Run-time Quantitative Operational Monitoring for BDI Agents

MARIE FARRELL, The University of Manchester, United Kingdom
ANGELO FERRANDO, University of Modena and Reggio Emilia, Italy
MENGWEI XU, Newcastle University, United Kingdom

The BDI (Belief-Desire-Intention) agent paradigm has been used to model decision-making in autonomous systems that operate with little to no human intervention, reasoning with beliefs, desires, and intentions on-the-fly to achieve goals. Since BDI agents are used to make decisions in safety-critical systems, verifying that they make correct decisions is vital. Existing design-time verification techniques are effective before deployment, but they cannot capture how agent behaviour evolves during execution, especially under imprecise actuation and probabilistic decision-making. This paper bridges the gap between design-time and run-time verification of probabilistic BDI agents. We propose a run-time quantitative operational monitoring methodology based on Discrete-Time Markov Chains (DTMCs), integrating execution traces with probabilistic model checking. Our approach constructs a DTMC representation of possible agent behaviours at design time, then incrementally updates it using observed execution traces to provide situational insight into agent operation. To demonstrate our work, we contribute a probabilistic BDI language extension, explicate our approach through smart manufacturing for probabilistic BDI agents, and use a rover case study to show the generality of our DTMC-based monitoring. We further implement this monitoring mechanism and evaluate its efficiency and scalability experimentally. This advances agent verification from pre-deployment analysis to ongoing quantitative assurance under uncertainty.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation → Verification by model checking**; **Operational semantics**; • **Computing methodologies → Multi-agent systems**.

Additional Key Words and Phrases: BDI Agents, Quantitative Monitoring, Run-time Analysis, Software Verification

ACM Reference Format:
Marie Farrell, Angelo Ferrando, and Mengwei Xu. 2026. Run-time Quantitative Operational Monitoring for BDI Agents. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (July 2026), 41 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The Belief-Desire-Intention (BDI) [11] architecture is a mature and well-studied rational agent framework for developing autonomous behaviour using explicit cognitive notions such as beliefs, goals, plans, and intentions. BDI languages and frameworks include, among others, AgentSpeak [48], 3APL [35], 2APL [16], Jason [10], and Conceptual Agent Notation (Can) [49, 51]. More broadly, recent work has adopted cognitive agents as fine-grained modular building blocks for contemporary software architectures, arguing that agent-oriented abstractions such as beliefs, intentions,

Authors' Contact Information: Marie Farrell, marie.farrell@manchester.ac.uk, The University of Manchester, Manchester, United Kingdom; Angelo Ferrando, angelo.ferrando@unimore.it, University of Modena and Reggio Emilia, Modena, Italy; Mengwei Xu, mengwei.xu@newcastle.ac.uk, Newcastle University, Newcastle Upon Tyne, United Kingdom.

ACM 1557-7392/2026/7-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

capabilities, and goal-oriented interaction can support higher-level structuring, reusability, and composability in software engineering, for example in the work [7]. In a BDI agent, the *(B)eliefs* represent what the agent knows, the *(D)esires* what the agent wants to bring about, and the *(I)ntentions* those desires that the agent has chosen to act upon. In this paper, we use the phrase "*BDI agent*" to refer specifically to agents that are programmed in BDI languages (*e.g.* AgentSpeak or Can) where beliefs, desires, and intentions are explicit constructs for agent reasoning. Examples of BDI agents in practical applications include robots [13]. For instance, a robot may maintain *beliefs* about its current position and nearby hazards (obtained from sensors), *desires* such as reaching a particular inspection location possibly in response to external requests from human operators, and *intentions* corresponding to the currently committed navigation plan (*e.g.* "go to waypoint w_1 via route r"). As new information arrives (*e.g.* the robot detects a hazard), the agent can revise its beliefs, desires, and intentions, and reconsider which destination or route to commit to next.

The design of BDI agents is far from trivial due to a mix of declarative specification (description of the state sought), procedural specification (set of instructions to perform), failure handling, and inherently interleaved concurrent behaviours (*e.g.* multi-tasking). Crucially, this complexity raises concerns about the safety and trustworthiness of the deployment of these agents. As such, there is a growing demand for verification techniques to analyse the behaviour of these BDI agents, ensuring their safety and boosting user confidence. There has been a proliferation of techniques and languages supporting BDI agent analysis, exemplified by [2–4, 18, 29], to ensure the correct design and provide evidence of the trustworthiness of these agents through formal analysis. Furthermore, many realistic deployments operate under uncertainty such as cyber-physical robotic systems (*e.g.* surveyed in [14]) with uncertain sensors and actuators. For instance, the outcome of an action may be probabilistic due to sensor noise and imprecise actuation. In addition, events, plans and intentions may differ in their domain-specific characteristics such as preference and urgency. As a result, the agent may assign probabilities to choices among enabled events, plans, or intentions, such as, to represent priority, urgency, or domain-specific scheduling assumptions. In other words, probabilistic BDI agents do not merely represent an uncertain environment; they also make explicit how uncertainty and probabilistic choice affect the agent's possible future behaviours. These uncertainty considerations have motivated *probabilistic BDI agents*, which extend the classical BDI agent with quantitative information under uncertainty.

Consider a robot that must inspect a location and can choose between a short but risky route and a longer but safer route. Before the route is selected, design-time analysis may compute the probability of eventually completing the inspection over all possible route choices and action outcomes from the initial configuration. Once the agent has selected the risky route, or once a movement action has failed, the relevant assurance question changes: the operator no longer needs the original probability from the initial configuration, and the probability of success or failure is now conditional on what has actually happened. Similar questions arise when a BDI agent chooses which event to handle first, which intention to progress, or which plan to select. This example highlights a general limitation of design-time assurance: it does not answer operational questions that arise after the agent has already executed part of its behaviour.

The prevailing focus in the community is on analysing BDI agents from some pre-fixed initial states prior to deployment, overlooking the nature of the run-time decision-making that is inherent to BDI agents, *i.e.* acting as they go. In practice, questions such as "what is the current probability of an agent achieving a task after doing action X or Y, or selecting plan A or B?" often arise, demanding an understanding of the implications of the choices that are made by the agent during operation. In principle, existing verification methods can re-verify a running system *e.g.* from a post-action state by resetting the initial configuration to the current (post-action) state and re-running verification. In practice, repeatedly applying this manual reset-and-reverify loop after

each iteration is computationally costly and error-prone; mistakes in re-initialisation can invalidate the resulting safety assurance. The challenge, therefore, lies not merely in the capability to change initial states or conduct re-verification but fundamentally in devising an error-resistant and efficient approach for continuously analysing BDI agents' decision-making at run time.

To address this challenge, this paper adopts a monitor-based verification approach, namely Run-time Verification (RV) [6], to analyse systems during operation. In our setting, RV provides the observation mechanism: it monitors the running agent, records the execution, and relates the observed behaviour to the formal model. Meanwhile, the main objective is to assess the quantitative implications of the agent's observed choices or actions under uncertainty. For example, after an action outcome, event handling decision, or plan selection has been observed, the monitor updates the probabilistic model and re-computes relevant probabilistic properties. If such a property is expressed with a probability threshold, this re-assessment yields a satisfaction or violation verdict; if it is expressed as a quantitative query, it yields an updated probability value.

In this paper, we employ RV to serve as one of the foundational methodological components to continuously observe the behaviours of BDI agents during operation. Our objective is to gain operational quantitative insights (*e.g.* determining the current probability of task achievement) based on the actions that have already been taken by the agent. To facilitate this, a formal probabilistic model of the BDI agent itself is required to allow us to capture its potential future behaviour. As a result, our methodology comprises two phases: (1) Design Time and (2) Run Time. At design time, probabilistic BDI agents (introduced later) are formally modelled using a suitable probabilistic specification language that is chosen by system analysts. From a defined initial state, all potential behaviours are exhaustively explored and captured in the transition system of a Discrete-Time Markov Chain (DTMC) [41] prior to deployment. At run time, a monitor is automatically constructed to observe the operational activities of the agent. The collected run-time data which reflects the agent's actual behaviour will be incrementally integrated into the existing DTMC, thereby updating it to accurately represent the agent's ongoing operational state. For instance, probabilistic transitions that correspond to observed actions are updated to certainty (probability of 1), clearly marking realised transitions.

A significant advantage of our approach compared to the default methods—which require continuously changing initial states followed by exhaustive re-verification—is that we perform the most computationally expensive step, namely DTMC construction and state-space exploration for the given agent model, only once at design time, rather than re-doing it after every change of initial state. Crucially, this makes the DTMC the central artefact of our approach: it can be constructed using whatever domain assumptions or modelling choices the developers or engineers prefer, and is then monitored and incrementally revised using run-time observations. Then the agent's behaviour space, captured by the DTMC, is therefore revised using only newly observed operational data. We note that, analysing a probabilistic system via an updated DTMC has a negligible computational overhead [2]. Thus, the incremental DTMC revision and analysis over the DTMC entail minimal computational overhead, allowing for immediate, precise, and operationally relevant quantitative insights that are seamlessly aligned with continuous monitoring. If verification at any stage yields inconclusive results within predefined bounds, system operations continue uninterrupted, naturally gathering richer operational data for subsequent, more definitive analyses. By combining continuous RV monitoring with probabilistic verification of the formally captured model (in this case, DTMCs), our method ensures consistent, accurate insights into system behaviour at any operational stage while maintaining low computational overhead, as confirmed through our empirical experimentation in Section 6.3.

To apply our methodology, we have taken a pragmatic approach that explicitly builds upon existing foundational work from [3], which provides an executable probabilistic semantics for

the Can language (an expressive BDI programming language). Concretely, given (i) a Can agent program, (ii) a probabilistic model of decision-making (*e.g.* which plan to select), and probabilistic distributions of action outcomes, and (iii) an initial agent configuration (*e.g.* initial beliefs and intentions), this executable probabilistic semantics can generate a Discrete-Time Markov Chain (DTMC) (via exhaustive state exploration) whose states correspond to reachable agent configurations in Can and whose transitions represent the agent's steps with associated probabilities. Our approach utilises and extends the DTMC representations from [3] specifically towards run-time verification and monitoring tasks. Thus, the primary target of this paper is probabilistic BDI agents programmed in Can, while the run-time verification and monitoring mechanism itself is defined over the resulting DTMC. This DTMC-level formulation is intentional: it keeps the BDI-specific modelling contribution explicit, while allowing the monitoring algorithm to be reused whenever an appropriate DTMC representation is available. Preliminary results of this run-time monitoring concept were initially introduced in [24]. In this paper, we make the following novel research contributions beyond our preliminary work [24] and regarding the foundational work [3].

- We significantly improve the formal presentation of the probabilistic semantics of BDI agents from [3] to make it less convoluted and more consistent with practical encoding. In particular, we separate agent-level operation selection into explicit probabilistic transition rules, thereby improving the alignment of BDI semantics with the underlying DTMC representation in modelling and analysis. (Section 4.2)
- For the first time, we provide a complete, rigorous formalisation of the probabilistic BDI agent's behavioural space explicitly as a DTMC, an important theoretical aspect that was not covered in [3]. The BDI semantics defines the stepwise evolution of agent configurations, while the DTMC formalisation defines the global probabilistic transitions used for model checking and run-time monitoring. We have accompanied this definition with a detailed graphical example that is, conceptually, in alignment with the usual BDI reasoning cycle. (Section 4.3)
- We present the formal design, practical implementation, and analysis of our computational algorithms that enable efficient run-time updates of the DTMC, the de-facto representation for probabilistic systems. The analysis establishes computational complexity which is important for practical implementation, while the implementation shows how the updated DTMC can be used with probabilistic model checkers to compute updated quantitative values or threshold verdicts. This was originally outlined in [24] at a very high-level and we provide more detail here. (Section 4.4)
- We show how our quantitative monitoring approach can support run-time quantitative analysis using a smart manufacturing example for both existing and new probabilistic selection strategies in [3]. This example illustrates our methodology from probabilistic Can to DTMC-based run-time monitoring. (Section 5)
- We demonstrate the generality of our methodology to any system as long as such a system can be modelled in a formal modelling language with tools to support DTMC construction and export. It implies that the DTMC-level monitoring mechanism is not strictly tied to Can once an appropriate DTMC has been obtained. This is illustrated through an autonomous rover example using the PRISM modelling language (*i.e.* reactive modules formalism [1]). (Section 6.1 and Section 6.2)
- We provide in-depth empirical results that demonstrate the computational efficiency and scalability of our underlying core monitoring technique with randomly generated, synthetic parameterised DTMCs. This evaluation isolates the cost of DTMC update and re-analysis from

domain-specific applications, allowing us to systematically vary DTMC size and monitored trace length. (Section 6.3)
- We provide a detailed reflection on the scope, prerequisites, and significance of the successful application of our methodology, in particular, our choice of a DTMC-based generic and analysable formal model structure. It clarifies the engineering conditions under which the approach is applicable and the role played by the DTMC abstraction beyond the specific BDI scenario. (Section 8)

Our research contributions in quantitative operational monitoring for probabilistic BDI agents include a methodology (Section 4), a DTMC representation of the probabilistic extension of BDI agents (Sections 4.2 and 4.3), an algorithm and implementation for monitors (Sections 4.4 and 4.5), a case study of BDI agents for feasibility (Section 5), a case study of non-BDI systems for broader generality (Section 6), and an extensive empirical computational evaluation to show the computational scalability (Section 6.3). We overview related work in Section 2. Section 8 reflects on our approach and outlines plans for future work. We conclude in Section 9.

2 Related Work

We consider related work from the following perspectives: (1) verification of BDI agents, (2) run-time verification, and (3) planning and scheduling.

(1) Verification of BDI Agents: Formal verification of BDI is well-summarised in a survey [43]. However, none of that work has addressed the run-time analysis of BDI agents with quantitative insights for agent behaviours. To make the comparison precise, we position prior work along two dimensions that are central to this paper: (i) design-time only vs. run-time monitoring, (ii) qualitative satisfaction vs. quantitative analysis of future behaviours.

A short commentary on the main existing BDI analysis is given below. One of the earliest publications on the verification of BDI agents is [9]. This employs the Promela specification language [37] (originally intended for communication-based systems) alongside the Spin model-checker [36] to analyse an agent which is specified in a restricted version of AgentSpeak. A major breakthrough is found in [20], which applies the program model checker (in this case, Java Path Finder[1]) to analyse an agent implemented in Java in the GWENDOLEN language [17] (a variation of AgentSpeak). Concretely, this breakthrough shifts verification from abstract encodings to checks over executable agent programs (via program model checking), which is conceptually close to our executable encoding. Though utilising Java PathFinder has the advantage of performing verification on the program code, it typically suffers from a significant performance bottleneck due to the symbolic execution of Java bytecode. Very recently, there has also been work *e.g.* [39, 52] that applies theorem proving to avoid the issue of state explosion commonly found in model-checking approaches. For example, [39] employs the Isabelle/HOL proof assistant [45] to model and verify agents that are programmed in GOAL [34], where GOAL is a pure reactive system and does not select pre-defined plans from a library but instead selects individual actions (or a sequence of actions). Complementing these theorem-proving approaches, [54] focuses on algebraically encoding the syntax of the Can language in Event-B theories [12], using inductive data types and polymorphic constructors to provide a faithful and reusable foundation for subsequent proof-based semantic modelling and analysis. To sum up, this line of work (either via model checking or theorem proving) provides strong qualitative design-time guarantees, but it does not support run-time monitoring about the agent's behaviour once execution has progressed.

[1] http://javapathfinder.sourceforge.net

Recent work has started to examine the probabilistic verification of BDI agents. For example, [19] uses a two-stage verification method that first *generates* a model through program model checking (of a system implementation), and then converts this model into PRISM input format for analysis. Similarly, [38] facilitates probabilistic verification of simplified BDI agents by encoding them directly in PRISM. One of the closest works to ours is [3] which encodes BDI agents through a probabilistic formal modelling language. Our approach utilised some of its notation and results to obtain DTMCs. However, like many others, it does not support operational monitoring (at run time) through DTMC revision, as we do in this paper. In summary, existing probabilistic BDI verification tends to focus on constructing a probabilistic model for offline model checking (via encoding or generation) and analysing a fixed model from a fixed initial state. In contrast, our contribution is to *retain* the precomputed DTMC of the full probabilistic behaviour space and to *update* it online from observed execution, so that standard probabilistic model checking can provide immediate quantitative insights about *future* behaviours from the current operational state.

(2) Run-time Verification. There is a vast community on the topic of run-time verification, *e.g.* [6]. In this paper, we highlight that we are not competing with the existing paradigm of run-time verification. Rather, we see the effectiveness of run-time verification, offering assurances against systems during execution, and employ it as one of the main components in our methodology. We compare run-time verification approaches in terms of: (i) whether they target BDI agents specifically, (ii) whether they support probabilistic reasoning, and (iii) whether they provide predictive quantitative analysis via an explicit probabilistic behaviour model (as opposed to only checking properties over the observed trace without models).

Although run-time verification has been applied to agent-based systems *e.g.* [13, 25], we are not aware of any run-time verification technique supporting quantitative analysis for probabilistic BDI agents. For example, [25] uses run-time verification as a way to enforce safety properties on non-probabilistic BDI (in this case, AgentSpeak) in the context of Linear Temporal Logic (LTL) [47] for property specifications. This is representative of RV for BDI agents: it strengthens safety during execution, but it remains qualitative (property satisfied/violated) and does not yield a quantitative analysis of future behaviours of BDI agents under uncertainty.

There are also existing state-of-the-art approaches in run-time verification that may be similar to ours, but applied in other types of systems. For example, there is a growing trend that applies probabilistic verification techniques for the run-time analysis of systems that may change or need to be re-constructed, *e.g.* adaptive software systems. For example, the work in [30] focuses on system re-configurations (*e.g.* a new device in the network), using a run-time probabilistic verification approach that re-uses the previous analysis results when some small changes are made to the model of the system being verified. However, these should not be confused with work in *e.g.* [22, 27] that learns/improves a DTMC's transition probabilities using system operation data to provide a more accurate model of the system that reflects its behaviour in its deployment environment. The key distinction is that reconfiguration-aware approaches [30] aim to maintain verification results under *model changes*, while learning-based approaches [22, 27] update *probabilities* to better fit the deployed environment. Our setting differs: the design-time DTMC already captures the agent's potential behaviour space, and the run-time task is to *update the DTMC to reflect the current operational state* based on observed agent steps, enabling run-time probabilistic analysis of *future* behaviour.

Other related work conducts run-time probabilistic verification by shifting the cost of model construction at design time to run time. For example, the approaches in [26] and [28], similar to our methodology, split the process into pre-computation and run-time verification phases. In the pre-computation phase, they consider (1) the model of the system as a DTMC, (2) a set of

transition variables, and (3) the reliability requirements of the system. The values of transition variables would become known only during execution (at run time). As such, the verification step that is performed at run time simply evaluates the formulae by replacing the variables with the actual values gathered by monitoring the system. In this paper, our approach to pre-computation is to construct the DTMC with the exact values of the transition (from the model) and run-time verification to provide quantitative insights of potential future behaviours of BDI agents based on the executed agent operational activities. Compared to parametric approaches [26, 28], we do not leave transition probabilities symbolic: instead, we precompute a concrete DTMC capturing the full behaviour space and then revise it to synchronise with the observed execution trace. This means that run-time analysis is not limited to parameter instantiation; it can also reflect the evolving operational activities within the behaviour space from the current state. We note that the distinction here is not in the class of properties that can be checked, but in how the model that is used for run-time analysis is updated: our approach conditions the concrete DTMC on the execution that has been observed so far, so that quantitative queries are evaluated over a trace-updated representation of the current operational state.

