A MAPE-K Instantiation for Dependable Normative Adaptation in Intelligent Agents

A L Partis, Radu Calinescu, Ana Cavalcanti, Sinem Getir Yaman, Alvaro Miyazawa, Ioannis Stefanakos
Department of Computer Science, University of York, York, UK
{a.l.partis, radu.calinescu, ana.cavalcanti,
sinem.getir.yaman, alvaro.miyazawa, ioannis.stefanakos}@york.ac.uk

***Abstract*—While intelligent agents are increasingly employed in human-centric applications, their inability to adapt to diverse social, legal, ethical, empathetic, and cultural (SLEEC) norms poses a significant threat to their dependability and trustworthiness. Traditional design-time verification is insufficient for managing the changing normative requirements of diverse users at runtime, where a failure to comply with SLEEC norms constitutes a critical system failure. In this paper, we propose a self-adaptive approach to enhance agent dependability by instantiating the MAPE-K architecture for normative adaptation. Our architecture enables an agent to dynamically reconfigure its workflow to ensure continuous compliance with SLEEC requirements that change with its active user. We introduce novel tools implementing a workflow adaptation algorithm to facilitate reliable runtime reconfiguration within the boundaries of pre-existing agent capabilities. We demonstrate the effectiveness of our approach in preventing normative failures through case studies in service robotics and chatbots.**

***Index Terms*—code synthesis, MAPE-K, normative reliability of human-centric software, self-adaptive agents**

I. Introduction

Humans are capable of seamlessly adapting to accommodate the social, legal, ethical, empathetic, and cultural (SLEEC) norms of the context in which they operate. For instance, a hospital cleaner may open a window before dusting a room if the patient in that room is allergic to dust, or may notify a patient in the next room before vacuum cleaning if that patient is easily startled by noise.

Intelligent agents—including cyber-physical systems (autonomous robots, self-driving cars, and so on) and software agents (chatbots and personal assistants, for instance)—lack this adaptability. So, the rapid adoption of intelligent agents across many domains carries a high risk of violating SLEEC norms. The possible consequences range from user dissatisfaction, distress, or offence, to violation of ethical norms and values, or of the law. This can undermine the trust of the public in technology that can, otherwise, address societal challenges.

Recent research has developed approaches for the elicitation [31], specification, validation, and verification [11], [17], [18] of SLEEC *rules* (i.e., formalised normative requirements) for intelligent agents. These methods support SLEEC requirements engineering, and verification of agent compliance with SLEEC rules at *design time*. Defining rules suitable for all possible operational environments, and every user, however, is typically impossible. For instance, a cleaner robot opening the window prior to dusting will not suit all patients, and individual user requirements may be unknown until runtime.

This paper addresses the need for reliable intelligent agent adaptation to changing normative requirements by instantiating the MAPE-K architecture [35]. We note that social robotics literature defines normative-requirement violations as system failures [19], and therefore we refer to the avoidance of such violations through adaptation as "dependability".

Our architecture uses novel tools to adapt an agent's workflow to: (i) perform all tasks already defined; (ii) comply with a set of SLEEC rules that correspond to preferences stated in a single user's profile at a time and are defined in terms of pre-existing agent capabilities; and, where applicable, (iii) generate a SLEEC-compliant orchestration controller which executes the adapted workflow. We provide an implementation of such an architecture using the ROS robotics middleware [23].

Our main contributions are: (1) the theory underpinning our approach to dynamic intelligent-agent workflow adaptation; (2) a reusable, application-independent implementation of the planning and execute MAPE-K stages; and (3) two case studies demonstrating the applicability of our approach to both cyber-physical and software intelligent agents.

The rest of the paper is organised as follows. Section II presents related work. A motivating example is given in Section III. Section IV presents the notation we adopt to define SLEEC rules. Our architecture is the topic of Section V. Its implementation is described in Section VI, and is evaluated in Section VII. We conclude with a brief summary and mention future work directions in Section VIII.

II. Related Work

There is consensus that where systems make decisions affecting user's experiences, rights and autonomy [8], [29], they must act in a "fair and ethical" manner [4], [7] to avoid negative impact on society. In many countries, responsible design is becoming a legal and moral necessity [29], [33].

There has been no work on automating adherence to SLEEC concerns arising at runtime. Here we summarise work in relevant related areas.

High-level principles: Floridi and Cowl [14] identified principles for ethical AI, prioritising beneficence, non-maleficence, autonomy, justice, and explainability: all essential, but too high-level to guide development [22].

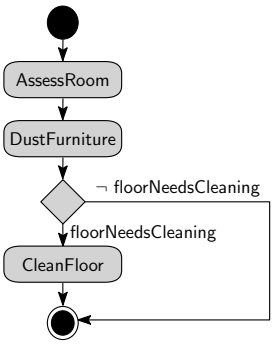


Fig. 1. Utility robot workflow

Elicitation of requirements: Most elicitation processes involve stakeholder input [20], [31]. In particular, [31] produces rules encoded using the SLEEC Domain-Specific Language (DSL) [16] we adopt here. Alfieri et al. [1] elicited user-value categories from ethics position theory, but the expressivity of this technique is limited. In our work here, we adopt the notion of profile presented in [1]. Liegl et al. [21] derives specific ethical requirements, but deployment of these requires a runtime adaptive planning approach [10] like ours.
Operationalisation of rules: Autili et al. [2] adopted a formal language to express preferences regarding online services. Whittle et al. [36] created a more complex language and formulated examples showing it could cover SLEEC issues. SLEEC is considered in [32] and [25], which mapped the SLEEC DSL to a similar logic for design-time verification. Bremner et al. [5] expressed ethical rules in propositional logic to guide a robotic controller that adopts a belief-desire-intention (BDI) architecture that uses simulations to make decisions, but not code synthesis.
Automated workflow adaptation: Workflow process management is emerging in robotic automation, but runtime adaptation is rare [9]. Outside robotics, Müller et al. [27] propose runtime adaptation of workflows according to complex rules to resolve functional errors. SLEEC issues have been considered in workflow process management, but no architectures specifically accommodate them [34]. Outside process management, Memon et al. [24] developed an "Ethical Context Manager" to support tractable SLEEC-based negotiation at runtime, though their framework is constrained to contended resource problems.

In summary, whilst considerable research addresses adaptive behaviors and ethical implications of autonomous agents, no prior work has addressed evolving normative requirements as a basis for runtime adaptation.

III. Motivating example

To motivate our work, we consider a service robot that cleans patient rooms at a hospital. The robot needs to execute the workflow in Figure 1 in each room. An initial *task*, called AssessRoom, involves entering the room and evaluating its cleanliness. The following task, DustFurniture, involves wiping the furniture. Finally, the robot uses the result of its initial assessment, recorded as a *measure* floorNeedsCleaning, to determine if the task CleanFloor is necessary.

The robot must dynamically adapt its behaviour by considering the SLEEC requirements of each patient. To that end, we assume that the robot has several additional *capabilities*: to display its cleaning plan, e.g., on a screen located on its chest (DisplayCleaningPlan); to detect whether the patient is resting (personResting) via a manual 'patient status' bedside control that the patient can set, or an AI perception component; to notify the patient of its next action (NotifyPatient); and to switch to a silent floor-cleaning mode (SilentFloorCleaning). Given these capabilities, we consider the requirements in Table I in the context of the following scenario.

Upon arrival, patients are received by a nurse who identifies their needs. The nurse defines a profile for the patient based on parameters such as disabilities, medical needs, religion, and others. This profile is recorded in the knowledge base, and can be retrieved using, for example, an RFID-tagged patient bracelet or information about the room allocated to the patient. The system can provide the nurse with a choice from a collection of profiles or from a collection of (SLEEC) rules (described in natural language) that define a profile. If the nurse chooses rules, before a profile is entered into the database, validation (for conflicts among rules) can be carried out automatically using techniques like those in [11], [18]. So, profiles can be dynamically created either by nurses or by technical hospital personnel based on rules written in natural language. In each case, the system can fully adapt to the profiles of the patients currently in the hospital.

