The Journal of Systems and Software 220 (2025) 112229

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Specification, validation and verification of social, legal, ethical, empathetic and cultural requirements for autonomous agents ☆

Sinem Getir Yaman [a,*], Pedro Ribeiro [a], Ana Cavalcanti [a], Radu Calinescu [a], Colin Paterson [a], Beverley Townsend [b]

[a] Department of Computer Science, University of York, UK
[b] York Law School, University of York, UK

ARTICLE INFO

Keywords:
Autonomous agents
Social, legal, ethical, empathetic and cultural requirements
Verification and validation
Sociotechnical systems
Formal methods

ABSTRACT

Autonomous agents are increasingly being proposed for use in healthcare, assistive care, education, and other applications governed by complex human-centric norms. To ensure compliance with these norms, the rules they induce need to be unambiguously defined, checked for consistency, and used to verify the agent. In this paper, we introduce a framework for formal specification, validation and verification of social, legal, ethical, empathetic and cultural (SLEEC) rules for autonomous agents. Our framework comprises: (i) a language for specifying SLEEC rules and rule *defeaters* (that is, circumstances in which a rule does not apply or an alternative form of the rule is required); (ii) a formal semantics (defined in the process algebra tock-CSP) for the language; and (iii) methods for detecting conflicts and redundancy within a set of rules, and for verifying the compliance of an autonomous agent with such rules. We show the applicability of our framework for two autonomous agents from different domains: a firefighter UAV, and an assistive-dressing robot.

1. Introduction

There is huge push to develop and use autonomous agents (software and cyber–physical systems) in high-stakes applications from health and social care, transportation, education, and other domains. Along functional and non-functional requirements such as dependability, performance and utility, a new class of non-functional requirements related to social, legal, ethical, empathetic, and cultural (SLEEC) concerns (Townsend et al., 2022) has become increasingly important and challenging for these applications (Wing, 2021; Moor, 2006; Inverardi, 2022; Calinescu et al., 2019). Despite that recognised importance, there is currently very little support for the elicitation, specification, validation, and verification of SLEEC requirements. Existing research in the area is promising, but only covers specific aspects of the problem. For example, there are results on the study (Bremner et al., 2019; Dennis et al., 2016) and verification (Dennis et al., 2015) of ethical concerns of autonomous agents, modelling of legal requirements for software systems (Boltz et al., 2022), and development of personalised ethical assistant tools based on the moral choices of the user (Alfieri et al., 2022).

In our paper, we build on this early research to provide support for the development of autonomous agents that need to perform tasks that raise SLEEC concerns (Townsend et al., 2022; Floridi, 2018). To that end, we introduce a tool-supported SLEEC requirement specification, consistency validation, and verification framework. It includes a language for defining these concerns as *SLEEC rules* that complement the functional and other non-functional requirements of an autonomous agent. Our language supports the use of defeasible logic (Horty, 2012; Zalta et al., 2005) to allow both the definition of SLEEC constraints and the specification of conditions under which these constraints do not apply or may need to be replaced with alternative constraints. Such conditions are expressed in terms of additional information coming from the environment or the agent components, and are specified within SLEEC rules as *defeaters*. Our language also supports the definition of deadlines for required responses, and of alternative responses when a deadline is missed.

As shown in Fig. 1, our approach for the specification, validation and verification of SLEEC requirements for an autonomous agent under development comprises three steps. In step (i), the set of SLEEC rules for the autonomous agent are formalised in tock-CSP (Baxter et al., 2022), a version of the communicating sequential processes (CSP) algebra (Hoare, 1978) that can describe discrete-time properties. This formalisation is carried out automatically, starting from rules specified by the relevant SLEEC experts (lawyers, ethicists, psychologists, engineers, etc.) in our SLEEC language. Next, in step (ii), our framework

☆ Editor: Prof Neil Ernst.

* Corresponding author.
E-mail address: sinem.getir.yaman@york.ac.uk (S. Getir Yaman).

https://doi.org/10.1016/j.jss.2024.112229
Received 2 November 2023; Received in revised form 7 September 2024; Accepted 23 September 2024
Available online 5 October 2024

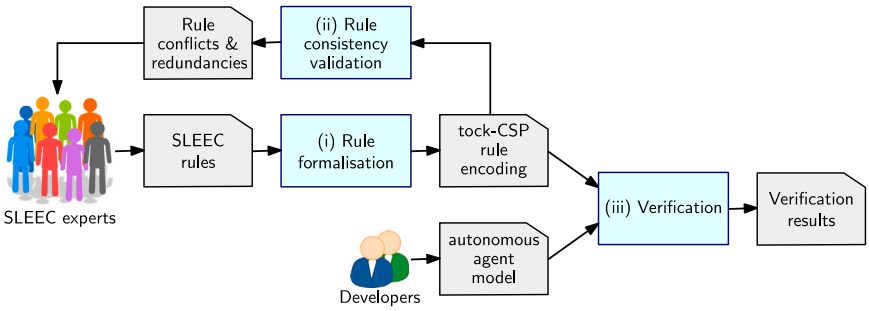


Fig. 1. SLEEC requirement specification, consistency validation, and verification framework.

performs a validation of the SLEEC ruleset consistency, to ensure that its rules are not conflicting, and to identify any redundant rules. The outcome of this rule consistency validation is fed back to the SLEEC experts, enabling them to revise the SLEEC rules in order to address any validation issues. These first two steps of the approach may need to be repeated several times, until all validation errors are resolved. Once a valid set of SLEEC rules is obtained, the compliance of a model of the autonomous agent (provided by its developers) with these rules is verified in step (iii) of the approach.

We have evaluated our framework with two case studies: a firefighter uncrewed aerial vehicle (UAV) and an assistive robot application from the healthcare domain. Their rules have been identified with the help of lawyers and ethicists. Our models representing the agent behaviour are also tock-CSP models, either developed by engineers with formal modelling expertise, or generated automatically from a high-level agent model specified in a domain-specific language for robotics, RoboChart (Miyazawa et al., 2019). This is a diagrammatic notation that can be used to model control software using state machines, time primitives to capture budgets and deadlines, and a simple component model. Since there is support to generate tock-CSP models of RoboChart diagrams automatically, we can use these models to formally verify designs of the autonomous agents' software against SLEEC rules.

The main contributions of our paper include:

(1) A domain-specific language supporting the specification of SLEEC rules for autonomous agents.
(2) The definition of a formal semantics for this language in tock-CSP, catering for the definition of time budgets, deadlines, and timeouts in the rules.
(3) A method for the formal validation of SLEEC specifications, to detect conflicting and redundant rules.
(4) A method for formally verifying the compliance of a tock-CSP-encoded agent specification or design with respect to a set of valid SLEEC rules.
(5) End-to-end tool support for SLEEC requirements specification, consistency validation and verification, using a combination of software components developed by our project and the FDR model checker (Gibson-Robinson et al., 2014).

These contributions complement our proposed SLEEC requirement elicitation process (Townsend et al., 2022), which describes how experts can determine the informal SLEEC rules for an agent at the start of the workflow described in Fig. 1. A preliminary presentation of the tool support available for the framework activities (i) and (ii) from Fig. 1 is available in our short tool paper (Getir Yaman et al., 2023). Here, we extend the results from Getir Yaman et al. (2023) considerably by providing for the first time: (a) a detailed description of our domain-specific language for the specification of SLEEC rules (Section 3); (b) the formal tock-CSP semantics for the SLEEC rules (Section 4); (c) our formal methods for SLEEC rule validation and verification (Section 5); and (d) a significantly expanded evaluation of the SLEEC framework (Section 7).

The remainder of the paper is structured as follows. Section 2 introduces the firefighter UAV, which we use as a running example in later sections. Section 3 presents the SLEEC language, and Section 4 defines its formal semantics. Section 5 describes our approach to conflict and redundancy checking, and our verification process for SLEEC specifications. Section 7 details our evaluation, describing the tool support provided, and the two case studies. Finally, Section 8 covers related work, and Section 9 concludes the paper with a brief summary and a discussion of directions for future work.

2. Running example

To illustrate the concepts, notation, methods, and application of our SLEEC framework, we use as a running example a firefighting UAV inspired by recent research on the use of drones to help tackle wildfires and urban fires (Alon et al., 2021; Cervantes et al., 2018; Innocente and Grasso, 2019). We consider that this UAV is tasked with: (i) using a thermal camera to detect a potential fire at a warehouse; (ii) determining the precise location of the fire (with its depth camera) to report to a human teleoperator; and (iii) using an onboard water spraying system to control the fire until the arrival of the fire brigade.

In addition to these functional goals, we suppose that the firefighter UAV from our running example needs to consider SLEEC concerns arising from its interactions with human firefighters, bystanders and teleoperators. For example, we assume that the UAV has an alarm which sounds when the battery is running low. However, there are social concerns about sounding a loud alarm too close to a human. As another example, we consider that reporting a (potential) fire involves sending video footage of the surveyed building to teleoperators. If, however, bystanders are present in the vicinity of the building, including them in this footage can raise legal and/or ethical privacy concerns. We explain the UAV capabilities and the associated SLEEC concerns in detail as we introduce our SLEEC notation and its semantics in the next sections.

3. The SLEEC Language

Our framework supports the definition of SLEEC rules for an autonomous agent by using a domain-specific language whose syntax we co-developed with autonomous-agent stakeholders including lawyers, ethicists, sociologists and psychologists (Townsend et al., 2022; Getir Yaman et al., 2023). While an alternative syntax/language (e.g., one using `if-then-else` constructs) could be used to express such rules, our stakeholders strongly preferred the syntax we chose, particularly because its constructs improve the readability of rules with multiple defeaters over the alternatives we considered.

As shown in Fig. 2, the set of SLEEC rules for an agent is provided in this language as a **specification** comprising two blocks. The first block (an element of the syntactic category **defBlock**) provides **definitions** for the functional capabilities and parameters of the agent. The second block (**ruleBlock**) defines the actual SLEEC **rules** in terms of those capabilities and parameters. These blocks are described next.

specification	::=	**defBlock ruleBlock**
defBlock	::=	def_start **definitions** def_end
definitions	::=	**definition** | **definition definitions**
definition	::=	event **eventID** | measure **measureID** : **type** | constant **constID** [= **value**]
type	::=	boolean | numeric | scale(**scaleParams**)
scaleParams	::=	**literal** | **literal**, **scaleParams**
ruleBlock	::=	rule_start **rules** rule_end
rules	::=	**rule** | **rule rules**
rule	::=	**ruleID** when **trigger** then **response**
trigger	::=	**eventID** | **eventID** and **mBoolExpr**
mBoolExpr	::=	**measureID** | not **mBoolExpr** | (**mBoolExpr**) | **mBoolExpr relOp value** | **mBoolExpr boolOp boolValue**
response	::=	**constraint** [**defeaters**] | { **constraint** [**defeaters**] }
constraint	::=	**eventID** [within **value timeUnit** [otherwise **response**]] | not **eventID** within **value timeUnit**
defeaters	::=	**defeater** | **defeater defeaters**
defeater	::=	unless **mBoolExpr** [then **response**]
relOp	::=	< | > | <> | <= | >= | =
boolOp	::=	and | or

Fig. 2. BNF syntax of the SLEEC language.

3.1. The definitions block

The **definitions** block (delimited by the keyword pair def_start...def_end) comprises declarations of events and measures that represent capabilities of the agent, and constants that represent parameters of the agent. Events and measures correspond to interactions between the agent and the environment, including any humans, to reflect aspects of the environment that are perceptible or affected by the agent. Measures differ from events in that they carry values, communicated to the agent on demand. A measure corresponds to a query whose answer is always immediately available. An event is an atomic interaction (input or output) that happens sporadically.

A constant represents a value for some parameter of the system configuration; its specific value may or may not be defined. If a value is not defined, the constant represents, for instance, a parameter that is defined at deployment time to reflect the hardware or environment in which the system is deployed, or the preferences of its user.

Example 3.1. An example of a definition block for our firefighter UAV is shown in Listing 1. BatteryCritical is an event that occurs when the battery is very low. This is an abstraction for a battery sensor that provides input to the UAV regarding its own hardware. CameraStart represents an interaction with a teleoperator, who can turn on the camera and start recording. SoundAlarm is associated with a loudspeaker that the UAV can use to sound an alarm. Finally, GoHome represents a navigation capability of the UAV, provided by its motors and the embedded software for using these motors to return the UAV to a home location. In addition, the SLEEC definition block from Listing 1 defines three measures. The first, personNearby, communicates a boolean to indicate whether, using its cameras and associated vision software, the firefighter UAV has detected the presence of a person. Whenever that information is needed, the agent can use the personNearby measure to obtain it. We declare also two measures for the temperature of the air, and the windSpeed level. Finally, the constant ALARM_DEADLINE records a "time budget" for the alarm to sound. We do not give its value in the specification, as we assume it is dependent on the actual deployment of the UAV. □

In summary, an event can be issued by an agent; GoHome is an example. Alternatively, an event can be a request issued by a user, such as CameraStart, or an input from another system component, such as BatteryCritical. Measures, on the other hand, provide to the agent information about the state of the system. Some measures may be known with a high degree of certainty from sensors, such as a temperature sensor or a heart-rate monitor. Others may be inferred from indirect measures or indeed the fusion of multiple sensors. For instance, a user's level of distress may be inferred from heart-rate monitors, images of the user's facial expression, and their tone of voice.

```
def_start
  event BatteryCritical
  event CameraStart
  event SoundAlarm
  event GoHome
  measure personNearby: boolean
  measure temperature: numeric
  measure windSpeed:
          scale(light,moderate,strong)
  constant ALARM_DEADLINE
def_end
```

Listing 1: Definition block for our firefighter robot.

Events, measures, and constants have a unique identifier (**eventID**, **measureID**, and **constID**). By convention, we use identifiers starting with a capital letter for events, and with a lowercase letter for measures. For constants, we use identifiers all in capitals. A measure declaration also defines the **type** of the values it communicates. The supported types are boolean, numeric, and ordinal scales, which introduce some literals and define an order among them. In our example, we have scale(light,moderate, strong) with the implicit order light < moderate < strong.

3.2. The rules block

SLEEC **rules** are defined in a **ruleBlock** (delimited by the keywords rule_start...rule_end). A rule has an identifier (**ruleID**), a **trigger**, and a **response**. A **trigger** is an event and, optionally, a condition (i.e., a Boolean expression) over measures (**mBoolExpr**); when the event in the **trigger** occurs and the condition, if any, is satisfied, then the rule specifies the required response, which defines a **constraint** indicating the event(s) that must or must not occur. The Boolean expression can include conjunctions (and), disjunctions (or), and equalities and inequalities (**relOp**) over numeric and scale measures.

Example 3.2. In Listing 2, Rule1 is concerned with the privacy of persons near the firefighter UAV, when its camera starts recording. The trigger of Rule1 has the event CameraStart and a condition requiring the value of the measure personNearby to be true. The response, in

```
rule_start
  Rule1 when CameraStart and personNearby
        then SoundAlarm
  Rule2 when CameraStart and personNearby
        then SoundAlarm within 2 seconds
  Rule3 when SoundAlarm
        then not GoHome within 5 minutes
  Rule4 when CameraStart then SoundAlarm
        unless personNearby then GoHome
        unless temperature > 35
rule_end
```

Listing 2: Sample SLEEC rules for a firefighter UAV

```
  Rule2_a when CameraStart and personNearby
          then SoundAlarm within 2 seconds
          otherwise GoHome
```

Listing 3: Extended version of Rule2 with otherwise construct

this case, consists of a constraint that requires SoundAlarm to occur, to warn the person that recording is underway. □

A distinctive feature of our SLEEC language is that it can be used to specify rules that impose time constraints: budgets (deadlines) for responses and required alternative responses in the case of a timeout. A time budget is specified using the within construct. The **timeUnit** is provided based on the context under consideration.

Example 3.3. In Listing 2, Rule2 is a more specific variant of Rule1. It has the same trigger as Rule1, but gives a time budget for the response: it must happen within 2 seconds. After all, if the person is not warned early enough, the recording might already have violated their privacy by the time the alarm sounds. □

For situations where the response may not happen within its budget, i.e., when there is a timeout, the otherwise construct can be used to define an alternative response.