Run-time verification provides quantitative analysis of the specification properties for cyber-physical systems while the system is being executed. For example, [5] provides the probability for the specification property being satisfied based on the partial execution through a prediction model using statistical learning. There are also some advanced and promising probabilistic run-time verification approaches for monitoring systems with imprecise sensors *e.g.* in [40]. In these cases, the challenge is that the monitor cannot directly observe the current system state. Hence, they aim to infer the probability of being in a state from a trace of observations due to the partial information about the system state in form of observations. These works address uncertainty in *state estimation* (partial observability) and learning-based prediction of satisfaction. In contrast, our approach is not designed to recover unobserved states or to learn probabilities from incomplete observations. Instead, it assumes that agent-level execution steps are observable at the level of the formal model and uses them to revise a precomputed DTMC of the agent's behaviour space. Thus, the quantitative results come from probabilistic model checking of an updated model rather than from statistical prediction.

Finally, another loosely related but quite different approach to run-time probabilistic verification is through statistical model checking [55]. This method, largely proposed to overcome the state space explosion issue that is inherent in traditional model checking, avoids the exhaustive exploration of all system states by sampling system executions and statistically estimating the probability that specific properties hold. As such, it can return a verification result relatively quickly, making it acceptable for run-time usage. However, it still remains "static" unless initial states in the model are manually updated each time to be in synchronisation with the current state of the actual system, and, importantly, by bypassing exhaustive exploration, it sacrifices the strong correctness guarantees that we are seeking for responsible autonomous systems. In contrast, our approach preserves exhaustive probabilistic verification on a DTMC: we do the expensive exploration/model construction once at design time, and then only apply lightweight DTMC revision to synchronise with the operational state at run time.

(3) Planning and Scheduling: Our work is related to automated planning and scheduling *e.g.* [31, 50]. In general, planning and scheduling are typically concerned with generating, selecting, or optimising action sequences that achieve specified goals under domain constraints *e.g.* to maximise rewards or the probability of reaching a goal. This is closely related to BDI agents because BDI reasoning also concerns goal-directed behaviour, plan selection, and intention progression. In fact, prior work has studied the integration of planning and BDI reasoning, including the use of planning

to synthesise new plans when no predefined BDI plan is applicable or when existing plans fail *e.g.* [53]. Here we refer the interested reader to a related survey on integrating BDI agents with planning [44].

Other work has connected BDI decision-making with formal verification and strategy synthesis over probabilistic models, for example, to reason about event, plan, and intention selection strategies *e.g.* in [4]. Interestingly, formal verification can coincide with planning when verification properties are expressed as reachability properties (i.e. a set of final desired states). In this case, formal verification can check if there exists a particular evolution of the system that makes reachability true and extract this evolution of the strategy as a side effect.

Our focus in this paper is complementary. We do not use planning and scheduling to synthesise plans or schedules at run time. Instead, we assume a probabilistic BDI agent whose possible behaviours have already been captured in a DTMC, and we monitor how the agent's observed run-time choices revise this DTMC and affect quantitative properties related to future behaviour. Thus, planning and scheduling provide a related perspective on goal-directed decision-making, while our contribution concerns run-time quantitative monitoring of probabilistic BDI execution.

3 Theoretical Background

We now provide the theoretical background for the paper. Section 3.1 introduces BDI agents using the Conceptual Agent Notation (Can) language [51]; Section 3.1.3 gives a simple BDI example to illustrate the syntax and semantics presented in Section 3.1.1 and Section 3.1.2, respectively. Section 3.2 then briefly introduces Discrete-Time Markov Chains (DTMCs) [41] and Probabilistic Computational Tree Logic (PCTL) [32].

3.1 BDI Agents

The Conceptual Agent Notation (Can), a superset of AgentSpeak [48], is a high-level agent programming language that captures BDI concepts with advanced behaviours such as reasoning with *declarative goals*, *concurrency*, and *failure recovery*, but without describing implementation details such as data structures. Here, we briefly summarise the Can syntax and semantics from [49, 51], to which we refer for more detailed accounts.

3.1.1 Syntax. A Can agent consists of a belief base $\mathcal{B}$, a plan library Π, an action description library Λ, and a set of intentions Γ.

The belief base $\mathcal{B}$ is a finite set of belief atoms that represents the agent's current beliefs. The belief base has operators to check whether a belief formula, φ, follows from the belief base (*i.e.* $\mathcal{B} \models \varphi$) and to revise the belief base. Entailment, $\mathcal{B} \models \varphi$, is defined under the closed world assumption: each belief atom, b, is true iff $b \in \mathcal{B}$, and false otherwise. Complex belief formulas, φ, are built from belief atoms using the standard logical connectives *i.e.* $\varphi ::= b \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$, and their truth is evaluated in the usual way from this assignment. Belief revision is performed at the level of belief atoms. To add a belief, we add a belief atom, b, into the belief base using set union (*i.e.* $\mathcal{B} \cup \{b\}$); to remove a belief, we use set minus (*i.e.* $\mathcal{B} \setminus \{b\}$).

A *plan library* Π contains the operational procedures of an agent and is a finite collection of plans of the form $e : \varphi \leftarrow P$. Here, e is the triggering event of the plan, φ is the context condition, and P is the plan-body. In short, the triggering event, e, specifies why the plan is triggered, while the context condition, φ, determines *when* the plan-body, P, is applicable (*i.e.* can be selected and executed) for that event. For convenience, we call the set of events from the external environment the external event set, denoted E^e. Meanwhile, the remaining events (which occur as part of the plan-body) are called either *sub-events* or *internal* events.

By convention *e.g.* in [10], the user-defined plan-body P in a plan $e : \varphi \leftarrow P$ may be referred to as the *program* or *agent program* and has the following syntax:

$$P ::= \ act \mid ?\varphi \mid \ + b \mid \ - b \mid e \mid P_1; P_2 \mid P_1 \parallel P_2 \mid goal(\varphi_s, e, \varphi_f)$$

with act an action, $?\varphi$ a test for φ entailment in the belief base, $+b$ and $-b$ represent belief addition and deletion respectively, and e is a sub-event (i.e. internal event). To execute a sub-event, a plan (corresponding to that event) is selected and the plan-body is added in place of the event. In this way, we allow plans to be nested (similar to sub-routine calls in other languages). Actions, act, (from the action description library, Λ) take the form $act = \psi \leftarrow \langle \phi^+, \phi^- \rangle$, where ψ is the precondition, and ϕ^+ and ϕ^- are the addition and deletion sets of belief atoms. For example, a belief base, $\mathcal{B}$, is revised with addition and deletion sets, ϕ^+ and ϕ^-, to be $(\mathcal{B} \setminus \phi^-) \cup \phi^+$ when the action executes. There are also composite programs denoted by $P_1; P_2$ for sequence and $P_1 \parallel P_2$ for interleaved concurrency. Finally, a declarative goal program, $goal(\varphi_s, e, \varphi_f)$, specifies a persistent goal whose success condition is φ_s and whose failure condition is φ_f. To achieve the success condition, the agent handles e by selecting and executing an applicable plan for this event. If φ_s becomes true, the goal succeeds and terminates; if φ_f becomes true, the goal fails. If neither condition holds after an attempt, the goal is retried, allowing the agent to continue pursuing φ_s through further handling of e.

When a plan, $e : \varphi \leftarrow P$, is selected to respond to an event, its plan-body, P, is adopted as an intention in the intention base, Γ, (a.k.a. the partially executed plan-body). A number of auxiliary agent programs are used internally for intentions, as shown below:

$$nil \mid e : (|\varphi_1 : P_1, \cdots, \varphi_n : P_n|) \mid P_1 \triangleright P_2 \mid goal(\varphi_s, P, \varphi_f) \mid false.$$

The first auxiliary program, nil, denotes a terminating program, *i.e.* there is nothing left to execute. The second auxiliary program captures a set of relevant plans for an event, e. The set of plans, whose triggering event matches the given event, e, is denoted compactly as $e : (|\varphi_1 : P_1, \cdots, \varphi_n : P_n|)$. It means that given an event, e, a plan-body, P_i, can be executed if its related context condition, φ_i, holds for the current belief base.

The auxiliary agent program, $P_1 \triangleright P_2$, provides the failure recovery by executing P_2 only on failure of P_1. The failure handling mechanism, $\triangleright$, which is a distinguishing feature of the Can language, specifies the kind of behaviour when the current plan does not go as expected. Intuitively, it enables trying alternative plans for addressing an event if the current strategy failed. Achieving this type of failure handling is typically accomplished by combining the programs $e : (|\varphi_1 : P_1, \cdots, \varphi_n : P_n|)$ and $P_1 \triangleright P_2$. For example, $P_1 \triangleright e : (|\varphi_2 : P_2, \cdots, \varphi_n : P_n|)$ represents an event-handling structure where a plan with plan-body P_1 has been selected from the relevant plans for event e, and P_1 is now the current program to execute. The remaining relevant plans for e, namely $e : (|\varphi_2 : P_2, \cdots, \varphi_n : P_n|)$, are retained as alternatives that may be tried if the plan-body P_1 cannot progress.

For a user-defined declarative goal, $goal(\varphi_s, e, \varphi_f)$, the agent progresses it by addressing the event e that is in the goal unless either the success or failure condition holds. When the event e is being addressed, this event is progressively replaced by the auxiliary program that is produced by the event-handling rules, for example, a set of relevant plans and then a selected plan-body with fallback alternatives. We therefore also use the auxiliary form $goal(\varphi_s, P, \varphi_f)$, where P denotes the current program that is being used to pursue the goal after the initial event e has begun to be progressed. Finally, *false* represents the failed program produced when the failure condition of a declarative goal holds.

3.1.2 Semantics. The Can semantics is specified using two types of transitions. The first, denoted by $\rightarrow$, specifies *intention-level* evolution on intention-level configurations, $\langle \mathcal{B}, P \rangle$, where $\mathcal{B}$ is the belief base, and P is the plan-body that is currently being executed. The second type, denoted by $\Rightarrow$,

$$\frac{act : \psi \leftarrow \langle \phi^+, \phi^- \rangle \quad \mathcal{B} \models \psi}{\langle \mathcal{B}, act \rangle \rightarrow \langle (\mathcal{B} \setminus \phi^-) \cup \phi^+, nil \rangle}\, act \qquad \frac{\Delta = \{\varphi : P \mid (e' = \varphi \leftarrow P) \in \Pi \wedge e' = e\}}{\langle \mathcal{B}, e \rangle \rightarrow \langle \mathcal{B}, e : (|\ \Delta\ |) \rangle}\, event$$

$$\frac{\varphi : P \in \Delta \quad \mathcal{B} \models \varphi}{\langle \mathcal{B}, e : (|\ \Delta\ |) \rangle \rightarrow \langle \mathcal{B}, P \rhd e : (|\ \Delta \setminus \{\varphi : P\}\ |) \rangle}\, select$$

$$\frac{\langle \mathcal{B}, P_1 \rangle \rightarrow \langle \mathcal{B}', P_1' \rangle}{\langle \mathcal{B}, P_1 \rhd P_2 \rangle \rightarrow \langle \mathcal{B}', P_1' \rhd P_2 \rangle}\, \rhd_; \qquad \frac{}{\langle \mathcal{B}, (nil \rhd P_2) \rangle \rightarrow \langle \mathcal{B}, nil \rangle}\, \rhd_\top$$

$$\frac{P_1 \neq nil \quad \langle \mathcal{B}, P_1 \rangle \nrightarrow \quad \langle \mathcal{B}, P_2 \rangle \rightarrow \langle \mathcal{B}', P_2' \rangle}{\langle \mathcal{B}, P_1 \rhd P_2 \rangle \rightarrow \langle \mathcal{B}', P_2' \rangle}\, \rhd_\bot \qquad \frac{\langle \mathcal{B}, P \rangle \rightarrow \langle \mathcal{B}', P' \rangle}{\langle \mathcal{B}, (nil; P) \rangle \rightarrow \langle \mathcal{B}', P' \rangle}\, ;_\top$$

$$\frac{\langle \mathcal{B}, P_1 \rangle \rightarrow \langle \mathcal{B}', P_1' \rangle}{\langle \mathcal{B}, (P_1; P_2) \rangle \rightarrow \langle \mathcal{B}', (P_1'; P_2) \rangle}\, ; \qquad \frac{\langle \mathcal{B}, P_1 \rangle \rightarrow \langle \mathcal{B}', P_1' \rangle}{\langle \mathcal{B}, (P_1 \| P_2) \rangle \rightarrow \langle \mathcal{B}', (P_1' \| P_2) \rangle}\, \|_1 \qquad \frac{\langle \mathcal{B}, P_2 \rangle \rightarrow \langle \mathcal{B}', P_2' \rangle}{\langle \mathcal{B}, (P_1 \| P_2) \rangle \rightarrow \langle \mathcal{B}', (P_1 \| P_2') \rangle}\, \|_2$$

$$\frac{}{\langle \mathcal{B}, (nil \| nil) \rangle \rightarrow \langle \mathcal{B}, nil \rangle}\, \|_\top$$

$$\frac{\mathcal{B} \models \varphi_s}{\langle \mathcal{B}, goal(\varphi_s, P, \varphi_f) \rangle \rightarrow \langle \mathcal{B}, nil \rangle}\, G_s \qquad \frac{\mathcal{B} \models \varphi_f}{\langle \mathcal{B}, goal(\varphi_s, P, \varphi_f) \rangle \rightarrow \langle \mathcal{B}, ?false \rangle}\, G_f$$

$$\frac{P \neq P_1 \rhd P_2 \quad \mathcal{B} \not\models \varphi_s \quad \mathcal{B} \not\models \varphi_f}{\langle \mathcal{B}, goal(\varphi_s, P, \varphi_f) \rangle \rightarrow \langle \mathcal{B}, goal(\varphi_s, P \rhd P, \varphi_f) \rangle}\, G_{init}$$

$$\frac{\mathcal{B} \not\models \varphi_s \quad \mathcal{B} \not\models \varphi_f \quad \langle \mathcal{B}, P_1 \rangle \rightarrow \langle \mathcal{B}', P_1' \rangle}{\langle \mathcal{B}, goal(\varphi_s, P_1 \rhd P_2, \varphi_f) \rangle \rightarrow \langle \mathcal{B}', goal(\varphi_s, P_1' \rhd P_2, \varphi_f) \rangle}\, G_;$$

$$\frac{\mathcal{B} \not\models \varphi_s \quad \mathcal{B} \not\models \varphi_f \quad \langle \mathcal{B}, P_1 \rangle \nrightarrow}{\langle \mathcal{B}, goal(\varphi_s, P_1 \rhd P_2, \varphi_f) \rangle \rightarrow \langle \mathcal{B}, goal(\varphi_s, P_2 \rhd P_2, \varphi_f) \rangle}\, G_\rhd$$

Fig. 1. Intention-level Can semantics derived from [4]

specifies *agent-level* evolution over agent-level configurations, $\langle E^e, \mathcal{B}, \Gamma \rangle$, detailing how to execute a complete agent where E^e is the set of pending external events to address (a.k.a. desires) and Γ is a set of partially executed plan-bodies (intentions). Fig. 1 summarises the rules for evolving a given intention.

Action execution. The *act* rule executes an action, *act*, with a guard (precondition), ψ, and a belief update, $\langle \phi^+, \phi^- \rangle$. If $\mathcal{B} \models \psi$, then the action can be executed, updating the belief base to $(\mathcal{B} \setminus \phi^-) \cup \phi^+$ and reducing the residual program to *nil*.

Event handling and plan selection. The *event* rule replaces an event, e, by the set of relevant plans for that event. Here, Π is the plan library, and $\Delta = \{\varphi : P \mid (e' = \varphi \leftarrow P) \in \Pi \wedge e' = e\}$ collects all plans whose triggering event e' matches the event e, paired with their context condition, φ, and plan-body, P. We write $e : (|\Delta|)$ for an event that is annotated with the set of $\varphi : P$ pairs from relevant plans for the event, e. The *select* rule chooses one applicable plan from the set of relevant plans $e : (|\Delta|)$. If $\varphi : P \in \Delta$ and $\mathcal{B} \models \varphi$, then the agent may commit to executing P, while retaining the remaining relevant plans as backups, yielding $P \rhd e : (|\Delta \setminus \{\varphi : P\}|)$. If several relevant plans are applicable, the semantics does not impose an ordering among them; any one of them may be selected non-deterministically.

Failure recovery ($\rhd$). The $P_1 \rhd P_2$ operator represents *fallback execution*: execute P_1, but if P_1 fails or becomes blocked, switch to P_2. The $\rhd_;$ rule propagates normal execution of the left component: if P_1 can step to P_1', then $P_1 \rhd P_2$ steps to $P_1' \rhd P_2$. The $\rhd_\top$ rule discharges the fallback execution structure when the left component has completed: $(nil \rhd P_2)$ becomes *nil* (*i.e.* no fallback is needed once the primary program has successfully finished). The $\rhd_\bot$ rule captures recovery on failure/blocking:

if P_1 cannot take a step ($\langle \mathcal{B}, P_1 \rangle \not\rightarrow$), but P_2 can progress ($\langle \mathcal{B}, P_2 \rangle \rightarrow \langle \mathcal{B}', P_2' \rangle$), then the execution transfers to P_2.

Sequential composition. The $(P_1; P_2)$ construct executes P_1 and then P_2. The ; rule progresses the first component: if P_1 can progress to P_1', then $(P_1; P_2)$ steps to $(P_1'; P_2)$. The $;_\top$ rule starts P_2 once P_1 has finished: $(nil; P)$ steps to P'.

Interleaved concurrency. The $(P_1 \| P_2)$ construct represents interleaved parallel execution. The $\|_1$ and $\|_2$ rules allow either component to progress (updating $\mathcal{B}$ and replacing the progressed component with its successor program), while leaving the other component unchanged. The $\|_\top$ rule terminates concurrent execution when both sides have completed: $(nil \| nil) \rightarrow nil$.

Declarative goals. A declarative goal has the form $goal(\varphi_s, P, \varphi_f)$, where φ_s is the success condition, φ_f is the failure condition, and P is the associated procedural program that is used to try to achieve the goal. The G_s rule terminates the goal successfully when $\mathcal{B} \models \varphi_s$. The G_f rule fails the goal when $\mathcal{B} \models \varphi_f$, replacing it with $?false$. Here, $?false$ denotes the test action $?\varphi$ instantiated with the formula *false*. Since *false* is never entailed by the belief base, this produces an unprogressable plan-body, thereby representing failure of the declarative goal. As a result, it allows the surrounding failure-recovery mechanism to treat the current attempt as failed and try an alternative if one is available. The G_{init} rule initialises goal persistence when neither φ_s nor φ_f holds, and the program has not yet been put into a persistent form: it rewrites the goal program to $P \triangleright P$, meaning "try P, and if it fails, try P again". The $G_;$ rule progresses a step of the persistent program inside the goal: if the left side of $P_1 \triangleright P_2$ progresses, then the whole goal steps accordingly. Finally, the $G_\triangleright$ rule restarts the persistent attempt when the current procedural attempt inside the declarative goal has become stuck, *i.e.* it cannot take a semantic step, and neither φ_s nor φ_f holds. Rather than waiting indefinitely, the declarative goal semantics re-establishes the fallback structure so that the agent can try again to achieve the goal.

The agent-level semantics are given in Fig. 2. Recall that an agent configuration has the form $\langle E^e, \mathcal{B}, \Gamma \rangle$, where E^e is the (finite) set of pending external events, $\mathcal{B}$ is the current belief base, and Γ is the current intention base (a set of *agent programs* to be executed concurrently). The transition relation, $\Rightarrow$, denotes one agent-level step.