Before entering a room, the robot retrieves the profile of the patient in that room, and thus the SLEEC rules that apply next. If these rules differ from those for the last room cleaned, adaptation is needed. As an example, we consider here a patient for whom information is important, and so they want to know the cleaning plan and to be notified before cleaning starts (requirements R1 and R2 in Table I). That patient also dislikes loud sounds, particularly when resting, and would like the robot to use its silent mode (requirement R3).

Discussion. The definition of SLEEC rules (and therefore profiles) depends fundamentally on the existing capabilities of the agent, with the SLEEC rules associated with the active user selected to guide the reordering of these capabilities within the agent's workflow. Thus, rule R1 from Table I is feasible thanks to the pre-existing robot capability DisplayCleaningPlan. Creating new capabilities on the fly and defining SLEEC rules that refer to non-existent agent capabilities is not feasible. For instance, a SLEEC requirement asking that the robot issues a spoken warning when a patient approaches a wet floor is unfeasible because the robot lacks speech generation capabilities and cannot assess patient proximity to wet floors.

TABLE I
SLEEC requirements for the motivating example

R1:	*After the room assessment, the robot shall display its room-cleaning plan.*
R2:	*Before the room assessment begins, the robot shall notify the patient.*
R3:	*When the patient is resting, the robot shall switch to silent cleaning mode.*

TABLE II
SLEEC RULES FOR THE MOTIVATING EXAMPLE

Rule1 when AssessRoomEnd then DisplayCleaningPlan
Rule2 when AssessRoomStart then NotifyPatient
Rule3 when CleanFloorStart and personResting then SilentFloorCleaning

IV. PRELIMINARIES: SLEEC DSL

To write the rules, we use a subset of the SLEEC DSL from [11], [16]. Developed by a multidisciplinary team of engineers, ethicists, lawyers, sociologists, and psychologists, this DSL offers the benefits of an intuitive notation, a formal semantics [18], and tool-supported methods [12], [13], [17] for validation and for verification of agent compliance with SLEEC requirements. A SLEEC rule has the general form:

id when **triggerEvent** [and **triggerGuard**] (1)
then **responseEvent**
[unless **defeaterGuard**[then **defeaterResponse**]]*

where **id** is a unique rule identifier, **triggerEvent** is an atomic, instantaneous *event* monitored by the agent (e.g., the start or end of a task), the optional **triggerGuard** is a Boolean expression over a set of *measures*, and the then clause is a response that the agent shall perform when the **triggerEvent** occurs and, if specified, the **triggerGuard** is true.

In the response, **responseEvent** is an atomic and instantaneous event signalling the execution of an action by the agent, and the optional clauses 'unless **defeaterGuard** [then **defeaterResponse**]' define *defeaters*, that is, circumstances in which the agent shall take an alternative action to **responseEvent**. If one or more defeaters are specified and at least one of their **defeaterGuard** Boolean expressions is true, the agent shall perform the **defeaterResponse** corresponding to the outermost such **defeaterGuard**, if that optional response is present, or be unrestricted if that outermost guard has no associated **defeaterResponse**. Table II shows the requirements from Table I expressed in the SLEEC DSL.

V. MAPE-K ARCHITECTURE INSTANTIATION

Many strategies for self-adaptation exist—including constraint-based solvers, feedback control loops, and model predictive control—but use of a MAPE-K loop is favoured. This architecture organises adaptation into five components. Monitoring collects data from the system and its environment. Analysis processes data to detect triggers. Planning formulates a strategy to meet goals. Execution implements the changes in the system. A Knowledge base is a repository providing the data for the other four components. Together, these components allow systems to dynamically adjust their actions, resources, and goals at runtime.

Figure 2 describes our instantiation of MAPE-K to achieve SLEEC-adherent workflow adaptation, code generation, and deployment. The monitor component uses system sensors to track the active user. The analysis component extracts SLEEC rules from the knowledge base and determines whether adaptation is required. The planning phase takes as input the SLEEC rules and the initial workflow and produces an adherent workflow described in a DSL with support for code generation. The execution component generates the new controller or configuration for the workflow, and deploys it on the agent. The components are outlined in further detail below.

We note that in order to adapt the behaviour of an agent, the SLEEC rules of its active user must map to a fixed set of pre-existing, abstracted agent capabilities. Rule authors do not need to understand the implementation of these capabilities; they compose rules using the exposed capability names. The new SLEEC-compliant orchestration controller generated by our approach or the reconfigured software agent executes services provided by the agent capabilities.

A. Knowledge base, monitoring, and analysis

The knowledge base records: (1) the user profiles and their SLEEC rulesets; (2) the user currently interacting with the system; (3) the base workflow; and (4) the current version of the workflow, and any cached variants.

The monitoring component keeps track of users that interact with the agent, handling one active user at a time. For instance, a robot can use facial recognition, and a chatbot can use the connection from a user's machine. The monitor determines the ID and matching user profile, and updates the knowledge base accordingly.

The analysis component extracts the active user's profile from the knowledge base, and compares it to the profile last used. If different, planning is triggered.

B. Planning

The planning component consists of an end-to-end pipeline that performs adaptation on the original workflow to produce SLEEC-adherent behaviour. The first component of the pipeline is a novel adaptation algorithm that produces a modified workflow that enforces the SLEEC rules. A second component is a model-to-model transformation that provides a description of the new workflow as a RoboSim [6] module.

RoboSim supports the diagrammatic modelling of control software. A RoboSim model is given by a module that describes the capabilities of the robot used by the software, and a collection of parallel state machines that define the behaviour of the software as a cyclic mechanism. Distinctively, RoboSim has a formal semantics compatible with that of the SLEEC DSL, described using a process algebra. Crucially, for the work presented here, RoboSim has a generator of code that preserves the behaviour of the RoboSim model.

This section first outlines the metamodel and well-formedness conditions for workflows, and then our novel adaptation algorithm. Finally, we describe the transformation from workflows to RoboSim modules.

Workflows are instances of the *Workflow* class of the metamodel in Figure 3. A workflow can be a Task, specified by an ID; a Sequence of two or more subworkflows, to be performed in order; a Loop comprising a body (a (sub)workflow) to be executed continuously whilst its guard (boolean expression) holds; or finally, a Decision comprising one or more options,

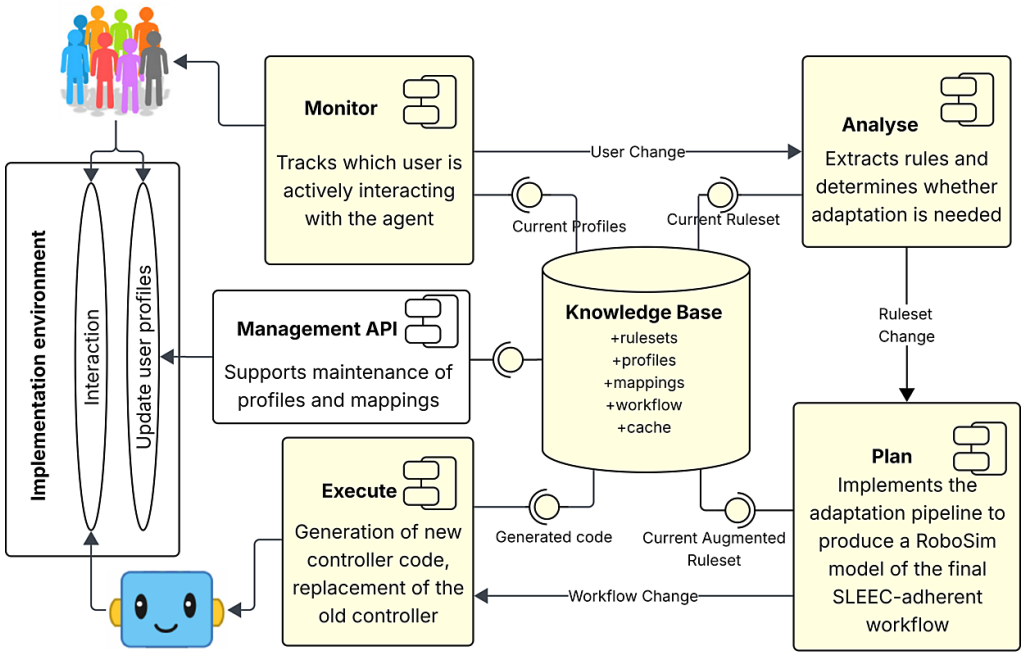


Fig. 2. Instantiation of the MAPE-K architecture.

each of which is a GuardedWorkflow. When one guard holds, its associated body is executed. If no guard holds, the execution of the Decision terminates with no task executed; equivalent to the predefined task with ID Skip. If multiple guards hold, the agent selects any body to execute.