Example 3.4. Revisiting Rule2 from Listing 2, we may realise that achieving SoundAlarm within 2 s is not guaranteed. The loudspeakers may be broken, or another SLEEC rule may specify that sounding the alarm is not a socially or ethically acceptable course of action, e.g., because the person is too close to the UAV. In this case, an alternative can be provided as shown in Rule2_a from Listing 3. Here, the UAV is required to return to base (GoHome) if the alarm cannot be sounded within 2 s. □

Thus, the otherwise construct allows us to provide a different response, in the particular case of a timeout arising from the definition of a related within.

Another form of **constraint** requires an event not to happen. In this case, a time budget must be defined via the within construct to not permanently disable the event.

Example 3.5. Rule3 from Listing 2 is triggered when SoundAlarm happens. In this case, for social reasons, the "output" event GoHome is blocked for 5 min. It may be the case, for example, that the teleoperators are in the home region, and the UAV should not come close to them while the alarm is sounding. □

The environment in which an autonomous agent is deployed is generally highly complex and the assumptions that underpin SLEEC rules may be invalid under certain conditions. To support resilience in such environments, we allow the use of defeasible reasoning to cope with scenarios leading to circumstances that outweigh or disable a constraint (Brunero, 2021). Defeasible reasoning is supported in our SLEEC language via chains of potentially nested unless clauses, which allow normative rules to be modified in light of additional information obtained from measures.

Example 3.6. Listing 2 presents a Rule4 for constraining CameraStart with a view different from that in the previous rules. In Rule4, CameraStart is required to lead to SoundAlarm. We have, however, an unless clause with a condition depending on the value of the Boolean measure personNearby. If this measure is true, then Rule4 requires the UAV to GoHome, so as to avoid the anti-social action of sounding an alarm near a person, likely a human firefighter. This is, however, once again defeated by a second unless clause based on the temperature measure. If this measure is greater than 35 °C, no response is required. That is because such a high temperature is deemed an indication that there is a fire nearby, which trumps the legal/ethical concerns about filming a bystander, and the firefighter UAV is permitted to use its camera without restrictions. □

Overall, multiple defeaters (grouped, if needed, within curly brackets {…} to indicate the constraint they apply to) alongside time constrains and timeouts can be defined in SLEEC rules. So, the semantics of the rules (formalised in the next section) can be rather subtle, and the interactions between multiple rules can be unexpected.

4. SLEEC semantics

This section defines the semantics of SLEEC using the process algebra tock-CSP, a timed variant of CSP (Roscoe, 1998). CSP is part of a large family of notations for specifying concurrent systems (Milner, 1983, 1999; Bergstra and Klop, 1985), and is distinctive in its denotational semantics, giving rise to notions of refinement useful for stepwise development. A powerful model checker called FDR (Gibson-Robinson et al., 2014) supports the validation and verification of CSP (and tock-CSP) specifications. In Section 4.1, we give a brief introduction to tock-CSP. Section 4.2 gives an overview of our semantics, with an example. The detailed semantics of SLEEC triggers and responses are described in Sections 4.3 and 4.4, respectively.

We note that alternative formalisms (such as variants of first-order logic or temporal logic) could have been used to capture the semantics of SLEEC rules. However, tock-CSP has the major advantage of easily supporting both (i) the validation of SLEEC rule sets (to identify conflicting and redundant rules, see Section 5.1) and (ii) the verification of autonomous-agent compliance with such rules (see Section 5.3). In contrast, other formalisms would only provide immediate support for one of these important activities. For instance, first-order logic could have been used to validate the consistency of a SLEEC ruleset, but not to verify autonomous-agent compliance using model checking. As another example, temporal logics could have been used to enable rule verification, but not ruleset validation.

4.1. Overview of tock-CSP

CSP processes specify patterns of interaction via synchronisation on channels, taking into account (non)determinism, deadlock, livelock, and termination. Communications between parallel processes and with the environment are achieved via channels. These communications are instantaneous, atomic CSP events, that can carry values: inputs and outputs. The dialect tock-CSP, in addition, allows processes to specify time budgets and deadlines using a special CSP event called *tock*. It represents the passage of one time unit, which we assume here to be 1 s.

In Table 1 we summarise the tock-CSP operators that we use in this paper. To illustrate the notation we present a tock-CSP process for a UAV firefighter's autopilot.

Table 1
List of tock-CSP operators, with basic processes at the top, followed by composite processes: P and Q are metavariables that stand for processes, d for a numeric expression, e for an event, a and c for channels, x for a variable, I for a set, v for an expression, g for a condition, and X for a set of events. For a channel c, $\{|c|\}$ is a set of events; if c is a typed channel then events are constructed using the dot notation, so that $\{|c|\} = \{|c.v_0, \ldots, c.v_n|\}$, where v_i ranges over the type of c.

Process	Description
Skip	**Termination**: terminates immediately
Wait(d)	**Delay**: terminates exactly after d units of time have elapsed
$e \rightarrow P$	**Prefix operator**: initially offers to engage in the event e while permitting any amount of time to pass, and then behaves as P
$a?x \rightarrow P$	**Input prefix**: same as above, but offers to engage on channel a with any value, and stores the chosen value in x
$a?x : I \rightarrow P$	**Restricted input prefix**: same as above, but restricts the value of x to those in the set I
$a!v \rightarrow P$	**Output prefix**: same as above, but initially offers to engage on channel a with a value v
if g **then** P **else** Q	**Conditional**: behaves as P if the predicate g is true, and otherwise as Q
$P \Box Q$	**External choice** of P or Q made by the environment
$P \; ; \; Q$	**Sequence**: behaves as P until it terminates successfully, and, then it behaves as Q
$P \setminus X$	Hiding: behaves like P but with all communications in the set X hidden
$P \;\|\|\|\; Q$	**Interleaving**: P and Q run in parallel and do not interact with each other
$P \;[\![X]\!]\; Q$	**Generalised parallel**: P and Q must synchronise on events that belong to the set X, with termination occurring only when both P and Q agree to terminate
$P \triangle Q$	Interrupt: behaves as P until an event offered by Q occurs, and then behaves as Q
$P \triangle_d Q$	**Strict timed interrupt**: behaves as P, and, after exactly d time units behaves as Q
$d \blacktriangleleft P$	**Deadline for visible interaction**: engages in an event of P in at most d time units
$\Box\, i : I \bullet P(i)$	**Replicated external choice**: offers an external choice over processes $P(i)$ for all i in I

Example 4.1. In this example, we define a process *AP* to model a simple autopilot. We use events *Navigate* and *Track* to represent capabilities of the drone to move to an area of interest (*Navigate*) and then search (*Track*) a fire. In *AP*, we specify that the autopilot first accepts a request to *Navigate* and then (→) starts *Track*ing. When a fire is found, *AP* behaves as defined in the process *FIRE*. When *FIRE* terminates, in sequence (;) *AP* recurses.

$$AP = Navigate \rightarrow Track \rightarrow FIRE \; ; AP$$

$$FIRE = 0 \blacktriangleleft (temperature?t \rightarrow$$
$$\textbf{if } t > 35$$
$$\textbf{then } 1 \blacktriangleleft (SoundFireAlarm \rightarrow \textbf{Skip})$$
$$\textbf{else Skip})$$

In *FIRE* the autopilot reads a value t using a channel *temperature* (*temperature*?t), and then behaves as defined by a conditional. The process with the communication *temperature*?t follow by the conditional is the argument of the operator (◀) that defines a deadline, here 0, for that process to exhibit visible behaviour. The deadline defines the number of time units that can pass, that is, the number of *tock* events that can occur, before the visible behaviour happens. With the deadline 0, we specify that the input must happen immediately: no *tock* events are allowed before the communication on the channel *temperature* occurs. In the conditional, if the temperature read (t) is greater than 35 Celsius, then an event *SoundFireAlarm* is required to happen in at most 1 time unit (1 s here). So, *SoundFireAlarm* can happen before a *tock* or after at most one *tock*. Afterwards, *FIRE* terminates (*Skip*) immediately: no more *tock* events can happen. If t is less than or equal to 35, *FIRE* just terminates. □

We note that the CSP event *temperature* corresponds to the SLEEC measure **temperature** in Listing 2. The other events are not mentioned there. In general, we can expect SLEEC rules to be concerned with some, but not all, capabilities of an agent. Verification needs to take that into account, as we discuss in Section 5.3.

4.2. Overview of SLEEC semantics

The semantics of a SLEEC **specification** as defined in Fig. 2 is given by a function $[\![\_]\!]_S$ defined in Table 2. This function maps the **specification** to a tock-CSP process, and is defined in terms of two other functions, $[\![\_]\!]_{DS}$ and $[\![\_]\!]_{RS}$, which capture the semantics of the definitions dB and rules rB of the **specification**. The semantic definitions in Table 2 are mechanised in our SLEEC tool (Getir Yaman et al., 2023), which automates the generation of the tock-CSP semantics of a SLEEC **specification** (see Section 6).

The semantics of **definitions** from Fig. 2 is given by corresponding declarations of channels and constants representing the SLEEC events, measures, and constants. The types boolean and numeric are given semantics as **Bool** and **Int**. For a scale type, the semantics is a CSP **datatype** that declares its literal parameters, and an associated Boolean function to record the order between those literals. A SLEEC constant becomes a CSP constant.

Example 4.2. In Fig. 3, we present the declarations for the definitions in Listing 1. For each event and measure, we have a **channel** declaration. For the type of the measure windSpeed, we define a **datatype** *STwindSpeed* and a Boolean function *STlewindSpeed* with arguments *v*1*windSpeed* and *v*2*windSpeed* (of type *STwindSpeed*). If *v*1*windSpeed* is the first literal light, then it is guaranteed to be less than or equal to *v*2*windSpeed*, no matter the value of *v*2*windSpeed*. If, however *v*1*windSpeed* is moderate, then the inequality holds if *v*2*windSpeed* is not light, since it is then at least moderate as well. Finally, if *v*1*windSpeed* is strong, then the inequality holds if, and only if, *v*2*windSpeed* is strong too. For model checking, we need to define a value for the constants. In this example, we use 3 as a time unit for the value of ALARM_DEADLINE. □

The recursive definition of $[\![\_]\!]_{DS}$ is given by two equations. For a single definition def, the semantics is given by another function $[\![\_]\!]_D$. For a list of definitions def defS, containing a single definition def followed by a list defS, the semantics is the sequence of CSP declarations determined by $[\![\mathsf{def}]\!]_D$ to capture the semantics of def, followed by the CSP declarations defined by a recursive application of $[\![\_]\!]_{DS}$ to defS. We do not consider the empty list of definitions, since a SLEEC specification defines restrictions on the use of the capabilities of the agent, and without a declaration of capabilities, there is no sensible specification.

The equations defining $[\![\_]\!]_D$ consider each form of **definition** separately. We assume that identifiers in SLEEC satisfy the usual lexical restrictions adopted in CSP, so that, for example, events and measures are represented by channels of the same name. For a constant, we

Table 2
Rules that define a tock-CSP semantics for SLEEC. We use the following *metavariables* in the definitions of the rules: def as a metavariable to stand for an element of the syntactic category definitions, defS to stand for an element of definitions, eID for an eventID, mID for a measureID, T for a type, cID for a constID, v for a value, sp and subscripted counterparts for a scaleParams, r for a rule, rrS for an element of rules, rID for a ruleID, trig for a trigger, and finally resp for a response. These metavariables are also used in rules in Tables 3 and 4.

$[\![\text{def\_start dB def\_end rule\_start rB rule\_end}]\!]_S$	$= [\![\text{dB}]\!]_{DS} \quad [\![\text{rB}]\!]_{RS}$
$[\![\text{def}]\!]_{DS}$	$= [\![\text{def}]\!]_D$
$[\![\text{def defS}]\!]_{DS}$	$= [\![\text{def}]\!]_D \quad [\![\text{defS}]\!]_{DS}$
$[\![\text{event eID}]\!]_D$	$=$ **channel** eID
$[\![\text{measure mID : T}]\!]_D$	$=$ **channel** mID : $[\![\text{T, mID}]\!]_T$
$[\![\text{constant cID = v}]\!]_D$	$=$ cID = v
$[\![\text{boolean, mID}]\!]_T$	$=$ **Bool**
$[\![\text{numeric, mID}]\!]_T$	$=$ **Int**
$[\![\text{scale}(\text{sp}_1, \ldots, \text{sp}_n), \text{mID}]\!]_T$	$=$ *ST*mID **datatype** *ST*mID = $\text{sp}_1 \mid \ldots \mid \text{sp}_n$ *STle*mID(*v*1mID, *v*2mID) = **if** *v*1mID == sp1 **then true** **else** (**if** *v*1mID == sp_2 **then** *v*2mID $\notin \{\text{sp}_1\}$ **else** … **else** *v*2mID == sp_n)
$[\![\text{r}]\!]_{RS}$	$= [\![\text{r}]\!]_R$
$[\![\text{r rS}]\!]_{RS}$	$= [\![\text{r}]\!]_R \quad [\![\text{rS}]\!]_{RS}$
$[\![\text{rID when trig then resp}]\!]_R$	$=$ rID = *Trigger*rID ; *Monitoring*rID ; rID *Trigger*rID = $[\![\text{trig}, \alpha_E(\text{resp}), \textbf{Skip}, \textit{Trigger}\text{rID}]\!]_{TG}$ *Monitoring*rID = $[\![\text{resp}]\!]_{RDS}$

$$
\begin{aligned}
&\textbf{channel}\ BatteryCritical\\
&\textbf{channel}\ CameraStart\\
&\textbf{channel}\ SoundAlarm\\
&\textbf{channel}\ GoHome\\
&\textbf{channel}\ personNearby : \textbf{Bool}\\
&\textbf{channel}\ temperature : \textbf{Int}\\
&\textbf{channel}\ windSpeed : STwindSpeed\\
&\textbf{datatype}\ STwindSpeed = light \mid moderate \mid strong\\
&STlewindSpeed(v1windSpeed, v2windSpeed) =\\
&\quad \textbf{if}\ v1windSpeed == light\\
&\quad \textbf{then true}\\
&\quad \textbf{else}\ (\ \textbf{if}\ v1windSpeed == moderate\\
&\qquad \textbf{then}\ (v2windSpeed \notin \{light\})\\
&\qquad \textbf{else}\ v2windSpeed == strong\ )\\
&ALARM\_DEADLINE = 3
\end{aligned}
$$

Fig. 3. CSP declarations for the definitions in Listing 1.

assume that a value is given to enable model checking. The type used in the declaration of a measure channel is given by the semantic function $[\![\_]\!]_T$ whose arguments are the type T and the identifier mID of the measure. The equations defining $[\![\_]\!]_T$ for boolean and numeric are straightforward. For a scale type, we use the name of the measure in defining the corresponding CSP type declarations.

For simplicity, we use an informal notation to represent a scale type with n parameters sp_1 to sp_n, namely, scale($\text{sp}_1, \ldots, \text{sp}_n$). The definition of a formal generative function for the semantics of a measure with such a type is, however, straightforward. The name of the **datatype** defined is that of the measure, that is, the argument mID, prefixed with *ST*. The name of the Boolean function that defines its order is prefixed with *STle* instead. The recursive definition of the semantic function $[\![\_]\!]_{RS}$ for a list of rules is similar to that of $[\![\_]\!]_{DS}$, but is based on the semantic function $[\![\_]\!]_R$ for a **rule**. The semantics of each rule

$$
\begin{aligned}
&Rule2 = TriggerRule2\ ;\ MonitoringRule2\ ;\ Rule2\\
&TriggerRule2 =\\
&\quad \textbf{let}\ MTrigger = 0 \blacktriangleleft (personNearby?vpersonNearby \rightarrow\\
&\qquad \textbf{if}\ vpersonNearby == true\\
&\qquad \textbf{then Skip}\\
&\qquad \textbf{else}\ TriggerRule2)\\
&\quad \textbf{within}\ \ CameraStart \rightarrow MTrigger\\
&\qquad\qquad \Box\\
&\qquad\qquad SoundAlarm \rightarrow TriggerRule2\\
&MonitoringRule2 = 2 \blacktriangleleft (SoundAlarm \rightarrow \textbf{Skip})
\end{aligned}
$$

Fig. 4. Semantics of Rule2 in Listing 2.

is given by a process, named after that rule, and defined using two processes that capture the meaning of its **trigger** and of its **response**. The process for every rule is defined by composing in sequence a *Trigger* and a *Monitoring* process. This reflects the fact that a rule imposes no constraints until its trigger is observed. At that point, it monitors (that is, determines) the allowed behaviour to enforce the response.