The A_{event} rule represents event adoption: if an external event, e, is present in E^e, the agent can remove it from the event set and add it to the intention base, Γ. Operationally, this indicates that the event has been acknowledged and queued for deliberation/handling as part of the agent's intention execution. The A_{step} rule performs one deliberation/execution step for one selected intention. It non-deterministically selects some $P \in \Gamma$ and applies one intention-level transition, $\langle \mathcal{B}, P \rangle \rightarrow \langle \mathcal{B}', P' \rangle$, thereby potentially updating beliefs to $\mathcal{B}'$ and replacing the selected intention, P, by its progressed version, P', in the intention sets (leaving all of the other intentions unchanged). This models the interleaved execution of multiple intentions at agent level. Finally, the A_{update} rule removes intentions that cannot make progress: if $P \in \Gamma$ and $\langle \mathcal{B}, P \rangle \not\rightarrow$, then P is discarded from Γ. This captures the termination of intentions, depending on the intention-level rules (*e.g.* reaching *nil*).

The two transition relations in the CAN semantics reflect the modular structure of BDI execution. An agent may have several pending external events and several active intentions, while each intention is itself a partially executed plan-body. The intention-level relation, $\rightarrow$, describes one local step of a single intention, such as executing an action, expanding an event into relevant plans, selecting an applicable plan, or progressing a sequential composition. The agent-level relation, $\Rightarrow$, describes one global step of the whole agent configuration, such as adopting an external event as a new intention, selecting an existing intention to progress, or removing an intention that can no

$$\frac{e \in E^e}{\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow \langle E^e \setminus \{e\}, \mathcal{B}, \Gamma \cup \{e\}\rangle} A_{event} \quad \frac{P \in \Gamma \quad \langle \mathcal{B}, P\rangle \rightarrow \langle \mathcal{B}', P'\rangle}{\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow \langle E^e, \mathcal{B}', (\Gamma \setminus \{P\}) \cup \{P'\}\rangle} A_{step}$$

$$\frac{P \in \Gamma \quad \langle \mathcal{B}, P\rangle \nrightarrow}{\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow \langle E^e, \mathcal{B}, \Gamma \setminus \{P\}\rangle} A_{update}$$

Fig. 2. Agent-level CAN semantics derived from [4].

```
1  // Initial belief base
2  at_base, power_high, short_route_clear, safe_route_clear
```

(a) Belief base.

```
1  // Initial external event
2  mission
3  // Plan library
4  mission : ⊤ <- goal(site_checked, inspect_site, mission_failed)
5  inspect_site : at_base ∧ short_route_clear <- move_short; check_site
6  inspect_site : at_base ∧ safe_route_clear <- move_safe; check_site
```

(b) External event and plan library.

```
1  // Action description library
2  move_short : at_base ∧ power_high ∧ short_route_clear <- ⟨{mission_failed, power_low},{at_base, power_high}⟩
3  move_safe : at_base ∧ power_high ∧ safe_route_clear <- ⟨{at_site, power_low},{at_base, power_high}⟩
4  check_site : at_site <- ⟨{site_checked},∅⟩
```

(c) Action description library.

Fig. 3. A simple BDI agent for a robot inspection scenario.

longer proceed. This separation keeps the semantics modular: changes to the rules for progressing individual plan-bodies can be made at the intention level, while the agent-level rules only need to specify how intentions are scheduled, added, updated, or removed within the complete agent configuration.

3.1.3 Simple BDI Example. We now give a small example to illustrate the syntax and the two-level semantics that was introduced above. The example is intentionally simplified for illustrative purposes. The agent is a simple inspection robot that receives a mission to inspect a site. It can either take a short route or a safe route. The short route is intended to represent a risky choice, while the safe route represents a more reliable but potentially more conservative choice.

Fig. 3a gives the initial belief base. The robot initially believes that it is at the base, that its power level is high, and that both the short and safe routes are clear. These are represented as belief atoms: `at_base`, `power_high`, `short_route_clear`, and `safe_route_clear`.

Fig. 3b gives the initial external event and the plan library. The initial external event is `mission`. The first plan states that, when the event `mission` is handled, the agent adopts a declarative goal: `goal(site_checked, inspect_site, mission_failed)`. This means that the agent should try to make `site_checked` true by addressing the event `inspect_site`, unless the failure condition, `mission_failed`, becomes true. The remaining two plans are alternative ways of addressing `inspect_site`. If the robot is at the base and the short route is clear, it can execute `move_short; check_site` (line 5). If the robot is at the base and the safe route is clear, it can execute the action of `move_safe; check_site` (line 6). In other words, the `inspect_site` event has two relevant and initially applicable plans.

Fig. 3c gives the action description library. Each action has a precondition and belief updates. For example, `move_safe` (on line 3) can execute when `at_base`, `power_high`, and `safe_route_clear`

hold; its effect is to add `at_site` and `power_low`, and to remove `at_base` and `power_high`. The `check_site` action can execute once `at_site` holds, and adds `site_checked` to the belief base (line 4). In contrast, `move_short` illustrates a risky route in this simple example: executing it adds `mission_failed` (line 2). This deliberately simple example lets us illustrate success and failure behaviours without adding probabilistic action outcomes at this point given the probabilistic BDI is not introduced yet.

We now explain how the two transition relations operate on this example. At the agent level, the initial configuration contains the pending external event, `mission`, the initial belief base from Fig. 3a, and an empty intention base. By the A_{event} rule, the agent may adopt the external event, `mission`: the event is removed from the external event set and added to the intention base. At this point, `mission` is no longer a pending external event, but an intention to be progressed.

The subsequent execution of `mission` is described by the intention-level relation. The event rule first replaces `mission` by the set of relevant plans for that event. Since the plan library contains the plan `mission : ⊤ <- goal(site_checked, inspect_site, mission_failed)`, and its context condition is $\top$, this plan is applicable. The select rule then selects its plan-body, namely the declarative goal, `goal(site_checked, inspect_site, mission_failed)`.

The declarative goal then tries to achieve `site_checked` by progressing the `inspect_site` event. Since neither `site_checked` nor `mission_failed` initially holds, the goal is not yet completed or failed. It will first be initialised as `goal(site_checked, inspect_site`$\triangleright$`inspect_site, mission_failed)`. Then the left part `inspect_site` will be progressed using the relevant plans in the plan library. Both plans for `inspect_site` are initially applicable, because the belief atoms of `at_base`, `short_route_clear`, and `safe_route_clear` all hold. The select rule may therefore choose either `move_short; check_site` or `move_safe; check_site`, retaining the remaining applicable plan as a possible fallback according to the failure-handling structure.

If the safe plan is selected, the sequence that rule first progresses is `move_safe` (line 3 of Fig. 3c). The action rule applies because its precondition, `at_base` $\wedge$ `power_high` $\wedge$ `safe_route_clear`, holds. The belief base is updated by adding `at_site` and `power_low`, and removing `at_base` and `power_high`. The sequence then progresses to `check_site`; since `at_site` now holds, executing `check_site` adds `site_checked`. The declarative goal can then terminate successfully because its success condition holds.

This example also explains why the semantics uses two transition relations. The intention-level relation, denoted by $\rightarrow$, describes the evolution of a single program fragment, such as expanding `inspect_site`, selecting a relevant plan, executing `move_safe`, or progressing the sequence `move_safe; check_site`. The agent-level relation, denoted by $\Rightarrow$, describes the evolution of the whole agent configuration, including adopting an external event as an intention, selecting an intention to progress, and removing intentions that can no longer proceed. The separation is useful because BDI agents may have several pending events and several intentions, while each individual intention still evolves according to its own structure.

3.2 Discrete-Time Markov Chain (DTMC)

Probabilistic systems can be described using Markov models/processes, where the probability of moving to a new state is strictly based on the current state [41]. A **Discrete-Time Markov Chain** (DTMC) is a 5-tuple $\mathcal{D} = \langle S, \bar{s}, P, AP, L\rangle$ where S is the set of states, $\bar{s} \in S$ a designated initial state, $P : S \rightarrow Dist(S)$ is a function mapping each state $s \in S$ to a probability distribution μ_s such that $\mu_s(s') \in [0, 1]$ is the transition probability from s to s', AP is a set of atomic propositions, and $L : S \rightarrow 2^{AP}$ is a labelling function mapping each state to a set of atomic propositions. For terminal states s_t, we have μ_{s_t} as the point distribution to itself with probability 1. Each state, s, in a DTMC represents one possible modelled system configuration. Each transition represents the possibility

of transiting from one configuration, s, to another, s', in one discrete step with the probability $\mu_s(s')$. To capture the evolution of a system, we have a *path* to represent any possible execution of a system. In a DTMC, a path is a sequence of states $s_0 s_1 s_2 \ldots$ starting from s_0 such that $\mu_{s_i}(s_{i+1}) > 0$ for all $i \geq 0$ and each pair $s_i s_{i+1} (i \in \mathbb{N})$ is called an execution.

To analyse probabilistic systems, formal logic is used to specify their behaviours. For DTMCs, specifications can be written in **Probabilistic Computation Tree Logic** (PCTL) [32] to enable reasoning about path-based properties. Such properties can be used to constrain the probability with which certain desirable or undesirable behaviours may occur. For example, a typical property includes "the probability of the system failing at some point is less than 0.01" and a typical query could be "what is the probability of the system failing at some point". The first expresses a *qualitative* specification—it can be checked as either true or false for a given model. The second expresses a *quantitative* query—it requires computing a numeric probability value. While both are supported in probabilistic model checkers such as PRISM [42] and STORM [33], it is important to note that *only the former falls within the strict syntax of PCTL*, while the latter is a *query-based extension* that is commonly used in tool-supported analysis.

Formally, the syntax of PCTL is given by:

$$\Phi ::= \text{true} \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid P_{\sim p}[\phi]$$

$$\phi ::= X\Phi \mid \Phi \, U^{\leq k} \, \Phi \mid \Phi \, U \, \Phi$$

where a is an atomic proposition, $\sim \in \{<, \leq, >, \geq\}$, $p \in [0, 1]$, and $k \in \mathbb{N}^+$. PCTL formulae are interpreted over the states of a DTMC, with satisfaction denoted $s \models \Phi$. The probabilistic operator, $P_{\sim p}[\phi]$, specifies that the probability of the path formula, ϕ, being satisfied from a given initial state meets the bound, $\sim p$. Path formulae include $X\Phi$ ("Φ holds in the next step"), $\Phi_1 \, U^{\leq k} \, \Phi_2$ ("Φ_2 holds within k steps, and Φ_1 holds until then"), and $\Phi_1 \, U \, \Phi_2$ ("Φ_2 eventually holds, and Φ_1 holds until that point"). Common derived operators include $F\Phi \stackrel{\text{def}}{=} \text{true} \, U \, \Phi$ for "eventually Φ" and $G\Phi \stackrel{\text{def}}{=} \neg F \neg \Phi$ for "globally Φ".

In practice, model checking tools (*e.g.* PRISM and STORM) extend PCTL with *query-based expressions* such as $P =?[\phi]$, which compute the exact probability of ϕ being satisfied, rather than asserting a probability bound. Strictly, such queries are *not* part of the classical PCTL syntax. But they are widely supported and crucial for probabilistic diagnostic and risk analysis tasks. In our setting, these quantitative queries are particularly useful for interpreting agent behaviour under uncertainty, and for returning precise probabilistic measures for task achievement that can be compared—thereby supporting richer forms of analysis beyond simple boolean satisfaction checking.

4 Methodology

This section introduces our quantitative operational monitoring methodology. We first give a high-level overview of the workflow (Fig. 4) in Section 4.1. We then formalise the probabilistic extension of Can (Section 4.2), define how probabilistic BDI behaviours are represented as DTMCs (Section 4.3), and present our run-time DTMC update procedure (Section 4.4) and monitor algorithm (Section 4.5) which is used throughout the remainder of the paper.

4.1 Overview

Our methodology (Fig. 4) consists of two phases: Design Time and Run Time. The Design Time phase constructs a probabilistic behavioural model of the agent as a DTMC, while the Run Time phase continuously *calibrates* this DTMC using observations from the running agent and re-analyses it at any given point to provide timely quantitative assurance. The distinction between the two

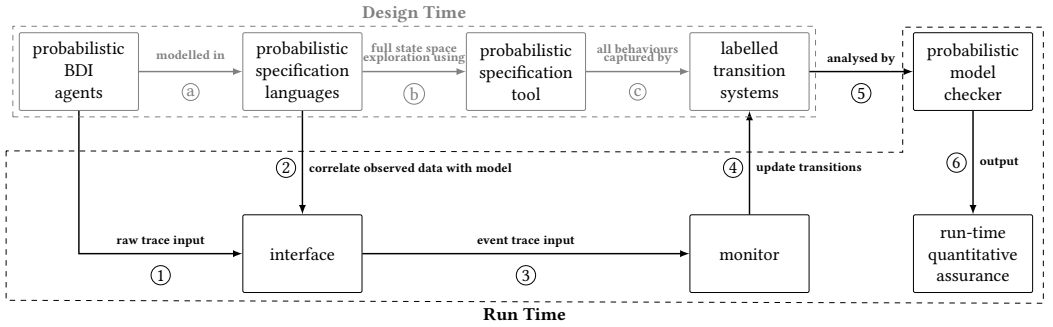


Fig. 4. Quantitative Operational Monitoring Methodology for BDI Agents, originally outlined in [24]. Boxes represent the main artefacts or tools in the workflow, and arrows represent the flow of models, traces, updates, and analysis results between them.

phases is the generation and update of a specific artefact: the DTMC. At design time, the DTMC is generated to capture the possible probabilistic behaviours of the agent from the specified initial configuration. At run time, the monitor does not reconstruct this DTMC from scratch after every observed step. Instead, it updates the existing DTMC based on the observed execution and invokes probabilistic model checking on this updated DTMC. The methodological focus is therefore on maintaining an analysis-ready DTMC during operation, avoiding repeated exhaustive exploration and manual DTMC re-generation after each observed step.

Design Time Phase. In the top-left of Fig. 4, probabilistic BDI agents (formalised in Section 4.2) are encoded (ⓐ) as a formal model in a suitable probabilistic specification language. A relevant probabilistic specification tool then takes this formal model of the BDI agent(s) and performs a full reachable-state exploration (ⓑ) from a chosen initial agent configuration (*e.g.* initial beliefs and a set of external events), producing (ⓒ) a DTMC. This DTMC captures all possible behaviours of such a probabilistic BDI agent that is reachable from the given initial state. In the DTMC, states correspond to agent configurations, and transitions correspond to enabled semantic rules with associated transition probabilities. A conventional use of this DTMC is to run a probabilistic model checker on the design-time model (⑤) to verify quantitative properties from the assumed initial state. However, such design-time analysis does not directly support analysing the system as it evolves during operation: to obtain guarantees for a later operational configuration, one typically has to reset the model's initial configuration and re-run exploration/model checking, which requires re-constructing a DTMC.

Run Time Phase. To avoid repeated manual resets and re-verification for each point of the agent operation, we employ a monitor-based verification approach that does not necessitate interfering with the model itself. This requires collecting and reasoning about run-time operational data from BDI agents, and, importantly, providing the run-time analysis at any given point of agent operation. This has motivated the bottom half of run-time design in our methodology in Fig. 4. As the probabilistic BDI agent runs, we observe it (①) and collect the latest operational data (*e.g.* system logs). We then correlate the observed data with the formal model of the probabilistic agent (②) to obtain a model-level trace of the executed activities of the agent (a.k.a. monitor events) for the monitor (③). Here, monitor events denote the observed agent execution activities that are used by the monitor; they are distinct from BDI external and internal events in the agent semantics. The monitor then updates (④) the DTMC based on the event trace by revising the DTMC. For example,

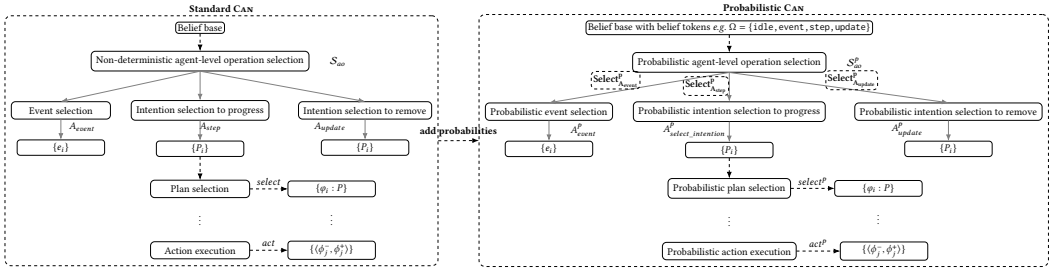


Fig. 5. Overview of the transition structure in standard Can and probabilistic Can. The left panel shows the standard non-deterministic choices in Can; the right panel shows how these choices are made explicit as probabilistic choices in Can^p. Solid arrows denote agent-level transitions, while dashed arrows denote intention-level transitions. The highlighted dashed-box labels indicate the new probabilistic rules for selecting agent-level operations.

at run time, once the monitor observes that the agent has moved from state s to state s', it conditions the DTMC on that observation by setting the probability of the observed transition $s \to s'$ to 1 (and all other outgoing transitions from s to 0), thereby aligning the model with the agent's current execution path. Importantly, by updating the DTMC, we avoid the most computationally expensive step (state-space construction), while the subsequent inspection of the system via the transition system to query quantitative properties has an extremely low computational overhead [2]. We can therefore invoke a probabilistic model checker on the updated DTMC (⑤) and report quantitative insights (⑥) with a latency that is comparable to monitoring, keeping the results timely as new observations arrive. We support this claim further with empirical experiments and analysis in Section 6.3.

Finally, we highlight that the key to the approach is the existence of a DTMC that captures the system's probabilistic dynamics and can be monitored and updated at run time. This is the main abstraction boundary: the DTMC can be constructed by the developers and engineers using any modelling assumptions or domain knowledge that is deemed appropriate, while our contribution operates on (and continuously revises) the DTMC to provide quantitative operational insights during execution.

4.2 Probabilistic Can

In this section, we introduce Can^p, our probabilistic extension of Can, which makes Can's implicit non-deterministic choices explicit via probability distributions. A key design goal is to preserve the structure of the Can semantics while exposing (i) *which agent-level operation is performed next* and (ii) *which element is selected within that operation* as explicit probabilistic decisions. This separation yields a semantics that is both easier to read when compared to original Can and directly aligned with DTMC encodings, where each probabilistic choice corresponds to a single transition.

Fig. 5 depicts the transition structure of standard Can and its probabilistic extension, Can^p, and serves as a roadmap for the semantic rules that are introduced in the remainder of this section.

The left panel summarises standard Can. At each agent cycle, exactly one of the three agent-level semantic rules is applied: adopting an external event, progressing an intention, or removing an unprogressable intention. The solid arrows represent these agent-level transitions. In standard Can, the choice of which agent-level rule to apply is implicit and non-deterministic. We write this implicit choice as the non-deterministic agent-level operation selector $\mathcal{S}_{ao}$.

The right panel shows CAN^p. The overall reasoning structure is unchanged: the agent still has the same three agent-level branches and the same distinction between agent-level and intention-level transitions. The difference is that the implicit choice points in standard CAN are made explicit and probabilistic. First, we augment the belief base with a small set of belief tokens that record the currently selected agent-level operation status, *i.e.* $\Omega = \{idle, event, step, update\}$, with exactly one token present at any time. This makes the agent-level operation choice an explicit part of the state, and therefore an explicit state-to-state transition in a DTMC. Correspondingly, the non-deterministic selector $\mathcal{S}_{ao}$ is lifted to its probabilistic counterpart, $\mathcal{S}^p_{ao}$, which returns a distribution over $\{A_{event}, A_{step}, A_{update}, \bot\}$. Sampling from this distribution is represented by the highlighted bold labels enclosed in dashed boxes, namely $\mathbf{Select}^{\mathbf{p}}_{\mathbf{A_{event}}}$, $\mathbf{Select}^{\mathbf{p}}_{\mathbf{A_{step}}}$, and $\mathbf{Select}^{\mathbf{p}}_{\mathbf{A_{update}}}$, (as shown in Fig. 5) which we introduce below.

