

Bridging the Gap: An Executable Simulator for the L-DINF Epistemic Logic of Cooperative Agents

Stefania Costantini^{1,†}, Lorenzo De Lauretis^{1,†} and Valentina Pitoni^{1,*,†}

¹Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Italy

Abstract

In this paper we present a modular Java simulator for the Logic of Inferable Actions (L-DINF), an epistemic framework for modelling cooperative agents subject to resource limitations. The formal definition of L-DINF provides the reference specification of the work, while the simulator is meant to act as its executable realization. Our goal is therefore not to revise the theory, but to show how its main constructs can be operationalized in software in a clear and modular way. We describe the architecture of the system, the mapping between the logical packages and the object-oriented implementation, and the engineering choices adopted to make the framework executable while preserving fidelity to the formal model. We then discuss two illustrative case studies showing that the simulator reproduces the expected qualitative behaviours of the framework, including preference-guided action selection, role-based executability, shared mental costs, group-level belief propagation, and cross-group collaboration. The result is a reproducible proof of concept that helps bridge the gap between formal semantics and executable multi-agent systems.

Keywords

Multi Agent Systems, Modal Logic, Epistemic Logic, Java Implementation

1. Introduction

The remarkable success of Multi-Agent Systems (MAS) in a wide variety of applications calls for the definition of predictable and rational behaviours, especially when autonomous entities must cooperate to achieve shared objectives [1]. Formal logics, and epistemic logics in particular, are well suited to this purpose, as they provide a precise language to specify agent properties and to track how their mental states evolve over time [2, 3].

L-DINF is one such framework: it models cooperative agents that operate under resource limitations, and combines explicit beliefs, background knowledge, mental and physical actions, costs, budgets, preferences, roles, action equivalence, and group-level interaction [4, 5]. These ingredients make the logic expressive enough to describe cooperative decision making in which agents do not merely select plans, but deliberate under epistemic, organizational, and resource constraints.

A formal theory alone, however, is not sufficient for practical experimentation. To study how the semantic components interact in concrete situations, one needs an operational model that can execute representative fragments of the logic. Along this line, we present a Java-based simulator for L-DINF. The simulator should be read as a concrete *semantic attachment* [6]: the formal semantics remains the reference specification, while the software provides an executable realization of the core constructs needed to run and inspect L-DINF scenarios.

A clarification about scope is important. The simulator is not a complete proof procedure for L-DINF. Since the satisfiability problem for the logic is computationally demanding, a full deductive engine would be unsuitable for an interactive simulation loop. Instead, the simulator implements a tractable operational cycle that preserves a transparent correspondence between the main semantic packages and their software counterparts. In particular, it supports explicit quantitative management of mental and

WOA 2026: the 27th Workshop From Objects to Agents, June 15–17, 2026, Salerno, Italy

*Corresponding author.

[†]These authors contributed equally.

✉ stefania.costantini@univaq.it (S. Costantini); lorenzo.delauritis@univaq.it (L. D. Lauretis); valentina.pitoni@univaq.it (V. Pitoni)

ORCID 0000-0002-5686-6124 (S. Costantini); 0000-0002-8796-3672 (L. D. Lauretis); 0000-0002-4245-4073 (V. Pitoni)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

physical budgets, preference-guided choice among equivalent actions, role-constrained executability, group-aware communication, and temporary lending of agents across groups.

The contribution of this paper is threefold. First, we take the formal definition of L-DINF as a fixed reference point and show how its main constructs can be mapped into software components without changing their intended meaning. Second, we design a modular Java simulator in which beliefs, background knowledge, budgets, costs, preferences, action equivalence, roles, and lending are represented explicitly and independently, in line with the package-based organization of the formal model. Third, we present two illustrative case studies showing that the simulator reproduces the qualitative behaviours predicted by the framework, namely preference-guided action selection, role-constrained execution, shared mental costs, intra-group belief propagation, and inter-group collaboration.

The paper is structured as follows. Section 2 positions the work with respect to logical agent languages, BDI platforms, JADE-based infrastructures, epistemic model checkers, DALI, and recent ASP, L-DINF integrations. Section 3 recalls only the L-DINF notions that are directly used by the implementation. Section 4 presents the architecture of the simulator and explains how the implementation realizes the formal packages in executable form. Section 5 discusses the case studies and their significance for validating the implementation. Section 6 clarifies the scope of the implementation and its relationship with proof procedures and agent platforms. Section 7 concludes and outlines future work.

2. Related Work

Research on epistemic and resource-bounded agents can be positioned along several complementary lines: formal logics for belief dynamics and cooperation, executable logic-based agent languages, BDI-oriented programming platforms, MAS middleware, epistemic verification tools, and recent hybrid approaches combining symbolic optimization with cognitive reasoning. The simulator presented in this paper is related to all these lines, but it occupies a different position in the design space. It is not intended as a general-purpose MAS platform, nor as a model checker or a full theorem prover for L-DINF. Rather, it provides an executable and inspectable realization of the main semantic packages of a specific epistemic logic for cooperative agents.

A first relevant line of work concerns formal accounts of belief dynamics, explicit knowledge, and resource-bounded reasoning. L-DINF originates from previous logical frameworks in which agents are not ideal reasoners, but entities equipped with limited inferential and operational resources. In this perspective, explicit beliefs, background knowledge, and action effects must be represented in a way that makes reasoning dependent on what the agent can actually derive or execute under given constraints [7, 8]. L-DINF extends this line by introducing a richer organization of cooperative agency, including mental and physical actions, costs, budgets, preferences, roles, action equivalence, group interaction, and lending mechanisms [9, 4, 5]. This formal background is essential for the present work because the simulator does not propose a new semantics. Instead, it takes the L-DINF package structure as the reference specification and makes selected parts of it executable.