Example 4.3. For Rule2 in Listing 2, the CSP process that defines its semantics is shown in Fig. 4. The behaviour of the process *Rule*2 is initially defined by that of *TriggerRule*2. When the trigger of Rule2 is observed, *TriggerRule*2 terminates (via the **Skip** from the conditional statement), and the process *MonitoringRule*2 takes over. When the response happens, *MonitoringRule*2 terminates and *Rule*2 recurses. □

In Table 2, the definition of $[\![\_]\!]_R$ uses the identifier rID of the rule to assemble the identifiers of the *Trigger* and *Monitoring* processes. The definition of the *Trigger* process is given by the semantic function $[\![\_]\!]_{TG}$ whose arguments are the trigger of the rule, the alphabet of events, that is, the set of all events used in the response of the rule, and two continuation processes. The alphabet of events is given by the function $\alpha_E(\text{resp})$. The first continuation process determines the behaviour when

Table 3
Rules that define a tock-CSP semantics for SLEEC triggers. Additional metavariables used here are as follows: AR for an alphabet (set) of events, sp and fp for tock-CSP processes, mBE for an mBoolExpr, and MIDs for a list of measureID elements.

$[\![\text{eID}, \text{AR}, \text{sp}, \text{fp}]\!]_{\text{TG}}$	$= \text{eID} \rightarrow \text{sp} \,\Box\, (\Box\ \text{e} : \text{AR} \bullet \text{e} \rightarrow \text{fp})$
$[\![\text{eID and mBE}, \text{AR}, \text{sp}, \text{fp}]\!]_{\text{TG}}$	$=$ **let** *MTrigger* $= [\![\alpha_{\text{ME}}(\text{mBE}), \text{mBE}, \text{sp}, \text{fp}]\!]_{\text{ME}}$ **within** $\text{eID} \rightarrow \mathit{MTrigger} \,\Box\, (\Box\ \text{e} : \text{AR} \bullet \text{e} \rightarrow \text{fp})$
$[\![\langle\rangle, \text{mBE}, \text{sp}, \text{fp}]\!]_{\text{ME}}$	$=$ **if** $\text{norm}(\text{mBE})$ **then** sp **else** fp
$[\![\langle \text{mID} \rangle \frown \text{mIDs}, \text{mBE}, \text{sp}, \text{fp}]\!]_{\text{ME}}$	$= 0 \blacktriangleleft (\text{mID}?\nu\text{mID} \rightarrow [\![\text{mIDs}, \text{mBE}[\nu\text{mID}/\text{mID}], \text{sp}, \text{fp}]\!]_{\text{ME}})$

the trigger happens. In the definition of $[\![\_]\!]_{\text{R}}$, this is **Skip**, since the *Trigger* process must terminate in this case. The second continuation process determines the behaviour if the event of the trigger takes place, but its condition does not hold. In the definition of $[\![\_]\!]_{\text{R}}$, this is the *Trigger* process itself, since in this case the *Trigger* process must recurse. To define the *Monitoring* process, we use the semantic function $[\![\_]\!]_{\text{RDS}}$. Its argument is the **response** that is to be monitored.

We define $[\![\_]\!]_{\text{TG}}$ next, and $[\![\_]\!]_{\text{RDS}}$ in Section 4.4. Those familiar with the use of CSP to specify properties might observe that we do not adopt the usual approach that considers the overall alphabet of events, and defines a rule that imposes no restrictions outside its own alphabet. That approach is convenient for verification by refinement, but does not easily support checks for conflicts and redundancy. With our semantics, we support validation, and, for verification, we adopt a more elaborate notion of correctness, using refinement and priorities (cf. Section 5).

4.3. Triggers

The definition of $[\![\_]\!]_{\text{TG}}$ is given in Table 3. For a **trigger** that has just an event eID, the process is a synchronisation on that event followed by the argument process sp that defines the continuation when the trigger happens. A choice allows the response events to happen freely, but their occurrence leads to a recursion so that the rule is not enforced if the trigger has not happened.

If the **trigger** has a Boolean mBE expression on measures, the process is defined using **let** and **within** clauses. The actual process is defined in the **within** clause, but in its definition we can use processes named in the **let** clause. In the process $[\![\text{eID and mBE}, \text{AR}, \text{sp}, \text{fp}]\!]_{\text{TG}}$, we have the synchronisation on eID followed by a process *MTrigger*, defined in the **let** clause using a semantic function $[\![\_]\!]_{\text{ME}}$.

Example 4.4. As shown in Fig. 4, if the trigger has a Boolean expression on measures, *MTrigger* first reads the values of the measures urgently. For Rule2 in Listing 2, the condition is just on the measure personNearby, so *MTrigger* inputs a value *vpersonNearby* using the channel *personNearby*. Afterwards, a conditional checks the measure expression. If it holds, the trigger has occurred, and *MTrigger* terminates, leading to *TriggerRule2* terminating as well. Otherwise, *MTrigger* recurses back to the *Trigger* process to wait for the trigger event again. □

The function $[\![\_]\!]_{\text{ME}}$ takes the list of measures used in the Boolean expression as arguments, that measure condition itself, and the continuation processes. In the definition of $[\![\_]\!]_{\text{TG}}$, the first argument $\alpha_{\text{ME}}(\text{mBE})$ of $[\![\_]\!]_{\text{ME}}$ is defined by a function α_{ME}, similar to α, but providing just measure identifiers used in the Boolean expression.

The inductive definition of $[\![\_]\!]_{\text{ME}}$ considers separately an empty list $\langle\rangle$ of measures and a list with at least one measure mID. In the process defined by this function, the value νmID of each measure mID is read urgently in sequence ($\rightarrow$). That value is then substituted for mID in the expression mBE. Once all of the measures are input, a conditional checks the value of the resulting expression.

In detail, for a list of identifiers $\langle \text{mID} \rangle \frown$ mIDs, the definition of $[\![\_]\!]_{\text{ME}}$ defines a process that reads the value of mID and records it into a local variable named νmID. To make that urgent, it uses the operator $\blacktriangleleft$ with deadline 0 over a process that starts with the communication $\text{mID}?\nu\text{mID}$. The behaviour that follows is defined by the process characterised by a recursive application of $[\![\_]\!]_{\text{ME}}$.

In that application of $[\![\_]\!]_{\text{ME}}$, the remaining measures in mIDs are considered. Moreover, the Boolean expression is changed to refer to the variable νmID, where the measure mID is used. We use $\text{mBE}[\nu\text{mID}/\text{mID}]$ to denote the Boolean expression obtained by replacing the occurrences of mID with νmID. In our example semantics for Rule2 in Listing 2, personNearby becomes *vpersonNearby*.

If the first argument of $[\![\_]\!]_{\text{ME}}$ is the empty list of measures, then all the relevant measure values have been read, and the Boolean expression is defined in terms of those values (recorded in local variables). So, $[\![\_]\!]_{\text{ME}}$ defines a conditional process that specifies the appropriate continuation behaviour depending on the measure condition.

The actual condition evaluated is specified using a normalisation function. In $\text{norm}(\text{mBE})$ the SLEEC relational operators applied to literals of scale types in mBE are encoded using the comparator functions of those scale types. (Strictly speaking, $\text{norm}(\_)$ requires an argument defining the type of the measures and the names of the comparator functions for the scale types.) Additionally, the use of a measure *mID* as a Boolean is transformed to an equality *mID* == **true** as required by the CSP notation.

4.4. Responses

The semantic function $[\![\_]\!]_{\text{RDS}}$ for **response** definitions is specified in Table 4. The semantics of a **response** enclosed in curly brackets is just the semantics of its **constraint** and **defeaters** itself. We omit that simple definition from Table 4. The semantics of a response that has just a constraint is given by the function $[\![\_]\!]_{\text{C}}$.

Example 4.5. Rule2 in Listing 2 provides an example of a response that has just a constraint, that is, the rule contains no defeaters. The *Monitoring* process in its semantics, shown in Fig. 4, captures the time constraint in the response. It requires that *SoundAlarm* takes place within 2 time units (2 s as indicated in Rule2). □

The SLEEC rules can refer to a variety of time units. To give semantics, we can either assume that *tock* represents the passage of a minimal period of time that can be considered (1 ms, for example), or calculate the greatest common divisor of all periods of time referenced, and adopt that to define the meaning of *tock*. Whatever the solution, when using a time period definition we need to normalise the value to describe it in terms of a number of *tock* events. For instance, if *tock* is deemed to represent a second, then "1 min" should be normalised to 60. As said, in our examples and in our tool, for simplicity, we take the view that 1 time unit corresponds to 1 s.

The definition of $[\![\_]\!]_{\text{C}}$ has one equation for each possible form of constraint. If it is just an event, the constraint process defined by $[\![\_]\!]_{\text{C}}$ requires that event to be accepted and then terminates. We recall that termination indicates that the constraint has been satisfied, and the rule process can recurse and wait for the next trigger.

If there is a time budget within v tU defining a number v of time units given by tU, then the process for the event is included in an $\text{norm}(\text{v}, \text{tU})\blacktriangleleft$. The deadline $\text{norm}(\text{v}, \text{tU})$ is determined using a normalisation function to calculate the number of *tock* events allowed, as explained above.

Table 4
Rules for the tock-CSP semantics of SLEEC responses. Additional metavariables used here are: const for a constraint, ARDS for a set of events, mp for a process, tU for a timeUnit, n for an index (a natural number), dfts for an element of defeaters, and dft for a defeater.

$[\![\mathsf{const}]\!]_{\mathsf{RDS}}$	$= [\![\mathsf{const}]\!]_{\mathsf{C}}$
$[\![\mathsf{const\ dfts}]\!]_{\mathsf{RDS}}$	$=$ **let** $[\![\langle \mathsf{const}\rangle \frown \mathsf{dfts}{\restriction_{\mathsf{RP}}}, 1]\!]_{\mathsf{LRDS}}$ **within** $[\![\alpha_{\mathsf{ME}}(\mathsf{dfts}), \mathsf{dfts}, \#\mathsf{dfts}+1]\!]_{\mathsf{CDS}}$
$[\![\mathsf{eID}]\!]_{\mathsf{C}}$	$=$ eID $\rightarrow$ **Skip**
$[\![\mathsf{eID\ within\ v\ tU}]\!]_{\mathsf{C}}$	$=$ norm(v, tU) $\blacktriangleleft$ (eID $\rightarrow$ **Skip**)
$[\![\mathsf{eID\ within\ v\ tU\ otherwise\ resp}]\!]_{\mathsf{C}}$	$=$ (eID $\rightarrow$ **Skip**) $\triangle_{\mathsf{norm(v,tU)}}$ ($[\![\mathsf{resp}]\!]_{\mathsf{RDS}}$)
$[\![\mathsf{not\ eID\ within\ v\ tU}]\!]_{\mathsf{C}}$	$=$ **Wait**(norm(v, tU))
$[\![\langle \mathsf{resp}\rangle, \mathsf{n}]\!]_{\mathsf{LRDS}}$	$=$ *Monitoring*n $=$ $[\![\mathsf{resp}]\!]_{\mathsf{RDS}}$, provided resp $\neq$ **NoRep**
$[\![\langle \mathbf{NoRep}\rangle, \mathsf{n}]\!]_{\mathsf{LRDS}}$	$=$ *Monitoring*n $=$ **Skip**
$[\![\langle \mathsf{resp}\rangle \frown \mathsf{resps}, \mathsf{n}]\!]_{\mathsf{LRDS}}$	$=$ $[\![\langle \mathsf{resp}\rangle, \mathsf{n}]\!]_{\mathsf{LRDS}}$ $[\![\mathsf{resps}, \mathsf{n}+1]\!]_{\mathsf{LRDS}}$
$[\![\langle\rangle, \mathsf{dfts}, \mathsf{n}]\!]_{\mathsf{CDS}}$	$=$ $[\![\mathsf{dfts}, \mathit{Monitoring}\,1, \mathsf{n}]\!]_{\mathsf{EDS}}$
$[\![\langle \mathsf{mID}\rangle \frown \mathsf{mIDs}, \mathsf{dfts}, \mathsf{n}]\!]_{\mathsf{CDS}}$	$= 0 \blacktriangleleft (\mathsf{mID}?v\mathsf{mID} \rightarrow [\![\mathsf{mIDs}, \mathsf{dfts}[v\mathsf{mID}/\mathsf{mID}], \mathsf{n}]\!]_{\mathsf{CDS}})$ $[\![\mathsf{unless\ mBE, fp, n}]\!]_{\mathsf{EDS}}$
$[\![\mathsf{unless\ mBE, fp, n}]\!]_{\mathsf{EDS}}$	$=$ **if** norm(mBE) **then** *Monitoring*n **else** fp
$[\![\mathsf{unless\ mBE\ then\ resp, fp, n}]\!]_{\mathsf{EDS}}$	$=$ **if** norm(mBE) **then** *Monitoring*n **else** fp
$[\![\mathsf{dfts\ dft, fp, n}]\!]_{\mathsf{EDS}}$	$=$ $[\![\mathsf{dft}, [\![\mathsf{dfts}, \mathsf{fp}, \mathsf{n}-1]\!]_{\mathsf{EDS}}, \mathsf{n}]\!]_{\mathsf{EDS}}$

```
Rule4_a when CameraStart then SoundAlarm
         unless personNearby then GoHome
```

Listing 4: Simpler version of Rule4

If the possibility of a timeout is considered, via an otherwise clause, instead of a $\blacktriangleleft$, we use a timed interrupt ($\triangle$) to specify that, if the budget norm(v, tU) is used up, the process that captures the semantics of the response associated with the otherwise takes over.

Example 4.6. The *MonitoringRule2_a* process for the SLEEC Rule2_a in Listing 3 is shown below.

(*SoundAlarm* $\rightarrow$ **Skip**) $\triangle_2$ (*GoHome* $\rightarrow$ **Skip**)

If after 2 s, for whatever reason, the alarm cannot be sounded, then *GoHome* is required. □

Finally, for a constraint that forbids the occurrence of an event, the semantics is the process **Wait**(norm(v, tU)) that pauses: only allows time to pass, that is, *tock* events to happen, for v tU time units, and then terminates.

If a response has one or more defeaters, $[\![\_]\!]_{\mathsf{RDS}}$ defines a process using **let** and **within** clauses. The **let** clause defines a number of local *Monitoring* processes used in the **within** clause to define the overall *Monitoring* process.

Example 4.7. Consider the simpler version of Rule4 in Listing 4. Its *Monitoring* process is as follows.

let

*Monitoring*1 = *SoundAlarm* $\rightarrow$ **Skip**

*Monitoring*2 = *GoHome* $\rightarrow$ **Skip**

within

0 $\blacktriangleleft$ (*personNearby*?*vpersonNearby* $\rightarrow$
if *vpersonNearby* == **true**
then *Monitoring*2 **else** *Monitoring*1)

The constraint in the then clause and the response in the unless clause are captured by local processes *Monitoring*1 and *Monitoring*2. In the **within** clause, after reading the relevant measures, the process chooses a local *Monitoring* process based on the unless condition. □

The local *Monitoring* processes are defined by $[\![\_]\!]_{\mathsf{LRDS}}$, which takes as argument a list containing the constraint of the response, and the responses in the defeaters dfts. We use $\mathsf{dfts}{\restriction_{\mathsf{RP}}}$ to represent the list of those responses. For an unless defeater without a response, we get **NoRep**. This is a special response defined in the semantics just for the purposes of simplifying the semantic rules. The additional argument is a counter for the *Monitoring* processes used to define their names. In the definition of $[\![\_]\!]_{\mathsf{RDS}}$, we define the (initial) value of the counter as 1.

The definition of $[\![\_]\!]_{\mathsf{LRDS}}$ has two equations for a singleton list of responses, and a third equation for a list $\langle \mathsf{resp}\rangle \frown \mathsf{resps}$ starting with a response resp followed by a list resps. For a singleton list with a proper response resp, the *Monitoring* process is defined by $[\![\_]\!]_{\mathsf{RDS}}$. For the special response **NoRep**, the response is just **Skip**.