Well-formedness An instance w of Workflow (Figure 3), and a set of $n \geq 1$ conflict and redundancy-free SLEEC rules (IV) input to our planning algorithm must satisfy the following assumptions.

Assumption 1) All **triggerEvent**s in the rules are of form taskIdStart or taskIdEnd, where taskId represents the ID of a Task that may be in w, and the suffix indicates whether the trigger event refers to the start or the end of the Task.

Assumption 2) All **responseEvent**s and **defeaterResponse**s in the rules correspond to actual capabilities of the agent and are not associated with any task from w, or with the $triggerEvent$ of any other SLEEC rule.

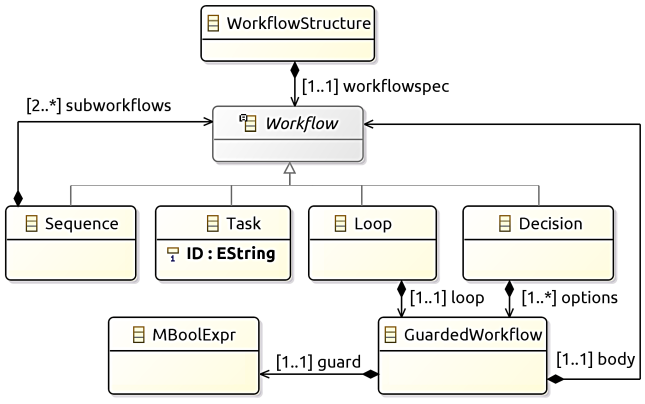


Fig. 3. Workflow metamodel

Adaptation algorithm The workflow adaptation is performed by the function AdaptWorkflow in Algorithm 1. This function takes two arguments, an input *workflow* (structured as defined in Figure 3) and a SLEEC *ruleset* (comprising rules (IV)), and returns a new workflow that is functionally equivalent to the input *workflow* and compliant with the specified *ruleset*. AdaptWorkflow operates by structural recursion over the *workflow* received as its first argument.

Case 1. When AdaptWorkflow receives a *workflow* of type Task (the base case, line 3 of Algorithm 1), all SLEEC rules with a trigger event matching the task ID and ending with the suffix Start or End are retrieved by function GetRulesByTrigger and placed into the rulesets *startRules* and *endRules* (lines 4 and 5). (The function GetRulesByTrigger performs a straightforward linear search through *ruleset*, and is not included in Algorithm 1 due to space constraints.) If both *startRules* and *endRules* are empty, no adaptation is required, and the original *workflow* is returned (lines 6–7).

Otherwise, a new *sequence* of tasks is: (1) created in line 8; (2) populated with additional sub-workflows required for SLEEC compliance with each of the *startRules* (lines 9–11), followed by the original task (that is, by *workflow*) in line 12, and by the sub-workflows required for compliance with each of the *endRules* (lines 13–15); and (3) returned as the adapted workflow in line 16. For each rule from *startRules* and *endRules*, the generation (in step 2) of the sub-workflow that ensures compliance with that rule is carried out by GenTasksFromSLEEC in lines 10 and 14, respectively.

The GenTasksFromSLEEC function (lines 31–35) assembles the required sub-workflow as follows. First, a new task corresponding to the response from the *rule* is generated in line 32 and stored into a response workflow $responseWf$.

Next, if the *rule* being processed includes the optional *triggerGuard* from (IV), then this response workflow is placed within a Decision whose options consist of a single GuardedWorkflow created from the rule's *triggerGuard* and the response workflow itself (lines 33–34). Finally, the response workflow is augmented as required by the *rule*'s defeaters by calling the function ADDDEFEATERS in line 35.

ADDDEFEATERS is defined recursively: in the absence of any defeaters, it returns the response workflow received as its first argument (line 38). Otherwise, it returns a two-option DECISION (line 40), where: (1) the first option executes a task associated with the outermost (and so the highest priority) defeater's response—if this defeater's guard holds; and (2) the second option executes the workflow generated by augmenting the response workflow *responseWf* as required by all other defeaters—if the outermost defeater's guard does not hold.

Case 2. When ADAPTWORKFLOW receives a *workflow* of type Sequence, it returns a new Sequence that is obtained by combining the adapted versions of each sub-workflow from the initial sequence (lines 17–22).

Case 3. When ADAPTWORKFLOW receives a *workflow* of type Decision, each *option* that is part of this Decision (see Figure 3) is adapted and placed into an array of adapted options (lines 25–26) that is used to create and return the adapted variant of the entire Decision (line 27).

Case 4. Finally, ADAPTWORKFLOW handles a Loop *workflow* (lines 28–30) by recursively assembling an adapted variant of the Loop's body (*SLEECbody*) in line 29, and using it to create and return a new Loop with the same guard as the initial Loop and this *SLEECbody* (line 30).

Example 1. Figure 4 shows the result of applying ADAPTWORKFLOW to the workflow from Figure 1, and the SLEEC ruleset from Table II.

Analysis of workflow adaptation algorithm. The following theorems, whose proofs are provided as supplementary material[1], show the correctness and complexity of adaptation.

Theorem 1. *Given a workflow with the structure from Figure 3, and a SLEEC ruleset satisfying Assumptions 1 and 2 above, the function* ADAPTWORKFLOW *from Algorithm 1 returns a workflow that is functionally equivalent to the original workflow, provided that any tasks added by* ADAPTWORKFLOW *are disregarded.*

Theorem 2. *Under the same premises as those specified for Theorem 1,* ADAPTWORKFLOW *returns a workflow that satisfies all SLEEC rules provided in the second argument, according to their intended semantics.*

Theorem 3. *The function* ADAPTWORKFLOW *executes in at most* $\mathcal{O}(tr(d+1))$ *steps, where:* t *is the number of tasks in the input workflow;* r *is the number of rules in the input ruleset; and* d *is the maximum number of defeaters associated with any single SLEEC rule within the ruleset.*

[1]https://github.com/a-l-partis/Normative-Agent-Adaptation/blob/main/Supplementary-material.pdf