A second related line is represented by executable logic-based agent languages, in particular DALI. DALI is an Active Logic Programming language designed for the executable specification of logical agents. It provides constructs for reactivity, proactivity, internal events, memory, and communication among agents [10, 11]. Later developments of DALI also explored web applications, cognitive robotics, and complex-event processing, showing that a Prolog-based agent language can support implemented multi-agent scenarios [12]. DALI is therefore an important precedent for executable logical agents. However, the aim of the present simulator is different. DALI provides a general agent programming framework, whereas our simulator provides a direct Java realization of the semantic components of L-DINF, including explicit budgets, action costs, action equivalence classes, preference-guided selection, group-aware communication, and temporary lending. In this sense, DALI is complementary to the present contribution and may represent a natural direction for future executable encodings of L-DINF agents, but it is not itself a direct implementation of the L-DINF package semantics considered here.

BDI-oriented programming platforms are also relevant when discussing executable cognitive agents.

Jason, for example, provides a mature implementation of AgentSpeak-style BDI programming and supports beliefs, goals, events, plans, and belief-base manipulation [13]. It is therefore close to the general objective of making cognitive agent models executable. However, Jason is based on a plan-oriented BDI execution cycle, while L-DINF is organized around epistemic and resource-sensitive semantic packages. Constructs such as shared mental costs, vector-valued physical budgets, action equivalence classes with preference-based selection, role-constrained executability, and inter-group lending are not native components of the Jason semantics. They could in principle be encoded, but doing so would require a substantial extension of the deliberation and resource-management mechanisms.

JADE is related in a different way and should not be characterized as a BDI platform. It is a FIPA-compliant Java middleware for the development of interoperable agent systems and communication infrastructures [14]. JADE is therefore useful for engineering distributed MAS applications, managing agents, and supporting communication protocols, but it does not provide a BDI semantics nor an epistemic-resource model of agency. Consequently, JADE and the present simulator address different layers of the MAS engineering problem. JADE focuses on infrastructure and interoperability, whereas our work focuses on the operationalization of logical constructs such as budgets, costs, preferences, roles, action equivalence, and lending. A future integration could exploit JADE as a communication infrastructure, but the L-DINF-specific reasoning cycle would still have to be provided by an additional semantic layer.

A further comparison concerns epistemic and temporal verification tools, such as MCK and MCMAS. These systems support the model checking of knowledge, temporal, and strategic properties of multi-agent systems [15, 16]. They are highly relevant when the objective is to verify whether a model satisfies certain epistemic or temporal specifications. The goal of our simulator is different. It does not primarily check formulas over a state space; rather, it executes scenarios in which agents consume resources, update beliefs, select among equivalent actions, communicate within groups, and temporarily reorganize cooperation through lending. For this reason, epistemic model checkers and the present simulator should be seen as complementary tools. A promising future development would be to export finite abstractions or execution traces from the simulator to model-checking environments, so that the operational behaviour of L-DINF scenarios could be independently verified.

Recent work on ASP and L-DINF provides another important point of comparison. Hybrid ASP-L-DINF approaches have been proposed in decision-support scenarios, especially in medical scheduling, where ASP is used to compute globally optimized solutions and L-DINF-style agents are used to represent local cognitive reasoning, preferences, intentions, and action feasibility [17]. This line is close in motivation because it also aims at connecting L-DINF with executable or computational settings. However, the focus is different. In hybrid ASP-L-DINF models, ASP typically provides an external optimization layer, while L-DINF enriches the solution with agent-level reasoning. The simulator presented here, by contrast, isolates the L-DINF semantic packages themselves and implements them as a standalone execution environment. The two approaches are therefore complementary: ASP can be used to generate optimized candidate decisions, whereas the simulator can be used to inspect how L-DINF agents enact, adapt, or explain such decisions under epistemic, organizational, and resource constraints.

On this basis, the novelty of the present work can be stated in a qualified form. We do not claim that executable logical agent systems, BDI platforms, MAS middleware, or epistemic verification tools are absent from the literature. Rather, to the best of our knowledge, this is the first standalone Java simulator that directly maps the main L-DINF semantic packages, beliefs, background knowledge, mental and physical budgets, costs, preferences, action equivalence, roles, group communication, and lending, to explicit software components and validates their interaction through executable traces. The contribution is therefore not a general executable agent language, but a direct and inspectable operationalization of the L-DINF package architecture.

3. L-DINF Background

This section recalls only the L-DINF notions that are used by the simulator. L-DINF is composed of a static component, L-INF, for explicit beliefs and background knowledge, and a dynamic component for the effects of mental and physical actions [4]. The simulator does not implement the full proof theory of the logic. It operationalizes the package structure needed to execute representative scenarios, while keeping the formal definitions as the reference specification.

3.1. Language Fragment Used by the Simulator

Let Atm be a countable set of atomic propositions, Atm_A a set of physical actions, Agt a set of agents, Grp a set of groups, and Res a set of resources. The implementation uses the following representative fragment of the L-DINF language:

$$\begin{aligned} \varphi, \psi ::= & p \mid \neg\varphi \mid \varphi \wedge \psi \mid \mathbf{B}_i\varphi \mid \mathbf{K}_i\varphi \mid \\ & do_G^P(\phi_A) \mid do_G(\phi_A) \mid can_do_G(\phi_A) \mid \\ & intend_G(\phi_A) \mid exec_G(\alpha) \mid pref_do_i(\phi_A, d) \mid \\ & pref_do_G(i, \phi_A) \mid [G : \alpha]\varphi \mid Cl(\phi_A, \phi'_A) \mid \\ & fCl_i(\phi_A, \phi'_A) \mid lend_G(i, H, \phi_A). \end{aligned}$$

Here $\mathbf{B}_i\varphi$ represents explicit belief, $\mathbf{K}_i\varphi$ background knowledge, $intend_G(\phi_A)$ a group intention, Cl and fCl action equivalence, and $lend_G(i, H, \phi_A)$ the possibility that group G temporarily obtains agent i from group H to perform action ϕ_A . The case studies use this fragment to model goals, rules, action costs, preferences, role-based executability, and inter-group collaboration.

3.2. Semantic Packages Relevant to Execution

The simulator follows the package-based organization of L-DINF. In the full semantics, an L-DINF model contains worlds, accessibility relations, neighbourhoods for explicit beliefs, and a family of functions describing executability, costs, budgets, preferences, equivalence, roles, and lending. For implementation purposes, the following packages are the relevant ones.

