ARTL.2007.13.3.303
31. Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., & Balan, G. C. (2005). MASON: A multiagent simulation environment. *Simulation, 81*(7), 517–527. https://doi.org/10.1177/0037549705058073
32. North, M. J., Collier, N. T., Ozik, J., Tatara, E. R., Macal, C. M., Bragen, M. J., & Sydelko, P. (2013). Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling, 1*, 3. https://doi.org/10.1186/2194-3206-1-3
33. Taillandier, P., Gaudou, B., Grignard, A., Huynh, Q.-N., Marilleau, N., Caillou, P., Philippon, D., & Drogoul, A. (2018). Building, composing and experimenting complex spatial models with the gama platform. *GeoInformatica, 23*(2), 299–322. https://doi.org/10.1007/s10707-018-00339-6

34. Drogoul, A., Michel, F., & Ferber, J. (2009). Multi-agent systems and simulation: A survey from the agent community's perspective. In A.M. Uhrmacher, & D. Weyns (Eds.), *Multi-Agent Systems - Simulation and Applications. Computational Analysis, Synthesis, and Design of Dynamic Systems* (pp. 3–51). CRC Press / Taylor & Francis, Boca Raton, FL, USA. https://doi.org/10.1201/9781420070248.PT1
35. Edmonds, B. (2000). The use of models - making MABS more informative. In S. Moss, & P. Davidsson (Eds.), *Multi-Agent-Based Simulation, Second International Workshop, MABS 2000, Revised and Additional Papers. Lecture Notes in Computer Science* (vol. 1979, pp. 15–32). Springer, Boston, MA, USA. https://doi.org/10.1007/3-540-44561-7_2
36. Fortino, G., & North, M. J. (2013). Simulation-based development and validation of multi-agent systems: AOSE and ABMS approaches. *Journal of Simulation, 7*(3), 137–143. https://doi.org/10.1057/JOS.2013.12
37. Fortino, G., Garro, A., & Russo, W. (2005). An integrated approach for the development and validation of multi-agent systems. *Computer Systems Science and Engineering*, *20*(4).
38. Miles, S., Winikoff, M., Cranefield, S., Nguyen, C. D., Perini, A., Tonella, P., Harman, M., & Luck, M. (2010). Why testing autonomous agents is hard and what can be done about it. In: AOSE Technical Forum. https://www.academia.edu/download/69061967/miles.pdf. Accessed 28 Mar 2024
39. Nguyen, C. D., Perini, A., Bernon, C., Pavón, J., & Thangarajah, J. (2011). Testing in Multi-Agent Systems. In *Agent-Oriented Software Engineering X* (pp. 180–190). Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-19208-1_13
40. Cossentino, M., Fortino, G., Garro, A., Mascillaro, S., & Russo, W. (2008). PASSIM: a simulation-based process for the development of multi-agent systems. *International Journal of Agent-Oriented Software Engineering.,2*(2), 132–170. https://doi.org/10.1504/IJAOSE.2008.017313. Publisher: Inderscience Publishers
41. Baiardi, M. (2024). Validation of BDI mass via simulation. In *28th International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2024, Urbino, Italy, October 7–9, 2024* (pp. 128–129). IEEE, Urbino, Italy. https://doi.org/10.1109/DS-RT62209.2024.00029
42. Briola, D., Vizzari, G., & Montinaro, M. (2025). In I. Lynce, N. Murano, M. Vallati, S. Villata, F. Chesani, M. Milano, A. Omicini, & M. Dastani (Eds.), *Enhancing Testing of MASs with a Simulator of JADE* (pp. 3687–3694). IOS Press, Amsterdam, Netherlands. https://doi.org/10.3233/faia251247
43. Bellifemine, F., Caire, G., & Greenwood, D. (2007). Developing Multi-Agent Systems with JADE. Wiley, Chichester, UK. https://doi.org/10.1002/9780470058411
44. Baiardi, M., Burattini, S., Ciatto, G., Pianini, D., Omicini, A., & Ricci, A. (2024). Concurrency model of BDI programming frameworks: Why should we control it? In *Proceedings AAMAS 2024* (pp. 2147–2149). https://doi.org/10.5555/3635637.3663089
45. Ricci, A., Croatti, A., Bordini, R. H., Hübner, J. F., & Boissier, O. (2020). Exploiting Simulation for MAS Development and Execution - The JaCaMo-Sim Approach. In *Engineering Multi-Agent Systems - 8th International Workshop, EMAS 2020, Revised Selected Papers* (vol. 12589, pp. 42–60). Springer, Auckland, New Zealand. https://doi.org/10.1007/978-3-030-66534-0_3
46. O'Brien, P. D., & Nicol, R. C. (1998). Fipa–towards a standard for software agents. *BT Technology Journal, 16*, 51–59.