After an agent-level operation has been selected, CAN^p then makes the intention-level choice probabilistic. For example, if event handling is selected, the agent probabilistically selects which external event to adopt; if intention progression is selected, it probabilistically selects which intention to progress; and if intention removal is selected, it probabilistically selects which unprogressable intention to remove. At the intention level, choices such as plan selection and probabilistic action outcomes are similarly represented by probabilistic rules, indicated by superscripts p such as $select^p$ and act^p.

Finally, the figure highlights an important structural point of probabilistic CAN. The probabilistic semantics separates each agent step into three conceptual parts: (i) selecting the agent-level operation, (ii) selecting the target element for that operation, such as which external events to adopt as an intention, and (iii) applying the selected operation. This separation matches the step-wise decision-making structure of BDI agents and aligns each probabilistic choice with a corresponding DTMC transition. We formalise this separation in the rules below.

We now introduce the probabilistic semantics that we use to construct the design-time DTMC. We note that the purpose of this part is not to define the run-time monitor itself, but to make explicit the probabilistic choices whose outcomes become transitions in the DTMC. In the CAN semantics, while the agent-level semantics allow the agent to respond to new events even when already dealing with other events, only one agent-level operation can be performed at each step. Such non-deterministic choice of agent-level operation is implicit in CAN, which we express explicitly as a function:

$$\mathcal{S}_{ao} : 2^{\overline{E^e}} \times 2^{\overline{\mathcal{B}}} \times 2^{\overline{\Gamma}} \to \mathcal{A} \cup \{\bot\}$$

where $\mathcal{A} = \{A_{event}, A_{step}, A_{update}\}$ denotes the set of agent-level rules (Fig. 2). We denote the set of all possible belief atoms—for a *specific program*—as $\overline{\mathcal{B}}$ and any given current belief base is given as $\mathcal{B} \subseteq \overline{\mathcal{B}}$. Similar notation applies for $\overline{E^e}$ and $\overline{\Gamma}$. The function $\mathcal{S}_{ao}$ returns a choice of (a)gent-level (o)perations (*ao*) to be applied given an agent-level configuration $\langle E^e, \mathcal{B}, \Gamma\rangle$ and $\bot$ stands for no applicable agent-level rules available (*e.g.* no tasks at hand).

In practice, selecting an agent-level operation is often achieved in a deterministic fashion such as by incorporating an external event (if there are any external events) before selecting an intention to execute a step. However, we may need to choose an agent-level operation from a probability distribution. For example, it may be better for the agent to *mainly* incorporate external events as intentions at the early operation stage and to "*mainly*" progress existing intentions at a later stage. Here the preference of "*mainly*" implies the probabilistic nature of the decision-making. To allow this, we can correspondingly represent it by sampling agent-level operations based on a probability distribution:

$$\mathcal{S}^p_{ao} : 2^{\overline{E^e}} \times 2^{\overline{\mathcal{B}}} \times 2^{\overline{\Gamma}} \to Dist(\mathcal{A} \cup \{\bot\})$$

$$\frac{idle \in \mathcal{B} \quad \mathcal{S}^p_{ao}(E^e, \mathcal{B}, \Gamma) = \mu \quad \mu(A_{event}) = p}{\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow_p \langle E^e, (\mathcal{B} \setminus \{idle\}) \cup \{event\}\}, \Gamma\rangle}\ \mathbf{Select}^{\mathbf{p}}_{\mathbf{A_{event}}}$$

$$\frac{idle \in \mathcal{B} \quad \mathcal{S}^p_{ao}(E^e, \mathcal{B}, \Gamma) = \mu \quad \mu(A_{step}) = p}{\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow_p \langle E^e, (\mathcal{B} \setminus \{idle\}) \cup \{step\}\}, \Gamma\rangle}\ \mathbf{Select}^{\mathbf{p}}_{\mathbf{A_{step}}}$$

$$\frac{idle \in \mathcal{B} \quad \mathcal{S}^p_{ao}(E^e, \mathcal{B}, \Gamma) = \mu \quad \mu(A_{update}) = p}{\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow_p \langle E^e, (\mathcal{B} \setminus \{idle\}) \cup \{update\}\}, \Gamma\rangle}\ \mathbf{Select}^{\mathbf{p}}_{\mathbf{A_{update}}}$$

Fig. 6. Rules to select which agent-level operations to apply

The probability of $\perp$ of any distribution $\mu \in Dist(\mathcal{A} \cup \{\perp\})$ is either $\mu(\perp) = 0$ (if there are agent-level operations available) or we have that $\mu(\perp) = 1$ otherwise. The superscript p denotes the probabilistic version of $\mathcal{S}^p_{ao}$.

To allow for such probabilistic selection of agent-level operations, we extend the agent-level semantics with explicit probabilistic transitions. In our formulation, the selection of an agent-level operation is represented as a separate transition, so that each probabilistic choice corresponds directly to a DTMC transition.

To move from non-deterministic transitions to probabilistic transitions, we employ probabilistic transitions $C \rightarrow_p C'$ or $C \Rightarrow_p C'$(*i.e.* move from C to C' with probability p depending on if it is an intention-level or agent-level transition) [21]. To extend the non-deterministic transition, the key is to assign a probability to each selection choice. To capture the choices of agent-level operation, we first formalise them as $\Omega = \{event, step, update, idle\}$ (which is a set of self-explanatory agent-operation statuses) and then augment the belief base with it (*i.e.* $\overline{\mathcal{B}} \cup \Omega$) as self-belief notes[2] such that $\mid \mathcal{B} \cap \Omega \mid = 1$ for $\mathcal{B} \subseteq \overline{\mathcal{B}}$ (any belief base $\mathcal{B}$—a subset of the set of all possible belief atoms $\overline{\mathcal{B}}$ *i.e.* $\mathcal{B} \subseteq \overline{\mathcal{B}}$—can have one and only one agent-level operational status from Ω at any moment *i.e.* $\mid \mathcal{B} \cap \Omega \mid = 1$).

Using $\mathcal{S}^p_{ao}$, we can now define these extra probabilistic rules to explicitly select different agent-level operations. These rules are given in Fig. 6. For example, the $\mathbf{Select}^{\mathbf{p}}_{\mathbf{A_{event}}}$ rule allows the agent to sample the agent-level operation of A_{event} from a probabilistic distribution and update the status from *idle* to *event*.

This formulation builds on the probabilistic BDI semantics of [3], but separates agent-level operation selection from the subsequent operation-specific choice. In [3], the selection of the agent-level operation is kept implicit in the premises of other rules. Here, we represent this selection as an explicit probabilistic transition, as shown in Fig. 6. This makes the rules less convoluted and improves the alignment between the semantics and DTMC representations, where such selections correspond to transitions between states.

Meanwhile, the outcome of an action may be probabilistic due to imprecise actuation, *e.g.* the robot tries to open a gate, but it might fail. Furthermore, plans, events, and intentions can differ in their domain-specific characteristics, such as preference and urgency, which may require different selection strategies (*e.g.* sampled from a probabilistic distribution). We now introduce new rules for these aspects, and we note that although these new rules have the same essence as those introduced in [3], they are significantly revised representations in the context of the new rules in Fig. 6. In

[2]Extending the tuple of agent-level configurations is also feasible here. We choose to take advantage of the belief base to keep the notation concise and, importantly, to make the encoding of such agents easier to manage. Specifically, by storing the agent-level operation status directly in the belief base, we can reuse existing belief update mechanisms and avoid introducing additional syntactic or structural machinery.

particular, we will show the contrast between some of the old rules in [3] and the newly devised rules to illustrate how the new rules map better to the DTMC representation.

Recall that the semantics of CAN is specified by two types of transitions. The first is the agent-level transition $\Rightarrow$ in Fig. 2 that specifies how to execute a complete agent. The second is the intention-level transition $\rightarrow$ in Fig. 1 that specifies how to evolve a given single intention. For example, to have a probabilistic version of A_{event} to probabilistically select an external event to address. We sample events based on a probability distribution, *i.e.* with the probabilistic event selection function: $\mathcal{S}_e^p : 2^{\overline{E^e}} \times 2^{\overline{\mathcal{B}}} \times 2^{\overline{\Gamma}} \rightarrow Dist(\overline{E^e} \cup \{\bot\})$ where the subscript e in $\mathcal{S}_e^p$ stands for events and $\bot$ denotes that there is no external event present to address. Using $\mathcal{S}_e^p$, we can define the probabilistic A_{event}^p rule:

$$\frac{event \in \mathcal{B} \quad \mathcal{S}_e^p(E^e, \mathcal{B}, \Gamma) = \mu \quad e \in E^e \quad \mu(e) = p}{\langle E^e, \mathcal{B}, \Gamma \rangle \Rightarrow_p \langle E^e \setminus \{e\}, (\mathcal{B} \setminus \{event\}) \cup \{idle\}, \Gamma \cup \{e\} \rangle} \mathbf{A}^{\mathbf{p}}_{\mathbf{event}}$$

The $\mathbf{A}^{\mathbf{p}}_{\mathbf{event}}$ rule says that if the agent-level operation of A_{event} is selected (*i.e.* $event \in \mathcal{B}$) and the probability of selecting a pending external event is p, then the probability of selecting this event at this step is p. The status is also updated from *event* back to *idle* to allow the next round of agent-level operation selection.

In contrast, the old rule of $\mathbf{A}^{\mathbf{p}}_{\mathbf{event}}$ in [3], which we denote as $\mathbf{A}^{\mathbf{p}}_{\mathbf{event\_old}}$ here, is given as follows:

$$\frac{\mathcal{S}_{ao}^p(E^e, \mathcal{B}, \Gamma) = \eta \quad \eta(A_{event}) = p_1 \quad \mathcal{S}_e^p(E^e, \mathcal{B}, \Gamma) = \mu \quad e \in E^e \quad \mu(e) = p_2}{\langle E^e, \mathcal{B}, \Gamma \rangle \Rightarrow_{p_1 \cdot p_2} \langle E^e \setminus \{e\}, \mathcal{B}, \Gamma \cup \{e\} \rangle} \mathbf{A}^{\mathbf{p}}_{\mathbf{event\_old}}$$

This $\mathbf{A}^{\mathbf{p}}_{\mathbf{event\_old}}$ rule conflates the selection of the agent-level operation (A_{event}) and the selection of the event to process into a single transition step. As a result, it fails to reflect the step-wise decision-making process that is typical of BDI agents, where the choice of what kind of agent-level operation to perform (*e.g.* incorporate an event) is conceptually and procedurally distinct from the content-specific selection within that operation (*e.g.* choosing a particular event). Moreover, this conflation obscures the correspondence with Discrete-Time Markov Chains (DTMCs), where each transition should represent a single, well-defined action with a clear source and target state. By making the selection implicit and combining multiple decisions in one rule, the old formulation limits clarity, modularity, and direct traceability between semantic rules and DTMC transitions. Our revised formulation addresses this by separating agent-level operation selection into its own explicit probabilistic transitions, as shown in Fig. 6, thereby both preserving the faithfulness to BDI semantics and improving its alignment with DTMC modelling practice.

Now we extend the agent-level rule, A_{step}, probabilistically. Similarly, we have the following function to allow intention selection from a distribution: $\mathcal{S}_i^p = 2^{\overline{E^e}} \times 2^{\overline{\mathcal{B}}} \times 2^{\overline{\Gamma}} \rightarrow Dist(\overline{\Gamma} \cup \{\bot\})$ where the subscript i in $\mathcal{S}_i^p$ stands for intentions. We note that the agent-level transitions of A_{step} depend on the intention-level transitions (which can itself be probabilistic *e.g.* probabilistic action execution). Again, to reflect the step-wise decision-making in BDI agents and align with DTMC transitions, we will need to split the probabilistic version of A_{step} into two steps.

The first step in our approach is to probabilistically select which intention to progress. To model this formally, we introduce a family of belief tokens that signal whether a specific intention has been selected. Given a set of intentions Γ, for any $P \in \Gamma$, we define a belief token, $\chi(P)$, that denotes that the intention P has been selected. We note that, by construction, the belief tokens $\chi(P)$ (which is a single belief token) range over the different steps of each intention that may arise during execution. Concretely, as an intention progresses, its current program, P, evolves step-by-step under the intention-level semantics into a new program, P', then P'', and so on. Strictly speaking, each of these programs represents a distinct intention and should therefore be treated as a separate P for the purpose of intention selection. However, for practical efficiency, these programs can often be treated

as equivalent by linking them to the external event that the intention ultimately addresses. To manage the constraint that *only one intention can be selected at any time*, we introduce another belief token, *no_intention_selected*. This token serves as a convenient signal that no intention is currently selected. Formally, we say: $no\_intention\_selected \in \mathcal{B} \iff \nexists P \in \Gamma$ such that $\chi(P) \in \mathcal{B}$. This token ensures that the agent can use existing belief update mechanisms in practical agent platforms to record either that no intention is currently selected, or that a specific intention P has been selected through $\chi(P)$, but not both. In detail, when an intention P is selected, *no_intention_selected* is removed, and $\chi(P)$ is added. When that intention is progressed (*i.e.* completes one step), $\chi(P)$ is removed and *no_intention_selected* is reinserted.

We now define two rules to capture the behaviour of the probabilistic version of A_{step}: the first one for selecting an intention, and the second one for progressing it.

$$\frac{step \in \mathcal{B} \quad no\_intention\_selected \in \mathcal{B} \quad \mathcal{S}_i^p(E^e, \mathcal{B}, \Gamma) = \mu \quad P \in \Gamma \quad \mu(P) = p_1}{\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow_{p_1} \langle E^e, (\mathcal{B} \setminus \{no\_intention\_selected\}) \cup \{\chi(P)\}, \Gamma\rangle} \ \mathbf{A}^{\mathbf{p}}_{\mathbf{select\_intention}}$$

$$\frac{step \in \mathcal{B} \quad \chi(P) \in \mathcal{B} \quad P \in \Gamma \quad \langle \mathcal{B}, P\rangle \rightarrow_{p_2} \langle \mathcal{B}', P'\rangle}{\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow_{p_2} \langle E^e, (\mathcal{B}' \setminus \{step, \chi(P)\}) \cup \{idle, no\_intention\_selected\}, (\Gamma \setminus \{P\}) \cup \{P'\}\rangle} \ \mathbf{A}^{\mathbf{p}}_{\mathbf{progress\_intention}}$$

The $\mathbf{A}^{\mathbf{p}}_{\mathbf{select\_intention}}$ rule states that, if the belief base contains the token *step* (indicating readiness to progress an intention) and no intention is currently selected, then we may probabilistically select an intention, P, based on a distribution, μ, provided by the selection strategy, $\mathcal{S}_i^p$. The token, $\chi(P)$, is added to mark that P is now selected, and *no_intention_selected* is removed. Meanwhile, the $\mathbf{A}^{\mathbf{p}}_{\mathbf{progress\_intention}}$ rule models the progression of a selected intention. If P has been selected (*i.e.* $\chi(P) \in \mathcal{B}$) and a probabilistic step can be applied to it, then P progresses to a new state, P', and the belief base is updated accordingly: the token, $\chi(P)$, is removed, the agent returns to the *idle* state, signalling readiness for the next agent-level operation, and *no_intention_selected* is added back to signal readiness for the next intention selection. Together, these rules implement a theoretically clean and practically feasible scheme for managing probabilistic intention selection and progression while in alignment with DTMC representation.

By contrast, removing an unprogressable intention is already an atomic operation. Here, we can extend the A_{update} rule to the probabilistic version as follows:

$$\frac{update \in \mathcal{B} \quad \mathcal{S}_i^p(E^e, \mathcal{B}, \Gamma) = \mu \quad P \in \Gamma \quad \mu(P) = p_1 \quad \langle \mathcal{B}, P\rangle \nrightarrow_1}{\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow_{p_1} \langle E^e, (\mathcal{B} \setminus \{update\}) \cup \{idle\}, \Gamma \setminus \{P\}\rangle} \ \mathbf{A}^{\mathbf{p}}_{\mathbf{update}}$$

The $\mathbf{A}^{\mathbf{p}}_{\mathbf{update}}$ rule re-uses the same function of $\mathcal{S}_i^p$ to select one intention from any present intentions. However, instead of updating the selected intention with the evolved form (as in the rules to progress an intention as per intention-level rules), it simply removes it from the intention set with the probability 1. Here, we write $\langle \mathcal{B}, P\rangle \nrightarrow_1$ to denote that no intention-level step is enabled from $\langle \mathcal{B}, P\rangle$ (*i.e.* P is blocked/terminated).

Finally, we show how to provide the probabilistic version of intention-level rules. Here, we only look at the *act* rule to execute an action to illustrate how it can be done. A full description of the new intention-level semantics can be found in the Appendix A, and we denote the full probabilistic extension of Can as Can^p from now on without causing any confusion. To allow uncertain action outcomes, we can sample outcomes based on a probability distribution, *i.e.* with an action outcome function: $\mathcal{S}_a^p : \Lambda \rightarrow Dist(2^{\overline{\mathcal{B}}} \times 2^{\overline{\mathcal{B}}})$. A probabilistic action execution is defined by:

$$\frac{\mathcal{S}_a^p(act) = \mu \quad \mu(\langle \phi^-, \phi^+\rangle) = p \quad \mathcal{B} \vDash \psi}{\langle \mathcal{B}, act\rangle \rightarrow_p \langle (\mathcal{B} \setminus \phi^-) \cup \phi^+, nil\rangle} \ \mathbf{act^p}$$

The **act^p** rule allows sampling action outcomes from a probabilistic distribution through the function $\mathcal{S}_a^p$ where the subscript a in $\mathcal{S}_a^p$ stands for actions. Since the intention-level rule, **act^p**, is

nested within the agent-level rule, it only handles the sampling of action outcomes. The agent-level rule then takes that outcome and applies any status changes.

We close this section by clarifying the role of probabilistic CAN in our methodology. CAN is used as the BDI programming language whose syntax and operational semantics provide the basis to develop autonomous agent behaviour. That said, probabilistic CAN is not intended to prescribe that every deployed BDI implementation must perform probabilistic intention selection or probabilistic action-outcome sampling. Rather, probabilistic CAN provides the formal basis for constructing an analysable probabilistic model. The concrete DTMC used by the run-time monitor is then generated by encoding this model in an executable modelling formalism, such as a Bigraph-based encoding in the prior work [3] or PRISM, where the relevant probabilities can be obtained from domain assumptions or modelling choices. Thus, what is required by our methodology is not necessarily a deployed implementation of probabilistic CAN, but a probabilistic model whose states and transitions correspond to the relevant configurations and steps of the agent in CAN, together with run-time observations that can be correlated with the generated DTMC.

4.3 Transforming BDI Behaviours to DTMC

In this section, we formalise the behaviours of a probabilistic BDI agent that is specified in CAN^p as a DTMC and show how to automate this transformation using the results and tools from the previous work [3].