Algorithm 1 Workflow Adaptation Algorithm

```
1: function AdaptWorkflow(workflow, ruleset)
2:   switch workflow.type do
3:     case Task:
4:       startRules ← GetRulesByTrigger(workflow.id + "Start")
5:       endRules ← GetRulesByTrigger(workflow.id + "End")
6:       if startRules = ∅ ∧ endRules = ∅ then
7:         return workflow
8:       sequence ← new Sequence
9:       for each rule ∈ startRules do
10:        SLEECwf ← GenTasksFromSLEEC(rule)
11:        sequence ← AddSubworkflow(sequence, SLEECwf)
12:      sequence ← AddSubworkflow(sequence, workflow)
13:      for each rule ∈ endRules do
14:        SLEECwf ← GenTasksFromSLEEC(rule)
15:        sequence ← AddSubworkflow(sequence, SLEECwf)
16:      return sequence
17:    case Sequence:
18:      sequence ← new Sequence
19:      for each subwf ∈ workflow.subworkflows do
20:        SLEECwf ←AdaptWorkflow(subwf, ruleset)
21:        sequence ←AddSubworkflow(sequence, SLEECwf)
22:      return sequence
23:    case Decision:
24:      SLEECoptions ← []
25:      for each option ∈ workflow.options do
26:        SLEECoptions ←
               SLEECoptions([new GuardedWorkflow(option.guard,
                     AdaptWorkflow(option.body, ruleset)]
27:      return new Decision(SLEECoptions)
28:    case Loop:
29:      SLEECbody ←
             AdaptWorkflow(workflow.loop.body, ruleset)
30:      return new Loop(new
   GuardedWorkflow(workflow.loop.guard, SLEECbody))

31: function GenTasksFromSLEEC(rule)
32:   responseWf ← new Task(rule.response)
33:   if rule.triggerGuard ≠ nil then
34:     responseWf ← new Decision(
          [new GuardedWorkflow(rule.triggerGuard,responseWf)])
35:   return AddDefeaters(responseWf, rule.defeaters)

36: function AddDefeaters(responseWf, dfs)
37:   if dfs = nil then
38:     return responseWf
39:   else
40:     return new Decision([new GuardedWorkflow(last(dfs).guard,
                    new Task(last(dfs).response)),
                    new GuardedWorkflow(¬last(dfs).guard,
                    AddDefeaters(responseWf,front(dfs)))])
```

RoboSim To enable generation of code for execution, we transform the adapted workflow to a RoboSim module. The capabilities defined in RoboSim are exactly those used in the SLEEC rules, covering the events defining triggers, responses, and the measures. We define behaviours in a single state machine that specifies the workflow as a cyclic mechanism.

Conversion from a workflow to RoboSim is carried out by a model-to-model transformation defined by the mappings in Table III. The control flow of a RoboSim machine is cyclic, with evolution defined by passage of time. In each cycle, all inputs are read and processed to define outputs, before time advances to the start of the next cycle. The special event exec marks the end of the processing. Given the sequential nature of a workflow, the main challenge of its transformation to a machine is the allocation of tasks and decisions to cycles.

Tasks are modelled in RoboSim as operations. So, each task

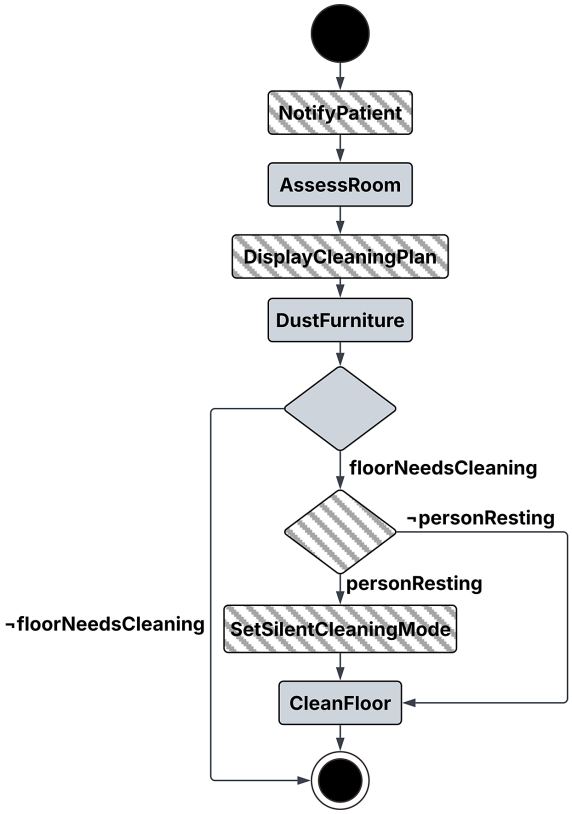


Fig. 4. Adapted workflow, with hashed shapes showing the added elements for each of the SLEEC rules R1, R2, and R3.

T is mapped to a state whose entry action calls that operation, which by convention is called $TStart() (see Table III). The cycle is then finished, as signalled by exec, and the machine transitions to another state. In that state, the RoboSim machine waits for the end of the task execution, signalled by $Tend. A self-transition ensures that cycles advance while the end is not signalled, and once it is signalled, the control flow proceeds. In Table III, we use black circles to represent the start and end of the control flow of a fragment of a workflow, and black squares to represent the start and end of the flow of a fragment of a RoboSim machine.

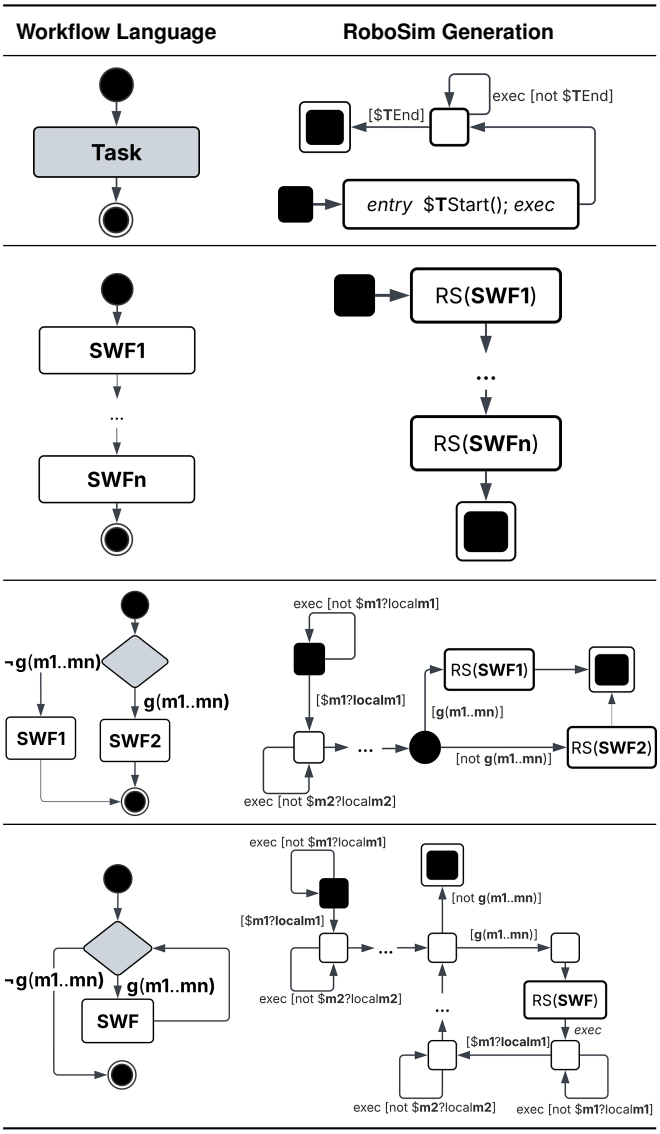
Sequences are available in RoboSim, so the translation is direct. A decision over a guard g(m1,...,mn) of measures is translated to a sequence of transitions that input the measures and assign them to local variables localm1, localm2, and so on. This ends in a junction where the decision over g is made using the local variables, leading to the translations of the workflows SW1 and SW2 in the decision denoted RS(SW1) and RS(SW2). The translation of a loop is similar, but the decision over g(localm1,...,localmn) may lead to the end of the flow or the translation of the loop body (SWF). The loop body includes an exec before measures are read again. A Skip task translates to a transition.

C. Execution

The execution component uses a RoboSim code generator to produce ROS nodes (i.e., software components) that can be used by the managed system to execute the adapted workflow. The code generator implements three model transformations. The first is a model-to-model transformation from RoboSim to an intermediate representation with concepts of imperative programming and parallelism. This encodes the RoboSim process algebraic semantics [6] after a behaviour-preserving elimination of some parallelism. In the semantics, parallelism simplifies reasoning, but it is eliminated to optimise execution.

The second transformation is from the intermediate representation to a specific programming language: we use C here. The last transformation implements the RoboSim definition of capabilities using the middleware adopted: ROS here. This transformation uses a library implementing parallelism and communication concepts of the intermediate representation.