Definition 3.1 (Core execution packages). For each agent i , group G , world w , mental action α , and physical action ϕ_A , the executable part of the model uses the following functions:

- $E(i, w)$: mental actions executable by i at w ;
- $B_1(i, w)$ and $C_1(i, \alpha, w)$: mental budget and mental-action cost;
- $A(i, w)$: physical actions executable by i ;
- $B_2(i, w)$ and $C_2(i, \phi_A, w)$: physical-resource budget and physical-action cost;
- $H(i, w)$: actions enabled by the role of i in its current group;
- $P(i, w, \phi_A)$: preference value for performing a physical action;
- $Q(\phi_A, w)$: equivalence class of physical actions that can realize the same abstract action;
- $F(i, w, \phi_A)$: selector function that chooses one action from $Q(\phi_A, w)$ according to P and affordability;
- $L(i, H, G, \phi_A, w)$: lending condition authorizing group G to borrow i from group H for ϕ_A .

Intuitively, E and A specify what an agent can perform, H restricts performance according to the agent's role inside its group, B_1 and B_2 determine whether enough resources are available, and C_1 and C_2 determine how resources are consumed. The preference function P induces an ordering over actions, while Q groups together actions that are interchangeable with respect to an abstract goal. The selector F then chooses the best executable and affordable representative. The lending function L accounts for temporary cross-group collaboration when a group lacks a locally available action.

The implementation preserves these distinctions explicitly. For example, the fact that an agent can physically execute an action is not sufficient: the action must also be enabled by its role and

affordable with respect to the relevant budget. Similarly, two actions may be equivalent for achieving a goal, but the simulator chooses among them by using the preference order and, in case of ties, an implementation-level cost criterion.

3.3. Group Execution, Budgets, and Action Choice

A group can progress towards a physical goal when at least one of its members can perform an action that realizes the goal. In executable terms, a physical candidate for ϕ_A is admissible for agent i only if it belongs to $A(i, w) \cap H(i, w)$ and its cost does not exceed the available physical budget $B_2(i, w)$. When $Q(\phi_A, w)$ contains several equivalent actions, the simulator realizes F by selecting an affordable representative with maximal preference value.

Mental actions are handled as group-relevant operations. If a mental action α is performed for group G , then the formal enabling condition checks whether some agent in G can perform α and whether the shared cost can be covered by every member of the group:

$$enabled_w(G, \alpha) \quad \text{iff} \quad \exists j \in G \left(\alpha \in E(j, w) \wedge \frac{C_1(j, \alpha, w)}{|G|} \leq \min_{h \in G} B_1(h, w) \right).$$

A deterministic predicate $doer_w(i, G, \alpha)$ selects one enabled agent as the performer. The simulator implements this choice by processing agents in configuration order and by applying the same affordability discipline at runtime.

3.3.1. Budget Update for Mental Actions

If a mental action α is executed by group G , its cost is shared among the members of the group. Hence, for each $i \in G$:

$$B_1^{[G:\alpha]}(i, w) = B_1(i, w) - \frac{C_1(i, \alpha, w)}{|G|}.$$

For agents outside G , the mental budget is unchanged. This rule is central in the case studies because it makes collective reasoning visible as a resource-consuming operation.

3.3.2. Budget Update for Physical Actions

If agent i performs physical action ϕ_A at world w and the execution leads to w' , then the physical budget is updated by subtracting the consumed resource tuple:

$$B_2(i, w') = B_2(i, w) - C_2(i, \phi_A, w).$$

The subtraction is defined component-wise over the resources declared in Res . The simulator represents these tuples as maps from resource names to quantities.

4. Simulator Architecture and Implementation

This section describes the architecture and the core mechanisms of the L-DINF simulator. The implementation acts as a *semantic attachment* [6], connecting the declarative logic to a concrete execution environment. The full source code is publicly available on GitHub¹.

4.1. Simulator Architecture

The simulation engine manages the execution lifecycle and coordinates the software components that realize the L-DINF constructs. Each simulation scenario is specified through an external `.ldinf` configuration file, keeping scenario data separate from the execution logic and allowing the same codebase to be reused across different experiments.

The main Java classes mirror the theoretical constructs of the framework:

¹<https://github.com/lolli91/LDINF-simulator>

Formal construct	Intended role	Used in the simulator for
K_i	Background rules	Rule-based inference toward goals
B_i	Explicit beliefs	Working memory and achieved facts
B_1, C_1	Mental budget and cost	Shared cost of inference steps
B_2, C_2	Physical budget and cost	Resource consumption for physical actions
H	Role-based enablement	Filtering actions allowed for each agent
P, Q, F	Preferences, equivalence, selector	Choosing the best action among alternatives
L	Lending authorization	Temporary borrowing of agents across groups

Table 1
Essential L-DINF concepts used by the simulator.

- Agent represents an active entity ($i \in \text{Agt}$) equipped with its own memory, budgets, and decision-making cycle. Each agent stores a `groupId` and the size $|G|$ of its group, both of which are used for group-scoped communication and for sharing the cost of mental actions.
- AgentMemory maintains two separate stores: a long-term knowledge base (inference rules, action costs, equivalence declarations, and preferences, corresponding to K_i) and a short-term working memory (the set of explicit beliefs, corresponding to B_i). It also records the set of *enabled actions* $H(i, w)$, which are loaded from the agent’s configuration.
- Budget tracks both the mental budget (B_1) and the physical resources (B_2) of an agent. It exposes two methods: `canAfford`, which checks whether a given cost can be covered by the current budget, and `consume`, which deducts the cost after execution.
- Logical constructs are represented as Java Records implementing a sealed interface `Formula`. Six record types partition the formula space: `Atom` (a predicate with an argument list), `Implication` (a list of premise atoms and one conclusion atom), `ActionCost` (an action name, a mental cost, and a resource-amount map), `Preference` (an action name and an integer preference value), `Equivalence` (a pair of action names), and `InferenceCost` (a base mental cost for inference steps). Because the interface is sealed, the compiler guarantees exhaustive pattern matching, so that no formula type can be left unhandled.
- `CommunicationChannel` enables agents to broadcast and receive facts within their group. The channel is *group-aware*: a broadcast is delivered only to agents that belong to the same group as the sender. A dedicated method, `broadcastToGroup`, supports directed communication to a specific group during agent-lending operations.
- Group encapsulates group-level state: the list of member agents, the common goals (intend_G), a shared knowledge-base template, and the total budget. It also implements the *agent-lending* mechanism described in Section 4.6.