47. Sakellariou, I., Kefalas, P., & Stamatopoulou, I. (2008). Enhancing NetLogo to simulate BDI communicating agents. In J. Darzentas, G. A. Vouros, S. Vosinakis, & A. Arnellos (Eds.), *Artificial Intelligence: Theories, Models and Applications* (pp. 263–275). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-87881-0_24
48. Drogoul, A., Amouroux, E., Caillou, P., Gaudou, B., Grignard, A., Marilleau, N., Taillandier, P., Vavasseur, M., Vo, D., & Zucker, J. (2013). GAMA: A spatially explicit, multi-level, agent-based modeling and simulation platform. In Y. Demazeau, T. Ishida, J. M. Corchado, J. Bajo (Eds.), *Advances on Practical Applications of Agents and Multi-Agent Systems, 11th International Conference, PAAMS 2013* (vol. 7879, pp. 271–274). Springer, Salamanca, Spain. https://doi.org/10.1007/978-3-642-38073-0_25
49. Taillandier, P., Bourgais, M., Caillou, P., Adam, C., & Gaudou, B. (2016). A BDI agent architecture for the GAMA modeling and simulation platform. In L.G. Nardin, & L. Antunes (Eds.), *Multi-Agent Based Simulation XVII - International Workshop, MABS 2016, Revised Selected Papers. Lecture Notes in Computer Science* (vol. 10399, pp. 3–23). Springer, Singapore. https://doi.org/10.1007/978-3-319-67477-3_1
50. Uhrmacher, A. M., & Schattenberg, B. (1998). Agents in discrete event simulation. In *European Simulation Symposium-ESS* (vol. 98, pp. 129–136). Citeseer.
51. Davoust, A., Gavigan, P., Ruiz-Martin, C., Trabes, G., Esfandiari, B., Wainer, G., & James, J. (2020). An Architecture for Integrating BDI Agents with a Simulation Environment. In *Engineering Multi-Agent Systems, Cham* (pp. 67–84). https://doi.org/10.1007/978-3-030-51417-4_4
52. Caballero, A., Botía, J. A., & Gómez-Skarmeta, A. F. (2011). Using cognitive agents in social simulations. *Engineering Applications of Artificial Intelligence, 24*(7), 1098–1109. https://doi.org/10.1016/J.ENGAPPAI.2011.06.006

53. Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning, 8*(3–4), 279–292. https://doi.org/10.1007/bf00992698
54. Figueiredo Prudencio, R., Maximo, M. R. O. A., & Colombini, E. L. (2024). (2024) A survey on offline reinforcement learning: Taxonomy, review, and open problems. *IEEE Transactions on Neural Networks and Learning Systems., 35*(8), 10237–10257. https://doi.org/10.1109/tnnls.2023.3250269
55. Weyns, D., Omicini, A., & Odell, J. (2007). Environment as a first class abstraction in multiagent systems. *Auton. Agents Multi Agent System, 14*(1), 5–30. https://doi.org/10.1007/S10458-006-0012-0
56. Issicaba, D., Rosa, M. A., Prostejovsky, A. M., & Bindner, H. W. (2017). Experimental validation of BDI agents for distributed control of electric power grids. In *IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)* (pp. 1–6. IEEE). Torino, Italy. https://doi.org/10.1109/ISGTEUROPE.2017.8260273
57. Pantoja, C. E., Jr., M. F. S., Lazarin, N. M., & Sichman, J. S. (2016). ARGO: an extended jason architecture that facilitates embedded robotic agents programming. In M. Baldoni, J. P. Müller, I. Nunes, & R. Zalila-Wenkstern (Eds.), *EMAS 2016. Lecture Notes in Computer Science* (vol. 10093, pp. 136–155). Springer, Singapore. https://doi.org/10.1007/978-3-319-50983-9_8
58. Moro, D. D., Robol, M., Roveri, M., & Giorgini, P. (2022). Developing bdi-based robotic systems with ROS2. In F. Dignum, P. Mathieu, J. M. Corchado, & F. Prieta (Eds.), *PAAMS 2022, Proceedings. Lecture Notes in Computer Science* (vol. 13616, pp. 100–111). Springer, L'Aquila, Italy. https://doi.org/10.1007/978-3-031-18192-4_9
59. Parry, O., Kapfhammer, G. M., Hilton, M., & McMinn, P. (2022). A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology, 31*(1), 17–11774. https://doi.org/10.1145/3476105
60. Wooldridge, M. J. (2000). *Reasoning About Rational Agents*. Cambridge, MA: Intelligent robots and autonomous agents. MIT Press.