Definition 4.1. Let the set of all possible belief atoms, external events and intentions for a *specific agent* be $\overline{\mathcal{B}}$, $\overline{E^e}$, and $\overline{\Gamma}$ respectively. At each agent step, the belief base (resp. events, intentions) is given as $\mathcal{B} \subseteq \overline{\mathcal{B}}$ (resp. $E^e \subseteq \overline{E^e}$, $\Gamma \subseteq \overline{\Gamma}$). We have a DTMC $\mathcal{D}^{\text{CAN}^p} = \langle S, \overline{s}, P, AP, L\rangle$ such that

- $S = 2^{\overline{E^e}} \times 2^{\overline{\mathcal{B}}} \times 2^{\overline{\Gamma}}$.
- $\overline{s} = \langle E^e_0, \mathcal{B}_0, \Gamma_0\rangle$.
- $P = \{rule \mid rule \in Rules(CAN^p)\}$ where $Rules(\text{CAN}^p)$ denotes the probabilistic semantic rules of CAN^p.
- $AP = \{\langle E^e, \mathcal{B}, \Gamma\rangle \mid E^e \in \overline{E^e}, \mathcal{B} \in \overline{\mathcal{B}}, \Gamma \in \overline{\Gamma}\}$.
- $L = S \rightarrow 2^{AP}$ where $L(s) = \{ap \in AP \mid ap \subseteq s\}$.

Definition 4.1 says that the behaviour space of a probabilistic BDI agent in CAN^p is all possible combinations of each element in the agent-level configuration. The initial state in such a DTMC corresponds to the initial agent-level configuration of such an agent. The probabilistic transition is captured by the set of semantic rules in CAN^p, which take a given agent-level configuration to the next agent-level configuration with a probability. In principle, the set of atomic propositions can be the snapshot of any agent-level configuration. An agent-level configuration, s, can be labelled with a proposition, ap, which is an agent-level configuration itself, if the former subsumes the latter (*i.e.* $ap \subseteq s$). In detail, each DTMC state $s \in S$ encodes a complete agent-level configuration of the form $\langle E^e, \mathcal{B}, \Gamma\rangle$. By contrast, an atomic proposition, $ap \in AP$, represents a partial agent-level configuration, used to label all DTMC states whose encoded configurations contain it. For a DTMC state s encoding the complete configuration $\langle E^e, \mathcal{B}, \Gamma\rangle$ and an atomic proposition $ap = \langle E^e ap, \mathcal{B}ap, \Gamma_{ap}\rangle$, we write $ap \subseteq s$ when $E^e_{ap} \subseteq E^e$, $\mathcal{B}_{ap} \subseteq \mathcal{B}$, and $\Gamma_{ap} \subseteq \Gamma$. Thus, s is labelled with ap when the complete configuration encoded by s contains the partial configuration described by ap.

Example 4.2. Let the agent initially be in configuration $\langle E^e_0, \mathcal{B}_0, \Gamma_0\rangle$, where $E^e_0 = \{e_1, e_2\}$ and $\Gamma_0 = \{act_1, e_3, act_2, act_3\}$. The DTMC in Fig. 7 illustrates how the agent probabilistically evolves based on the semantics of CAN^p. Each state represents a possible agent configuration, with transitions guided by operational rules and belief token updates. For illustration purposes, we assume that

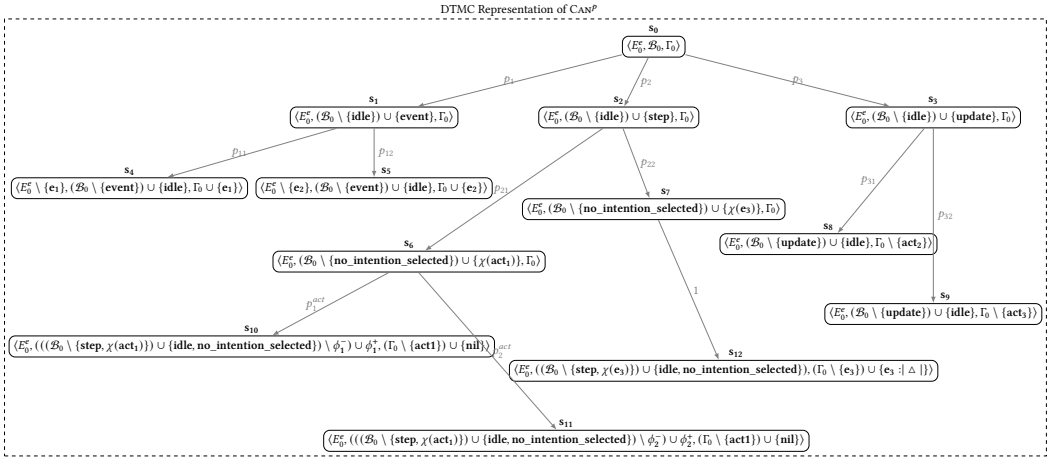


Fig. 7. DTMC Representation of CAN^p with initial configuration $\langle E_0^e, \mathcal{B}_0, \Gamma_0 \rangle$ where $E_0^e = \{e_1, e_2\}$ and $\Gamma = \{act_1, e_3, act_2, act_3\}$. For illustration purposes, we assume that act_1 is not blocked with two probabilistic effects, e_3 is ready for relevant plan collection, and act_2 and act_3 are blocked (*e.g.* preconditions unmet).

act_1 is not blocked with two probabilistic effects (*i.e.* p_1^{act} and p_2^{act}), e_3 is ready for relevant plan collection, which is deterministic, and act_2 and act_3 are blocked (*e.g.* preconditions unmet).

From the initial state, $\mathbf{s_0}$, the agent probabilistically chooses one of three agent-level operations: handling an event (with probability p_1), selecting an intention (p_2), or performing an intention update (p_3), leading to $\mathbf{s_1}$, $\mathbf{s_2}$, and $\mathbf{s_3}$ respectively. From $\mathbf{s_1}$, an event can be selected (either e_1 or e_2), resulting in a new intention and returning to *idle* status ($\mathbf{s_4}$ and $\mathbf{s_5}$) with the probability p_{11} and p_{12}, respectively. In $\mathbf{s_2}$, the agent selects a progressable intention, *e.g.* act_1, resulting in state $\mathbf{s_6}$ with the probability of p_{21}, by replacing *no_intention_selected* with $\chi(act_1)$ in the belief base. Selected intentions then progress. In the state $\mathbf{s_6}$ where act_1 is chosen, one of two probabilistic effects of act_1 applies ($\mathbf{s_{10}}$ and $\mathbf{s_{11}}$) with the probability of p_1^{act} and p_1^{act}. If e_3 is chosen in the state $\mathbf{s_7}$, a set of relevant plans is deterministically constructed ($\mathbf{s_{12}}$), hence with the probability of 1. Each case concludes by restoring *idle* and *no_intention_selected* when progressing an intention. In $\mathbf{s_3}$, the agent chooses to update unprogressable intentions, and in this case, we remove two blocked actions act_2 and act_3 with the probabilities p_{31} and p_{32}, respectively. We highlight that belief tokens such as *step*, *idle*, $\chi(P)$, and *no_intention_selected* are useful to manage internal control and enforce mutual exclusion in intention selection. The DTMC captures both the probabilistic and sequential aspects of agent behaviour in a way that is directly representable in DTMCs and semantically grounded in BDI reasoning.

We note that the probabilistic CAN semantic rules either transition with probability 1 or through probability distributions. These probability distribution functions, *e.g.* S_{ao}^p, are abstract at the semantic level: they identify where probabilistic choices occur, but they do not by themselves prescribe how the numerical probabilities are obtained.

In our methodology, probabilities are part of the design-time model. They may be supplied directly by the modeller, estimated from empirical data, derived from domain knowledge, or computed from explicit modelling assumptions. These probabilities must then be represented in a concrete executable model so that the DTMC can be generated. This is the sense in which we refer to the choice of a modelling language or formalism: the probabilistic CAN semantic rules define the abstract probabilistic transition structure, but an executable encoding is still needed for exhaustive

state-space exploration and DTMC export. For example, this executable encoding may be provided by the Bigraph-based encoding used in [3], or by another probabilistic modelling formalism such as PRISM.

In such a concrete encoding, additional syntax may be needed to instantiate the abstract probability distribution functions that are associated with agent-level operation selection, event selection, intention selection, plan selection, and action outcomes. To obtain concrete probability distributions from the abstract selection functions introduced above, we use *situation value functions* [46]. A situation value is a non-negative score that is assigned to an enabled choice, such as a plan, event, intention, or agent-level operation, based on the current agent configuration. These scores are not probabilities themselves; rather, they are unnormalised weights that are normalised across the currently enabled alternatives to obtain the corresponding probability distribution. Thus, choices with higher situation values are selected with higher probability, while the absolute scale of the values is less important than their relative magnitudes.

Formally, a situation value function can be viewed as a mapping $\theta : \mathcal{S} \to \mathbb{R}_{\geq 0}$ from the current situation/state to a non-negative value. In this paper, we use two common forms. First, a guarded-sum description has the form $\langle d_0, \{(\varphi_1, d_1), \ldots, (\varphi_n, d_n)\}, f \rangle$, where d_0 is a default value and the values d_i are aggregated using f (e.g. *sum*) whenever the corresponding guard φ_i holds in the current belief base, i.e. $\mathcal{B} \models \varphi_i$. Second, a situation value can be given directly as an arithmetic expression over state variables, such as deadlines or progress counters. Both forms return unnormalised values, which are then normalised over the available alternatives. We will use the smart manufacturing case study in Section 5 to show how these situation value descriptions are instantiated for concrete event, plan, intention, and agent-level operation selection.

Finally, since our focus is to showcase quantitative monitoring for BDI agents, we re-use the results and tools of [3] to automate the generation of the DTMC for probabilistic BDI agents. This route takes a specification of CAN^p, including the probability information that is required by the selected concrete encoding, explores all possible behaviours of the agent, and outputs the DTMC used in the Design Time process in Fig. 4. At run time, the monitor does not infer these probabilities from the observed trace; instead, it updates the generated DTMC according to the execution that has actually been observed. Furthermore, we will reflect the importance of the fact that successful formal analysis (in our case, quantitative monitoring) will always need to go hand-in-hand with good formal modelling in the first place.

4.4 Run-time DTMC Update

Our methodology is centred on a design-time DTMC that is carried into operation and updated using run-time observations. This DTMC is the core artefact: it can be created in any developer-chosen manner (*e.g.* from domain knowledge, data, or existing models), and our monitoring/update procedure is agnostic to how the DTMC was obtained, provided it faithfully represents the system's probabilistic behaviour at the level of interest.

We clarify that the monitor is not learned from data. Rather, it is applied as a run-time component that is configured with the design-time DTMC, observed monitor events, and the property or query of interest. During execution, the monitor observes a monitor-event trace, matches the observed events to DTMC transitions, revises the DTMC accordingly, and then invokes probabilistic model checking on the updated DTMC.

We now consider how our methodology can conduct the quantitative operational monitoring for a given DTMC. Often, a DTMC remains fixed when used for system verification at design time, capturing all potential behaviours from a pre-determined initial state. However, we are interested in run-time verification, where the current state of the system in operation may differ from the initially fixed state. As such, we need to reflect such operational state changes in the DTMC. The

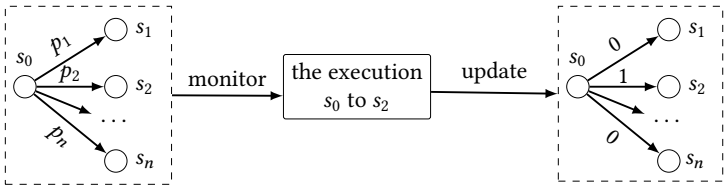


Fig. 8. On the left, a state s_0 has the probability p_i transitioning the state s_i ($i \in \{1, \cdots, n\}$). In the middle, the transition from s_0 to s_2 is observed. To reflect this change in the DTMC, the probability from the state s_0 to s_2 is known to be 1 and the rest of the probability from s_0 to s_j ($j \in \{1, 3, \cdots, n\}$) now becomes 0.

key to our methodology is to reflect the system's run-time operational data that is observed by the monitor in the DTMC generated at the design time. To do so, it essentially requires the revision of such a DTMC. In this section, we will first look at the non-loop (a.k.a. sequential) revision in a DTMC and examine how to perform loop revision afterwards.

4.4.1 Non-Looping/Sequential Revision in a DTMC. We first look at the non-looping/sequential revision in a DTMC, which is depicted in Fig. 8. The system evolves according to its probabilistic behaviours *i.e.* DTMCs. Let us assume that an execution is detected from the state s_0 to s_2 in Fig. 8. To reflect such execution and current state change in the pre-generated DTMC, we update the DTMC such that the probability of transitioning from s_0 to s_2 is known to be one, and it now has zero chance of transitioning from s_0 to any other states, as shown in the right-hand side of Fig. 8. We note that our approach presupposes that the initially generated DTMC at design time accurately represents the probabilistic dynamics of the system in question. We use the term *non-looping* (or *sequential*) to denote the case where the monitored execution moves to a state that has not been previously visited in the current trace. The update then affects only the outgoing distribution of the current state: the observed transition is assigned the probability value 1, while the other outgoing transitions from that state are assigned probability 0. The rest of the DTMC remains unchanged, so unreached states retain their design-time probabilities. Looping executions require separate treatment, because returning to a previously visited state may invalidate an earlier 1/0 commitment for that state's outgoing distribution; this is addressed next.

We also note that this update should not be interpreted as a statistical estimation of transition probabilities from repeated visits. The monitor does not learn empirical transition frequencies. Instead, it conditions the DTMC on the concrete transition that has been observed in the current execution. Thus, if the current visit to a state s is observed to move to s', the outgoing distribution of s is updated to assign probability 1 to s' for this trace-specific revision, and probability 0 to the alternatives. If the execution later returns to s, this is handled by the to-be-introduced loop-revision mechanism, which restores the relevant design-time probabilities before processing the next observed transition from s.

4.4.2 Loops Revision in a DTMC. Fig. 8 showed the simple non-looping case but loops are fundamental in complex probabilistic systems. Fig. 9 illustrates how to reset the transition probabilities of all of the descendant nodes of the state where a looping transition occurs. The key point is that a loop can bring the execution back to a state that has been visited before. If we kept the previous visit's 1/0 update, then returning to that state would inherit the earlier choice and would no longer represent the behaviour for this new visit. Therefore, when a loop transition returns to s_0 in Fig. 9 (and similarly for any loop entry state), we reset s_0 and all of its descendants to the design-time DTMC, so monitoring continues from the fresh start on each new visit.

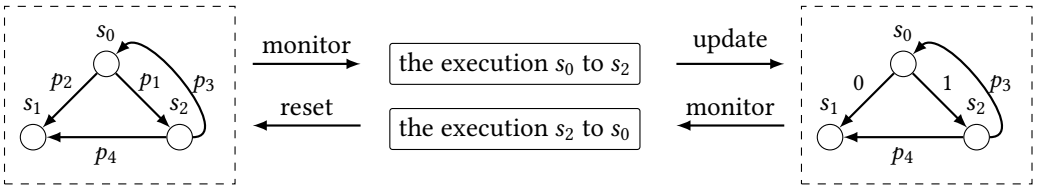


Fig. 9. On the left, a state s_0 has the probability p_i transitioning the state s_i ($i \in \{1, 2\}$). The non-loop transition s_0 to s_2 is observed and updated to the right according to our previous non-loop revision. When a loop transition s_2 to s_0 occurs, we *reset* the probability of any transitions from s_0, including all of its descendants, to the original DTMC generated at the design time.

4.5 Monitor Algorithm

We now present the algorithm and provide a protocol of our methodology for monitoring probabilistic properties in a given DTMC. Algorithm 1 takes three input parameters: the DTMC representing all of the possible behaviours of a probabilistic system, the PCTL formula (including query-based formula) to verify, and the trace of events (*e.g.* executed path) that are observed through the labelled predicates over the DTMC. This algorithm checks PCTL queries *e.g.* $\mathcal{P}_{=?}$ [Fϕ] from observed events σ (*i.e.* executed operational activities of the agent) at run time.

Algorithm 1 is intended to be invoked whenever new run-time observation events are available. In a typical execution, observations are produced as the BDI agent progresses *e.g.* through event handling and plan selection. The DTMC is then updated according to the newly observed trace prefix. If the agent remains idle, or if no new observation has been received since the previous invocation, there is no new operational information to incorporate and no DTMC update is required. Here, Algorithm 1 focuses on the core DTMC update and run-time analysis once such observations are available.

Algorithm 1 starts by iterating through all of the events (lines 5–19) in the event trace σ. Events are represented as labelled predicates from one state to another, *i.e.* $\mu_s(s') \neq 0$ and $L(s') = \sigma_i$. For each event, σ_i, and the current state, s, the monitor identifies the new state, s', to which the system transitions based on the trace of events (line 7). Note that we assume the correctness of the DTMC as a representation of the running system. Therefore, the observed events will always be consistent with the probabilistic model (*i.e.* if σ_i is observed in s, we have s' such that $L(s') = \sigma_i$ and $\mu_s(s') \neq 0$). For clarity, Algorithm 1 is presented for the nominal case in which the formal model accurately represents the monitored system and each observed monitor event matches an enabled DTMC transition. This consistency is a prerequisite for applying the DTMC update. A violation of this consistency may arise from an incomplete or inaccurate modelling step at design time; in such a case, the monitor cannot apply the update.[3]

Once the next state, s', is identified, depending on whether it is a loop execution, we apply the corresponding revision strategies from Figs. 8 and 9. The extra procedures for finding loops and refreshing the DTMC are presented in Algorithms 2–4. After processing the available events in the event trace σ, the updated DTMC can be analysed through standard probabilistic model checking (line 20), and the results of the model checking are returned (line 21).

Theorem 4.3. *The algorithm described in Algorithm 1 terminates in polynomial time with respect to the size of the DTMC and the length of the input event trace.*

[3]This mismatch-handling case is omitted for readability, as it does not pertain to the core update procedure.

Algorithm 1 Monitor($\langle S, \bar{s}, P, AP, L\rangle$, φ, σ)

1: $s = \bar{s}$ ▷ Set DTMC initial state as current state
2: $i = 0$ ▷ Set index of events in trace to 0
3: $InitialDist = Dist$ ▷ Store the initial probability distribution $Dist$
4: $loops$ = FindLoops($\langle S, \bar{s}, P, AP, L\rangle$) ▷ Get all loops in the DTMC
5: **while** $i < |\sigma|$ **do** ▷ For each event in the trace σ
6: **for** $s' \in S$ **do** ▷ For each state in the DTMC
7: **if** $\mu_s(s') \neq 0$ **and** $L(s') = \sigma_i$ **then** ▷ transition & trace
8: $s'' = s'$ ▷ Store the state in auxiliary variable
9: **break**
10: **end if**
11: **end for**
12: **if** $s'' \in$ $loops$ **then** ▷ Check if s'' is a loop state
13: Refresh(s'', P, $InitialDist$) ▷ Loop update in Fig. 9
14: **else**
15: $\mu_s = \mu'_s$ s.t. $\mu'_s(s'') = 1$ and $\mu'_s(s''') = 0$ for $s''' \neq s''$ ▷ Non-loop update in Fig. 8
16: **end if**
17: $s = s''$ ▷ Set s'' as the new current state
18: $i = i + 1$
19: **end while**
20: $output = Analyse(DTMC, \varphi)$ ▷ PRISM or STORM
21: **result** $output$ ▷ quantiative insights *e.g.* $\mathcal{P}_{=?}$ [$\mathbf{F}\phi$])

Algorithm 2 FindLoops($\langle S, \bar{s}, P, AP, L\rangle$)

1: $visited = \emptyset$ ▷ Initialise the set of visited states to the empty set
2: $loops = \emptyset$ ▷ Initialise the set of loops states to the empty set
3: FindLoopsAux($\bar{s}$, $visited$, $loops$) ▷ Call auxiliary procedure to find loops
4: **return** $loops$

Algorithm 3 FindLoopsAux(s, $visited$, $loops$)

1: **if** $s \in visited$ **then** ▷ s has been visited previously, so s is a loop state
2: **add** s **in** $loops$ ▷ Add s to the set of loops states
3: **else** ▷ otherwise, it is the first time s is visited
4: **add** s **in** $visited$ ▷ Add s to the set of visited states
5: **for** $s' \in S$ **do** ▷ For each state s' in the DTMC
6: **if** $P(s)(s') \neq 0$ **then** ▷ reached by s
7: FindLoopsAux(s', $visited$, $loops$) ▷ Find loops starting from s'
8: **end if**
9: **end for**
10: **remove** s **from** $visited$ ▷ Remove s from the set of visited states
11: **end if**

Proof. Algorithm 1 exhibits linear time complexity with respect to the length of the event trace, σ, (lines 5–19) where the algorithm iterates over the trace. In each iteration, Algorithm 1 traverses the states of the DTMC to determine the new state given the currently observed event. This step

Algorithm 4 RefreshMonitor(s, P, $InitialP$)

```
1: for s′ ∈ S do                          ▷ For each state s′ in the DTMC
2:     if s′ is reachable from s then     ▷ reachable from s in one or more steps
3:         P(s′) = InitialP(s′)           ▷ Restore the probability distribution of s′
4:     end if
5: end for
```

has linear complexity for the size of the DTMC. After the while loop, on line 20, probabilistic model checking is performed. This process is known to have polynomial time complexity concerning the size of the DTMC and linear to the formula [8, 15]. Since both auxiliary procedures present polynomial time complexity as well, we can conclude that Algorithm 1 terminates in polynomial time with respect to the size of the DTMC, the PCTL formula, and the length of the input event trace. □

5 Case Study of BDI agents

We build on a smart manufacturing example from [3]. We extend it with new probabilistic selection strategies and, importantly, show how to provide run-time quantitative operational insights as the agent operates. The models are freely available online[4] using off-the-shelf model checkers, and users can "run" models with different settings, e.g. initial belief base, external events, plan libraries, and customised situation value functions.