Example 4.8. The *Monitoring* process for Rule4 in Listing 4 is shown below. It is similar to that in Example 4.7, but has an extra local *Monitoring*3 process since we have an extra unless clause. In the **within** clause, both relevant measures are read urgently, and then conditionals identify the right local process to monitor the behaviour.

let

*Monitoring*1 = *SoundAlarm* $\rightarrow$ **Skip**

*Monitoring*2 = *GoHome* $\rightarrow$ **Skip**

*Monitoring*3 = **Skip**

within

0 $\blacktriangleleft$ (*personNearby*?*vpersonNearby* $\rightarrow$
0 $\blacktriangleleft$ (*temperature*?*vpersonNearby* $\rightarrow$
if *vtemperature* > 35 **then** *Monitoring*3
else (**if** *vpersonNearby* == **true**
then *Monitoring*2 **else** *Monitoring*1)))

*Monitoring*3 corresponds to the unless clause for the condition temperature > 35, which does not have an associated response. So the rule imposes no restrictions, and the overall *Monitoring* process should just terminate.

The process in the **within** clause of a response process (as defined by $[\![\_]\!]_{\mathsf{RDS}}$) is specified by a $[\![\_]\!]_{\mathsf{CDS}}$ function. Its arguments are the alphabet $\alpha_{\mathsf{ME}}(\mathsf{dfts})$ of measures of the defeaters dfts, the defeaters dfts themselves, and the number of responses to be handled, namely, the number #dfts of defeaters, plus 1, to consider the constraint in the rule overall response. The definition of $\alpha_{\mathsf{ME}}(\mathsf{dfts})$ is similar to that of $\alpha_{\mathsf{ME}}(\mathsf{mBE})$, but applies to a list of defeaters, considering the Boolean expressions that they use.

The inductive definition of $[\![\_]\!]_{\mathsf{CDS}}$ is simple. For a list of measures $\langle \mathsf{mID}\rangle \frown \mathsf{mIDs}$, the process inputs the values vmID of the measure mID

urgently and then behaves as the process defined by $[\![\_]\!]_{CDS}$ for mIDs. In the defeaters used as argument for the recursive application of $[\![\_]\!]_{CDS}$, the references to mID are replaced with vmID. In Example 4.8, this defines the two urgent communications to input values of the measures personNearby and temperature.

For an empty list of measures, the process is defined by $[\![\_]\!]_{EDS}$. This is again an inductive definition, whose arguments are the list of defeaters, the local *Monitoring*1 process that applies when no defeater in the list does, and the number of defeaters in that list. We recall that *Monitoring*1 is the process that captures the behaviour of the overall constraint of the rule (see definitions of $[\![\_]\!]_{RDS}$ and $[\![\_]\!]_{LRDS}$). If the list of defeaters dfts dft has more than one defeater, the result is the application of $[\![\_]\!]_{EDS}$ to the last defeater, whose monitoring process is given by a recursive application of $[\![\_]\!]_{EDS}$ to define a process for the other defeaters, which applies when dft does not.

If we have just one defeater, then the process is a conditional that checks whether its condition applies. It does, the local *Monitoring* process identified by the counter n is used. Otherwise, the continuation process fp is used.

Example 4.9.

We show below the use of $[\![\_]\!]_{EDS}$ to define the conditional in Example 4.8.

$$
\begin{aligned}
&[\![\ \ \textsf{unless}\ \mathit{vpersonNearby}\ \textsf{then}\ \mathsf{GoHome}\\
&\ \ \textsf{unless}\ \mathit{vtemperature} > 35,\\
&\ \ \mathit{Monitoring}\,1, 3\ ]\!]_{EDS}\\
&=\\
&[\![\ \ \textsf{unless}\ \mathit{vtemperature} > 35,\\
&\ \ [\![\textsf{unless}\ \mathit{vpersonNearby}\ \textsf{then}\ \mathsf{GoHome}, \mathit{Monitoring}\,1, 2]\!]_{EDS},\\
&\ \ 3\ ]\!]_{EDS}\\
&=\\
&[\![\ \ \textsf{unless}\ \mathit{vtemperature} > 35,\\
&\ \ (\ \textbf{if}\ \mathit{vpersonNearby} == \mathit{true}\\
&\ \ \ \textbf{then}\ \mathit{Monitoring}\,2\ \textbf{else}\ \mathit{Monitoring}\,1\ ),\\
&\ \ 3\ ]\!]_{EDS}\\
&=\\
&\ \textbf{if}\ \mathit{vtemperature} > 35\\
&\ \textbf{then}\ \mathit{Monitoring}\,3\\
&\ \textbf{else}\ (\ \textbf{if}\ \mathit{vpersonNearby} == \mathit{true}\\
&\qquad \textbf{then}\ \mathit{Monitoring}\,2\ \textbf{else}\ \mathit{Monitoring}\,1\ )
\end{aligned}
$$

□

In the next section, we explain how we use the semantics.

5. Validation and verification

When writing SLEEC rules, it is possible to make a mistake and introduce redundant or conflicting rules, especially given the possibility that these rules are provided by stakeholders with different expertise (lawyers, ethicists, sociologists, etc.) and comprise complex defeaters. Redundant rules may help stakeholders to understand the consequences of the rules; for verification, however, these rules are unnecessary and so should be flagged. Conflicting rules, on the other hand, mean that there is no implementation that can satisfy them all. They need to be flagged and the conflicts need to be resolved. In Section 5.1, we present an approach that uses the semantics of our rules to detect conflicts, and in Section 5.2, we present redundancy checks. Finally, in Section 5.3, we discuss the verification of an agent model against a set of SLEEC rules.

Table 5
Conjunction of rules r1 and r2.

$$
\begin{aligned}
[\![\mathsf{r1}, \mathsf{r2}]\!]_{CP} = \ & \mathsf{idCC}(\mathsf{r1}, \mathsf{r2}) = \textbf{let}\\
& \quad \mathit{Env} = |\!|\!|\ \mathsf{e} : \alpha_M(\mathsf{r1}, \mathsf{r2}) \bullet \mathit{Env}\mathsf{e}\\
& \quad \mathit{Env}\mathsf{e} = \mathsf{e}?x \rightarrow \mathit{VEnv}\mathsf{e}(x)\\
& \quad \mathit{VEnv}\mathsf{e}(x) = \mathsf{e}!x \rightarrow \mathit{VEnv}\mathsf{e}(x)\\
& \quad \textbf{within}\\
& \qquad (\mathsf{id}(\mathsf{r1})[\![\alpha_E(\mathsf{r1}) \cap \alpha_E(\mathsf{r2})]\!]\mathsf{id}(\mathsf{r2}))\\
& \qquad\quad [\![\alpha_M(\mathsf{r1}, \mathsf{r2})]\!]\\
& \qquad \mathit{Env}
\end{aligned}
$$

5.1. SLEEC conflict detection

Two rules r1 and r2 are conflict free if there is no scenario in which both rules apply and the restriction of r1 makes it not possible to satisfy the restriction of r2, or vice-versa. Conjunction is specified in CSP using parallelism. So, roughly speaking, conflict freedom requires the process formed by the parallel combination of the processes for r1 and r2 never to reach a state in which the only event that can happen, if any, is *tock*. In this case, a system that satisfies both rules cannot make useful progress.

In practical terms, we only need to check for conflict between rules that have an overlap in their alphabet of events. If the rules have no such overlap, the restrictions they impose cannot interfere with each other. Moreover, overlap in the alphabet of measures is irrelevant, as rules do not need to agree on the reading of measures. The measures represent information about the system and the environment that is available at any time.

Table 5 presents the function $[\![\mathsf{r1}, \mathsf{r2}]\!]_{CP}$, which defines the conjunction process $\mathsf{idCC}(\mathsf{r1}, \mathsf{r2})$ for the rules r1 and r2. In the **within** clause of this definition, we compose the processes $\mathsf{id}(\mathsf{r1})$ and $\mathsf{id}(\mathsf{r2})$ (which define the semantics of r1 and r2) in parallel ($[\![...]\!]$), synchronising on their common alphabet of events, i.e., on the intersection alphabets $\alpha_E(\mathsf{r1}) \cap \alpha_E(\mathsf{r2})$ of their alphabets. Here, $\alpha_E(\mathsf{r})$ is the set of events of a rule r, or more precisely, the set of CSP events that represent the SLEEC events used in r.

An additional parallel process *Env* captures the environment in which the rules are considered, by recording the values of the measures for sharing between $\mathsf{id}(\mathsf{r1})$ and $\mathsf{id}(\mathsf{r2})$. The definition of *Env* is given in the **let** clause as the interleaving, that is, the parallel combination without synchronisation, of processes *Env*e for each event e in the alphabet of measures $\alpha_M(\mathsf{r1}, \mathsf{r2})$ of r1 and r2. These processes input the value of the measure e when a rule first requires that measure ($\mathsf{e}?x$). They then record the value x input as a parameter for another process *VEnv*e, which outputs x ($\mathsf{e}!x$) whenever a rule needs that measure. With *Env* we ensure that, for conflict checking, the rules are considered when the measures take the same value.

Using $[\![\mathsf{r1}, \mathsf{r2}]\!]_{CP}$, we define conflict freedom for the rules r1 and r2 below. For that, we use the process operator 'P after t', which defines the process that behaves like P after it has already engaged in its trace of events t.

Definition 5.1. The rules r1 and r2 are conflict free, if, and only if, for every trace t_1 of $[\![\mathsf{r1}, \mathsf{r2}]\!]_{CP}$, there is a trace t_2 of $[\![\mathsf{r1}, \mathsf{r2}]\!]_{CP}$ after t_1 that contains at least one event and at least one event different from *tock*.

With this definition, we require that, at no point, enforcing both rules, as defined by the process $[\![\mathsf{r1}, \mathsf{r2}]\!]_{CP}$, leads to a deadlock, so that no more events are possible, or to a situation in which there is no deadlock, but only the passage of time can be observed. The latter scenario is a timed deadlock: time can progress, but no event is possible.

Definition 5.1 is given in terms of the semantics, that is, the set of traces, of the conjunction process $[\![\mathsf{r1}, \mathsf{r2}]\!]_{CP}$. For automation, we can check conflict freedom using FDR using two assertions. The first is a standard FDR assertion for deadlock freedom, and the second is

$$
\begin{aligned}
RuleARule3 = \ & \textbf{let} \ \ Envtemperature = temperature?x \rightarrow VEnvtemperature(x) \\
& \quad VEnvtemperature(x) = temperature!x \rightarrow VEnvtemperature(x) \\
& \quad Env = Envtemperature \\
& \textbf{within}(RuleA \ [\![\{ GoHome \}]\!] \ Rule3) \ [\![\{\!| temperature |\!\}]\!] \ Env
\end{aligned}
$$

Fig. 5. Conjunction process for RuleA in Listing 5 and Rule3 in Listing 2.

```
rule_start
  RuleA when BatteryCritical and temperature
        < 25 then GoHome within 1 minutes
rule_end
```

Listing 5: Conflicting rule for a firefighter robot

an assertion based on our mechanisation of a timed-deadlock freedom check in the context of *tock*-CSP that is inspired by work in Roscoe (2013).

Example 5.1. Listing 5 presents another rule (RuleA) for the firefighter UAV. This rule requires that, if the battery reaches a critical level, and there is no risk of fire nearby, as indicated by the temperature measure, then the robot should return to base so that it can continue to work at a later point. We can imagine that, if there is risk of fire, the UAV should continue its mission even if it means that it will exhaust its battery in action. However, RuleA is in conflict with Rule3 from Listing 2. We show in Fig. 5 the conjunction CSP process for the two rules. In this case, both rules restrict the GoHome event and use just one measure, temperature. So, the *Env* process is just the *Envtemperature* process for this measure. The deadlock check (using the FDR model checker) gives a counterexample that indicates the reason for the deadlock. Namely, it provides a trace with the events *BatteryCritical* and *temperature*.20, and after 13 occurrences of *tock*, then the event *SoundAlarm*, followed by 47 occurrences of *tock*. We recall that we are identifying a time unit with 1 s. So, the counterexample, indicates that if RuleA is triggered, and after 13 s, Rule3 is triggered, then, after another 47 s, we have a deadlock, as RuleA requires *GoHome* to take place, but Rule3 forbids it. □

If the assertion for deadlock freedom holds, there is no guarantee that timed deadlock freedom holds.

Example 5.2. Listing 6 presents two other conflicting rules for the firefighter UAV. In the case of these rules, their conjunction does not lead to a deadlock. It is the case, however, that there is a situation in which the only possible behaviour allowed by these two rules is the passage of time. The check for timed deadlock freedom provides the following counterexample. First *BatteryCritical* happens, so that both rules are triggered. Afterwards, the measures *personNearby* and *temperature* are read and the values provided are *true* and 33. So, RuleC requires the robot to *GoHome* and RuleD requires it to *SoundAlarm* instead. We do not have a deadlock, as time can pass in the absence of a deadline. It so happens, however, that RuleC forbids *SoundAlarm* and RuleD forbids *GoHome* (because the two events are in the alphabets of both rules, and *SoundAlarm* is not mentioned in the relevant defeater of RuleC, while *GoHome* is not mentioned in the relevant defeater of RuleD). So, neither event can happen. □

If a pair of rules are not conflicting, but their alphabets overlap, then one of them may be redundant. We next consider how to check for redundancy.

5.2. Detection of redundant rules

For a pair of rules r1 and r2 that have overlapping alphabets and are not conflicting, we define redundancy below, using $t \upharpoonright E$ to denote the trace obtained from t by removing all events that are not in the set E.

```
rule_start
  RuleC when BatteryCritical
        then CameraStart
        unless personNearby then GoHome
        unless temperature > 35
        then SoundAlarm

  RuleD when BatteryCritical
        then CameraStart
        unless personNearby then SoundAlarm
        unless temperature > 35 then GoHome
rule_end
```

Listing 6: Conflicting rule for the firefighter UAV

Definition 5.2. For conflict-free rules r1 and r2, we say r2 is redundant with respect to r1 if, and only if, for every trace t_1 of id(r1), there is a trace of t_2 of id(r2), such that, (1) $t_1 \upharpoonright \alpha_E(r1) = t_2 \upharpoonright \alpha_E(r1)$, and (2) for every event e of $\alpha_E(r1)$ in a position i of these traces, for every measure m in $\alpha_M(r1)$, the value of m recorded in t_1 and t_2 at position i are the same.

The traces of the process that characterises a rule identify the behaviours allowed by the rule. So, the smaller that set of traces, the more restrictive is that rule. In this context, however, the reading of measures is irrelevant, since, as already said, rules use measures just to obtain information that indicates how the events are to be restricted. So, in Definition 5.2, we characterise a rule r2 as redundant, with respect to another rule r1, by considering the traces $t_1 \upharpoonright \alpha_E(r1)$, where t_1 is a trace of r1 and $\alpha_E(r1)$, we recall, is the set of events of r1 or, more precisely, the set of CSP events that represent the SLEEC events of r1. These traces characterise the restrictions of r1. Similarly, the traces $t_2 \upharpoonright \alpha_E(r1)$ characterise the restrictions of r2. If every behaviour allowed by r1 is also allowed by r2, then r2 imposes no additional restrictions, and so is redundant.

Example 5.3. In Listing 2, Rule1 is weaker, as it does not have a deadline, and can be eliminated. This can be automatically checked using the FDR model checker via a trace refinement. Since the refinement holds, there is no counterexample, but a clear indication of the weaker rule between the two. □

Normally, a rule r2 should not be redundant with respect to another rule r1 if r2 involves events not referenced in r1. This is, however, not necessarily the case, since the responses that refer to the extra events may be unreachable. So, in general, it is worth checking every pair of non-conflicting rules with overlapping alphabets of events. It is also possible to check for unreachable responses.