The source code is organized into two Java packages that reflect the core/packages split of the formal model. The `com.ldinf.sim.core` package contains the runtime components (`Agent`, `AgentMemory`, `Budget`, `CommunicationChannel`, `Group`, and `DirectLDinfLoader`), while the `com.ldinf.sim.model` package defines the declarative data types (`Formula` and its six record subtypes, together with the `LDinfParser` that translates textual directives into typed formula objects).

This organization also identifies the main customization points of the simulator. New reasoning policies can be introduced by replacing the action selector used by `reasonAndAct`; alternative cost-sharing policies can be localized in `Budget` and in the group-level inference update; additional formula types can be added by extending the sealed `Formula` hierarchy and the parser; and alternative lending policies can be implemented inside `Group` without changing the memory or budget representation. Thus, the modularity claimed by the implementation is not only architectural, but also experimental:

individual L-DINF packages can be enabled, restricted, or replaced while keeping the rest of the execution loop stable.

4.2. Scenario Configuration

Each simulation scenario is specified in a single declarative `.ldinf` file whose syntax closely mirrors the notation used in the L-DINF paper. The file is organized into nested `GROUP` and `AGENT` blocks, each containing directives that correspond directly to the formal constructs:

- K_i : Knowledge-base rules (implications), corresponding to $\mathbf{K}_i$.
- B_i : Working-memory facts for each agent, corresponding to $\mathbf{B}_i$.
- H_i : Enabled physical actions for each agent, corresponding to $H(i, w)$.
- $C1$: Equivalence declarations between actions, corresponding to $Q(\phi_A, w)$.
- $C1$: Base mental inference cost, corresponding to C_1 .
- $C2$: Physical action costs (resource tuples), corresponding to C_2 .
- P : Preference values for physical actions, corresponding to $P(i, w, \phi_A)$.
- G : Common group goals, corresponding to $intend_G(\phi_A)$.
- G_i : Personal goals for individual agents.
- $B1$: Mental budget (energy), corresponding to B_1 .
- $B2$: Physical budget (resource amounts), corresponding to B_2 .
- `LENDING`: Enables or disables the cross-group agent-lending mechanism for the enclosing group, as an executable realization of the inter-group collaboration envisaged by the framework.

The following compact fragment illustrates how the domain-level declarations remain close to the formal notation while being directly executable by the loader:

```

1 GROUP group_1 {
2   B1: 100
3   B2: food=10, utensils=4
4   G: done(prepare_banquet)
5   Ki: done(cook_main_dish), done(set_table) -> done(prepare_banquet)
6   C1: cook_main_dish = cook_steak | cook_pasta
7   C2: cook_steak food=4, utensils=1
8   C2: cook_pasta food=2, utensils=1
9   P: cook_steak = 10
10  P: cook_pasta = 5
11  AGENT agent_1 { Hi: cook_steak, cook_pasta, set_table }
12 }
```

Listing 1: Configuration fragment for equivalent actions, preferences, and resources.

The `DirectLDinfLoader` class parses the `.ldinf` file and directly instantiates the simulation objects (`Group`, `Agent`, `Budget`, `AgentMemory`) without any intermediate representation. Internally, the loader reads the file line by line and maintains a brace-depth counter to track the nesting of `GROUP` and `AGENT` blocks. Each directive is recognized by its prefix (e.g. `Ki:`, `C2:`, `P:`) and delegated to the `LDinfParser`, which converts the textual representation into the corresponding `Formula` record. Group-level directives populate a shared `AgentMemory` template; when an agent is created, it receives a copy of this template enriched with its own working-memory facts (B_i) and enabled actions (H_i).

At load time, the total mental budget B_1 declared for a group is divided equally among its $|G|$ agents, so that each agent starts with $B_1/|G|$ units of mental energy. Physical resources (B_2) are split in the same way. This division reflects the group-level budget semantics of the formal model while giving each agent an individual `Budget` object that can be updated independently during the simulation.

4.3. Mapping of Theoretical Packages to Implementation

Table 2 summarizes the correspondence between the formal packages of L-DINF (Def. 3.1) and their executable counterparts in the simulator. The logic remains the reference specification; the implementation provides one concrete realization. Whenever the simulator adopts a specific engineering solution, such a choice should be understood as an implementation-level refinement, not as a modification of the underlying theory.

Package	Paper	Java Implementation
Exec. (mental)	$E(i, w)$	KB rules in AgentMemory
Budget (mental)	$B_1(i, w)$	Budget.mentalBudget
Cost (mental)	$C_1(i, \alpha, w)$	AgentMemory.baseInferenceCost
Exec. (physical)	$A(i, w)$	ActionCost entries
Budget (physical)	$B_2(i, w)$	Budget.physicalResources
Cost (physical)	$C_2(i, \phi_A, w)$	ActionCost record
Roles	$H(i, w)$	Hi: directive, AgentMemory
Preferences	$P(i, w, \phi_A)$	AgentMemory.preferences
Equivalence	$Q(\phi_A, w)$	Equivalence records
Selector	$F(i, w, \phi_A)$	performBestActionFor
Lending	$L(i, H, G, \phi_A, w)$	Group class

Table 2

Mapping of L-DINF packages to Java classes.

4.4. Reasoning and Action Selection Mechanism

The `reasonAndAct` method implements the operational cycle through which each agent pursues its goals. The current version covers the executable fragment needed for the case studies: rule-based inference, affordability checks, preference-guided selection among equivalent physical actions, group-aware communication, and cross-group collaboration. This section should therefore be read as a description of how the formal framework is realized in software, rather than as an alternative specification of the framework itself.