Consider a smart manufacturing setup where a robot selects and packages products which can decay if left unpackaged for too long before transporting them to storage. We first describe the scenario at a high level, then instantiate it as a BDI agent program (Fig. 10) to make the design concrete. Figure 10 provides a compact presentation of the case-study agent design by collecting the initial beliefs, external events, plans, action descriptions, and situation value functions in one place. This notation is intended as a concise presentation format rather than a standalone input language with a dedicated compiler. According to the methodology in Fig. 4, the resulting probabilistic BDI agent design is then modelled in a suitable probabilistic specification language from the previous work [3], using the semantic rules introduced in Section 4.2. We explain the main components of the design below.

To package products, the robot must choose between premium and standard wrapping. Premium wrapping is costly but always prevents decay and never fails, while standard wrapping is cheaper, only works when the product has not been left unwrapped for too long, and may break (an adverse outcome). To avoid implying any first-principles thermal modelling, we represent perishability as a discrete *deadline* counter using *temporal agent time*: for each product i, $de_i \in \mathbb{N}$ on line 2 in Fig. 10 denotes the maximum number of agent steps that this product i can wait before being considered spoiled, and de_i decreases by 1 after each agent-level step. By observing the robot's decisions at run time, one can assess success probabilities and intervene if necessary to manage risk. A short agent program managing this process with two products is provided in Fig. 10 with the following commentary.

Lines 4 and 5 give the external events of products awaiting processing *e.g.* `e_product1` with situation value function θ_{13} (explained below). The agent uses a declarative goal (line 7) to achieve `success1` (wrapped and moved), responding to `e_process_product1`, but it fails if `failure1` (dropped or decayed) becomes true. Two plans (lines 8–9) implement wrapping choices, each with different situation value functions. Event `e_product2` is handled similarly (lines 10–12). Actions

[4]https://github.com/Mengwei-Xu/Smart_Manufacturing_Monitoring

```
1  // initial_belief_base
2  de_1 = 10 ,  de_2 = 14
3  // External events
4  e_product1 [θ_13]
5  e_product2 [θ_23]
6  // Plans
7  e_product1 : true ← goal(success1,e_process_product1,failure1)
8  e_process_product1 : φ_11 ← wrap_standard1; move_product_standard1 [θ_11]
9  e_process_product1 : φ_12 ← wrap_premium1; move_product_premium1 [θ_12]
10 e_product2 : true ← goal(success2,e_process_product2,failure2)
11 e_process_product2 : φ_21 ← wrap_standard2; move_product_standard2 [θ_21]
12 e_process_product2 : φ_22 ← wrap_premium2; move_product_premium2 [θ_22]
13 // Action descriptions
14 wrap_standard1 = true ← [⟨φ_1^- = ∅, φ_1^+ = {product_1_packed}⟩ ↦ 1]
15 move_product_standard1 = product_1_packed ←
       [⟨φ_2^- = ∅, φ_2^+ = {success1}⟩ ↦ 0.9, ⟨φ_2^- = ∅, φ_2^+ = {failure1}⟩ ↦ 0.1]
16 wrap_premium1 = true ← [⟨φ_1^- = ∅, φ_1^+ = {product_1_packed}⟩ ↦ 1]
17 move_product_premium1 = product_1_packed ← [⟨φ_2^- = ∅, φ_2^+ = {success1}⟩ ↦ 1]
18 wrap_standard2 = true ← [⟨φ_3^- = ∅, φ_3^+ = {product_2_packed}⟩ ↦ 1]
19 move_product_standard2 = product_2_packed ←
       [⟨φ_4^- = ∅, φ_4^+ = {success2}⟩ ↦ 0.9, ⟨φ_4^- = ∅, φ_4^+ = {failure2}⟩ ↦ 0.1]
20 wrap_premium2 = true ← [⟨φ_3^- = ∅, φ_3^+ = {product_2_packed}⟩ ↦ 1]
21 move_product_premium2 = product_2_packed ← [⟨φ_4^- = ∅, φ_4^+ = {success2}⟩ ↦ 1]
22 A_event [θ_31]
23 A_step [θ_32]
24 A_update [θ_33]
```

where we have $\varphi_{x1} = de_x \geq 3$, $\varphi_{x2} = de_x \geq 0$, $\varphi_{x3} = 3 \geq de_x \geq 0$, $\theta_{x1} = \{1, \{(\varphi_{x1}, 1)\}, sum\}$, $\theta_{x2} = \{1, \{(\varphi_{x3}, 1)\}, sum\}$, $\theta_{x3} = (de_x + pr_x)^{-3}$, and $\theta_{3y} = (4 - y) \cdot (pr_1 + pr_2)^y$ such that $x \in \{1, 2\}$, $y \in \{1, 2, 3\}$. And de_i is the deadline of product i, and pr_i denotes the current progress counter for product i.

Fig. 10. Compact presentation of the smart manufacturing BDI agent using the notation introduced in Section 3.1. Lines 1–2 define the initial belief values, lines 3–5 define the external events, lines 6–12 define the plans for handling the two products, lines 13–21 define the probabilistic action outcomes, and lines 22–24 define situation value functions for agent-level operation selection. The notation is used for presentation only; the main components are explained in the surrounding text.

are described on lines 14–21. For example, `move_product_standard1` has a 10% chance to fail, whereas `move_product_premium1` always succeeds. Probabilistic distributions depend on two discrete agent-time variables: progress, which records how far an intention has advanced, and deadline, which records the remaining agent steps before spoilage. In practice, the initial values of deadlines can be derived from temperature logs via an application-specific mapping; here we assume that they are given and focus on the agent reasoning rather than thermal dynamics.

We now explain the relevant quantitative details, in particular, by instantiating the situation value descriptions for the smart manufacturing example. On line 2, $de_1 = 10$ and $de_2 = 14$ are the initial perishability deadlines (in agent steps) of `e_product1` and `e_product2`, respectively, *i.e.* the maximum number of agent steps that each product can wait before being spoiled. The precondition $\varphi_{11} = de_1 \geq 3$ checks whether de_1 is greater than or equal to 3. The situation value specification $\theta_{11} = \langle d, S, agg \rangle$ consists of (i) a default value $d = 1$, (ii) a set of value specification pairs $S = \{(\varphi_{11}, 1)\}$ of weighted values, and (iii) an aggregation function $agg = sum$. Its evaluated

Agent-level Operation Selection Strategies	Event and Intention Selection Strategies	Plan Selection Strategies
ProD: Progress Distribution	**UD**: Urgency Distribution **CUD**: Conditioned Urgency Distribution **OCUD**: Optimised Conditioned Urgency Distribution	**PreD**: Preference Distribution **SMP**: Select Most Preferred

Table 1. Selection strategies.

value is obtained by aggregating the weights of all conditions in S that hold in the current belief base together with the default value. For example, if φ_{11} holds, then θ_{11} evaluates to $1 + 1 = 2$. In other words, the purpose of the situation value function is to facilitate the calculation of a plan value which can be obtained dynamically based on a specific situation. As another example, the plan on line 9 in Fig. 10 has the context, φ_{12} *i.e.* as long as the deadline is greater than zero. However, the corresponding situation value, $\theta_{12} = \{1, \{(\varphi_{13}, 1)\}, sum\}$, will have an increased evaluated value when $\varphi_{13} = 3 \geq de_x \geq 0$ holds, *i.e.* only when it is closely approaching the deadline.

As another example of arithmetic expression for situation value, for the external event, `e_product1`, we define $\theta_{13} = (de_1 + pr_1)^{-3}$, where de_1 is the remaining perishability deadline (in agent steps) and pr_1 is the current progress counter for `e_product1`. As $de_1 + pr_1$ decreases (*i.e.* the product is more urgent with small progress counters and has a smaller deadline left), θ_{13} increases rapidly, prioritising products that are close to spoiling and are insufficiently progressed. The exponent is a modelling parameter controlling the sharpness of the normalised probability distribution. It therefore affects the quantitative probabilities assigned to different selections, but not the run-time quantitative operational monitoring method itself. The value -3 was chosen to make urgency differences sufficiently visible in this case study: larger absolute exponents make the distribution more strongly favour more urgent options, while smaller absolute exponents produce flatter distributions.

Similarly, there are situation value functions for agent-level rules (lines 22–24), ensuring that A_{event} is more likely to be applied early on when $pr_1 + pr_2$ is small (*i.e.* relatively low overall progress). As $pr_1 + pr_2$ grows, the distribution shifts preference to A_{step} and A_{update}, with A_{update} eventually having the highest value. These values are normalised automatically to determine the rule selection probabilities by the selected off-the-shelf formal modelling tools.

To illustrate our approach in action, we show how our quantitative monitoring approach can support run-time quantitative analysis for both existing and new probabilistic selection strategies in this smart manufacturing example.

We adopt the selection strategies given in Table 1. These strategies are domain-instantiated rather than domain-independent: they use manufacturing-specific features such as progress counters, deadlines, and wrapping preferences. More specifically, each strategy uses such features to rank or weight the available choices (operation, intention, or plan). If probabilistic decision-making is used, these weights are normalised to obtain a probability distribution over the available choices. If deterministic decision-making is used, domain features can instead be used to impose an ordering over the applicable choices. The general process is therefore to identify domain-relevant features for a choice, encode them either as a deterministic ordering or as situation value functions, and then evaluate the resulting strategies quantitatively under common event traces. We choose the **ProD** (Progress Distribution) strategy that selects an agent-level operation from a probability distribution that is based on the current progress of the agent (lines 22-24). Intuitively, **ProD** makes the agent

(ProD, UD, SMP)	(ProD, CUD, SMP)	(ProD, OCUD, SMP)	Event Trace
0.0548	0.4829	0.9046	[]
0.0548	0.4829	0.9046	[A_{event}]
0.0461	0.3875	0.8278	[A_{event},e_product2]
0.0592	0.5101	0.9693	[A_{event},e_product2, A_{event}]
0.0592	0.5101	0.9693	[A_{event},e_product2, A_{event}, e_product1]

Table 2. Quantitative operational monitoring of two products eventually completed with success under existing selection strategies from [3].

(ProD, UD, PreD)	(ProD, CUD, PreD)	(ProD, OCUD, PreD)	Event Trace
0.0551	0.4869	0.9149	[]
0.0551	0.4869	0.9149	[A_{event}]
0.0462	0.3894	0.8338	[A_{event},e_product2]
0.0595	0.5133	0.9794	[A_{event},e_product2, A_{event}]
0.0595	0.5133	0.9794	[A_{event},e_product2, A_{event}, e_product1]

Table 3. Quantitative operational monitoring of two products eventually completed with success under three sets of new probabilistic selection strategies *not* considered in [3].

more likely to adopt external events as intentions when overall progress is low, and more likely to progress already adopted intentions as overall progress increases.

For event/intention, we choose three different selection strategies. The **UD** (Urgency Distribution) strategy selects an intention by sampling from a distribution where a situation value function is given by $(de + pr)^{-3}$. This assigns larger values to products with less remaining deadline and/or insufficient progress, so **UD** tends to prioritise products that are closest to spoiling or have lagged behind. The **CUD** (Conditioned Urgency Distribution) strategy only deems an intention to be urgent if the product is not packed or spoiled. This avoids assigning probabilities to intentions that can no longer contribute to achieving the goal. The **OCUD** (Optimised Conditioned Urgency Distribution) strategy selects an intention similarly to **CUD** but the situation value description is revised to be $|de + pr - steps\_expected|^{-3}$, which accounts for the steps remaining to pack a product (to avoid spoilage) where $steps\_expected$ is a domain knowledge that is accessible. In practice, **OCUD** reduces the chance of repeatedly selecting products that are already hopeless (too late) or trivially safe, focusing instead on products that are "just-in-time" and therefore most sensitive to delay.

Finally, we have **PreD** which selects a plan by sampling the distribution based on the situation values of plans whereas **SMP** always prefers the cheapest bag if possible (in this case the standard wrapping). That is, **PreD** is probabilistic (weighted by plan values), while **SMP** is deterministic and always chooses the minimum-cost option when applicable.

In comparison with [3], we note that Table 2 uses exactly the three strategy sets evaluated in [3], and therefore provides a direct baseline comparison. The key difference is that [3] reports strategy quality from a fixed initial model/design-time perspective, whereas we compute the same success property conditionally on the observed execution trace and update the probability online as the trace grows. Meanwhile, Table 3 uses entirely new strategy sets that are not included in [3], to show that the monitoring mechanism is strategy-agnostic while still enabling a quantitative comparison between strategy families at run time.

Table 2 shows the monitored probability of eventually successfully completing two products under three sets of existing selection strategies from [3]. Initially, with an empty event trace (no agent progress), each strategy set provides distinct predictions, with **(ProD, OCUD, SMP)**

appearing to be the most promising. As the system advances, the event trace becomes longer. In the beginning, the agent only has the operation choice of applying the A_{event} agent-level operation (probability of 1 of applying A_{event}). Hence, the probabilities remain the same even as the event trace becomes [A_{event}]. Next, if the agent selects `e_product2` (the trace becomes [A_{event}, `e_product2`]), all probabilities drop because focusing on the product with a longer deadline raises the risk that `e_product1` will spoil. The updated probability could inform an external engineering decision at this point. For example, an agent engineer may decide that the agent should next handle `e_product1`, corresponding in the model to selecting the A_{event} agent-level operation. We emphasise that our contribution is to quantify the effect of such modelled operational choices; implementing a mechanism that enforces those choices is outside the scope of this paper. As there is only one event left here (*i.e.* `e_product1`), this modelled decision immediately increases the probability even higher than the one from the initial conditions, because it removes the chance of progressing `e_product2` at this step.

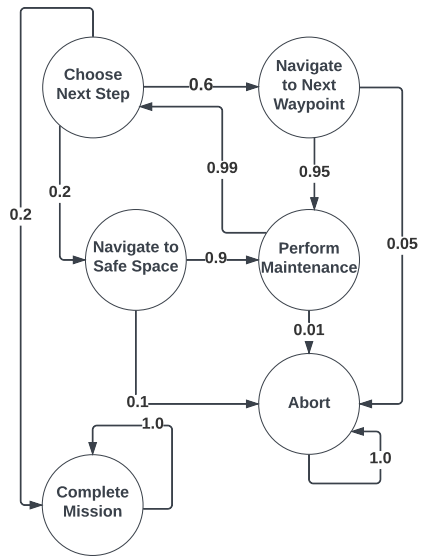
We highlight that our monitoring approach itself does not rely on which selection strategies are used; it simply tracks how the evolving event trace—the agent's progress—affects success probabilities. To illustrate this, Table 3 provides the monitoring results for the same trace under three new strategies that were not included in [3]. As expected, because the same trace is monitored in each case, the results follow a similar trend. This illustrates that the monitor can be applied across different selection strategies, while the resulting probabilities still reflect the trace-updated DTMC generated under each strategy. The difference between Table 2 and Table 3 is that the new strategies are marginally better than the existing strategy. This is due to the probabilistic strategy **PreD** avoiding the over-greedy nature of the non-probabilistic strategy of **SMP** that always selects the standard wrapping (a cost-effective option) that has only marginally higher chance of breaking than the premium counterpart.

6 Case Study of Non-BDI Systems

Although our approach is initially targeted at BDI systems, we recognise the broader benefits and more general applicability that our algorithms can have. To demonstrate the broader feasibility and generality of our methodology, we also apply our methodology to non-BDI-based systems, in particular, an autonomous rover. This case study differs not only in the nature of the system type but also in the use of probabilistic specification languages and model checkers. In addition, we show that it is important to distinguish between the tool's modelling aspect and its analysis capabilities. For instance, PRISM can serve dual roles: as a probabilistic specification language or as an independent model checker. The following sections outline the actual modelling and implementation of such a scenario.

Autonomous rovers, for example Mars rovers such as Perseverance, explore hazardous and unknown environments [23]. Fig. 11 depicts a basic DTMC representing the expected system behaviour for a rover carrying out a remote inspection mission of multiple waypoints. The rover should safely navigate to waypoints and conduct routine self-maintenance. This example is intentionally simplified, and the values chosen for the transition probabilities are for illustrative purposes only; in reality, the system would be more complex. That said, it suffices to demonstrate our approach.

In exploring other planets, rovers face constraints on battery power. To support system verification, run-time monitors may be employed, but they can consume substantial computational and energy resources. This also motivates the use of quantitative run-time information to support decisions about monitoring effort. For example, if the monitor reports a sufficiently high probability of incident-free navigation to the next waypoint, this could provide evidence that intensive monitoring is less urgent for the current mission segment. That said, we emphasise that our current



PCTL Properties:

```
1  P>=0.5[X navigateToNextWaypoint]
2  P<=0.25[F abort]
3  P<=0.2[X (navigateToNextWaypoint)
4         & (F abort)]
5  P>=0.7[F performMaintenance]
```

```
1  module rover
2    chooseNextStep:bool init true;
3    navigateToNextWaypoint:bool init
         false;
4    navigateToSafeSpace:bool init false;
5    abort:bool init false;
6    performMaintenance:bool init false;
7    completeMission:bool init false;
8
9    [] chooseNextStep =true ->
10     0.6 : (navigateToNextWaypoint' =
           true)
11     & (chooseNextStep' = false)
12     + 0.2 :(navigateToSafeSpace' = true)
13     & (chooseNextStep' = false)
14     + 0.2 : (completeMission' = true)
15     & (chooseNextStep' = false);
16   ...
17   [] performMaintenance =true ->
18     0.99 : (chooseNextStep' = true)
19     & (performMaintenance' = false)
20     + 0.01 :(abort' = true)
21     & (performMaintenance' = false);
22 endmodule
```

Fig. 11. Rover example ($dtmc_{rover}$): On the left, a graphical representation of the model is shown, featuring 6 states and 11 transitions (with loops), each displaying the probability of taking a specific transition. Below, PCTL properties are presented. On the right, the PRISM model of this simple rover is provided.

model does not itself implement automatic monitor activation or deactivation. Rather, we treat such decisions as a possible future use of the quantitative information produced by our approach, not as functionality required in this paper.