Our experience with the case studies presented in this paper and with a number of other examples of autonomous agents, as well as discussions with SLEEC experts suggest that the number of SLEEC rules for an autonomous agent will not run into the hundreds: it is more like tens. Moreover, a single rule is unlikely to have a very long or very deep list of defeaters. So, although our checks require a pairwise analysis of the rules, we expect that the checks for conflicts and redundancy within a SLEEC specification for an autonomous agent will remain tractable. Importantly, as we avoid dealing with the whole set of rules in a single check, model checking is also likely to remain feasible. The treatment

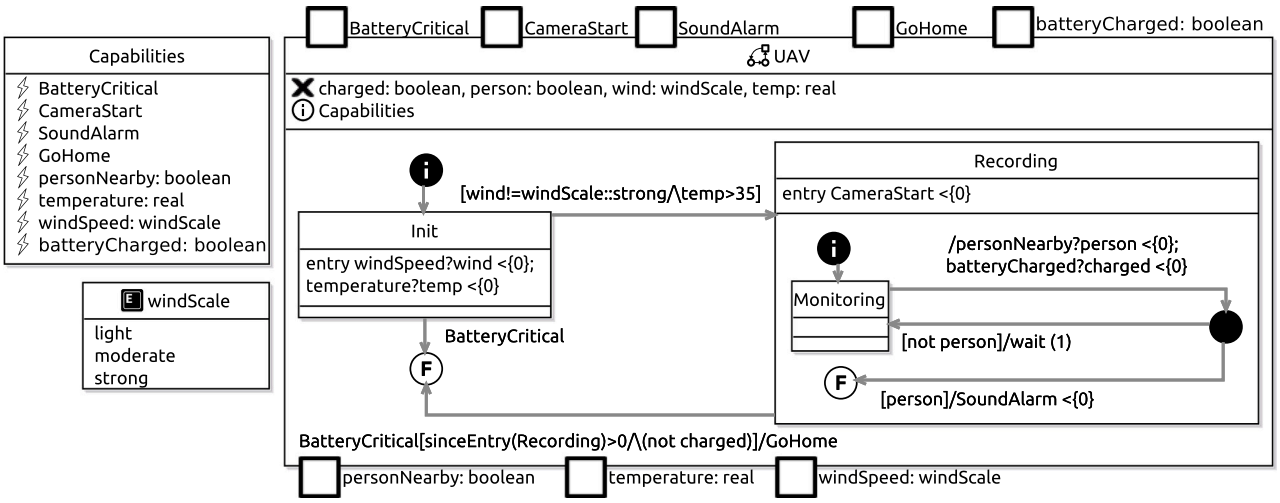


Fig. 6. Sketch of RoboChart model for a simple firefighter UAV.

of more complex data types provided by measures, however, is likely to impose a challenge.

5.3. Verification of compliance with SLEEC rules

This section describes our method for checking a system under verification (SUV) against a SLEEC rule r that refers only to events, that is, capabilities of the SUV, by means of refinement in tock-CSP. We use the term "relevant" to refer to these rules of interest for the verification of a given SUV. We note that a relevant rule may use measures not considered in the SUV, but the rule can only refer to events representing capabilities of the SUV. This is because, first of all, if a rule has a trigger not available in the SUV, then it will never apply. Moreover, if the rule has a response involving an event not available in the SUV, then it can never be satisfied, if triggered. We do not consider here the undesirable scenario where the response with the unavailable capability is unreachable.

To define a notion of conformance for an SUV with respect to a rule, we assume the existence of a tock-CSP model for the SUV. Such models can be generated automatically from other design models, or devised manually by developers with CSP expertise. Here, we consider examples where a RoboChart (Miyazawa et al., 2019) model for the agent is available and used as a basis to generate a tock-CSP model *SUV* automatically. So, we can use model checking to establish conformance as defined here.

Our notion of conformance $\mathsf{r} \vDash_{TT} SUV$ is defined below. In words, it corresponds to traces refinement in tock-CSP, where the specification is defined in terms of the process id(r) that captures the semantics of r (cf. Table 2). Traces refinement in tock-CSP ensures that the events of the SUV occur in the order and time specified, so that time budgets and deadlines are respected. Like in the check for redundancy, refinement disregards the measures; here, however, we require the values of the measures recorded in the specification and in the SUV to be the same.

Definition 5.3. An SUV conforms to a rule r, written $\mathsf{r} \vDash_{TT} SUV$, where *SUV* is the tock-CSP model of SUV, if, and only if, for every trace t_1 of the process $SUV; Stop$, there is a trace t_2 of id(r) such that: (1) $t_1 \restriction \alpha_E(\mathsf{r}) = t_2 \restriction \alpha_E(\mathsf{r})$ and; (2) for every event e of $\alpha_E(\mathsf{r})$ in a position i of these traces, for every measure m in $\alpha_M(\mathsf{r})$, the value of m recorded in t_1 and t_2 at position i are the same.

We consider the process $SUV; Stop$, rather than just *SUV*, because the processes that give semantics to a rule do not terminate. If SUV terminates, subsequent composition with *Stop* ensures that we do not erroneously flag a problem just because the rule does not allow termination. According to Definition 5.3, a conforming SUV may engage in additional events and read additional measures.

As mentioned earlier, the mechanisation of conformance checking is based on refinement, but the specification is a weakening of id(r), with respect to refinement, to allow occurrence of additional events and any order in the reading of measures. We illustrate the verification process using a simplified model of the control software of the firefighting UAV. In the next section, we consider additional examples.

Example 5.4. As said, for modelling we use RoboChart. In Fig. 6, we show a RoboChart interface Capabilities that declares those capabilities of the firefighter UAV identified in the SLEEC specification. Other interfaces in the model may declare additional capabilities not related to SLEEC concerns, but needed to implement the firefighter UAV mission. We also show a RoboChart state machine called UAV that specifies the control software of our simple firefighter in terms of these Capabilities and using local variables charged, person, wind, and temp. In the initial state Init of UAV (the target of the transition out of the initial junction indicated by a dark circle with an i), an entry action reads the windSpeed, recording it in the variable wind, and the temperature, recording it in temp. The notation '<{0}' specifies that these inputs need to be immediately available. There are two transitions out of Init. The first has the event BatteryCritical as a trigger. If it happens, the UAV cannot proceed, and terminates by transitioning to the final state, indicated by a clear circle with an F. The other transition has no trigger, but a guard requires the wind not to be strong (wind != windScale::strong, where windScale is the enumeration type of wind defined on the left in Fig. 6), and the temperature to be high (greater than 35), indicating a possible fire. That transition leads to a composite state Recording whose entry action starts the camera. Its own state machine is concerned with whether there is a personNearby and whether the battery is charged. Every time unit, this machine reads those measures and records them in the variables person and charged. Depending on whether there is a person or not, the machine raises the event SoundAlarm. A transition out of Recording ensures that, when BatteryCritical is signalled and the battery is not sufficiently charged, the UAV goes back to base by raising the event GoHome. The guard ensures that the amount of time since Recording has been

entered (sinceEntry(Recording)) is greater than 0, so that the check for the presence of a person is always carried out before the robot returns to base.

There are several simplifications in this example, but our focus is on the rules in Listing 2. We have identified that Rule1 is redundant, so we do not need to be concerned with it. Our technique identifies that the model satisfies Rule2, but not Rule3. The counterexample provided by the FDR model checker has the following events:

windSpeed.*light*, *temperature*.36,

CameraStart, *personNearby*.*true*,

batteryCharged.*false*, *SoundAlarm*, *tock*,

BatteryCritical, *GoHome*

This counterexample is a trace that leads to a forbidden event, here *GoHome*. The trace corresponds to a scenario in which, in the Init state, the measures windSpeed and temperature read are *light* and 36, respectively. With that, in the state Recording, the camera is started, then there is a personNearby and the battery is not sufficiently charged. So, after one time unit (i.e., one *tock*), the alarm is sounded, but the battery is indicated as critical. In this situation the UAV goes home, but Rule3 forbids that for 5 min. Indeed, the projection of this UAV trace to the events of Rule3 is *SoundAlarm*, *GoHome*, which is not a trace of the process for Rule3 (without the events that refer to measures). So, considering Definition 5.3, condition (1) is not satisfied. □

Before providing additional examples in the next section, we note that the relatively low complexity of SLEEC rules is expected to make the verification of SUV compliance with each individual rule feasible, under the assumption that the SUV tock-CSP model is itself of manageable size. As is often the case with model checking, this assumption may not always hold because of state explosion, in particular as the FDR model checker is not optimised for dealing with timed (i.e., tock-CSP) models despite supporting them. On the positive side, RoboChart is part of a framework that includes support for alternative verification approaches, based on theorem proving, simulation, and testing (Cavalcanti et al., 2021). In particular, theorem proving is promising, and amenable to automation if we use automatically generated semantics, like that of SLEEC.

6. Tool support

We have implemented a tool that allows SLEEC experts with limited IT expertise to specify and edit normative requirements in the domain-specific language from Fig. 2. The tool includes a friendly user interface with syntax highlighting (Fig. 7), and a parser for the SLEEC language. The translation of SLEEC documents to tock-CSP is based on the definitions presented in Tables 2–4 and is automated via approximately 700 lines of code written using Xtend (Anon, 2010), a lightweight version of Java, defining model-to-text transformation rules. Each semantic function is implemented as a recursive function in Xtend, so that there is a direct correspondence between our equations and their Xtend encoding. The parameters of the Xtend functions are exactly those of the functions, and the simple metanotation we use is directly available and adopted. The pattern matching, used to identify the different forms of SLEEC terms considered in the function definitions via different equations, is captured by conditionals. Local variables are used to record the various elements of the patterns used in the equations. The target notation is encoded directly, and distinguished in Xtend using quotes.

As an example, the implementation of the mechanisation of the function $[\![\_]\!]_C$ from Table 4 is given in Listing 7. The Xtend method is called `C`, and its result is a text (`CharSequence`) defining the CSP process specified in our equations. The arguments of `C` are exactly those of $[\![\_]\!]_C$. The first argument `const` is an arbitrary `Constraint`; the local variables `eID`, `v`, `tU`, and `resp` record the (optional) elements of `const` used in the definition of $[\![\_]\!]_C$ from Table 4. In the body of `C`, the (nested) conditional determines the equation from Table 4 that applies to define the semantics of `const`. If there is no value (that is, `v == null`), we have a simple constraint `eID`, whose semantics is given by the first equation from Table 4. The resulting text is exactly as defined in that equation; the guillemots indicate elements of the metanotation, here `eID`. Similarly, if there is no response associated with an otherwise clause (`resp == null`) and the constraint negates an event (`const.not`), the semantics is given by the third equation. If an event is not negated, the second equation applies. In the FDR syntax for a deadline, we use a process constructor `StartBy`, which takes the process and the deadline as arguments. For the last equation, we use the FDR constructor `TimedInterruptSeq` for the timeout operator; it takes three arguments: the event, the timeout value, and the timeout process.

With a faithful encoding of the semantic definitions as model-to-text transformations, our tool validates the semantics as well playing a key role in validating the SLEEC language. We have carried out enough tests to provide full coverage of the syntactic structure of SLEEC specifications, and used FDR to inspect the automatically generated tock-CSP semantics. Our tests are evidence that our rule definitions are well typed, that there are enough rules, and that they produce valid tock-CSP models. Validation that, in addition, the semantics is appropriate is provided by our examples as discussed in the next section.

The implementation of norm(mBE) allows seconds, minutes, hours and days in the concrete syntax and it normalises each value to seconds in the current implementation. The resulting tool is described in Getir Yaman et al. (2023). All code and the models are publicly available (Anon, 2023). In addition, a recently extended tool version that also integrates RoboTool and FDR, and invokes FDR directly for checking the conformance of a RoboChart model against SLEEC rules can be found in Getir Yaman et al. (2024).

For the semantics, we translate from the trace-based definitions of conflict, redundancy, and conformance, to refinement checks via a mechanisation of tock-CSP (Baxter et al., 2022) for verification using the CSP model-checker FDR (Gibson-Robinson et al., 2014). This enables the automatic analysis of SLEEC rules and verification of conformance against system models with tock-CSP semantics, such as in the case of RoboChart models.

Fig. 7 shows a screenshot of our tool, where we can see the encoding of the SLEEC definitions and rules from Listings 1 and 2 in the left pane, and the automatically generated tock-CSP script for the SLEEC specification in the right pane. We note that, because model checking operates with finite models, measures of the type *Int* need to be specified using finite intervals such as {0..35}.

7. Evaluation

We evaluated our notation, processes, and tool by conducting two case studies aimed at answering the following research questions (RQs).

RQ1 (Expressivity) — Does the SLEEC language support the specification of rules covering a wide range of SLEEC concerns? With this RQ, we seek to establish whether the language allows SLEEC experts (ethicists, lawyers, sociologists, psychologists, etc.) to define the normative rules for an autonomous agent.

RQ2 (Correctness) — Are the framework's methods and tools identifying rule conflicts, redundancies, and violations by an autonomous agent specification correctly? As a key objective of our work is to enable consistency validation and verification of a SLEEC rule set, we examined the results produced by these tool-supported checks for SLEEC rules defined by multiple stakeholders.

RQ3 (Scalability) — How does the time required to perform rule validation and verification grow as the number of rules increases? As

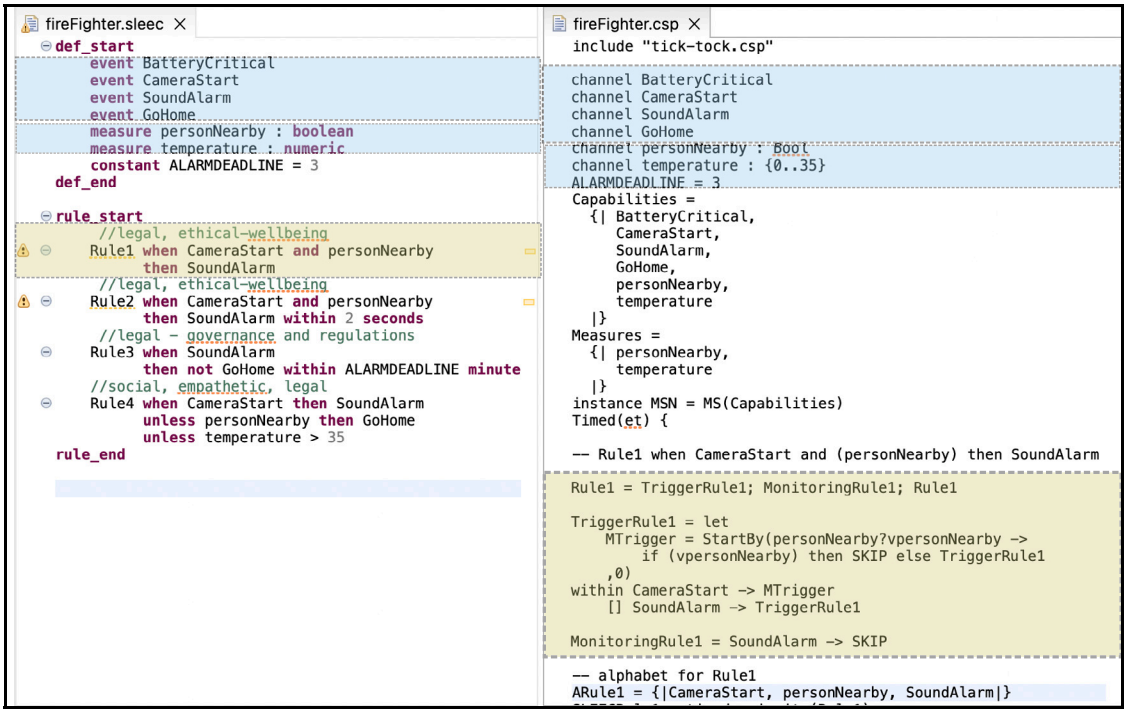


Fig. 7. Tool for editing and supporting reasoning about SLEEC rules.

```
private def CharSequence C(Constraint) {
    val eID = const.event.name
    val v = const.value
    val tU = const.unit
    val resp = const.response

    //[[eID]] C
    if (v === null)
      '''≪eID≫ -> SKIP'''
    // [[eID within v tU, trig, ARDS, mp]] C
    else if (resp === null) {
    // [[not eID within v tU]] C
      if (const.not)
      '''WAIT(≪norm(v, tU)≫)'''
    // [[eID within v tU]] C
      else
      '''StartBy(≪eID≫ -> SKIP,≪norm(v, tU)≫)'''
        }
    // [[eID within v tU otherwise resp]]C
    else {
      '''TimedInterruptSeq(≪eID≫,≪norm(v,tU)≫,
           ≪RDS(resp)≫)'''

    }
  }
```

Listing 7: Code snippet in Xtend illustrating the implementation of the recursive function $[\![\_]\!]_C$ defined in Table 4.

model checking, which underpins our framework, is known to have scalability issues, we assessed its execution time for SLEEC rulesets of growing sizes.

RQ4 (Usability) – How easy is it for the intended users of the framework to specify SLEEC rules, to use the tools and process we propose, and to understand the output of the tools? As shown in Fig. 1, the purpose of our framework is to support both the specification and validation of normative rules by SLEEC experts with limited technical expertise, and the verification of an autonomous agent by its developers. To assess whether the framework can be used effectively, we obtained feedback from these users.