TABLE III
MAPPING WORKFLOW TO ROBOSIM STATE MACHINES, WHERE T REFERS A TASKID, SWF TO ANY SUBWORKFLOW, G(M1,...MN) TO A BOOLEAN EXPRESSION OVER MEASURES M1,...,MN. THE FIRST ROW SHOWS THE STRUCTURE OF A TASK, THE SECOND ROW A SEQUENCE, THE THIRD A DECISION, AND THE FOURTH A LOOP.



With the library, the translation is straightforward, so that under the assumption that the library is correct, the resulting code preserves the behaviour of the RoboSim model.

ROS adopts a publish/subscribe mechanism. Accordingly, in the ROS code generator, parallelism is implemented through ROS nodes and communication via topics. Each parallel component of the RoboSim model yields a ROS node that subscribes to input-event topics, stores input data in the local state, publishes messages to output-event topics, and implements the cyclic execution pattern. This is a loop where, in each cycle, (1) inputs are read from the local state; (2) a step function that captures the semantics of the state machine is called as many times as possible until time must be allowed to pass as defined by exec; and (3) the end of the cycle is published. The step function uses the node's publishers to send output events via functions defined in the ROS-specific library.

The capabilities are provided by the platform, a pre-existing ROS node obtained at design time that implements the robot's events and operations. Input events are implemented via a loop that reads information from the robot to decide whether to issue an event, publishing it to the appropriate topic. Output events are implemented as subscribers using the robot's API. Operations are implemented as ROS actions.

The generated ROS node acts as controller to the platform. On initialisation it publishes the output event corresponding to the first task in the workflow. On receipt, the platform executes the operation implementing this task. Once it concludes, the platform publishes a message indicating task completion. This is received by the generated node which publishes the next task. Throughout execution the platform continuously publishes information on the detected state of the world, constantly read by the generated node to resolve the guards.

The generated nodes must be managed nodes to enable adaptation. Once a new adapted node is available, the executor shuts down and destroys the previous implementation, and initialises and activates the new one.

VI. Implementation

The implementation of the MAPE-K loop in ROS consists of five main nodes, one for each MAPE-K component. The knowledge-base node uses a local NoSQL database (TinyDB) to store user information, and provides topics and services to all other nodes, enabling access to the centralised database. The communication protocol between the remaining nodes forms a ring, with each node communicating only with the knowledge base and its two neighbouring nodes.

Those remaining nodes extend a `MAPEK_Node` abstract class, supporting the implementation of an application that uses our architecture. `MAPEK_Node` defines a ROS2 node with one subscriber (input) and one publisher (output), with the names and types of the associated topics given as constructor parameters. This class requires the definition of functions to declare application-specific publishers, subscribers and clients, state, and initialisation, as well as the input callback function. The class provides default implementations for subscriber and publisher creation, message emission through the output publisher, initialisation, and the main function. The latter enforces the use of appropriate executors and callback groups, which enable multiple behaviours to execute in the same node.

The monitor node extends the `MAPEK_NODE` class and defines the input topic as `monitor` of type `Empty`, and the output topic as `analyse` of type `String`; this enables the monitor to request the analysis of the user it has observed. This observation takes place via an application-specific subscription to the topic `user_id`. The monitor requests the current user from the knowledge base via a call to a ROS2 service, compares the result with the new user, and, if they differ, requests the analyser to evaluate the need for replanning.

The analyser node similarly extends `MAPEK_NODE`; it has input topic `analyse` and output `plan`, and waits for a request to analyse the new user. The input topic's callback function uses the knowledge base node's services to obtain the current state: current user, and the rules, events, and measures associated with the current and new users. It compares the rulesets to determine whether replanning is needed. If it is, it sends the planner a request via `plan`.

The planner node waits for requests from the analyser (on the topic `analyse`), recovers the relevant rules, events and measures associated with the new user from the knowledge base node, cleans a local `scratch` folder, assembles the workflow specification and SLEEC model, executes the Java implementation of the workflow adaptation algorithm, stores the generated RoboSim model locally, and requests (on the topic `execute`) the executor to rebuild a ROS implementation.

The executor node waits for a request from the planner (on the topic `execute`), obtains the RoboSim model generated by the planner, executes the Java implementation of the RoboSim code generator to produce a ROS2 implementation of the model and compiles it. This enables the executor to shut down the previous controllers and initialise and activate the newly generated controllers. The MAPE-K loop and the implementations of the RoboSim controllers and state machines interact with a simulation of the robot in Gazebo, but can also be deployed into a real-world robot that supports ROS2.

All the infrastructure of tools and code that enable the implementation of an application that uses our architecture is available [28]. Use of ROS2 makes this work of wide relevance for the robotics community.

VII. Evaluation

To evaluate our approach, we carried out extensive experiments aimed at answering the following research questions.

RQ1 (Correctness): Do all components of our MAPE-K instantiation operate and interact as intended in order to produce dependable adherence to user normative requirements?

RQ2 (Breadth of Applicability): Is our proposed architecture applicable to different types of intelligent agents from different application domains?

RQ3 (Scalability and Overheads): Can workflows of realistic size can be augmented and executed with low latency? For example, the adaptation of its workflow should not delay the hospital utility robot in Section III by more than a few seconds.

In the absence of existing baselines for normative-driven runtime adaptation (as concluded in Section II), our evaluation focuses on assessing the approach's breadth of applicability, scalability and (absolute) overheads. The remainder of this section outlines our experimental methodology and presents the results for each research question.

RQ1 (Correctness). To assess the correctness of the core elements of our MAPE-K instantiation, we performed component-level testing of the implementations of Algorithm 1, and of the RoboSim converter described in Table III. The other components employed by our approach are either part of the RoboStar framework and had been been verified by their authors [26], or were simple enough to be verified through code inspection by the project team. Details of this component-level verification are provided below:

1) Monitor component. This component, with under 70 lines of code designed to detect when the agent's active user is changing, was verified through code inspection.
2) Analysis component. This component, comprising under 80 lines of simple code that compare the SLEEC rules of a new active user to those of the previously active user, was also verified through code inspection.
3) Planning component. This consists of two modules:
   - Implementation of Algorithm 1. As Theorems 1 and 2 already show that Algorithm 1 is correct, we created a suite of JUnit tests to also verify the correctness of its implementation. These tests were designed to maximise coverage of the SLEEC DSL and workflow notation, and passed with branch coverage of 98.1%.
   - Adapted-workflow-to-RoboSim generator. We assembled a test suite designed to achieve full coverage of workflow and SLEEC-rule constructs by combining a subset of the JUnit tests above with three new tests addressing structurally distinct workflow configurations, achieving 98.6% code coverage of the underlying implementation. For each of these tests, the generated RoboSim controller was statically validated for syntax and type consistency, compiled into CSP semantics [30], and confirmed to (i) comply with the adapted workflow's task execution order and, and (ii) disallow unauthorized execution paths via automated trace-refinement testing in the refinement checker FDR4 (https://cocotec.io/fdr/).
4) Execution component. The RoboSim-to-code transformation performed by this component uses a straightforward combination of established RoboSim functionality [6], [26] already validated by its developers.

We implemented the MAPE-K architecture described in Section VI using a Turtlebot 4 research robot in Gazebo simulation. We used the hospital robot workflow and the rules from Section III to demonstrate adaptation, and defined a ROS platform containing the publishers and subscribers that provide an interface between the Turtlebot 4 and the controller produced by the MAPE-K execute phase. The platform also abstracts the tasks described in the workflow Figure 4 as Turtlebot behaviours (see Table IV), and publishes their IDs to Turtlebot's display. The guards that appear either in the initial workflow or emerge through the SLEEC rulesets used in the evaluation, were **floorNeedsCleaning** and **personResting**. Platform mappings implemented these measures as boolean variables initialised to false, and Turtlebot 4 buttons were programmed to change these variables when pressed. ROS publishers send them from the platform to the generated controller continuously.