6.1 Modelling in PRISM and Verification with STORM

A snippet of our PRISM model for this is given on the right of Fig. 11. We model whether each state has been reached using a boolean variable named after that state. Each such variable is initialised to `false`, except for `chooseNextStep`, which denotes the initial state and is initialised to `true`. Note that PRISM denotes post-transition variable values using primed names.

On the bottom left of Fig. 11, we included some of the properties that we wish to verify of this rover. The first is that the probability of the next state being `navigateToNextWaypoint` is greater than or equal to 0.5. The second states that the probability of the rover aborting in the future should be less than or equal to 0.25. The third one states that the probability of eventually reaching `abort` when the next state is `navigateToNextWaypoint` is less than or equal to 0.2. Finally, we want to ensure that the rover reaches `performMaintenance` with a high probability (greater than or equal to 0.7). All of these properties return true when we verify the model against them. This, of course,

evaluates the properties from the initial state and not while the system state is evolving as it does during execution.

6.2 Monitoring at Run Time

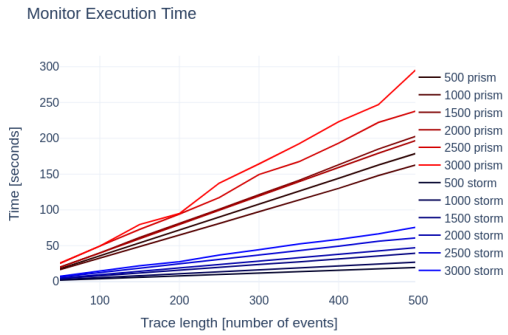
We begin with this example in which the running system moves from the state of `chooseNextStep` to the state of `navigateToNextWaypoint`. The first two properties become false. For the first, the state `navigateToNextWaypoint` cannot be reached from itself at the next time step. The second is also violated due to an increased probability from 0.22 to 0.27 of reaching `abort`. Importantly, this is an *eventual* reachability probability: **F** `abort` specifies reaching `abort` at any future step, not only the direct one-step transition. Although there is a direct 0.05 transition to `abort` from `navigateToNextWaypoint`, the overall **F**-probability is larger because the DTMC may loop (*e.g.* via `performMaintenance` back to `chooseNextStep`) and encounter `abort` later. The third property evaluates to true because it's first part **X** `navigateToNextWaypoint` is not true while we are in the `navigateToNextWaypoint` state, hence the probability is indeed less than or equal to 0.2 (it is 0).

Next, consider the fourth property (`P` $\geq_{0.7}$ `[F performMaintenance]`), and an execution trace consisting of the event `navigateToNextWaypoint`. This example illustrates the difference between (i) model checking in the initial state (the standard offline result) and (ii) run-time re-assessment conditioned on what has been observed so far. In the context of Algorithm 1, we invoke `Monitor(`$dtmc_{rover}$`,` φ`,` σ`)`, where $dtmc_{rover}$ is the DTMC in Fig. 11, φ is any of the PCTL properties in Fig. 11, and σ represents the events that are observed by the monitor. In particular, $\sigma = \langle$`navigateToNextWaypoint`$\rangle$ indicates that the monitor has observed the transition out of `chooseNextStep` into `navigateToNextWaypoint`. In this case, when calling the monitor function (Algorithm 1), we update the DTMC probability distribution in a non-loop fashion by assigning a probability of 0 to the transition from the `chooseNextStep` to *e.g.* `navigateToSafeSpace`, while assigning a probability of 1 from `chooseNextStep` to `navigateToNextWaypoint`. This non-loop update can be understood as conditioning the branching at `chooseNextStep` on the observed outcome, *i.e.* eliminating the alternatives that are inconsistent with the trace. We then perform the model-checking step on this revised DTMC. Importantly, the new result probability differs from the one that is obtained by performing the verification in the initial state, increasing from 0.75 (in the initial state) to 0.95 (in the new state). After moving into the `navigateToNextWaypoint` state, the probability of eventually reaching `performMaintenance` is dominated by the outgoing transition from `navigateToNextWaypoint` to `performMaintenance` (0.95), whereas the initial state of `chooseNextStep` also accounts for other branches from `chooseNextStep` (*e.g.* paths that complete the mission without visiting `performMaintenance`). At such a high likelihood, a decision may be made to conserve battery power and discontinue further monitoring of this property. More generally, monitoring can identify when a property has become almost surely satisfied (or violated) under the current execution context, enabling adaptive resource allocation (*e.g.* reducing further checks when additional information is unlikely to change the decision). This kind of insight is a clear benefit to monitoring the probabilistic behaviour during execution.

6.3 Experimental Analysis

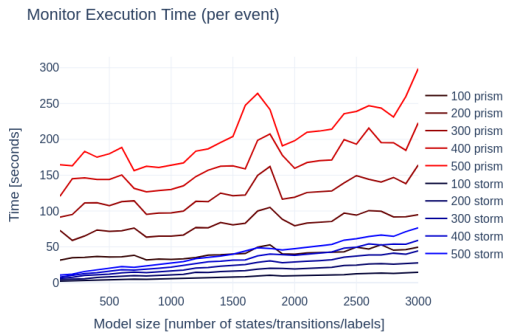
We have given two detailed case studies above. However, to stress-test the computational costs of the crucial monitoring step, independent of particular BDI designs and types of systems, ensuring its broader applicability, we conducted experiments with randomly generated, synthetic parameterised DTMCs. We have implemented our quantitative operational monitoring as a prototype in Python (version 3.10.12), incorporating the described algorithms.[5] We focused on assessing it against some

[5]https://github.com/AngeloFerrando/PRVPrototype

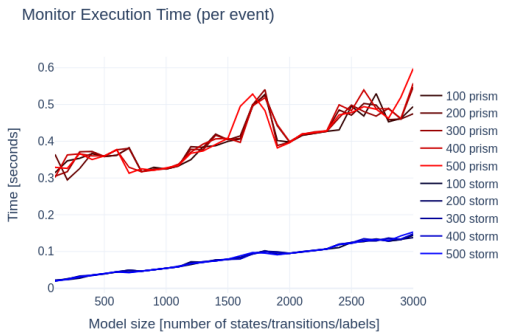


(a) Monitor execution time regarding a number of events to analyse.

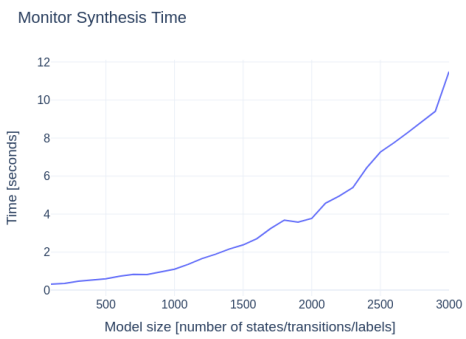
(b) Monitor execution time per event regarding a trace of events to analyse.



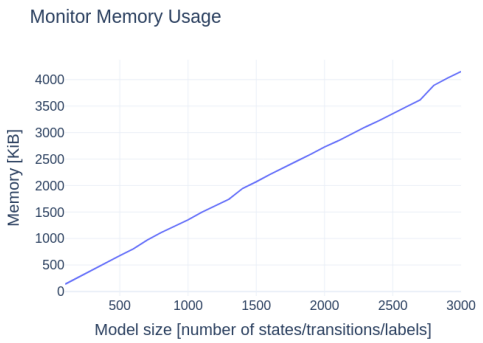
(c) Monitor execution time regarding the size of the probabilistic model.



(d) Monitor execution time per event regarding the size of the probabilistic model.



(e) Monitor synthesis time regarding the size of the probabilistic model.



(f) Monitor memory usage regarding the size of the probabilistic model.

Fig. 12. We consider execution time with respect to the number of events to analyse (12a and 12b) and the size of the generated probabilistic model (12c and 12d). We also examine monitor synthesis time (12e) and memory usage (12f).

standard formal properties,[6] in particular, liveness, which is essential in autonomous systems, *e.g.* $\mathcal{P}_{=?}$ [$F\phi$] that determines the likelihood of ϕ eventually being true. By varying the DTMC size and trace length, we can assess the computational costs and scalability. Event traces were generated using state labels, simulating system operation with events chosen randomly from a uniform distribution for each state.

Our study examines two main aspects of our approach: *synthesis time* (the time taken to generate a monitor from a formal property) and *verification time* (the time that a monitor takes to check both an entire trace of events and individual events). Both are critical for real-time applications, with synthesis being computationally demanding and verification determining the system latency. All of the experimental results given in Fig. 12 were repeated to mitigate the machine noise and the plots display the average times. A short commentary is as follows.

Fig. 12e shows that synthesis time is influenced by DTMC size rather than formula size (which is typical for standard RV), as our method uses probabilistic model checkers. Although our approach eliminates the need to transform the formal properties, it includes a synthesis step based on DTMCs. Fig. 12e indicates that this step exhibits pseudo-quadratic behaviour. Even with a non-negligible number of states (*e.g.* 3000), the total synthesis time remains less than 12 seconds. As our monitors also store transition systems, we show that the memory footprint in Fig. 12f is negligible, staying below 4 MiB even for large models.

Fig. 12a shows the verification time for the entire trace of events, while Fig. 12b displays the verification time per event, indicating the time that the monitor takes to provide its verdict for each query. Fig. 12a demonstrates that our approach grows linearly with the length of the event trace. This is further confirmed by Fig. 12b, where a constant time behaviour is observed to analyse every single event in the trace. Additionally, our prototype is agnostic to the specific model checker that is used. However, we examine the effects of the choice of model checkers in this section. With the same model and formula, STORM outperforms PRISM by completing verification in less than 70s whereas 300s for PRISM. We also conducted experiments from the perspective of the size of the DTMC analysed. Fig. 12c and Fig. 12d illustrate the verification time while fixing the length of the trace and varying the size of the DTMC, both demonstrating linear behaviour. We also observe a local irregularity in PRISM's monitor execution time per event around 1500–2000 states in Fig. 12c and Fig. 12d. Since this behaviour is not mirrored by STORM under the same setting, it can appear to be tool-specific variation rather than a feature of the monitoring algorithm.

7 Threats to Validity

We summarise the following internal and external threats to validity to our experimentation in Section 6.3:

Internal Threats: Our experiments use PRISM and STORM, two state-of-the-art probabilistic model checkers that are widely adopted in the research community. While we believe that the results obtained are broadly generalisable due to the maturity and popularity of these tools, we recognise that emerging model checkers or verification frameworks could yield different performance profiles. In particular, the PRISM results show a local performance irregularity in monitor execution time per event for medium-sized models that is not present in the corresponding STORM results. Importantly, this can suggest that the irregularity is more likely due to tool-specific performance behaviour than to the monitoring algorithm, although identifying the precise cause would require a separate profiling study of PRISM, which is outside the scope of this paper. This reinforces the need to assess

[6]We support any PCTL formula and checking a PCTL formula is linear time in regard to the number of logical connectives and temporal operators for a given DTMC [32].

tool-specific factors in future work, for example, by incorporating a broader range of probabilistic model checkers.

External Threats: While the empirical results presented in Section 6.3 demonstrate the feasibility and scalability of our approach, there remain several opportunities for further improvement. In particular, our current evaluation employed randomly generated DTMCs to systematically assess the various performance perspectives of our monitoring approach. Though effective for controlled experimentation, this poses an external threat to validity because this approach may not fully capture the structural characteristics of real-world systems. Future work will therefore explore more guided or domain-informed generation techniques to increase the representativeness validity of the agent behaviours.

Moreover, the PCTL properties examined in our study were limited to non-nested forms, chosen to provide a focused evaluation of run-time efficiency and responsiveness. A more comprehensive assessment—including complex, nested, and query-based PCTL properties—could offer deeper insights into the expressiveness and computational implications of our method when applied to richer specification languages. In future work, we will assess more complex, nested PCTL properties.

8 Discussion and Future Work

8.1 Modelling Prerequisites

Our goal is to support quantitative run-time analysis of BDI agents. We have shown that this can be effectively achieved via a DTMC representation of the agent's behaviour. Accordingly, we have developed and refined a probabilistic extension of BDI semantics that aligns with the structure of DTMCs. Specifically, each semantic rule in the BDI framework is mapped to a corresponding transition in the DTMC, enabling the complete behaviour of a BDI agent to be captured as a formal, probabilistic transition system.

Once such a DTMC model is in place, quantitative run-time analysis becomes a challenge for monitoring and updating this representation based on the observed execution data. Such a challenge requires coordination between DTMCs, the monitor, and probabilistic model checking tools. To evaluate the general scalability and responsiveness of our approach, we have conducted comprehensive empirical studies in a setting that is intentionally application-agnostic and system-independent. This demonstrates the feasibility, scalability, and generality of our methodology for a broad class of both BDI and non-BDI systems as long as it can be captured as a DTMC. Our results show, that even in stressed scenarios with models having thousands of states, our instantiated methodology reacts in less than 0.15 seconds.

That said, constructing a DTMC representation for any system, in particular, BDI agents, is non-trivial. For example, the main modelling challenge lies in obtaining a faithful model for BDI agents in the first place: designers must choose an appropriate abstraction, encode the agent behaviour in the selected probabilistic specification language, provide suitable probability distributions, and define properties that capture the quantities of interest. Thus, to fully realise the benefits of our monitoring approach, system designers must be equipped with the relevant modelling expertise. We recognise this as a key precondition for the methodology's successful adoption, with formal analysis going hand-in-hand with formal modelling.

8.2 Beyond BDI Agents

Our main goal is to provide run-time quantitative operational monitoring for probabilistic BDI agents, while using DTMCs as the intermediate representation that also makes the methodology applicable to other systems modelled in the same form.

A key design choice is to use a DTMC as an enabling intermediate representation: it makes quantitative checking possible and supports incremental revision of DTMC from run-time observations. In our BDI setting, the DTMC is obtained from CAN^P programs through exhaustive exploration of the agent's behaviour space using suitable formal modelling and analysis languages and tools. Once the DTMC is available, the monitoring and DTMC revision components can be applied independently of the particular route by which the DTMC was constructed.

This also clarifies the scope of the "beyond BDI" claim. The monitoring and DTMC revision components of our pipeline apply whenever (i) the system can be modelled or approximated as a DTMC and (ii) the required events are observable, so executions can be mapped to DTMC states and transitions. This observability requirement is a strong but explicit assumption: the monitor must receive observations at the level of abstraction used for DTMC revision. The observed monitor events need not expose the full internal state of the running system, but they must be sufficient to identify the corresponding transition or successor state in the DTMC. If the available observations are coarser, noisy, or ambiguous, then obtaining a suitable monitor-event trace is outside the scope of this paper; our contribution assumes such a trace and uses it to revise the DTMC. Accordingly, Section 6 (Mars Rover) is included as a feasibility demonstration of the re-usability of these two components when these requirements are met, rather than as a shift in focus away from our main contribution on probabilistic BDI agents.

8.3 Future Work

As summarised in our Threats to Validity in Section 7, there are a number of places where our evaluation could be improved. In future work, we will seek to address these points by (1) providing more generation of DTMCs to evaluate against representative realistic systems, (2) examine more complex and nested PCTL properties and (3) including more novel probabilistic model checkers in our analysis to ensure that our approach is capable with both the state-of-the-art (PRISM and STORM) as well as upcoming tools that are under development. For (1) and (2), we will also perform case studies alongside our industrial partners on other projects.

DTMCs provide an ideal and natural place to start for our approach, but extending to more complex representations, such as Markov Decision Processes (MDPs), would further improve the applicability of our approach. The challenges of such an extension include (1) the formal modelling of BDI agents as MDPs from both a theoretical and practical perspective, (2) extending our monitoring technique for MDPs, (3) the revision of MDPs to reflect the executed agent operations, and, most importantly, the holistic and suitable integration of (1)-(3). For example, in MDPs, the query-based properties become either minimum or maximum probability evaluations for a particular property of interest. This poses a challenge for the revision of the models that we use in our approach and some form of pruning must be performed for the underlying MDP representation. Fortunately, there is some good progress on the challenging aspects of (1) and (2) in [4] for MDP-based BDI agents and MDP monitoring in [40]. That said, putting them together into a coherent framework is not a trivial task and is something that we will investigate in future work.

Taken together, these directions form a natural trajectory for extending the scope, realism, and generality of our methodology, and for increasing its impact across a wider range of probabilistic and autonomous systems.

9 Conclusion

Quantitative operational monitoring allows us to analyse systems that must react quickly under uncertainty and exhibit probabilistic behaviours due to imprecise actuators and decision policies during operation. This enables proactive decision-making and advanced mitigation strategies in autonomous systems based on the likelihood of specific properties being satisfied or violated.

Applications where this approach is beneficial are vast, ranging from situations where we want to invoke mitigations as quickly as possible to avoid hazards to situations where platform resource budgets (*e.g.* compute/energy budgets) mean that we must reduce the number of monitors running at a given time by adaptively selecting which properties to monitor. We treat such budgets as external constraints and do not model the internal dynamics of the underlying subsystem platforms; our focus is solely on agent-level decision-making and run-time quantitative monitoring. These scenarios can greatly benefit from a quantitative approach where if a specific probability threshold is met then we can invoke mitigations before hazardous behaviour occurs and/or turn off monitors if it is highly likely that a given property is satisfied or violated (they can be turned back on later as needed).

To address this challenge, this paper has presented a run-time quantitative operational monitoring methodology for probabilistic BDI agents. In the methodology, it firstly contributes an improved formal presentation of the probabilistic semantics of BDI agents and a formalisation of the resulting probabilistic BDI behavioural space as a DTMC. Then, computational algorithms are given for run-time DTMC updates using observations from the running agent, and for re-analysing the updated DTMC using probabilistic model checking to provide updated quantitative values or threshold verdicts during operation. These algorithms have been formally introduced, practically implemented, and empirically evaluated, confirming the feasibility and scalability of the core monitoring technique. The examples and experiments play complementary roles. The smart manufacturing example illustrates the methodology from probabilistic BDI agents to DTMC-based run-time monitoring. The rover example illustrates that the DTMC-level monitoring mechanism is not strictly tied to BDI agents once an appropriate DTMC has been obtained. Such generality is important to pave the way for detailed quantitative verification of any probabilistic systems that operate in uncertain environments and/or exhibit probabilistic behaviours. Finally, the synthetic parameterised DTMC experiments isolate the computational cost of the monitoring algorithms, including DTMC update and re-analysis, from domain-specific scenarios, allowing scalability to be assessed systematically.

Acknowledgments

This work is partially supported by a Royal Academy of Engineering Research Fellowship, EPSRC grant EP/Y001532/1, and the Centre for Robotic Autonomy in Demanding and Long Lasting Environments (CRADLE) under EPSRC grant EP/X02489X/1. Authors are listed alphabetically to indicate equal contributions.