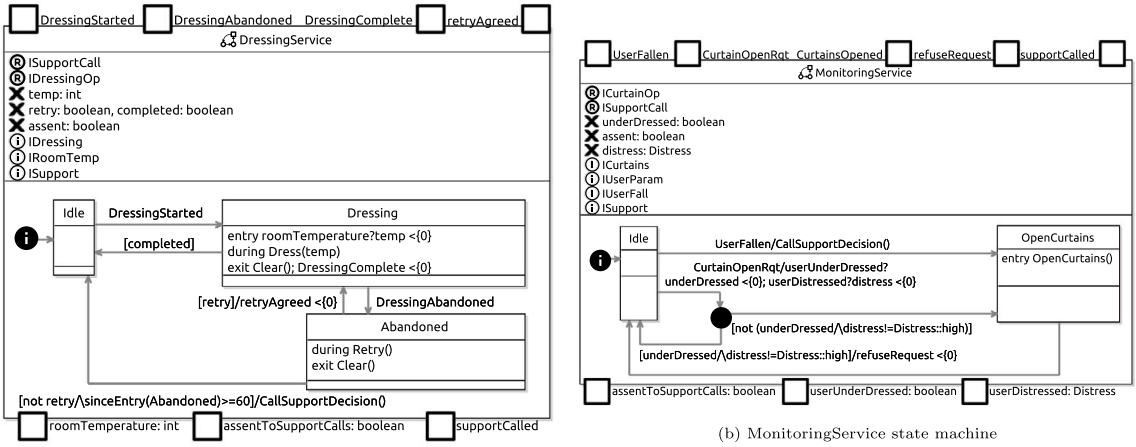
The first case study was based on the firefighter UAV from our running example. The autonomous agent from this case study is described in Section 2 and Example 5.4. In the second case study, we used an autonomous agent from the robotic assistive-care domain. This autonomous agent is described in Section 7.1. Our evaluation methodology, the results for each RQ, and their discussion are presented in Section 7.2.

7.1. Robotic assistive-care application

We considered in our second case study a robotic assistive-dressing (RAD) system tasked with helping a physically impaired user to put on a garment, such as a gown or coat. The need for daily assistance with dressing is a reality for over 80% of those in skilled nursing homes (Mitzner et al., 2014). As this need often exceeds the capacity of the nursing home caregivers, the development of robotic-assisted dressing solutions has received significant attention from the research community in recent years, e.g., Camilleri et al. (2022), Jevtić et al. (2018), Zhang and Demiris (2022), Pirhonen et al. (2020).

Our particular RAD system is for use by a person living independently at his or her home; it is adapted from the solution presented in Camilleri et al. (2022), and has the additional function of monitoring the health of its user, who is liable to fall. When falls are detected, RAD is expected to contact support services. Health and fall monitoring is achieved through a smart watch worn by the user, and by visual sensors mounted on the RAD platform and in the user's home. To communicate with the user, RAD is equipped with voice recognition and speech modules. RAD can communicate with a support operator located off site by transmitting audio and video feeds. Finally, RAD can control temperature, lighting, and the opening or closing of the room curtains through home automation functionality.

RAD's control software is specified as a RoboChart model that includes multiple state machines defining parallel behaviour. The two



(a) DressingService state machine

(b) MonitoringService state machine

Fig. 8. RoboChart state machines of the RAD application.

machines relevant to the discussion here are DressingService and MonitoringService, depicted in Fig. 8. DressingService specifies the operation of the control software for user dressing, and interacts with the platform via events such as DressingStarted and DressingAbandoned. MonitoringService mediates the opening of the curtains, and support calls. In what follows, we describe each state machine in further detail.

DressingService is initially in an Idle state. A transition to a Dressing state is triggered by the event DressingStarted, corresponding to a request from the user. In that state's entry action, the current roomTemperature is input into a local variable temp, and then an operation Dress is called with the current temperature passed as a parameter. Here, Dress is a software operation that captures the time the actual dressing can take, after which it sets the value of a Boolean variable completed to true. Because Dress is called in a during action, this behaviour can be interrupted by any of Dressing's outgoing transitions: either because completed is true, or as a result of DressingAbandoned being triggered. In both cases, before Dressing is exited, its exit action is executed: it calls an operation Clear that sets the variables completed and retry to false followed by an output on DressingComplete to indicate that dressing has completed, irrespective of whether it has succeeded. If the transition out of Dressing is to the state Abandoned, there is a call to an operation Retry. This operation captures a protocol for agreeing to retry dressing. If there is agreement from the user, then retry is set to true and the transition back to Dressing becomes enabled and is taken. Otherwise, if no agreement has been reached and more than two minutes have elapsed since entering Abandoned, the guard over the transition to Idle becomes true. When that transition is taken there is a call to a software operation CallSupportDecision that calls support depending on whether there is user assent for that.

The machine MonitoringService also starts in an Idle state, from where two outgoing transitions can be triggered by events UserFallen and CurtainOpenRqt. If a user has fallen, then the operation CallSupportDecision is called, followed by the opening of curtains in the entry action of the state OpenCurtains, and then there is a transition back to Idle. If there is a request to open the curtains via CurtainOpenRqt, then there are two readings of measures userUnderDressed and userDistressed to determine if the user is under dressed and their level of distress. If the user is neither underdressed nor highly distressed, then the curtains are opened, and otherwise the request is refused as indicated by the output event refuseRequest.

7.2. Evaluation methodology, results and discussion

RQ1 (Expressivity). To answer the first question, we recruited a group of five SLEEC experts from different disciplines and backgrounds: (i) Psychology and Ethics; (ii) Law and Ethics; (iii) Ethics; (iv) Moral Psychology and Law; and (v) Engineering and Goal Modelling.

These experts were asked to define SLEEC rules for variants of the firefighter UAV and RAD system with the capabilities and functionalities described earlier in the paper *and any additional capabilities and functionality* they envisaged for future firefighter UAV and RAD extensions. We allowed and even encouraged our study participants to imagine such extensions to significantly expand the range of SLEEC rules they defined, and thus to assess better the ability of our SLEEC language to encode a wide variety of such rules.

During the experiment, the experts initially independently established the rules, but we encouraged them to engage in negotiation and collaboration through additional sessions. The resulting SLEEC specifications for the case studies can be found in our GitHub repository (Anon, 2023).

Case study 1 (firefighter UAV). The SLEEC experts provided a set of 13 SLEEC rules for the firefighter UAV. We combined these rules with our four sample SLEEC rules from Listing 2, and carried out a manual analysis of the resulting set of 17 rules. Through this analysis, we found that eight of the rules are relevant to the firefighter UAV, and the remaining nine refer to events and/or measures that do not match the functionality from the UAV model in Fig. 6. Table 6 depicts a selection of both types of rules, with the non-relevant rules shown in shaded rows.

Case study 2 (RAD). By manual analysis, we identified that, among a total of 14 rules defined by the SLEEC experts, six rules are relevant to the RAD system (i.e., the capabilities mentioned in the rules are a subset of the system capabilities), while eight rules involve capabilities imagined by the experts. We have also created seven synthetic rules to cover different additional elements of our language. Several rules from the combined set of 21 rules are presented in Table 7, where Rules 5 and 6 (shaded in the table) are examples of non-relevant rules and the rest are examples of rules relevant to the RAD capabilities.

Discussion. Our language was observed to be intuitive for expressing normative rules in a comprehensive manner. However, certain language constraints have prompted the experts to modify their rules. The main constraint discussed was that the specification of negation

Table 6
Partial set of SLEEC rules for the firefighter UAV.

Rule id	SLEEC specification	Rule type	Implication	Relevance
Rule3	when SoundAlarm then not GoHome within 5 minutes	ethical	ensuring safety	✓
...				
Rule13	when BatteryCritical then GoHome unless personNearby and temperature>MAX_TEMP then SoundAlarm	legal ethical	preventing harm	✓
Rule7	when FireConfirmed and temperature > 60 then SprayWater unless personNearby and sprayPressure > 100 then InformBystanderAndSprayWater within 45 seconds	empathetic cultural	promoting well-being	×
...				
Rule14	when FullyCharged then TestEfficiencyPreserved within 24 hours	legal	ensuring operational efficiency	×

Table 7
Partial set of SLEEC rules for the RAD system.

Rule id	SLEEC specification	Rule type	Implication	Relevance
Rule1	when DressingStarted and userUnderDressed then DressingComplete within 2 minutes unless roomTemperature < 19 then DressingComplete within 90 seconds unless roomTemperature < 17 then DressingComplete within 60 seconds	empathetic ethical	promotes and supports user well-being	✓
Rule2	when CurtainOpenRqt then CurtainsOpened within 60 seconds unless userUnderDressed then RefuseRequest within 30 seconds unless userDistressed > medium then CurtainsOpened within 60 seconds	cultural empathetic	respect for privacy and cultural sensitivity	✓
Rule3	when UserFallen then SupportCalled within 1 minutes unless not assentToSupportCalls unless emergency	legal ethical social	respect for autonomy and preventing harm	✓
Rule4	when DressingAbandoned then {RetryAgreed within 2 minutes otherwise {SupportCalled unless not assentToSupportCalls}}	legal ethical	promoting user benef icence and respecting autonomy	✓
Rule5	when UserRequestInfo then ProvideInfo unless not informationAvailable then InformUserandReferToHumanCarer unless informationDisclosureNotPermitted then InformUserandReferToHumanCarer	legal ethical	promoting privacy and transparency	×
Rule6	when DressingStarted and dressPreferenceTypeA and genderTypeB then DressinginClotingX unless userAdvices unless medicalEmergency unless clothingItemNotFound then InformUser	cultural ethical	respecting cultural norms and values	×

(not) without a timeout is not allowed (as that would block a capability permanently). The second constraint of note was the inability to define conjunction of events in the response. Lastly, experts were asked to indicate time units explicitly rather than relying on temporal (vague) phrases (e.g., "in a few minutes"). Overall, they found the tool user-friendly and were able to express their requirements using our DSL.

RQ2 (Correctness). We evaluated the correctness of our SLEEC framework in two stages:

1. In a first stage, we used our consistency validation tool to identify SLEEC rule conflicts and redundancies across the entire rulesets mentioned earlier. All identified conflicts and redundancies were then checked with the experts, and the SLEEC rulesets were manually examined both by our project team and the experts for any additional conflicts or redundancies that the validation tool might have missed.
2. In a second stage, we retained the expert-defined SLEEC rules that were relevant to the firefighter UAV and RAD presented in Fig. 6 and Fig. 8, respectively, and we used our SLEEC verification tool to check whether that corresponding autonomous agent satisfies these rules. The results of this verification were then manually examined by our project team.

In the first stage, we presented to the experts every pair of rules that was identified to present a conflict or redundancy as a result of our analysis in a session of approximately two hours. After examining each result, we first inquired whether they comprehended the conflict or redundancy. If so, the expert was asked to propose a resolution action. This could be removing or modifying a rule, or even combining these rules. The experts also had the option to choose no further action.

```
Rule3-modified when SoundAlarm
  and batteryCharged then not GoHome
  within 5 minutes
```

Listing 8: One of firefighter SLEEC rules verified in RQ2

Table 8
RQ2 evaluation results for firefighter UAV.

	#Rules	Resolution action
Conflicts	17 / 8	Removing 2 rules, sanitising the event and measure names
Redundancies	17 / 22	Removing 5 rules, modifying 2 rules, streamlining defeaters
Non-conformance	2 / 1	Revising system design

Case study 1 (firefighter UAV). In the first stage, our SLEEC tool from Section 6 reported eight conflicting rule pairs and 22 pairs of redundant rules (Table 8). The expert examination of these consistency issues confirmed that all were real issues. The large number of redundancies was due to the limited functionality available for the firefighter UAV from our running example, which meant that most study participants defined similar SLEEC rules for the autonomous agent. Most of these redundancies were resolved easily by removing five of the rules. The remaining cases were then resolved by combining a couple of rules, and by reorganising rule defeaters. For the conflict resolution, two additional SLEEC rules were removed, one rule was split into two rules, and the names of events and measures were further aligned. This process produced a set of nine valid SLEEC rules at the end of the first stage.

In the second stage, we considered for verification two rules out of the nine valid rules obtained in the previous stage. These were the only rules relevant for the model presented in Fig. 6, and included:

- a modified version of Rule3 from Table 6 (see Listing 8), as obtained in the first stage of our analysis;
- Rule13 from Table 6, which was not modified as a result of the analysis.

Of these two rules, the system from Fig. 6 satisfied Rule3-modified, and failed to satisfy Rule13. We discuss these conformance checking results next.

Rule3-modified is concerned with allowing the alarm to be heard for a sufficient amount of time when the battery is sufficiently charged, so the drone should not go home within 5 min after the alarm sounds. Rule6, on the other hand, requires the drone to GoHome when the battery is critical, but if nearbyPerson is detected and the temperature is higher than MAX_TEMP, then it should sound the alarm instead of going home. In the RoboChart model, if the transition out of the state Recording is triggered, then the drone will GoHome even if a personNearby is detected at that point, as there is no further checking of whether a person is nearby, whereas the rule requires that the alarm is triggered first. Resolving this error requires changing the design from Fig. 6.

Case study 2 (RAD). We identified three conflicts and six redundancies. All of these results were acknowledged by the experts as genuine conflicts and redundancies. Conflicts, in particular, stemmed from different experts defining the same rule, but using different names for the capabilities. Consequently, the experts decided to delete one of the rules for each conflict. Regarding redundancies, several interferences

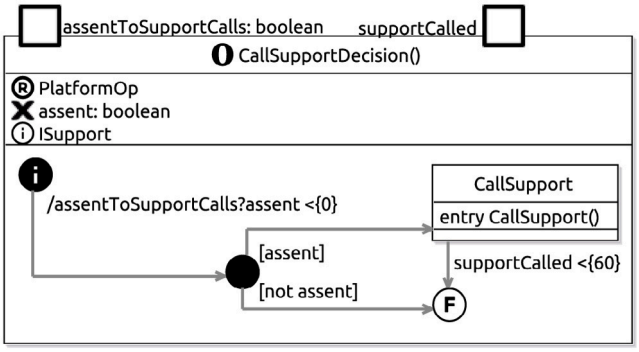


Fig. 9. RoboChart model for CallSupportDecision.

existed among different rules. This meant that the removal of a single rule resolved multiple redundancies. The summary of these results is presented in Table 9. As a result of these efforts, the overall ruleset reduced from its initial count of 21 rules to a consistent set of 16 rules.

In the second stage, we considered for verification five out of these 16 rules, namely, we considered just the rules that are relevant for our model presented in Section 7.1. Of the five rules, the system failed to satisfy one of them. We present in Table 7 four of these rules and, in the following, discuss the conformance checking results.

Rule1 is concerned with the time taken by a dressing episode: at most 2 min to complete, unless the room temperature is low, in which case it should be faster. Rule2 regulates the opening of the curtains, which should consider the privacy of the user, but also be sensitive to the distress that can be caused if a user request for the curtains to be opened is denied. If a user fall is detected then support must be called, but Rule3 requires that assent from the user is available for this if it is not an emergency case. Finally, if the dressing is abandoned, Rule4 requires that there should be an attempt to retry, and, eventually support must be called. Again, however, the user's assent to make the call should be gained beforehand.

Using our technique, we get confirmation that the first three rules are satisfied, but Rule4 is not. The issue is related to the design of the operation CallSupportDecision(), shown in Fig. 9. Since CallSupport() is a platform operation, it is asynchronous and does not block, so SupportCalled is used to flag termination of CallSupport(). In this version, the design engineer has followed an extra requirement to call support in no more than 1 min, if the user falls and there is consent. So, in CallSupportDecision() there is a deadline 60 on SupportCalled. With this deadline, we require that any actions involved in implementing CallSupport(), such as establishing a phone connection and dialling, are completed within 1 min.

The extra requirement to call support in 1 min is incompatible with the requirement to satisfy both Rule3 and Rule4, because Rule4 requires a delay of 2 min before calling support in case dressing is abandoned. As already mentioned, there is no conflict between Rule3 and Rule4, so, in principle, they could be both satisfied by a design. In the context of the extra requirement, however, this is no longer possible. The counterexample shown below, which is generated by our SLEEC tool, reveals the issue:

UserFallen, *DressingStarted*,
assentToSupportCalls.true, *CallSupport*,
roomTemperature.−2, *DressingAbandoned*

This trace indicates that DressingService has gone through DressingStarted, got the measure −2 for the roomTemperature, but finally DressingAbandoned occurs. In MonitoringService, UserFallen has led to a call to CallSupportDecision(), where assentToSupportCalls was found to be true, so a call to support is triggered, and then a

Table 9
RQ2 evaluation results for RAD system.