The method proceeds through the following steps:

1. **Goal identification.** The agent selects an active goal—either a personal goal or a common group goal—that has not yet been achieved (i.e., the corresponding atom is not present in the agent’s working memory).
2. **Applicable rule search.** The agent scans its knowledge base for inference rules whose conclusion matches the selected goal.
3. **Premise analysis.** For each matching rule, the agent determines which premises are already satisfied (i.e., present in working memory) and which are missing. Each missing premise becomes a candidate sub-goal that the agent may try to achieve first.
4. **Candidate action generation.** For every action that could satisfy either the goal directly or a missing premise, the agent expands the equivalence class $Q(\phi_A, w)$ to obtain all equivalent concrete actions. A *physical* candidate is retained only if:
 - it belongs to $A(i, w) \cap H(i, w)$, i.e., the agent is both physically capable of performing it *and* enabled to do so by its group role;
 - the `canAfford` check confirms that the mental cost does not exceed $B_1(i, w)$ and that every required physical resource is covered by $B_2(i, w)$.

For *mental* actions (pure inference steps), the enabling function H does not apply. Instead, the shared inference cost $C_1(i, \alpha, w)/|G|$ is checked against $\min_{h \in G} B_1(h, w)$, i.e., the minimum mental budget across all group members, as prescribed by the *enabled_w* condition. Since all agents within a group share the same base inference cost, the first agent in the processing order that satisfies the enabling condition acts as the *doer*, consistently with the *doer_w* predicate.

5. **Best action selection (function F).** Each affordable candidate is wrapped in an internal `ActionCandidate` record carrying the action name, its `ActionCost`, the preference value, and a flag distinguishing physical from mental actions. The agent sorts the candidate list using a two-level Java `Comparator`: it first maximizes the preference value $P(i, w, \varphi'_A)$; among candidates with equal preference, it selects the one with the lowest total physical cost $sum(C_2(i, \varphi'_A, w))$. The top-ranked candidate is chosen for execution.
6. **Execution and state update.** The agent executes the selected action. For a physical action, it deducts the cost from B_2 and records the outcome (e.g., `done(cook_steak)`) in its working memory. For a mental action (inference), the simulator applies the group-level budget discipline specified in Section 3.3.1: the cost $C_1/|G|$ is deducted from the budget of *every* member of the group. In both cases, the result is broadcast to the other group members through the `CommunicationChannel`.

The following pseudo-code shows the implementation idea behind the selector. It makes explicit how action equivalence, role enablement, affordability, and preference are combined in the executable fragment:

```

1 List<ActionCandidate> candidates = new ArrayList<>();
2 for (String a : equivalenceClass(goalAction)) {
3     if (memory.isEnabledByRole(a) && budget.canAfford(costOf(a))) {
4         candidates.add(new ActionCandidate(a, costOf(a), preferenceOf(a)));
5     }
6 }
7 candidates.sort(
8     Comparator.comparing(ActionCandidate::preference).reversed()
9     .thenComparing(ActionCandidate::totalPhysicalCost)
10 );
11 return candidates.isEmpty() ? Optional.empty() : Optional.of(candidates.get(0));

```

Listing 2: Executable realization of the selector over equivalent physical actions.

The cycle is repeated at each simulation step, allowing agents to progress jointly towards their goals.

4.5. Simulation Loop

The overall execution is orchestrated by the `SimulationMain` class, which runs a step-based loop with three phases per step:

1. **Agent execution.** Every agent in the system executes its `reasonAndAct` method. Agents are processed in the order in which they appear in the configuration file, ensuring deterministic and reproducible results across runs.
2. **Stall detection and lending.** For each group that has lending enabled, the simulator checks whether the group is *stalled*, i.e., it has unmet goals but none of its available agents can make further progress. If a stall is detected, the lending mechanism (described in Section 4.6) is invoked.
3. **Global termination check.** The loop terminates successfully when every group has achieved all of its common goals. An upper bound on the number of steps (default: 15) prevents infinite loops. The simulator also halts early if an entire step completes without any agent performing an action (global stall).

4.6. Agent Lending Between Groups

A distinctive feature of the simulator is the support for *agent lending* across groups, addressing situations in which a group cannot make progress because none of its members can perform a required physical action.

The lending workflow, implemented in the `Group` class, proceeds as follows:

1. **Stall detection.** After each simulation step, the simulator checks whether a group is *stalled*: the group has unmet goals, but no available agent can make progress towards any of them.

2. **Sub-goal decomposition.** The stalled group analyses its unmet goals by invoking the recursive method `findMissingPremises`. Starting from each unmet goal, the method inspects the shared knowledge base for matching rules and collects every premise that no group member currently holds in its working memory. The recursion continues on each missing premise, so that deeply nested dependencies are flattened into a set of leaf sub-goals that can be targeted directly. For example, if the goal `done(organize_event)` requires `done(cook_main_dish)` and this premise is missing, the lending mechanism targets `done(cook_main_dish)`.
3. **Candidate search.** The group iterates over all other groups whose lending is enabled, searching for an available agent that can satisfy the missing sub-goal (via the method `canMakeProgressOn`). Availability is determined by the `lentOut` set maintained by each group: an agent that is currently lent to another group is excluded from further lending.
4. **Temporary lending.** Once a suitable agent is found, it is temporarily borrowed: the lending group marks the agent in its `lentOut` set, the requesting group adds the agent to its borrowed list, and the agent performs the required action using its own budget and capabilities (via `performBestActionFor`). The result is then propagated to the requesting group's agents through their working memories. After the action completes, the agent is returned to its original group by restoring both sets.

Operationally, the lending condition $L(i, H, G, \phi_A, w)$ is implemented as a conjunction of four checks: lending must be enabled for both the requesting and the donor group; the candidate agent must not already be lent out; the candidate must be able to make progress on the missing sub-goal according to the same $A \cap H$ and budget checks used in ordinary execution; and the selected action must produce a fact that is useful for the requesting group. The following pseudo-code summarizes the control flow:

```

1  if (requestingGroup.isStalled()) {
2      for (Fact subGoal : requestingGroup.missingLeafSubgoals()) {
3          for (Group donor : otherGroupsWithLendingEnabled()) {
4              Optional<Agent> a = donor.findAvailableAgentFor(subGoal);
5              if (a.isPresent()) {
6                  donor.markLentOut(a.get());
7                  boolean ok = a.get().performBestActionFor(subGoal);
8                  if (ok) requestingGroup.importProducedFact(subGoal);
9                  donor.restore(a.get());
10                 if (ok) return true;
11             }
12         }
13     }
14 }
15 return false;

```

Listing 3: Control flow of the lending mechanism.