TABLE IV
ABSTRACTING WORKFLOW TASKS USING TURTLEBOT 4 CAPABILITIES

Workflow Tasks	Turtlebot Capabilities
NotifyPatient	Blinking red LED
AssessRoom	Undock and rotate
DisplayCleaningPlan	Blinking red LED
DustFurniture	Move
SetSilentFloorCleaning	Set the LED for CleanFloor
CleanFloor	Move with red LED or no LED

The full adaptation process—from detection of user ID, to generation, compilation, and deployment of a new ROS node that communicates with the platform via ROS topics—was executed and verified to be correct through manual inspection. This was carried out using the examples of two different users.

We first ran the MAPE-K architecture for a user with no SLEEC rules. The knowledge base retrieved the empty ruleset and the planner produced a RoboSim model implementing Figure 1. The executor generated and set running the ROS node, and we visualised its interaction with the platform in Gazebo simulation. The Turtlebot 4 successfully executed the platform abstractions (Table IV) for AssessRoom, DustFurniture, and CleanFloor. The Turtlebot 4's buttons were used to set the value of the guard floorNeedsCleaning, and was shown to control whether the task CleanFloor executed.

Next we ran the MAPE-K architecture for a user with the SLEEC rules from Table II. The change of user was detected, the correct rules were extracted, and a RoboSim model implementing Figure 4 generated. Again the executor created and set up the ROS node, and we inspected in Gazebo. The Turtlebot 4 successfully executed the platform abstractions (Table IV) for NotifyPatient, AssessRoom, DisplayCleaningPlan (see Figure 5), DustFurniture, SetSilentCleaningMode, and CleanFloor. Through changing the values of floorNeedsCleaning and personResting via the turtlebot buttons, we determined that the correct decision branches were selected.

Thus we show the correctness of our adaptation approach in practice. The code for the full MAPE-K simulation is available at [28], alongside a video documenting these experiments.

RQ2 (Breadth of Applicability). To evaluate the applicability and reusability of our approach, we applied it to two distinct case studies: the hospital service robot above, and a customer support chatbot. These two applications are different in many respects, including their system type, operational domain, and technical implementation. These multifaceted differences are summarised in Table V.

TABLE V
SUMMARY OF CASE STUDY DIFFERENCES

Characteristic	Hospital Service Robot	Customer Support Chatbot
System	Cyber-physical system (robot)	Software agent (chatbot)
Application Domain	Healthcare / Service robotics	E-commerce / Customer service
Platform & Language	ROS2 (Middleware) / C++	Rasa (ML framework) / Python
Implementation Depth	In-house Gazebo simulation; model-to-code synthesis	Off-the-shelf framework; no code synthesis
Managing-managed System Communication	ROS2 Publish/Subscribe topics	JSON files

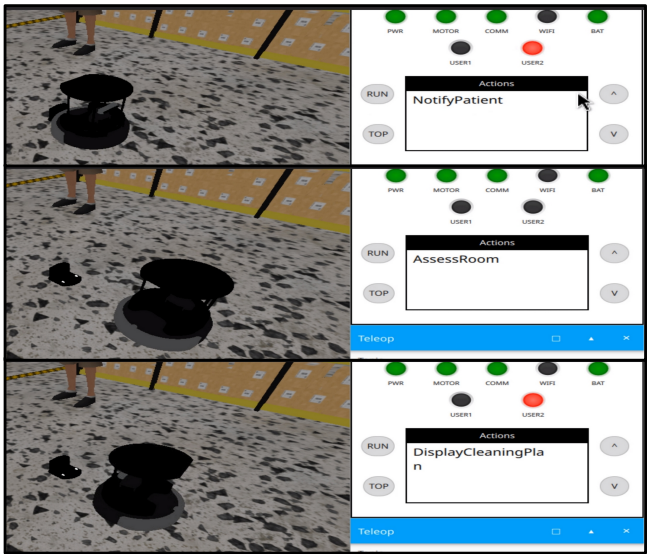


Fig. 5. Screenshots from simulation of the adapted motivating example. In the top image NotifyPatient is indicated by LED2 blinking red. The next image shows AssessRoom in progress: the Turtlebot undocks and rotates. In the third image DisplayCleaningPlan is indicated by the red blinking LED2.

The first case study, involving a service robot, has been presented throughout the paper to illustrate our core methodology. To further demonstrate the breadth of our approach, we also instantiated our architecture for an online chatbot designed to handle customer queries. This chatbot was implemented using Rasa [3], a widely used open-source machine learning framework for building conversational assistants. The chatbot handles two categories of customer queries: inquiries regarding the status of an order, and requests concerning the return of a purchased item. The initial, default workflow encoding these interactions is shown in Figure 6.

In addition to its core functional capabilities (shown in Figure 6), the chatbot includes a suite of predefined capabilities that can be used to specify SLEEC rules for different user profiles: setting a more thorough explanation mode (SetMoreThoroughExplanationMode), issuing an apology (Apologise), transferring the session to a human operator (PassOntoHumanOperator), adopting a sympathetic conversational tone (SetSympatheticTone), warning the user about the chatbot's next action (WarnUser), and (by using a sentiment-analysis module) detecting whether the user is frustrated (userFrustrated). SLEEC rulesets corresponding to different user profiles are assembled, validated and refined beforehand. In a real deployment, these activities would be carried out by developers and representatives from both the user base and

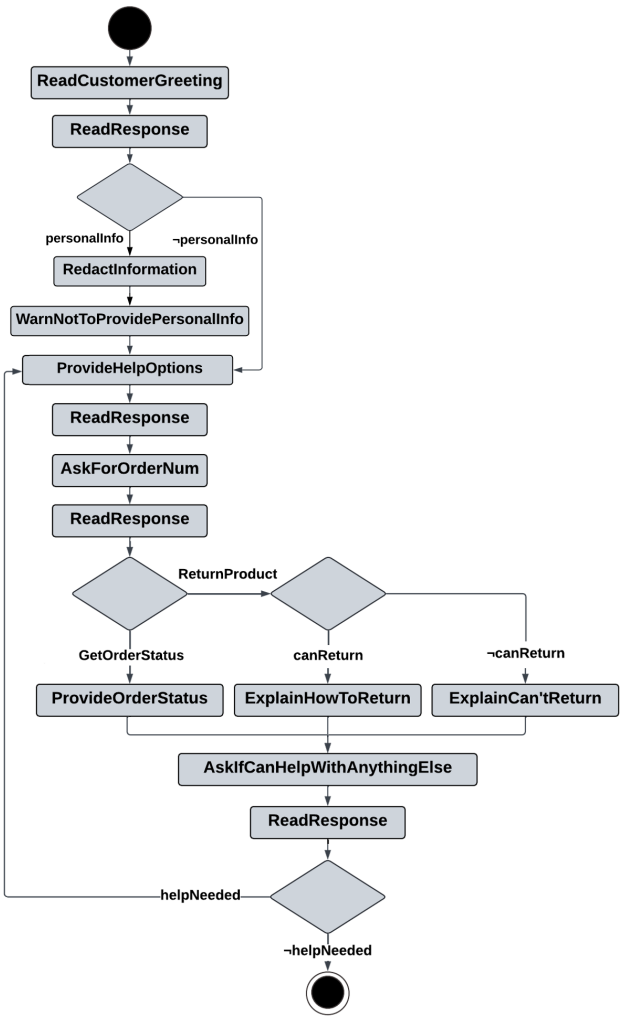


Fig. 6. Initial chatbot workflow

the company operating the chatbot, with consultation from regulators and legal experts as necessary. For the purposes of our case study, we assembled simple SLEEC rulesets for first-time and anxious users (Table VI) ourselves, as the objective of the evaluation was to test the adaptation engine rather than to assess SLEEC-rule elicitation processes.