	#Rules	Resolution action
Conflicts	21 / 3	Removing 3 rules, sanitising event names
Redundancies	21 / 6	Removing 3 rules revising 1 defeater
Non-conformance	5 / 1	Relaxing the deadline

deadline requires SupportCalled to take place in 60 s. At this point, however, SupportCalled is forbidden by Rule4 for two minutes so that a RetryAgreed has a chance to occur.

In this situation where the system design violates a SLEEC rule, we have to consult the SLEEC experts and other requirements stakeholders. A few outcomes may be possible. In our experiment, a domain expert agreed that a 1 min deadline is too strict, and, in this case, the design needs to be changed to relax the deadline.

RQ3 (Scalability). We assessed the scalability of our conflict and redundancy checking by examining the time required to complete these operations for SLEEC rulesets of increasing size. As the complete SLEEC ruleset for the RAD system is larger than the one for the firefighter UAV, we only ran the scalability experiments for the RAD system. To that end, we used the 21 RAD SLEEC rules from RQ1 to assemble four rulesets of increasing size. These rulesets comprised five, 10, 15, and all 21 SLEEC rules, respectively. The rules included in the first three sets were specifically chosen to maximise the number of rule pairs referring to non-disjoint sets of autonomous agent capabilities, and therefore requiring conflict and redundancy checking — we recall that two SLEEC rules that refer to disjoint sets of agent capabilities cannot be conflicting or redundant, so they are trivially consistent with each other, and thus our consistency validation skips their analysis.

As the scalability of tock-CSP analysis is known to be affected by the magnitude of the durations (i.e., by the number of *tock* events in which the CSP processes need to engage), we used each of the four rulesets mentioned above to synthetically create two additional sets of rules by increasing all the deadlines from every rule by a factor of two, four, and eight, respectively. As an example, when starting from a SLEEC ruleset that included Rule1 from Table 7, the deadline 'within 2 minutes' from this rule was replaced by 'within 4 minutes' in the first additional ruleset, by 'within 8 minutes' in the second, and by 'within 16 minutes' in the third. In this way, we obtained a total of 16 SLEEC rulesets with different combinations of rule numbers and timeout magnitudes.

Each of the 16 SLEEC rulesets were analysed separately for conflicts and redundancies. All the experiments were run on a Ubuntu server with a Dual AMD EPYC 7501 processor and 2.0 TiB of RAM (with a six-hour timeout), and the execution times for these analyses are reported in Fig. 10. Before discussing these results, we note that, for our four rulesets of increasing size, the number of rule pairs that need to be analysed (because their rules refer to non-disjoint sets of agent capabilities) grows only slightly faster than linearly, from five rule pairs for the five-rule set to 12 rule pairs for the 10-rule set, 32 rule pairs for the 15-rule set, and 49 rule pairs for the 21-rule set. This is a far slower increase in the number of rule pairs than the (quadratic) worst-case scenario, in which each of the $n(n-1)/2$ rule pairs that can be formed for an n-rule set contains interdependent rules that need to be analysed. We expect this sub-quadratic growth in the number of interdependent rules to be representative of most SLEEC rulesets, as it is unlikely that some of the agent capabilities appearing in a rule will also appear in most other rules, especially for large numbers of capabilities and large rulesets.

For all four deadline magnitudes, the experimental results show an approximately linear increase in the analysis time for conflict checking, and an exponential increase for redundancy checking as the number of rule pairs being analysed grows. The exponential growth stems from rule pairs that make use of more measures and deadlines, mainly reflected in the times for rule sets with 49 rule pairs. For those pairs, the experiment with eight-fold longer deadlines did not terminate within the six-hour timeout, and so its data point is omitted from Fig. 10(b). Additionally, for all 16 rulesets, the conflict checking completes faster than redundancy checking (by between ~ 2 and ~ 19.7 times). This is due to the fact that the FDR assertions required to check conflicts are simpler than those used to check redundancies. For conflicts we use a deadlock and a divergence-freedom check, and for redundancies, we use a refinement check. FDR has very efficient algorithms for checking deadlock and divergence-freedom (Roscoe, 1998), and they involve just one CSP process, while a refinement check is concerned with two processes.

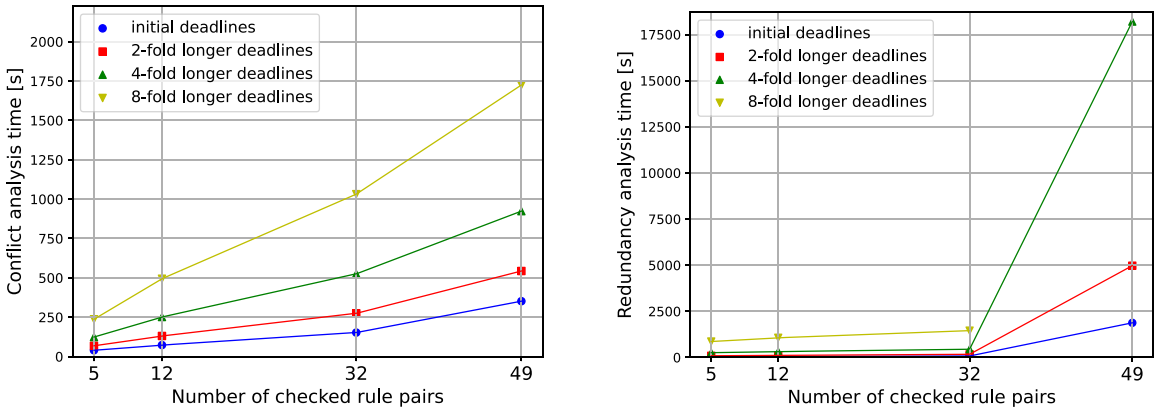
Looking now at the effect of increasing deadline magnitudes, we note that, for every ruleset size, the analysis times (for both types of consistency checking) increase quickly as the deadline magnitudes grow from the durations specified by our SLEEC experts. This reflects a known scalability result for tock-CSP. The number of states of each tock-CSP process grows linearly with these durations. In our mechanisation, however, in each time unit, the values of the measures can change. So, each additional *tock* event creates a number of states related to the number of possible combinations of values of the measures. Hence, the exponential growth observed in Fig. 10.

This scalability issue can be alleviated by using a single *tock* CSP event for a number of time units corresponding to the greatest common factor (GCF) of all timeouts from the pair of SLEEC rules under analysis (an optimisation not implemented in the current version of our tools). This cannot, however, eliminate the issue, as the ratio between the largest deadline and this GCF can still be very large. Another way of addressing scalability is by restricting the possible combinations of values of the measures, if there are requirements that invalidate some combinations. This, however, requires encoding environment assumptions.

We complete the discussion of the experimental results from Fig. 10 by noting that the combined time required to complete the conflict and redundancy checking does not exceed 2300 s for the SLEEC rulesets assembled using rules with unmodified deadlines, 5517 s for the SLEEC rulesets assembled using rules with deadlines increased two-fold, 19123 s for the rulesets assembled with four-fold, and 2485 s (excluding the ruleset with 49 pairs) for the SLEEC rulesets assembled using rules with deadlines increased 8-fold. While these analysis times are significant, they are all acceptable for development-time consistency validation of a set of requirements. Furthermore, the analyses of different pairs of SLEEC rules can be carried out independently and therefore in parallel, e.g., using cloud-computing resources.

Finally, the verification time required to check a system under verification against a SLEEC rule as described in Section 5.3 depends primarily on the complexity of that system's tock-CSP model. Evaluating the scalability of analysing tock-CSP system models is beyond the scope of this article (and is covered elsewhere, e.g. Roscoe, 1998; Mestel and Roscoe, 2020). As such, we only measured the total FDR execution time for the conformance checking of the: (i) firefighter UAV tock-CSP model obtained from the RoboChart state machines in Figs. 6 against the two relevant SLEEC rules from RQ2; and (ii) RAD tock-CSP model obtained from the RoboChart state machines in Fig. 8 against the four relevant SLEEC rules from Table 7. This verification was carried out on the same Ubuntu server as the SLEEC rule conflict and redundancy analyses described earlier, and took 14 s and 863 s, respectively. Again, we note that these times are acceptable for an offline verification activity — and that the verification of different SLEEC rules can be performed independently (and thus in parallel).

RQ4 (Usability). As previously noted, the stakeholders worked both individually and in cooperation to formulate the SLEEC requirements.



(a) Execution time for conflict analysis

(b) Execution time for redundancy analysis

Fig. 10. Execution time for analysing conflicts and redundancies for SLEEC rulesets of different sizes and timeout magnitudes; the horizontal axis shows the number of rule pairs being analysed (because they refer to non-disjoint sets of autonomous capabilities) for four rulesets comprising five, 10, 15 and 21 SLEEC rules, respectively.

Their feedback suggests that our language possesses an intuitive structure, and found value in having a DSL editor that incorporates features like highlighting and type checking. The analysis of conflicts and redundancies was particularly useful in identifying and understanding problems with the rules. Notably, they highlighted the usefulness of pairwise analysis, as dealing with issues within an extensive set of rules collectively can be overwhelming. Some suggestions for enhancements were put forth, including the incorporation of diagnostic capabilities and providing more comprehensive feedback e.g. an explaination of the causes of conflicts. Additionally, users indicated that a guide detailing the integration of the tool with the whole requirements elicitation process would be beneficial. Overall, the users appreciated the help of the tool for encoding, understanding, and validating SLEEC requirements.

7.3. Threats to validity

Construct validity threats may arise due to assumptions made about the systems from our running example and case studies, or about their SLEEC requirements. To limit these threats, the two autonomous systems used in the paper are adapted from the research literature on firefighting UAVs (Alon et al., 2021; Cervantes et al., 2018; Innocente and Grasso, 2019) and robotic assistive dressing (Camilleri et al., 2022; Jevtić et al., 2018; Zhang and Demiris, 2022; Pirhonen et al., 2020), and the SLEEC rules for these systems were provided by stakeholders with expertise in social, legal, ethical, empathetic and cultural norms of such systems (i.e., lawyers, ethicists, psychologists, etc.).

Internal validity threats may stem from bias in establishing cause–effect relationships in the experiments from our case study. To mitigate these threats, we used SLEEC experts from multiple disciplines in the evaluation of our framework's expressivity and usability, and we carried out the correctness and scalability evaluation using a diverse set of rules provided by these experts — including rules that cover additional autonomous agent capabilities proposed by these experts. Furthermore, we have enabled replication by making all our models, SLEEC rules and code available in the project's GitHub repository (Anon, 2023).

External validity threats exist if the SLEEC requirements of other autonomous agents cannot be expressed in our SLEEC domain-specific language, if the behaviour of such agents cannot be modelled using the process algebra employed by our framework, or if their tock-CSP models are too large and complex to be verified efficiently. We limited the first threat by developing the language with constant input from SLEEC experts from other disciplines (in particular lawyers, ethicists, philosophers, and social psychologists), and assessed its expressiveness as part of case studies in which we encouraged the participants to imagine additional capabilities for a robotic assistive-dressing system., To mitigate the second threat, we chose tock-CSP as the underpinning modelling paradigm for our framework because tock-CSP models for a wide range of autonomous agents can be obtained automatically from RoboChart models developed in an easy-to-use, validated domain-specific language for robotics (Miyazawa et al., 2019). Nevertheless, additional case studies are needed to establish the applicability of our SLEEC framework in domains with characteristics that differ from those of our running example and case study. The last threat is a known problem of formal verification/model checking, which can often be alleviated by using suitable levels of abstraction (to focus on the relevant aspects of the verified system). Furthermore, the last several decades of advances in model checking algorithms and tools have greatly improved the scalability of these techniques, leading to their wide adoption, including for autonomous agents, e.g., Miyazawa et al. (2019), Li et al. (2024).

8. Related work

The normative themes found in many recently developed artificial intelligence (AI) and autonomous systems ethics and guidance instruments inform a 'normative core' of a principled approach to development, deployment, and adoption (Jobin et al., 2019; UNESCO, 2021; OECD, 2022) of agents whose behaviour relies on the use of artificial intelligence and autonomous decision-making techniques. Significant work has been done in the development of autonomous systems from the perspective of normative ideas (Dennis et al., 2015; Fjeld et al., 2020), including work on transparency (Winfield et al., 2022), explainability, and accountability (Inverardi, 2022). Another research perspective related to our SLEEC framework proposes a data-driven personalised tool based on the moral choices of the user (Alfieri et al., 2022). Our SLEEC framework, however, is concerned with the operationalisation of norms (Townsend et al., 2022; Solanki et al., 2023), and defines a formalisation and an automated process for validating and verifying rules that capture these norms. As such, our work complements the ongoing research on normative aspects of AI and autonomous systems.

Requirement consistency checking is an established (Heitmeyer et al., 1996; Nuseibeh and Easterbrook, 2000) and active (e.g., Bertram et al., 2023; Gärtner and Göhlich, 2024) area of research. Formal techniques such as SMT-based consistency analysis have been successfully used to check the consistency of both functional (Filipovikj et al., 2017, 2018; Bendík, 2017) and non-functional requirements (Becker, 2019). However, none of these approaches has addressed the consistency checking of normative requirements, which is the focus of our SLEEC framework.

There also exists significant research on the development and verification of cyber–physical and autonomous systems, e.g., Luckcuck et al. (2019), Nordmann et al. (2014), Menghi et al. (2019), Bennaceur et al. (2019). Most of these approaches verify the autonomous agents using formal verification methods, such as model checking and theorem proving, by introducing new formalisms, but — complementary to our SLEEC framework — they focus on the safety and reliability requirements of the agents. To the best of our knowledge, the use of formal methods for the consistency validation and for the verification of SLEEC requirements is novel to our framework.

Concerns regarding some level of ethical constraints and legal aspects (Bhuiyan et al., 2020) have been recently studied (Bremner et al., 2019; Dennis et al., 2016) and investigated from a verification perspective (Dennis et al., 2015), although not from the perspective of operationalisation of these requirements. In the approach proposed in Dennis et al. (2015), verification deals with robots that select their actions by evaluating the outcomes of these actions using simulation. Bremner et al. (2019) present a technique for verification of transparency and ethical concerns using a belief–desire–intention model and a simulation module to obtain ethical rules. This line of work is complementary to ours: we focus on formalisation and validation of rules, prior to verification, and cover defeasible reasoning and timed properties. Furthermore, unlike our SLEEC framework, these approaches do not provide a notation dedicated to the encoding of SLEEC-related concerns as requirements like we do here.

9. Conclusion

We have introduced a tool-supported framework for the end-to-end specification, validation and verification of social, legal, ethical, empathetic and cultural requirements for autonomous agents. The framework supports (1) the specification of these requirements as SLEEC rules formalised in a timed domain-specific language grounded in defeasible logic (Horty, 2012; Zalta et al., 2005); and (2) the translation of the rules into the process algebra tock-CSP for redundancy and conflict checking, and for verifying autonomous agent compliance with SLEEC rules. By enabling operationalisation of SLEEC requirements for autonomous agents, our framework complements the significant international efforts to define ethical principles for AI and autonomous systems (UNESCO, 2021; OECD, 2022; IEEE Global Initiative for Ethics of Autonomous and Intelligent Systems, 2019), and our own recent work to elicit SLEEC rules for autonomous agents by starting from relevant normative principles and stakeholder needs (Townsend et al., 2022).

In future work, we will explore several opportunities for extending the applicability and usability of our SLEEC framework. First, we plan to augment the SLEEC language with probabilistic constructs, and thus to provide support for modelling the uncertainty in the environment and decisions of autonomous agents. Second, we intend to augment our tool support with a module that converts the counterexample traces produced by FDR into error messages that are easier to understand for framework users who do not have FDR expertise. Finally, we plan to continue to evaluate the framework in additional case studies from different domains, and with a larger number of SLEEC experts, to identify and address any remaining applicability and usability issues, and to assess the scalability and generality of the framework further.

In the longer term, we will consider also extending the framework with two additional techniques. The first is the runtime verification of autonomous-agent decisions. Many autonomous systems learn, adapt and evolve in operation, e.g., in response to changes in their environment, and therefore cannot be fully verified at development time. The second technique is the online synthesis of SLEEC-compliant adaptation plans for autonomous agents.