This mechanism provides an executable strategy for realizing the kind of temporary cross-group collaboration envisaged by the framework. The goal is not to redefine the logic, but to offer a concrete and controlled execution model that is consistent with the cooperative perspective of L-DINF. The case study in Section 5 makes this mechanism concrete: Group 2 can decorate the venue but cannot cook the main dish; lending succeeds only because Group 1 contains an agent whose role, preferences, and resources make a cooking action executable.

4.7. Engineering Notes and Reproducibility

Technology stack. The simulator is implemented in Java 21 and built with Apache Maven. Three Java 21 features improve structural fidelity to the logical model: *sealed interfaces* restrict the `Formula` hierarchy to exactly the six permitted subtypes; *records* provide concise immutable representations for constructs such as `Atom` and `ActionCost`; and *pattern matching for instanceof* allows the

`AgentMemory.addKnowledge` dispatcher to route each incoming formula to the correct store in a single exhaustive chain without explicit casting.

Determinism, build, and structure. The simulation loop is fully deterministic: agents are processed in configuration-file order and candidate actions are ranked by a stable two-level comparator (preference descending, total physical cost ascending), ensuring reproducible traces across runs. The Maven artifact is compiled with `mvn clean package -DskipTests` and the resulting jar invoked as:

```
java -jar L-DINF_simulator.jar examples/case_study_B.ldinf
```

The repository ships with three scenario files (`banquet.ldinf`, `case_study_A.ldinf`, `case_study_B.ldinf`); traces are written to standard output. The thirteen source files are split across `com.ldinf.sim.core` (runtime components) and `com.ldinf.sim.model` (declarative data types), mirroring the core-versus-packages organization of the formal model.

5. Case Studies

We validate the simulator on two scenarios, each specified in a separate `.ldinf` configuration file:

- **Case Study A** (`case_study_A.ldinf`): Intra-group cooperation with preference-based action selection. Each group is self-sufficient; lending is disabled.
- **Case Study B** (`case_study_B.ldinf`): Inter-group agent lending. Group 2 depends on a capability that is only available in Group 1, thereby forcing the lending mechanism.

Both scenarios share the same domain, banquet preparation and event organization, but differ in Group 2's goal structure and in the `LENDING` directive.

Although deliberately compact, the domain is meant to represent a general class of resource-constrained cooperative problems: a group may have goals whose achievement depends on capabilities, costs, and preferences distributed across different agents. The banquet vocabulary is used only to keep traces readable. The same configuration pattern can encode applied domains such as service teams, hospital units, rescue groups, or distributed maintenance crews, where local insufficiency is overcome either by internal cooperation or by temporary cross-group support.

5.1. Experimental Setup

Group 1 (Banquet preparation) [both cases]. Two agents share the goals $\text{intend}_G(\text{done}(\text{prepare_banquet}))$ and $\text{intend}_G(\text{done}(\text{clean_banquet}))$. The total mental budget is $B_1=100$, divided equally (50 per agent). The knowledge base contains the inference rules, action costs, equivalences, and preferences for cooking, setting the table, and clearing the table. Agent 1 is enabled (Hi) for cooking and setting the table; Agent 2 is enabled for clearing and setting the table.

The abstract action `cook_main_dish` admits two equivalent concrete actions (C1), with the following preference values:

- P: `cook_steak` = 10
- P: `cook_pasta` = 5

Group 2 — Case A (Decoration only, no lending). A single agent has the goal $\text{intend}_G(\text{done}(\text{decorate_venue}))$. The agent is enabled (Hi) for `decorate_venue`, and the knowledge base contains only the corresponding rule. The group is self-sufficient; no cross-group dependency exists. The directive `LENDING: disabled` is set for both groups.

Group 2 — Case B (Event organization, lending required). A single agent has the goal $intend_G(done(organize_event))$, which requires both decorating a venue *and* cooking a main dish. The agent is enabled (Hi) only for `decorate_venue` and has no cooking capability or cost entry. This configuration forces the lending mechanism. The directive `LENDING: enabled` is set for both groups.

5.2. Execution Trace — Case A (No Lending)

Case A completes all goals in a single simulation step. The excerpt below is a condensed English rendering of the simulator log (the actual output uses longer Italian-language messages):

```

--- L-DINF LOADER ---
-> group_1 (2 agents, 4 Ki rules,
    2 G goals, lending=disabled)
    Budget per agent: B1=50
    agent_1 Hi=[cook_steak, set_table, cook_pasta]
    agent_2 Hi=[set_table, clear_table]
-> group_2 (1 agent, 1 Ki rule,
    1 G goal, lending=disabled)
    Budget per agent: B1=60
    agent_1 Hi=[decorate_venue]

--- STARTING SIMULATION ---

STEP 1
group_1_agent_1: cook_steak (Pref: 10)
-> done(cook_steak)
group_1_agent_1: set_table (Pref: 0)
-> done(set_table)
group_1_agent_2: clear_table (Pref: 5)
-> done(clear_table)
group_2_agent_1: decorate_venue (Pref: 8)
-> done(decorate_venue)
>>> ALL GROUPS ACHIEVED GOALS at step 1

--- FINAL REPORT ---
Group: group_1 (lending=disabled)
  G(done(prepare_banquet)): YES
  G(done(clean_banquet)): YES
  agent_1 - B1(remaining)=40
  agent_2 - B1(remaining)=5
Group: group_2 (lending=disabled)
  G(done(decorate_venue)): YES
  agent_1 - B1(remaining)=57

```

5.3. Execution Trace — Case B (With Lending)

Case B requires two simulation steps. At the end of step 1, Group 2 detects a stall and borrows an agent from Group 1 (same formatting convention as above):

```

--- L-DINF LOADER ---
-> group_1 (2 agents, 4 Ki rules,
    2 G goals, lending=enabled)
    Budget per agent: B1=50