With these SLEEC rulesets in place, when a user connects to the chatbot, they are associated with a specific user profile (e.g., a first-time user, an anxious user, or an anonymous default user) using standard tracking cookies or through information provided (optionally) by the user. The monitoring component of the MAPE-K architecture identifies whether the active user matches one of these profiles using these cookies,

TABLE VI
SLEEC RULES FOR THE CHATBOT

Chatbot – First-time user's rules	
Rule1f	when ReadCustomerGreetingEnd and newUser then SetMoreThoroughExplanationMode
Rule2f	when ExplainCantReturnEnd then Apologise unless userRequestsHumanOperator then PassOntoHumanOperator
Chatbot - Anxious user's rules	
Rule1a	when ReadCustomerGreetingEnd and userFrustrated then SetSympatheticTone
Rule2a	when RedactInformationStart then WarnUser

or, if cookies are unavailable, by explicitly asking the user at the start of the session whether they are willing to provide such information. Once the profile is identified, the monitor updates the knowledge base, and the corresponding ruleset is used for workflow adaptation. A representative fragment of the adapted chatbot workflow generated for an anxious user (based on the SLEEC rules in Table VI) is shown in Figure 7.

Our MAPE-K architecture (Figure 2) applies to both ROS-based cyber-physical agents and non-ROS software intelligent agents. The Rasa chatbot, which belongs to the second class of supported agents, uses modified variants of the planning and execution MAPE-K stages. Thus, the planning stage only includes the workflow adaptation part, whilst in the execution stage the communication between the MAPE-K loop and the agent occurs via JSON files rather than ROS topics: the adapted workflow is placed directly into the chatbot's directory in JSON format instead of generating compiled controller code. Aside from these technical deployment mechanisms, this JSON-based MAPE-K variant maintains a one-to-one relationship with the ROS-enabled variant. This architectural flexibility is a strength of our approach, showing its ability to support non-ROS, software agents as well as ROS-based cyber-physical agents (Table V). Both implementations are available at [28] and are reusable.

Figure 8 shows the chatbot's communication logs recording the chatbot server activities, and its sessions with three distinct customers (CLIENT1, CLIENT2, and CLIENT3). The server activity log (fourth panel in Figure 8) shows how the chatbot detects a user connection and creates a JSON file with the user's ID, which is detected by the monitoring phase of the MAPE-K architecture (see the fifth panel in Figure 8). The analysis stage selects a user profile based on this user ID and checks if the associated SLEEC rules are different from those used to adapt the previously active workflow. If so, the planning phase produces a new workflow in JSON format via Algorithm 1, before the execution phase delivers this workflow in JSON format to the chatbot.

For CLIENT1 (see the top panel of Figure 8), the monitoring component detects no tracking cookies and the user declines to provide profile preferences at the beginning of the session. As such, the chatbot conducts the session using the default workflow shown in Figure 6 to process the user's item return query. In contrast, CLIENT2 (an anxious user, panel 2) and CLIENT3 (a first-time user, panel 3) each have a set

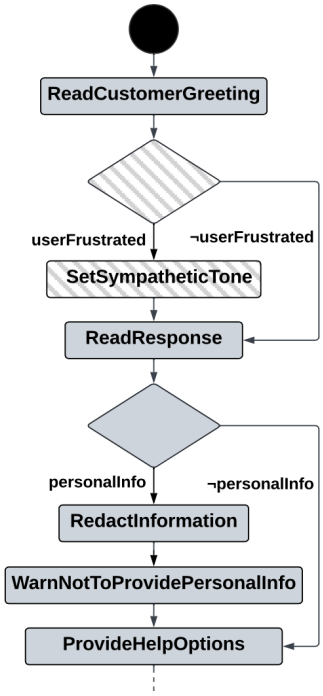


Fig. 7. Fragment of the adapted workflow for CLIENT2's (an anxious user) interaction with the Chatbot. Hashed shapes show elements added to conform with SLEEC rules.

of SLEEC rules that the analyser detects. As a result, the full planning MAPE-K stage is initialised for each of these users (see the bottom panel) to perform adaptation, and then the execution stage provides the resulting SLEEC-adherent workflow to the chatbot's directory in JSON format.

The differences in the chatbot's behaviour can be observed in Figure 8. The chatbot capability SetMoreThoroughExplanationMode is applied for the first-time user (CLIENT3), in line with the SLEEC rule Rule1f from Table VI. This changes the chatbot to deliver more detailed explanations. Similarly, the capability SetSympatheticTone is adopted in the session with the anxious, frustrated user (CLIENT2) to modify the responses of the chatbot to be considerate, in line with the SLEEC rule Rule1a. Neither is applied in the session with the user that lacks SLEEC rules (CLIENT1).

RQ3 (Scalability and Overheads). Scalability is crucial for an algorithm intended for runtime execution in service robots. We conducted experiments to investigate the constants associated with the linear complexity established in Theorem 3. All experiments reported were performed on a standard Ubuntu laptop with an Intel Core i7 processor and 64 GB of memory.

The large workflows required for testing were generated with a tool [28] that assigns each workflow type (Figure 3) a probability of generation. As the workflow approaches a set quantity of tasks, the probability of producing a task approaches 1, ending generation. We provide a similar tool for random generation of sets of SLEEC rules. The tools were designed to maximise coverage of SLEEC and workflows.

We conducted two sets of experiments. For the first, we varied the number of workflow tasks from 1 to 10,000, and considered sets of SLEEC rules with 25, 125, and 625,



Fig. 8. Adaptive chatbot architecture. The server loads and executes the active workflow, while the MAPE-K loop is responsible for generating and updating it dynamically based on user interactions and contextual conditions.

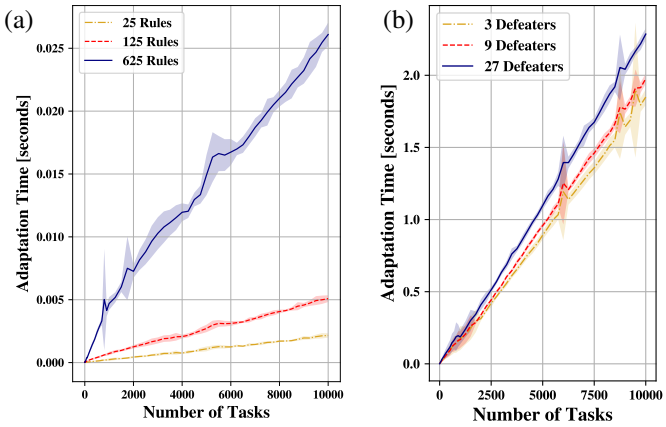


Fig. 9. Workflow adaptation execution times (averaged over 10 runs) for: (a) defeater-free SLEEC rulesets with 25, 125 and 256 rules; and (b) SLEEC rulesets with 10,000 rules and 3, 9 and 27 defeaters per rule. The shaded regions depict the mean ± one standard deviation.

(defeater-free) rules. The results are in Figure 9. The second experiment investigated defeaters. We varied the number of tasks from 1 to 10,000, fixed the number of SLEEC rules at 10,000, and varied the defeaters per rule from 3 to 27. The results are reported in Figure 9. All rule **triggerEvent**s were to mapped to the tasks in the input $workflow$.

The results confirm the linear dependency of execution time on the number of tasks, rules, and rule defeaters. Even for very large workflows and sets of SLEEC rules (up to 10,000 tasks and 625 rules) and for complex configurations (up to 10,000 SLEEC rules with 27 defeaters), the workflow adaptation completed in under 0.03 seconds and under 2.5 seconds, respectively. This performance is perfectly acceptable for the runtime requirements of most agents.

In our Turtlebot-based simulation of the MAPE-K architecture, which included the workflow-to-RoboSim and RoboSim-to-code transformations in addition to the workflow adaptation, the full adaptation cycle took 20 seconds to complete. Code generation and compilation processes within the execution phase took up the majority (over 90%) of this time. For many applications, this total latency is manageable; the hospital cleaning robot, for example, can pause as it enters a SLEEC-governed environment. If this is not appropriate, because a system has to react to SLEEC rule changes immediately, caching of profile-based code (at the point in which patients enter the hospital or customers register) can be beneficial. This type of technique was shown to reduce overheads significantly [15].