One final direction of future work involves the extension of the SLEEC framework with support for multi-user collaboration, enabling experts from all fields to specify rules together, and even to manage (in an interactive way) the contradictory and redundant rules that may emerge.

CRediT authorship contribution statement

Sinem Getir Yaman: Conceptualization, Formal analysis, Investigation, Methodology, Project administration, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Pedro Ribeiro:** Formal analysis, Software, Validation, Visualization, Writing – review & editing, Conceptualization. **Ana Cavalcanti:** Conceptualization, Formal analysis, Investigation, Methodology, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing, Funding acquisition, Software. **Radu Calinescu:** Conceptualization, Formal analysis, Methodology, Project administration, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing, Funding acquisition. **Colin Paterson:** Conceptualization, Methodology, Writing – original draft. **Beverley Townsend:** Conceptualization, Methodology, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgements

The work described in this paper was funded by the EPSRC, United Kingdom project EP/V026747/1 ‘UKRI Trustworthy Autonomous Systems Node in Resilience’. The work of Ana Cavalcanti is also funded by the Royal Academy of Engineering, United Kingdom grant CiET1718/45 and ‘UKRI TAS Verifiability Node EP/V026801/1’. The work of Pedro Ribeiro is funded by EPSRC RoboTest EP/R025479/1. Last but not least, we are grateful to the anonymous reviewers for their constructive comments, which helped us improve the article significantly.

References

Alfieri, C., Inverardi, P., Migliarini, P., Palmiero, M., 2022. Exosoul: Ethical profiling in the digital world. In: Schlobach, S., Pérez-Ortiz, M., Tielman, M. (Eds.), HHAI 2022: Augmenting Human Intellect - Proceedings of the First International Conference on Hybrid Human-Artificial Intelligence. 13–17 June 2022, In: Frontiers in Artificial Intelligence and Applications, vol. 354, IOS Press, pp. 128–142.

Alon, O., Rabinovich, S., Fyodorov, C., Cauchard, J.R., 2021. Drones in firefighting: A user-centered design perspective. In: 23rd International Conference on Mobile Human-Computer Interaction. pp. 1–11.

Anon, 2010. Eclipse XTend. https://www.eclipse.org/xtend/. (Accessed November 2023).

Anon, 2023. SLEECVAL GitHub repository. URL https://github.com/SLEEC-project/SLEECVAL.

Baxter, J., Ribeiro, P., Cavalcanti, A., 2022. Sound reasoning in tock-CSP. Acta Inform. 59 (1), 125–162. http://dx.doi.org/10.1007/s00236-020-00394-3.

Becker, J.S., 2019. Analyzing consistency of formal requirements. Electron. Commun. EASST 76.

Bendík, J., 2017. Consistency checking in requirements analysis. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 408–411.

Bennaceur, A., Ghezzi, C., Tei, K., Kehrer, T., Weyns, D., Calinescu, R., Dustdar, S., Hu, Z., Honiden, S., Ishikawa, F., et al., 2019. Modelling and analysing resilient cyber-physical systems. In: 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS, IEEE, pp. 70–76.

Bergstra, J.A., Klop, J.W., 1985. Algebra of communicating processes with abstraction. Theoret. Comput. Sci. 37, 77–121.

Bertram, V., Kausch, H., Kusmenko, E., Nqiri, H., Rumpe, B., Venhoff, C., 2023. Leveraging natural language processing for a consistency checking toolchain of automotive requirements. In: Schneider, K., Dalpiaz, F., Horkoff, J. (Eds.), 31st IEEE International Requirements Engineering Conference. RE 2023, Hannover, Germany, September 4–8, 2023, IEEE, pp. 212–222. http://dx.doi.org/10.1109/RE57278.2023.00029.

Bhuiyan, H., Olivieri, F., Governatori, G., Islam, M.B., Bond, A., Rakotonirainy, A., 2020. A methodology for encoding regulatory rules. In: MIREL@JURIX. pp. 1–13.

Boltz, N., Sterz, L., Gerking, C., Raabe, O., 2022. A model-based framework for simplified collaboration of legal and software experts in data protection assessments. In: INFORMATIK 2022. Gesellschaft für Informatik, Bonn.

Bremner, P., Dennis, L., Fisher, M., Winfield, A., 2019. On proactive, transparent, and verifiable ethical reasoning for robots. Proc. IEEE PP, 1–21. http://dx.doi.org/10.1109/JPROC.2019.2898267.

Brunero, J., 2021. Reasons and Defeasible Reasoning. Philos. Q. 72 (1), 41–64. http://dx.doi.org/10.1093/pq/pqab013.

Calinescu, R., Cámara, J., Paterson, C., 2019. Socio-cyber-physical systems: Models, opportunities, open challenges. In: 2019 IEEE/ACM 5th International Workshop on Software Engineering for Smart Cyber-Physical Systems. SEsCPS, IEEE, pp. 2–6.

Camilleri, A., Dogramadzi, S., Caleb-Solly, P., 2022. A study on the effects of cognitive overloading and distractions on human movement during robot-assisted dressing. Front. Robot. AI 9.

Cavalcanti, A., Barnett, W., Baxter, J., Carvalho, G., Filho, M.C., Miyazawa, A., Ribeiro, P., Sampaio, A., 2021. RoboStar technology: A roboticist's toolbox for combined proof, simulation, and testing. In: Software Engineering for Robotics. Springer International Publishing, pp. 249–293. http://dx.doi.org/10.1007/978-3-030-66494-7_9 (Chapter 9).

Cervantes, A., Garcia, P., Herrera, C., Morales, E., Tarriba, F., Tena, E., Ponce, H., 2018. A conceptual design of a firefighter drone. In: 15th International Conference on Electrical Engineering, Computing Science and Automatic Control. IEEE, pp. 1–5.

Dennis, L., Fisher, M., Slavkovik, M., Webster, M., 2016. Formal verification of ethical choices in autonomous systems. Robot. Auton. Syst. 77, 1–14. http://dx.doi.org/10.1016/j.robot.2015.11.012.

Dennis, L., Fisher, M., Winfield, A.F.T., 2015. Towards verifiably ethical robot behaviour. arXiv, arXiv:1504.03592.

Filipovikj, P., Rodriguez-Navas, G., Nyberg, M., Seceleanu, C., 2017. SMT-based consistency analysis of industrial systems requirements. In: Proceedings of the Symposium on Applied Computing. pp. 1272–1279.

Filipovikj, P., Rodriguez-Navas, G., Nyberg, M., Seceleanu, C., 2018. Automated SMT-based consistency checking of industrial critical requirements. ACM SIGAPP Appl. Comput. Rev. 17 (4), 15–28.

Fjeld, J., Achten, N., Hilligoss, H., Nagy, A., Srikumar, M., 2020. Principled artificial intelligence: Mapping consensus in ethical and rights-based approaches to principles for AI. Berkman Klein Cent. Res. Publ. 2020–1.

Floridi, L., 2018. Soft ethics and the governance of the digital. Philo. Technol. 31, 1–8. http://dx.doi.org/10.1007/s13347-018-0303-9.

Gärtner, A.E., Göhlich, D., 2024. Towards an automatic contradiction detection in requirements engineering. Proc. Des. Soc. 4, 2049–2058.

Getir Yaman, S., Burholt, C., Jones, M., Calinescu, R., Cavalcanti, A., 2023. Specification and validation of normative rules for autonomous agents. In: 26th International Conference on Fundamental Approaches to Software Engineering. Springer-Verlag, pp. 241–248. http://dx.doi.org/10.1007/978-3-031-30826-0_13.

Getir Yaman, S., Ribeiro, P., Burholt, C., Jones, M., Cavalcanti, A., Calinescu, R., 2024. Toolkit for specification, validation and verification of social, legal, ethical, empathetic and cultural requirements for autonomous agents. Sci. Comput. Program. 236, 103118.

Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A., 2014. FDR3 — A Modern Refinement Checker for CSP. In: Ábrahám, E., Havelund, K. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. In: LNCS, vol. 8413, pp. 187–201.

Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G., 1996. Automated consistency checking of requirements specifications. ACM Trans. Softw. Eng. Methodol. (TOSEM) 5 (3), 231–261.

Hoare, C.A.R., 1978. Communicating sequential processes. Commun. ACM 21 (8), 666–677. http://dx.doi.org/10.1145/359576.359585.

Horty, J.F., 2012. Reasons as defaults. Oxford University Press.

IEEE Global Initiative for Ethics of Autonomous and Intelligent Systems, 2019. Ethically aligned design: A vision for prioritizing wellbeing with autonomous systems and intelligent systems. Version 2. URL https://standards.ieee.org/wp-content/uploads/import/documents/other/ead_v2.pdf.

Innocente, M.S., Grasso, P., 2019. Self-organising swarms of firefighting drones: Harnessing the power of collective intelligence in decentralised multi-robot systems. J. Comput. Sci. 34, 80–101.

Inverardi, P., 2022. The challenge of human dignity in the era of autonomous systems. In: Perspectives on Digital Humanism. Springer International Publishing, Cham, pp. 25–29. http://dx.doi.org/10.1007/978-3-030-86144-5_4 (Chapter 4).

Jevtić, A., Valle, A.F., Alenyà, G., Chance, G., Caleb-Solly, P., Dogramadzi, S., Torras, C., 2018. Personalized robot assistant for support in dressing. IEEE Trans. Cogn. Dev. Syst. 11 (3), 363–374.

Jobin, A., Ienca, M., Vayena, E., 2019. The global landscape of AI ethics guidelines. Nat. Mach. Intell. 1 (9), 389–399.

Li, W., Ribeiro, P., Miyazawa, A., Redpath, R., Cavalcanti, A., Alden, K., Woodcock, J., Timmis, J., 2024. Formal design, verification and implementation of robotic controller software via RoboChart and RoboTool. Auton. Robots 48 (6), 1–22.

Luckcuck, M., Farrell, M., Dennis, L.A., Dixon, C., Fisher, M., 2019. Formal specification and verification of autonomous robotic systems: A survey. ACM Comput. Surv. 52 (5), http://dx.doi.org/10.1145/3342355.

Menghi, C., Tsigkanos, C., Pelliccione, P., Ghezzi, C., Berger, T., 2019. Specification patterns for robotic missions. IEEE Trans. Softw. Eng. 47, 2208–2224.

Mestel, D., Roscoe, A.W., 2020. Translating between models of concurrency. Acta Inform. 57 (3–5), 403–438.

Milner, A.J.R.G., 1983. Calculi for synchrony and asynchrony. Theoret. Comput. Sci. 25, 267–310.

Milner, R., 1999. Communicating and Mobile Systems: the π-calculus. Cambridge University Press.

Mitzner, T.L., Chen, T.L., Kemp, C.C., Rogers, W.A., 2014. Identifying the potential for robotics to assist older adults in different living environments. Int. J. Soc. Robot. 6, 213–227.

Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J., Woodcock, J., 2019. RoboChart: Modelling and verification of the functional behaviour of robotic applications. Softw. Syst. Model. 18, 1–53. http://dx.doi.org/10.1007/s10270-018-00710-z.

Moor, J.H., 2006. The nature, importance, and difficulty of machine ethics. IEEE Intell. Syst. 21 (4), 18–21.

Nordmann, A., Hochgeschwender, N., Wrede, S., 2014. A survey on domain-specific languages in robotics. In: Brugali, D., Broenink, J.F., Kroeger, T., MacDonald, B.A. (Eds.), Simulation, Modeling, and Programming for Autonomous Robots. Springer International Publishing, Cham, pp. 195–206.

Nuseibeh, B., Easterbrook, S., 2000. Requirements engineering: a roadmap. In: Proceedings of the Conference on the Future of Software Engineering. pp. 35–46.

OECD, 2022. Recommendation of the Council on Artifical Intelligence, OECD/LEGAL/0449. http://legalinstruments.oecd.org. (Accessed 27 March 2023).

Pirhonen, J., Melkas, H., Laitinen, A., Pekkarinen, S., 2020. Could robots strengthen the sense of autonomy of older people residing in assisted living facilities?—A future-oriented study. Ethics Inf. Technol. 22 (2), 151–162.

Roscoe, A.W., 1998. The Theory and Practice of Concurrency. Prentice-Hall Series in Computer Science, Prentice-Hall.

Roscoe, A.W., 2013. The automated verification of timewise refinement. In: First Open EIT ICT Labs Workshop on Cyber-Physical Systems Engineering.

Solanki, P., Grundy, J., Hussain, W., 2023. Operationalising ethics in artificial intelligence for healthcare: a framework for AI developers. AI Ethics 3 (1), 223–240. http://dx.doi.org/10.1007/s43681-022-00195-z.

Townsend, B., Paterson, C., Arvind, T.T., Nemirovsky, G., Calinescu, R., Cavalcanti, A., Habli, I., Thomas, A., 2022. From pluralistic normative principles to autonomous-agent rules. Minds Mach. 32 (4), 683–715. http://dx.doi.org/10.1007/s11023-022-09614-w.

UNESCO, 2021. Recommendation on the Ethics of Artificial Intelligence. https://unesdoc.unesco.org/ark:/48223/pf0000380455. (Accessed 18 March 2022), Document code: SHS/BIO/REC-AIETHICS/2021.

Winfield, A., Watson, E., Egawa, T., Barwell, E., Barclay, I., Booth, S., Dennis, L.A., Hastie, H., Hossaini, A., Jacobs, N., Markovic, M., Muttram, R.I., Nadel, L., Naja, I., Olszewska, J., Rajabiyazdi, F., Rannow, R.K., Theodorou, A., Underwood, M.A., von Stryk, O., Wortham, R.H., 2022. IEEE Standard for Transparency of Autonomous Systems. Institute of Electrical and Electronics Engineers (IEEE), p. 52. http://dx.doi.org/10.1109/IEEESTD.2022.9726144.

Wing, J.M., 2021. Trustworthy AI. Commun. ACM 64 (10), 64–71.

Zalta, E.N., Nodelman, U., Allen, C., Anderson, R.L., 2005. Defeasible Reasoning,stanford Encyclopedia of Philosophy. Stanford University, Palo Alto CA.

Zhang, F., Demiris, Y., 2022. Learning garment manipulation policies toward robot-assisted dressing. Sci. Robot. 7 (65), eabm6010.

Sinem Getir Yaman is a Postdoctoral Research Associate in the University of York's Department of Computer Science. She holds a Ph.D. in Computer Science from Humboldt University of Berlin. Her research focuses on methods for the modelling and verification of non-functional requirements, including SLEEC requirements, for software quality evaluation under uncertainty.

Pedro Ribeiro is a lecturer in Computer Science at the University of York and a member of the RoboStar centre of excellence. He has over a decade of experience on formal modelling and verification with applications to modern software engineering approaches, and is one of the core developers of RoboTool and its integration with the SLEEC toolkit. He has served on multiple programme committees in the area of formal methods, and is a co-founder of Formal Method's Europe communications committee.

Ana Cavalcanti is Professor of Software Verification at the University of York and Royal Academy of Engineering Chair in Emerging Technologies working on Software Engineering for Robotics. She currently leads the RoboStar centre of excellence, whose approach to model-based software engineering complements the current practice of design and verification of mobile and autonomous robots, covering simulation, testing and proof.

Radu Calinescu is Professor of Computer Science at the University of York, UK. His research interests include formal methods for self-adaptive, autonomous, secure and dependable software, cyber–physical and AI systems, and in performance and reliability software engineering. He is an active promoter of formal methods at runtime as a way to improve the integrity and predictability of self-adaptive, autonomous and AI systems and processes.

Colin Paterson is a Senior Lecturer in Computer Science at the University of York, UK. He holds Ph.D.s in Control Systems Engineering, gained in collaboration with Jaguar Cars, and Computer Science, in which techniques for the verification of operational processes, using observation data, were developed. His current research considers techniques for the development and assurance of resilient autonomous and AI systems operating in uncertain environments.

Beverley Townsend is a BRAID (Bridging Responsible AI Divides) Research Fellow at the University of York, UK, specialising in the law and ethics of emerging digital technologies. Her expertise is in data protection and in integrating the law and ethics into safe and resilient autonomous systems (robots). Her research has focused on digital health, privacy, data protection law, data sharing and international data transfers, ethics, human rights, biotechnologies, AI, and regulation and governance.