```

```

    agent_1 Hi=[cook_steak, set_table, cook_pasta]
    agent_2 Hi=[set_table, clear_table]
-> group_2 (1 agent, 2 Ki rules,
    1 G goal, lending=enabled)
    Budget per agent: B1=60
    agent_1 Hi=[decorate_venue]

--- STARTING SIMULATION ---

STEP 1
group_1_agent_1: cook_steak (Pref: 10)
-> done(cook_steak)
group_1_agent_1: set_table (Pref: 0)
-> done(set_table)
group_1_agent_2: clear_table (Pref: 5)
-> done(clear_table)
group_2_agent_1: decorate_venue (Pref: 8)
-> done(decorate_venue)
>>> STALL DETECTED: group_2
>>> LENDING: group_2 borrows
    group_1_agent_1 from group_1
    for done(cook_main_dish)
    cook_steak -> done(cook_steak)
>>> LENDING SUCCESS

STEP 2
group_1_agent_1: Inference
    'prepare_banquet' (Energy: 1, shared)
group_1_agent_2: Inference
    'clean_banquet' (Energy: 1, shared)
group_2_agent_1: Inference
    'organize_event' (Energy: 2, shared)
>>> ALL GROUPS ACHIEVED GOALS at step 2

--- FINAL REPORT ---
Group: group_1 (lending=enabled)
    G(done(prepare_banquet)): YES
    G(done(clean_banquet)): YES
    agent_1 - B1(remaining)=30
    agent_2 - B1(remaining)=3
Group: group_2 (lending=enabled)
    G(done(organize_event)): YES
    agent_1 - B1(remaining)=55

```

5.4. Analysis

The two execution traces show that the simulator exhibits the qualitative behaviours expected from the formal framework.

1. **Preference-guided selection.** In both cases, Agent 1 of Group 1 selects `cook_steak` over `cook_pasta`, in accordance with the higher preference value. This confirms that the selector function F follows the ordering induced by the preference function P .

2. **Role-based executability.** Each agent performs only those actions for which it is both physically capable and enabled by its group role ($A(i, w) \cap H(i, w)$). This is consistent with the interaction between physical executability and role-based enabling prescribed by the formal model.
3. **Group-level reasoning and budget discipline.** Mental actions are treated as group-relevant operations: the inference cost is shared equally among all group members, as prescribed by the budget-update rule of Section 3.3.1. This ensures that collective progression towards common goals is reflected in the budgets of all participants.
4. **Belief propagation within the group.** Facts produced by one agent are made available to the other members of the same group via the `CommunicationChannel`. This realizes the cooperative dimension of the framework at the implementation level.
5. **Contrast between self-sufficiency and collaboration.** In Case A, every group is self-sufficient, and no inter-group collaboration is needed. In Case B, by contrast, Group 2 cannot achieve its goal with its own capabilities alone; successful completion depends on borrowing an agent from Group 1.
6. **Operational realization of cross-group collaboration.** The lending workflow in Case B demonstrates how the simulator operationalizes collaboration across groups when a required capability is missing locally, providing a concrete execution model for the cooperative phenomena described at the logical level.

Overall, the case studies support the claim that the simulator is a faithful executable realization of the L-DINF mechanisms considered in this paper. They show that the implementation preserves the intended qualitative interaction among preferences, executability constraints, budget conditions, group communication, and cross-group collaboration.

6. Implementation Scope and Pragmatic Approach

While the formal semantics of L-DINF provides a rich and expressive framework for reasoning about cooperative agents, translating such a logic into an executable system requires careful engineering choices. In this section, we clarify the scope of the simulator, its relation to formal proof procedures, and its relation to existing agent platforms.

6.1. Computational Complexity and Executable Coverage

L-DINF, like many expressive modal and epistemic logics, is computationally demanding. A complete proof procedure would have to address a PSPACE-complete satisfiability problem. For this reason, the simulator does not attempt to prove arbitrary L-DINF formulae at runtime. It implements an executable fragment centered on the constructs needed for simulation: explicit working memories, background rules, resource-bounded inference, physical-action execution, action equivalence, preferences, role constraints, group communication, and lending.

The resulting implementation is therefore sound as a proof of concept of the operational interaction among these packages, but it is not logically complete with respect to the full language. This is a deliberate engineering trade-off. The simulator sacrifices unrestricted theorem proving in favour of a deterministic and inspectable execution loop in which every state change can be traced to a corresponding L-DINF package.

6.2. Why Existing Platforms Are Not Sufficient

Jason is a mature BDI/AgentSpeak platform, and JADE is a mature Java middleware for FIPA-compliant multi-agent systems [13, 14]. Both are valuable for MAS development, but they address a different level of abstraction. Jason is centered on events, plans, and intentions in a BDI execution cycle; JADE provides agent containers, communication services, and interoperability. Neither platform natively represents the full combination of L-DINF packages required here.

The mismatch is not limited to syntax. L-DINF requires explicit quantitative budgets for both mental and physical actions, shared costs for group reasoning, vector-valued physical resources, equivalence classes of actions, preference-guided selection among equivalent candidates, role-constrained executability, and lending as temporary group reconfiguration. Encoding these mechanisms inside an existing BDI or middleware platform would require redefining the deliberation cycle, the representation of group state, and the resource update layer. Such an encoding would obscure the direct relationship between semantic packages and executable components, which is precisely the methodological goal of the simulator.

6.3. Expressive Power for Non-Trivial MAS Modelling

Despite its pragmatic scope, the simulator is expressive enough to model non-trivial cooperative behaviours. The case studies show the interaction of five mechanisms that are usually handled separately: preference-based action choice, role-based action filtering, budget-constrained execution, group-level belief propagation, and cross-group lending. Importantly, the observed behaviours are not hard-coded into the traces; they emerge from the combination of scenario declarations and the generic execution loop.

The modular architecture also makes the simulator useful as an experimental platform. A researcher may disable lending, alter the preference ordering, change the budget-sharing rule, restrict equivalence classes, or replace the selector function, and then compare the resulting traces. This supports the workshop-oriented goal of turning an abstract formal model into a reproducible tool for inspecting cooperative reasoning under resource constraints.

7. Conclusion and Future Work

We presented a modular Java simulator for L-DINF, an epistemic logic for cooperative agents operating under resource constraints. The contribution is an executable and inspectable realization of the main semantic packages of the framework: explicit beliefs, background knowledge, mental and physical budgets, costs, preferences, action equivalence, role-based executability, group communication, and temporary lending across groups. The implementation is intentionally pragmatic: it is not a complete proof procedure for L-DINF, but a reproducible simulator that makes the interaction among the core packages observable in concrete traces.