References

[1] Rajeev Alur and Thomas A Henzinger. 1999. Reactive Modules. *Formal Methods in System Design* 15 (1999), 7–48. doi:10.1023/a:1008739929481

[2] Blair Archibald, Muffy Calder, Michele Sevegnani, and Mengwei Xu. 2022. Modelling and Verifying BDI Agents with Bigraphs. *Science of Computer Programming* 215 (2022), 102760. doi:10.1016/j.scico.2021.102760

[3] Blair Archibald, Muffy Calder, Michele Sevegnani, and Mengwei Xu. 2023. Quantitative Modelling and Analysis of BDI Agents. *Software and Systems Modeling* (2023). doi:10.1007/s10270-023-01121-5

[4] Blair Archibald, Muffy Calder, Michele Sevegnani, and Mengwei Xu. 2023. Quantitative Verification and Strategy Synthesis for BDI Agents. In *NASA Formal Methods Symposium*. Springer, 241–259. doi:10.1007/978-3-031-33170-1_15

[5] Reza Babaee, Arie Gurfinkel, and Sebastian Fischmeister. 2018. Prevent: A Predictive Run-Time Verification Framework Using Statistical Learning. In *International Conference on Software Engineering and Formal Methods*. Springer, 205–220. doi:10.1007/978-3-319-92970-5_13

[6] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. *Lectures on Runtime Verification: Introductory and Advanced Topics* (2018), 1–33. doi:10.1007/978-3-319-75632-5_1

[7] Federico Bergenti and Stefania Monica. 2026. Rethinking software agents as building blocks of software systems. In *The Agents Journey: Twenty-Five Years of Multi-agent Systems*. Springer, 181–212. doi:10.1007/978-3-032-22940-3_7

[8] Andrea Bianco and Luca de Alfaro. 1995. Model Checking of Probabilistic and Nondeterministic Systems. In *Foundations of Software Technology and Theoretical Computer Science*, P. S. Thiagarajan (Ed.). Springer Berlin Heidelberg, 499–513. doi:10.1007/3-540-60692-0_70

[9] Rafael H Bordini, Michael Fisher, Carmen Pardavila, and Michael Wooldridge. 2003. Model Checking Agentspeak. In *International Joint Conference on Autonomous Agents and Multiagent Systems*. 409–416. doi:10.1145/860638.860641

[10] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. 2007. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Vol. 8. John Wiley & Sons. doi:10.1002/9780470061848

[11] Michael Bratman. 1987. *Intention, Plans, and Practical Reason*. Harvard University Press.

[12] Michael Butler and Issam Maamria. 2013. Practical Theory Extension in Event-B. In *Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*. Springer, 67–81. doi:10.1007/978-3-642-39698-4_5

[13] Rafael C Cardoso, Marie Farrell, Matt Luckcuck, Angelo Ferrando, and Michael Fisher. 2020. Heterogeneous Verification of an Autonomous Curiosity Rover. In *NASA Formal Methods*. Springer, 353–360. doi:10.1007/978-3-030-55754-6_20

[14] Hong Chen. 2017. Applications of Cyber-Physical System: A Literature Review. *Journal of Industrial Integration and Management* 2, 03 (2017), 1750012. doi:10.1142/s2424862217500129

[15] Costas Courcoubetis and Mihalis Yannakakis. 1995. The Complexity of Probabilistic Verification. *Journal of the ACM* 42, 4 (1995), 857–907. doi:10.1145/210332.210339

[16] Mehdi Dastani. 2008. 2APL: A Practical Agent Programming Language. *Autonomous Agents and Multi-Agent Systems* 16, 3 (2008), 214–248. doi:10.1007/s10458-008-9036-y

[17] Louise A Dennis and Berndt Farwer. 2008. Gwendolen: A BDI Language for Verifiable Agents. In *AISB Symposium on Logic and the Simulation of Interaction and Reasoning*. 16–23.

[18] Louise A Dennis, Michael Fisher, Nicholas K Lincoln, Alexei Lisitsa, and Sandor M Veres. 2016. Practical Verification of Decision-Making in Agent-Based Autonomous Systems. *Automated Software Engineering* 23 (2016), 305–359. doi:10.1007/s10515-014-0168-9

[19] Louise A Dennis, Michael Fisher, and Matt Webster. 2018. Two-Stage Agent Program Verification. *Journal of Logic and Computation* 28, 3 (2018), 499–523. doi:10.1093/logcom/exv002

[20] Louise A Dennis, Michael Fisher, Matthew P Webster, and Rafael H Bordini. 2012. Model Checking Agent Programming Languages. *Automated Software Engineering* 19, 1 (2012), 5–63. doi:10.1007/s10515-011-0088-x

[21] Alessandra Di Pierro and Herbert Wiklicky. 1998. An Operational Semantics for Probabilistic Concurrent Constraint Programming. In *International Conference on Computer Languages*. IEEE, 174–183. doi:10.1109/iccl.1998.674168

[22] Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. 2009. Model Evolution by Run-Time Parameter Adaptation. In *International Conference on Software Engineering*. IEEE, 111–121. doi:10.1109/icse.2009.5070513

[23] Kenneth A Farley, Kenneth H Williford, Kathryn M Stack, Rohit Bhartia, Al Chen, Manuel de la Torre, Kevin Hand, Yulia Goreva, Christopher DK Herd, Ricardo Hueso, et al. 2020. Mars 2020 Mission Overview. *Space Science Reviews* 216 (2020), 1–41.

[24] Marie Farrell, Angelo Ferrando, and Mengwei Xu. 2025. Quantitative Operational Monitoring for BDI Agents. In *International Conference on Autonomous Agents and Multiagent Systems*. ACM, 2517–2519. https://dl.acm.org/doi/10.5555/3709347.3743922

[25] Angelo Ferrando and Rafael C Cardoso. 2023. Failure Handling in BDI Plans via Runtime Enforcement. In *European Conference on Artificial Intelligence*. IOS Press, 716–723. doi:10.3233/faia230336

[26] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. 2011. Run-Time Efficient Probabilistic Model Checking. In *International Conference on Software Engineering*. 341–350. doi:10.1145/1985793.1985840

[27] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. 2012. A Formal Approach to Adaptive Software: Continuous Assurance of Non-Functional Requirements. *Formal Aspects of Computing* 24 (2012), 163–186. doi:10.1007/s00165-011-0207-2

[28] Antonio Filieri and Giordano Tamburrelli. 2013. Probabilistic Verification at Runtime for Self-Adaptive Systems. *Assurances for Self-Adaptive Systems: Principles, Models, and Techniques* (2013), 30–59. doi:10.1007/978-3-642-36249-1_2

[29] Michael Fisher, Viviana Mascardi, Kristin Yvonne Rozier, Bernd-Holger Schlingloff, Michael Winikoff, and Neil Yorke-Smith. 2021. Towards a Framework for Certification of Reliable Autonomous Systems. *Autonomous Agents and Multi-Agent Systems* 35 (2021), 1–65. doi:10.1007/s10458-020-09487-2

[30] Vojtech Forejt, Marta Z. Kwiatkowska, David Parker, Hongyang Qu, and Mateusz Ujma. 2012. Incremental Runtime Verification of Probabilistic Systems. In *International Conference on Runtime Verification*, Vol. 7687. Springer, 314–319. doi:10.1007/978-3-642-35632-2_30

[31] Hector Geffner and Blai Bonet. 2013. *A concise introduction to models and methods for automated planning*. Morgan & Claypool Publishers.

[32] Hans Hansson and Bengt Jonsson. 1994. A Logic for Reasoning About Time and Reliability. *Formal Aspects of Computing* 6 (1994), 512–535. doi:10.1007/bf01211866

[33] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. 2022. The Probabilistic Model Checker Storm. *International Journal on Software Tools for Technology Transfer* 24, 4 (2022), 589–610. doi:10.1007/S10009-021-00633-Z

[34] Koen V Hindriks. 2009. Programming Rational Agents in GOAL. In *Multi-Agent Programming: Languages, Tools and Applications*. Springer, 119–157. doi:10.1007/978-0-387-89299-3_4

[35] Koen V. Hindriks, Frank S. De Boer, Wiebe Van der Hoek, and John-Jules Ch Meyer. 1999. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems* 2, 4 (1999), 357–401. doi:10.1023/a:1010084620690

[36] Gerard J. Holzmann. 1997. The Model Checker SPIN. *Transactions on Software Engineering* 23, 5 (1997), 279–295. doi:10.1109/32.588521

[37] Gerard J Holzmann and William Slattery Lieberman. 1991. *Design and Validation of Computer Protocols*. Vol. 512. Prentice Hall Englewood Cliffs.

[38] Paolo Izzo, Hongyang Qu, and Sandor M Veres. 2016. A Stochastically Verifiable Autonomous Control Architecture with Reasoning. In *IEEE Conference on Decision and Control*. 4985–4991. doi:10.1109/cdc.2016.7799031

[39] Alexander Birch Jensen. 2022. Machine-Checked Verification of Cognitive Agents.. In *International Conference on Agents and Artificial Intelligence*. 245–256. doi:10.5220/0010838700003116

[40] Sebastian Junges, Hazem Torfah, and Sanjit A Seshia. 2021. Runtime Monitors for Markov Decision Processes. In *International Conference on Computer Aided Verification*. Springer, 553–576. doi:10.1007/978-3-030-81688-9_26

[41] Marta Kwiatkowska, Gethin Norman, and David Parker. 2010. Advances and Challenges of Probabilistic Model Checking. In *Allerton Conference on Communication, Control, and Computing*. IEEE, 1691–1698. doi:10.1109/allerton.2010.5707120

[42] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *International Conference on Computer Aided Verification (Lecture Notes in Computer Science, Vol. 6806)*. Springer, 585–591. doi:10.1007/978-3-642-22110-1_47

[43] Matt Luckcuck, Marie Farrell, Louise A Dennis, Clare Dixon, and Michael Fisher. 2019. Formal Specification and Verification of Autonomous Robotic Systems: A Survey. *Comput. Surveys* 52, 5 (2019), 1–41. doi:10.1145/3342355

[44] Felipe Meneguzzi and Lavindra De Silva. 2015. Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning. *The Knowledge Engineering Review* 30, 1 (2015), 1–44.

[45] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer.

[46] Lin Padgham and Dhirendra Singh. 2013. Situational Preferences for BDI Plans. In *International Conference on Autonomous Agents and Multi-Agent Systems*. 1013–1020. doi:10.65109/cjop2465

[47] Amir Pnueli. 1977. The Temporal Logic of Programs. In *Symposium on Foundations of Computer Science*. ieee, 46–57. doi:10.1109/SFCS.1977.32

[48] Anand S. Rao. 1996. AgentSpeak (L): BDI Agents Speak Out in a Logical Computable Language. In *European Workshop on Modelling Autonomous Agents in a Multi-Agent World*. Springer, 42–55. doi:10.1007/bfb0031845

[49] Sebastian Sardina and Lin Padgham. 2011. A BDI Agent Programming Language with Failure Handling, Declarative Goals, and Planning. *Autonomous Agents and Multi-Agent Systems* 23 (2011), 18–70. doi:10.1007/s10458-010-9130-9

[50] A Sima Uyar, Ender Ozcan, and Neil Urquhart. 2013. *Automated scheduling and planning: from theory to practice*. Vol. 505. Springer.

[51] Michael Winikoff, Lin Padgham, James Harland, and John Thangarajah. 2002. Declarative & Procedural Goals in Intelligent Agent Systems. *KR* 2002 (2002), 470–481. doi:10.5555/3087093.3087132

[52] Thomas Wright, Louise A Dennis, Jim Woodcock, and Simon Foster. 2024. Formal Verification of BDI Agents. In *The Combined Power of Research, Education, and Dissemination: Essays Dedicated to Tiziana Margaria on the Occasion of Her 60th Birthday*. Springer, 302–326. doi:10.1007/978-3-031-73887-6_20

[53] Mengwei Xu, Tom Lumley, Ramon Fraga Pereira, and Felipe Meneguzzi. 2024. A Practical Operational Semantics for Classical Planning in BDI Agents. In *Proceedings of the 27th European Conference on Artificial Intelligence*. 1365 – 1372. doi:10.3233/FAIA240636

[54] Mengwei Xu, Peter Rivière, Toshiaki Aoki, Marie Farrell, Yamine Aït Ameur, Neeraj Kumar Singh, and Guillaume Dupont. 2026. Encoding BDI Syntax with Theories in Event-B. In *International Conference on Rigorous State-Based Methods*. Springer, 210–228. doi:10.1007/978-3-032-26752-8_13

[55] Håkan LS Younes, Marta Kwiatkowska, Gethin Norman, and David Parker. 2004. Numerical vs. Statistical Probabilistic Model Checking: An Empirical Study. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 46–60.

A Probabilistic Intention-Level Can Semantics

$$\frac{\mathcal{S}_a^p(act) = \mu \quad \mu(\phi^-, \phi^+) = p \quad \mathcal{B} \vDash \psi}{\langle \mathcal{B}, act\rangle \rightarrow_p \langle (\mathcal{B} \setminus \phi^-) \cup \phi^+, nil\rangle}\ \mathbf{act^p}$$

$$\frac{\Delta = \{\varphi : P \mid (e' = \varphi \leftarrow P) \in \Pi \wedge e' = e\}}{\langle \mathcal{B}, e\rangle \rightarrow_1 \langle \mathcal{B}, e : (\mid \Delta \mid)\rangle}\ \mathbf{event^p}$$

$$\frac{\mathcal{S}_p^p(\mathcal{B}, \Delta) = \mu \ \mu(\bot) = 0 \ \varphi : P \in \Delta \ \ \mu(\varphi : P) = p}{\langle \mathcal{B}, e : (\mid \Delta \mid)\rangle \rightarrow_p \langle \mathcal{B}, P \rhd e : (\mid \Delta \setminus \{\varphi : P\} \mid)\rangle}\ \mathbf{select^p}$$

$$\frac{\langle \mathcal{B}, P_1\rangle \rightarrow_p \langle \mathcal{B}', P_1'\rangle}{\langle \mathcal{B}, P_1 \rhd P_2\rangle \rightarrow_p \langle \mathcal{B}', P_1' \rhd P_2)\rangle}\rhd^{\mathbf{p}}_{;} \qquad \frac{}{\langle \mathcal{B}, (nil \rhd P_2)\rangle \rightarrow_1 \langle \mathcal{B}, nil\rangle}\rhd^{\mathbf{p}}_{\top}$$

$$\frac{P_1 \neq nil \ \langle \mathcal{B}, P_1\rangle \nrightarrow_1 \ \langle \mathcal{B}, P_2\rangle \rightarrow_p \langle \mathcal{B}', P_2'\rangle}{\langle \mathcal{B}, P_1 \rhd P_2\rangle \rightarrow_p \langle \mathcal{B}', P_2'\rangle}\ \rhd^{\mathbf{p}}_{\bot}$$

$$\frac{\langle \mathcal{B}, P\rangle \rightarrow_p \langle \mathcal{B}', P'\rangle}{\langle \mathcal{B}, (nil; P)\rangle \rightarrow_p \langle \mathcal{B}', P'\rangle};^{\mathbf{p}}_{\top} \qquad \frac{\langle \mathcal{B}, P_1\rangle \rightarrow_p \langle \mathcal{B}', P_1'\rangle}{\langle \mathcal{B}, (P_1; P_2)\rangle \rightarrow_p \langle \mathcal{B}', (P_1'; P_2)\rangle};^{\mathbf{p}}$$

$$\frac{\mathcal{S}_c^p(\mathcal{B}, P_1 \| P_2) = \mu \ \mu(\bot) = 0 \ \mu(P_1) = p_1}{\langle \mathcal{B}, (P_1 \| P_2)\rangle \rightarrow_{p_1} \langle \mathcal{B} \cup \{left\}, (P_1 \| P_2)\rangle}\ \|^{\mathbf{p}}_{\mathbf{select\_left}}$$

$$\frac{left \in \mathcal{B} \ \langle \mathcal{B}, P_1\rangle \rightarrow_{p_1'} \langle \mathcal{B}', P_1'\rangle}{\langle \mathcal{B}, (P_1 \| P_2)\rangle \rightarrow_{p_1'} \langle \mathcal{B}' \setminus \{left\}, (P_1' \| P_2)\rangle}\ \|^{\mathbf{p}}_{\mathbf{progress\_left}}$$

$$\frac{\mathcal{S}_c^p(\mathcal{B}, P_1 \| P_2) = \mu \ \mu(\bot) = 0 \ \mu(P_2) = p_2}{\langle \mathcal{B}, (P_1 \| P_2)\rangle \rightarrow_{p_2} \langle \mathcal{B} \cup \{right\}, (P_1 \| P_2)\rangle}\ \|^{\mathbf{p}}_{\mathbf{select\_right}}$$

$$\frac{right \in \mathcal{B} \ \langle \mathcal{B}, P_2\rangle \rightarrow_{p_2'} \langle \mathcal{B}', P_2'\rangle}{\langle \mathcal{B}, (P_1 \| P_2)\rangle \rightarrow_{p_2'} \langle \mathcal{B}' \setminus \{right\}, (P_1 \| P_2')\rangle}\ \|^{\mathbf{p}}_{\mathbf{progress\_right}}$$

$$\frac{}{\langle \mathcal{B}, (nil \| nil)\rangle \rightarrow_1 \langle \mathcal{B}, nil\rangle}\|^{\mathbf{p}}_{\top} \qquad \frac{\mathcal{B} \models \varphi_s}{\langle \mathcal{B}, goal(\varphi_s, P, \varphi_f)\rangle \rightarrow_1 \langle \mathcal{B}, nil\rangle}\mathbf{G}^{\mathbf{p}}_{\mathbf{s}}$$

$$\frac{\mathcal{B} \models \varphi_f}{\langle \mathcal{B}, goal(\varphi_s, P, \varphi_f)\rangle \rightarrow_1 \langle \mathcal{B}, ?false\rangle}\mathbf{G}^{\mathbf{p}}_{\mathbf{f}} \qquad \frac{P \neq P_1 \rhd P_2 \quad \mathcal{B} \nvDash \varphi_s \quad \mathcal{B} \nvDash \varphi_f}{\langle \mathcal{B}, goal(\varphi_s, P, \varphi_f)\rangle \rightarrow_1 \langle \mathcal{B}, goal(\varphi_s, P \rhd P, \varphi_f)\rangle}\mathbf{G}^{\mathbf{p}}_{\mathbf{init}}$$

$$\frac{\mathcal{B} \nvDash \varphi_s \quad \mathcal{B} \nvDash \varphi_f \quad \langle \mathcal{B}, P_1\rangle \rightarrow_p \langle \mathcal{B}', P_1'\rangle}{\langle \mathcal{B}, goal(\varphi_s, P_1 \rhd P_2, \varphi_f)\rangle \rightarrow_p \langle \mathcal{B}', goal(\varphi_s, P_1' \rhd P_2, \varphi_f)\rangle}\mathbf{G}^{\mathbf{p}}_{;}$$

$$\frac{\mathcal{B} \nvDash \varphi_s \quad \mathcal{B} \nvDash \varphi_f \quad \langle \mathcal{B}, P_1\rangle \nrightarrow_1}{\langle \mathcal{B}, goal(\varphi_s, P_1 \rhd P_2, \varphi_f)\rangle \rightarrow_1 \langle \mathcal{B}, goal(\varphi_s, P_2 \rhd P_2, \varphi_f)\rangle}\mathbf{G}^{\mathbf{p}}_{\rhd}$$

Fig. 13. Probabilistic intention-level Can semantics

In Fig. 13, the major rules have a deterministic transition such as **event**$^{\mathbf{p}}$. Meanwhile, the rule **select**$^{\mathbf{p}}$ use the plan selection function $\mathcal{S}_p^p : 2^{\overline{\mathcal{B}}} \times 2^{\Pi} \rightarrow Dist(\Pi \cup \{\bot\})$ such that any non-relevant and non-applicable plans are assigned the probability 0 and $\mu(\bot) = 0$ stands there is at least an applicable plan. Also, the rule $\|^{\mathbf{p}}_{\mathbf{select\_left}}$ and $\|^{\mathbf{p}}_{\mathbf{select\_right}}$ use a selection function $\mathcal{S}_c^p : 2^{\overline{\mathcal{B}}} \times P \rightarrow Dist(P \cup \{\bot\})$ to choose one program from a concurrent program to progress where $P \subseteq \Gamma$ is all possible plan-body programs. And the original concurrent rules $\|_1$ and $\|_2$ are again split into two rules with belief tokens used for execution flow control.