Threats to Validity We mitigated **construct validity threats**, which could arise from assumptions in our methodology, by selecting the chatbot and hospital examples. They embed interaction with intelligent agents in practice, allowing comparisons to existing approaches using similar platforms (for example, TurtleBot deployment as in [24]).

We reduced **internal validity threats**, which could introduce bias in determining cause-effect relationships experimentally, by backing up results with formal proofs where feasible. A threat to internal validity is the reliance of our approach on external library functions for the execution MAPE-K com-

ponent, described in Section V-C; to mitigate this threat, we used a library (RoboSim [6], [26]) validated by its authors. Finally, we enabled replication and verification of our findings by making our experimental results publicly available online.

We addressed **external validity threats** by evaluating applicability across agent types and domains, with workflows modelled similar to UML activity diagrams. However, only one of the case studies used the ROS-based variant of the MAPE-K implementation, and therefore the adapted-workflow-to-RoboSim and RoboSim-to-code transformations were only evaluated in this case study and through component-level testing. Evaluation for additional ROS-based cyber-physical agents is required to confirm the generality of the approach.

VIII. Conclusion

We described a new approach for dependable intelligent-agent adaptation to normative rules that change, providing guarantees of soundness with respect to the original mission (workflow) and rules. The work is integrated in the RoboStar framework [26], ensuring it can automatically generate code for use at runtime. The approach is supported by a novel instantiation of the MAPE-K architecture that preserves at the code level the design concepts. This provides traceability from code to rules, and facilitates adaptation. Unlike learning-based agents that leverage LLMs or world models to achieve broad adaptation, our approach is limited to reconfiguring workflows and regenerating controllers that orchestrate pre-existing agent capabilities. However, this limitation comes with the significant advantage of normative-compliance guarantees, which is essential for high-stakes agents.

In future work, we will consider the timed semantics of SLEEC [18], coping with the expected deadlines of rule responses, and dealing with timeouts. In a more ambitious agenda, we plan to: (a) use the normative rules as a basis to design explainable norm-sensitive robots, and (b) extend our approach to use cases involving multiple active users and/or multiple agents.

Acknowledgments

This work was partially funded by RoboStar, the UK EPSRC Grants EP/R025479/1, EP/V026801/2 and EP/V026747/1, the UK Royal Academy of Engineering Grant number CiET1718/45, the EU Horizon project RoboSAPIENS under agreement number 101133807, and the UKRI Responsible AI project DOMINOS.

References

[1] Alfieri, C., et al.: Exosoul: Ethical Profiling in the Digital World. In: HHAI'22. IOS Press (Sep 2022)
[2] Autili, M., et al.: A Software Exoskeleton to Protect and Support Citizen's Ethics and Privacy in the Digital World. IEEE Access **7**, 62011–62021 (2019)
[3] Bocklisch, T., et al.: Rasa: Open Source Language Understanding and Dialogue Management (Dec 2017), arXiv:1712.05181
[4] Boltz, N., et al.: Human Empowerment in Self-adaptive Socio-technical Systems. SEAMS'24 pp. 200–206 (Apr 2024)
[5] Bremner, P., et al.: On proactive, transparent, and verifiable ethical reasoning for robots. Proc. IEEE **107**(3), 541–561 (2019)
[6] Cavalcanti, A.L.C., et al.: Verified simulation for robotics. Science of Computer Programming **174**, 1–37 (2019)
[7] De Sanctis, M., Inverardi, P.: Engineering Ethical-Aware Collective Adaptive Systems. In: REoCAS Colloquium. pp. 238–252. Springer Nature Switzerland (2025)
[8] Donati, D., et al.: On Representing Humans' Soft-Ethics Preferences As Dispositions (Jul 2024), arXiv:2408.06355
[9] Enríquez, J.G., et al.: Robotic Process Automation: A Scientific and Industrial Systematic Mapping Study. IEEE Access **8**, 39113–39129 (2020)
[10] Farahani, A., et al.: On Adaptive Fairness in Software Systems. In: SEAMS'21. pp. 97–103. IEEE (May 2021)
[11] Feng, N., et al.: Analyzing and Debugging Normative Requirements via Satisfiability Checking. In: ICSE'24. pp. 1–12 (2024)
[12] Feng, N., et al.: Towards a formal framework for normative requirements elicitation. In: ASE'23. pp. 1776–1780 (2023)
[13] Feng, N., et al.: Normative requirements operationalization with large language models. In: RE'24. pp. 129–141 (2024)
[14] Floridi, L., et al.: A Unified Framework of Five Principles for AI in Society. Harvard Data Science Review **1**(1) (Jul 2019)
[15] Gerasimou, S., et al.: Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration. In: SEAMS'14. pp. 115–124 (2014)
[16] Getir Yaman, S., et al.: Specification and validation of normative rules for autonomous agents. In: FASE'23. pp. 241–248 (2023)
[17] Getir Yaman, S., et al.: Toolkit for specification, validation and verification of social, legal, ethical, empathetic and cultural requirements for autonomous agents. Science of Computer Programming **236**, 103118 (2024)
[18] Getir Yaman, S., et al.: Specification, Validation and Verification of Social, Legal, Ethical, Empathetic and Cultural Requirements for Autonomous Agents. Journal of Systems and Software **220**, 112229 (2025)
[19] Honig, S., et al.: Understanding and resolving failures in human-robot interaction: Literature review and model development. Frontiers in Psychology **9** (2018). https://doi.org/10.3389/fpsyg.2018.00861
[20] IEEE: IEEE Standard Model Process for Addressing Ethical Concerns during System Design. IEEE Std 7000-2021 pp. 1–82 (Sep 2021)
[21] Liegl, M., et al.: Designing for ethical innovation: A case study on ELSI co-design in emergency. International Journal of Human-Computer Studies **95**, 80–95 (Nov 2016)
[22] Lu, Q., et al.: Towards a roadmap on software engineering for responsible AI. In: CAIN. pp. 101–112. ACM, New York, NY, USA (Oct 2022)
[23] Macenski, S., et al.: Robot Operating System 2: Design, architecture, and uses in the wild. Science Robotics **7**(66) (2022)
[24] Mashal Afzal Memon, et al.: RobEthiChor: Automated context-aware ethics-based negotiation for autonomous robots. Journal of Systems and Software **232**, 112692 (Feb 2026)
[25] Mirani, M., et al.: Towards Efficient Norm-Aware Robots' Decision Making Using Datalog. In: BEWARE@AI*IA (2024)
[26] Miyazawa, A., et al.: Diagrammatic physical robot models. Software and Systems Modelling (2025)
[27] Müller, R., et al.: AgentWork: a workflow system supporting rule-based workflow adaptation. Data and Knowledge Engineering **51**(2), 223–256 (Nov 2004)
[28] Normative-Adaptation project: GitHub repository (2026), https://github.com/a-l-partis/Normative-Agent-Adaptation
[29] Parliament, E.: EU AI Act: first regulation on artificial intelligence (Jun 2023)
[30] Roscoe, A.W.: Understanding concurrent systems. Springer (2010)
[31] Townsend, B., et al.: From pluralistic normative principles to autonomous-agent rules. Minds and Machines **32**(4), 683–715 (2022)
[32] Troquard, N., et al.: Social, legal, ethical, empathetic, and cultural rules: compilation and reasoning. In: AAAI. AAAI'24/IAAI'24/EAAI'24, AAAI Press (2024)
[33] Tzavaras, A., et al.: Ethical and trustworthy Artificial Intelligence. British Standards Institution (2023)
[34] Weber, B., et al.: Beyond rigidity – dynamic process lifecycle support. Computer Science - Research and Development **23**(2), 47–65 (2009)
[35] Weyns, D., et al.: Model-Based Simulation at Runtime for Self-Adaptive Systems. In: ICAC'16. pp. 364–373 (Jul 2016)
[36] Whittle, J., et al.: RELAX: A language to address uncertainty in self-adaptive systems requirement. Requirements Eng **15**(2), 177–196 (2010)