61. Brooks, R. A. (1991). Intelligence without reason. In J. Mylopoulos, R. Reiter (Eds.), *Proceedings of the 12th International Joint Conference on Artificial Intelligence* (pp. 569–595). Morgan Kaufmann, Sydney, Australia. http://ijcai.org/Proceedings/91-1/Papers/089.pdf
62. Baiardi, M., Burattini, S., Ciatto, G., & Pianini, D. (2025). Blending BDI agents with object-oriented and functional programming with JaKtA. *SN Computer Science*. https://doi.org/10.1007/s42979-024-03244-y
63. Baiardi, M., & Pianini, D. (2024). anitvam/uav-circle-2024-jakta-alchemist: 1.0.4. Zenodo. https://doi.org/10.5281/ZENODO.13921206
64. Zambonelli, F., Omicini, A., Anzengruber, B., Castelli, G., De Angelis, F. L., Di Marzo Serugendo, G., Dobson, S. A., Fernandez-Marquez, J. L., Ferscha, A., Mamei, M., Mariani, S., Molesini, A., Montagna, S., Nieminen, J., Pianini, D., Risoldi, M., Rosi, A., Stevenson, G., Viroli, M., & Ye, J. (2015). Developing pervasive multi-agent systems with nature-inspired coordination. *Pervasive and Mobile Computing, 17*, 236–252. https://doi.org/10.1016/j.pmcj.2014.12.002
65. Montagna, S., Pianini, D., & Viroli, M. (2012). A model for drosophila melanogaster development from a single cell to stripe pattern formation. In *Proceedings of ACM SAC 2012* (pp. 1406–1412). https://doi.org/10.1145/2245276.2231999
66. Pianini, D., Viroli, M., & Beal, J. (2015). Protelis: practical aggregate programming. In *Proceedings of ACM SAC* (pp. 1846–1853). https://doi.org/10.1145/2695664.2695913
67. Casadei, R., Viroli, M., Aguzzi, G., & Pianini, D. (2022). Scafi: A scala DSL and toolkit for aggregate programming. *SoftwareX., 20*, 101248. https://doi.org/10.1016/j.softx.2022.101248
68. Koos, S., Mouret, J., & Doncieux, S. (2013). The transferability approach: Crossing the reality gap in evolutionary robotics. *IEEE Trans. Evol. Comput., 17*(1), 122–145. https://doi.org/10.1109/TEVC.2012.2185849
69. Aguzzi, G., Bacchini, L., Baiardi, M., Casadei, R., Cortecchia, A., Domini, D., Farabegoli, N., Pianini, D., & Viroli, M. (2025). A demonstrator for self-organizing robot teams. In C. D. Giusto, & A. Ravara (Eds), *Coordination Models and Languages - 27th IFIP WG 6.1 International Conference, COORDINATION 2025, Held as Part of the 20th International Federated Conference on Distributed Computing Techniques, DisCoTec 2025, Lille, France, June 17–19, 2025, Proceedings. Lecture Notes in Computer Science* (vol. 15731, pp. 230–244). Springer, Lille, France. https://doi.org/10.1007/978-3-031-95589-1_12

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.