The revised positioning of the work is deliberately precise. Existing logical agent languages such as DALI, BDI platforms such as Jason, MAS middleware such as JADE, epistemic model checkers, and ASP-L-DINF integrations all provide important and complementary tools. The novelty of the present work lies in the standalone and direct mapping of the L-DINF package architecture to Java components, together with case studies that show how the formal mechanisms interact during execution.

The case studies confirm that the simulator reproduces the qualitative behaviours predicted by the theory. Preferences guide action selection, budgets constrain both inference and physical execution, roles regulate which agents may perform which actions, group communication propagates achieved facts, and lending enables cross-group reconfiguration when local capabilities are insufficient. These behaviours emerge from the implementation without ad hoc case-specific rules.

Future work will proceed in four directions. First, we plan to extend belief revision and inconsistency management so that richer belief dynamics can be handled during long simulations. Second, we will enlarge the executable fragment towards more of the deductive apparatus of L-DINF, while keeping computational feasibility under control. Third, we will connect the simulator with larger and more realistic scenarios, including scheduling and service-coordination domains where ASP-based optimization can provide candidate allocations and L-DINF agents can reason about their execution. Fourth, we intend to add a query and inspection interface, and to study export mechanisms towards verification tools, so that simulation traces can be checked against epistemic and temporal properties.

Declaration on Generative AI

During the preparation of this work, the author(s) used AI-based tools for grammar and spelling verification. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

- [1] A. S. Rao, M. P. Georgeff, Modeling agents within a BDI-architecture, in: Proc. of Intl. Conf. on Principles of Knowledge Representation and Reasoning (KR), Morgan Kaufmann, Cambridge, Massachusetts, 1991.
- [2] R. Fagin, J. Y. Halpern, Belief, awareness, and limited reasoning, *Artificial Intelligence* 34 (1987) 39–76.
- [3] H. van Ditmarsch, J. Y. Halpern, W. van der Hoek, B. Kooi, *Handbook of Epistemic Logic*, College Publications, 2015.
- [4] S. Costantini, A. Formisano, V. Pitoni, Preference management in epistemic logic L-DINF, in: *Proceedings of CILC 2023 – 38th Italian Conference on Computational Logic*, volume 3428 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023. URL: <https://ceur-ws.org/Vol-3428/paper10.pdf>.
- [5] S. Costantini, A. Formisano, V. Pitoni, An epistemic logic for formalizing group dynamics of agents, *Interaction Studies* 23 (2023) 391–426.
- [6] R. W. Weyhrauch, Prolegomena to a theory of mechanized formal reasoning, *Artificial Intelligence* 13 (1980) 133–170.
- [7] P. Balbiani, D. Fernández-Duque, E. Lorini, A logical theory of belief dynamics for resource-bounded agents, in: C. M. Jonker, S. Marsella, J. Thangarajah, K. Tuyls (Eds.), *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, Singapore, May 9–13, 2016, ACM, 2016, pp. 644–652. URL: <http://dl.acm.org/citation.cfm?id=2937020>.
- [8] E. Lorini, Reasoning about cognitive attitudes in a qualitative setting, in: *Logics in Artificial Intelligence - 16th European Conference, JELIA 2019, Rende, Italy, May 7–11, 2019, Proceedings*, 2019, pp. 726–743. URL: https://doi.org/10.1007/978-3-030-19570-0_47. doi:10.1007/978-3-030-19570-0_47.
- [9] S. Costantini, A. Formisano, V. Pitoni, An epistemic logic for modular development of multi-agent systems, in: N. Alechina, M. Baldoni, B. Logan (Eds.), *Engineering Multi-Agent Systems – EMAS 2021, Revised Selected Papers*, volume 13190 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 72–91.
- [10] S. Costantini, A. Tocchio, A logic programming language for multi-agent systems, in: *Logics in Artificial Intelligence: 8th European Conference, JELIA 2002*, volume 2424 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 1–13. doi:10.1007/3-540-45757-7_1.
- [11] S. Costantini, A. Tocchio, The DALI logic programming agent-oriented language, in: *Logics in Artificial Intelligence: 9th European Conference, JELIA 2004*, volume 3229 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 685–688. doi:10.1007/978-3-540-30227-8_57.
- [12] S. Costantini, G. De Gasperis, V. Pitoni, A. Salutati, DALI: A multi agent system framework for the web, cognitive robotic and complex event processing, in: *Proceedings of the 32nd Italian Conference on Computational Logic (CILC 2017)*, volume 1949 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2017, pp. 75–89. URL: <https://ceur-ws.org/Vol-1949/CILCpaper05.pdf>.
- [13] R. H. Bordini, J. F. Hübner, BDI agent programming in AgentSpeak using *Jason*, in: F. Toni, P. Torroni (Eds.), *CLIMA VI, selected papers*, volume 3900 of *LNCS*, Springer, 2005, pp. 143–164.
- [14] F. Bellifemine, G. Caire, A. Poggi, G. Rimassa, Jade: A software framework for developing multi-agent applications. lessons learned, *Information and Software Technology* 50 (2008) 10–21. doi:10.1016/j.infsof.2007.10.008.
- [15] P. Gammie, R. Meyden, Mck: Model checking the logic of knowledge, in: *Computer Aided*

Verification, 16th International Conference, CAV 2004, Proceedings, 2004, pp. 479–483. doi:10.1007/978-3-540-27813-9_41.

- [16] A. Lomuscio, H. Qu, F. Raimondi, MCMAS: A Model Checker for the Verification of Multi-Agent Systems, volume 5643, 2009, pp. 682–688. doi:10.1007/978-3-642-02658-4_55.
- [17] S. Costantini, V. Pitoni, Integrating cognitive reasoning into medical scheduling: A transition from ASP personas to L-DINF agents, in: PRIMA 2025: Principles and Practice of Multi-Agent Systems, volume 16366 of *Lecture Notes in Computer Science*, Springer, Cham, 2026, pp. 286–293. doi:10.1007/978-3-032-13562-9_21.