

This is a repository copy of *Specification, Validation and Verification of Social, Legal, Ethical, Empathetic and Cultural Requirements for Autonomous Agents*.

White Rose Research Online URL for this paper:

<https://eprints.whiterose.ac.uk/id/eprint/202062/>

Preprint:

Yaman, Sinem Getir, Cavalcanti, Ana orcid.org/0000-0002-0831-1976, Calinescu, Radu orcid.org/0000-0002-2678-9260 et al. (3 more authors) (2023) Specification, Validation and Verification of Social, Legal, Ethical, Empathetic and Cultural Requirements for Autonomous Agents. [Preprint]

Reuse

Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

Specification, Validation and Verification of Social, Legal, Ethical, Empathetic and Cultural Requirements for Autonomous Agents

Sinem Getir Yaman¹, Ana Cavalcanti¹, Radu Calinescu¹, Colin Paterson¹, Pedro Ribeiro¹, and Beverley Townsend²

1. Department of Computer Science, University of York, UK
2. York Law School, University of York, UK

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.

Keywords: software specification; timed systems; verification and validation; sociotechnical systems

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 [1] has become increasingly important and challenging for these applications [2, 3, 4]. 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 [5, 6] and verification [7] of ethical concerns of autonomous agents, modelling of legal requirements for software systems [8], and development of personalised ethical assistant tools based on the moral choices of the user [9].

In this 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 [1, 10]. To that end, we introduce a tool-supported SLEEC requirement specification and verification framework that 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 [11, 12] 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*. Given a set of SLEEC rules for an autonomous agent, our framework automates: (i) their formalisation in tock-CSP [13], a version of the communicating sequential processes (CSP) algebra [14] that can describe discrete-time properties; (ii) the validation of their consistency, to ensure that the rules are not conflicting, and to identify redundant rules; and (ii) the verification of an agent's compliance with the validated rules.

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 developed using a domain-specific language for robotics, RoboChart [15]. 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 [16].

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 6 details our evaluation, describing the tool support provided, and the two case studies. Finally, Section 7 covers related work, and Section 8 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 [17, 18, 19]. 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 is defined in Figure 1. 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.

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 that is always responsive. 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.

□

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

Figure 1: BNF syntax of the SLEEC language

```

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.

```

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

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.

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 over measures can include conjunctions (**and**), disjunctions (**or**), and equalities and inequalities (**relOp**) over **numeric** and **scale** measures and relevant values.

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

```

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

```

Listing 3: Extended version of Rule2 with `otherwise` construct

A distinctive feature of our SLEEC language is that it can specify time constraints: time budgets 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. $\square$

For situations where the response event 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 seconds 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 seconds. $\square$

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, so as 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 minutes. 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. $\square$

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 for the use of defeasible reasoning when there are scenarios leading to reasons that outweigh or disable a constraint [20]. Defeasible reasoning is supported in our SLEEC language via `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. $\square$

Overall, multiple defeaters (grouped, if needed, within curly brackets `{...}` to indicate the constraint they apply to) alongside time constraints 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 [21]. CSP is part of a large family of notations for specifying concurrent systems [22, 23, 24], and is distinctive in its denotational semantics, giving rise to notions of refinement useful for stepwise development. A powerful model checker called FDR [16] 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.

4.1. Overview of *tock-CSP*

CSP processes specify patterns of interaction via synchronisation on channels, taking into account (non)determinism, deadlock, 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*.

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.

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

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 , $\llbracket c \rrbracket$ is a set of events; if c is a typed channel then events are constructed using the dot notation, so that $\llbracket c \rrbracket = \llbracket c.v_0, \dots, c.v_n \rrbracket$, 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
$P \square Q$	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
$P \parallel Q$	Interleaving: P and Q run in parallel and do not interact with each other
$P \llbracket X \rrbracket Q$	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
$\square i : I \bullet P(i)$	Replicated external choice: offers an external choice over processes $P(i)$ for all i in I

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 ($\rightarrow$) starts *Tracking*. 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  $\rightarrow$ 
    if t > 35
    then 1  $\blacktriangleleft$  (SoundFireAlarm  $\rightarrow$  Skip)
    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 ($\blacktriangleleft$) 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. 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. $\square$

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** from Figure 1 is given by a function $\llbracket _ \rrbracket_S$ defined in Table 2. This function maps the **specification** to a *tock*-CSP process, and is defined in terms of two other functions, $\llbracket _ \rrbracket_{DS}$ and $\llbracket _ \rrbracket_{RS}$, that capture the semantics of the definitions **dB** and rules **rB** of the **specification**. The definitions in Table 2 are mechanised in our SLEEC tool [25], which automates the generation of the *tock*-CSP semantics of a SLEEC **specification** (see Section 6.1).

The semantics of **definitions** from Figure 1 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

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.

$\llbracket \text{def_start } dB \text{ def_end } rule_start \text{ rB } rule_end \rrbracket_S = \llbracket dB \rrbracket_{DS} \quad \llbracket rB \rrbracket_{RS}$	
$\llbracket \text{def} \rrbracket_{DS}$	$= \llbracket \text{def} \rrbracket_D$
$\llbracket \text{def defS} \rrbracket_{DS}$	$= \llbracket \text{def} \rrbracket_D \quad \llbracket \text{defS} \rrbracket_{DS}$
$\llbracket \text{event } eID \rrbracket_D$	$= \text{channel } eID$
$\llbracket \text{measure } mID : T \rrbracket_D$	$= \text{channel } mID : \llbracket T, mID \rrbracket_T$
$\llbracket \text{constant } cID = v \rrbracket_D$	$= cID = v$
$\llbracket \text{boolean}, mID \rrbracket_T$	$= \text{Bool}$
$\llbracket \text{numeric}, mID \rrbracket_T$	$= \text{Int}$
$\llbracket \text{scale}(sp_1, \dots, sp_n), mID \rrbracket_T$	$= STmID$
	$\text{datatype } STmID = sp_1 \mid \dots \mid sp_n$
	$STlemID(v1mID, v2mID) =$
	if $v1mID == sp_1$ then true
	else (if $v1mID == sp_2$ then $v2mID \notin \{sp_1\}$
	else ...
	else $v2mID == sp_n$)
$\llbracket r \rrbracket_{RS}$	$= \llbracket r \rrbracket_R$
$\llbracket r \text{ rS} \rrbracket_{RS}$	$= \llbracket r \rrbracket_R \quad \llbracket rS \rrbracket_{RS}$
$\llbracket rID \text{ when } trig \text{ then } resp \rrbracket_R$	$= rID = TriggerrID ; MonitoringrID ; rID$
	$TriggerrID = \llbracket trig, \alpha_E(resp), Skip, TriggerrID \rrbracket_{TG}$
	$MonitoringrID = \llbracket resp, trig, \alpha_E(resp), MonitoringrID \rrbracket_{RDS}$

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 Figure 2, 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 *v1windSpeed* and *v2windSpeed* (of type *STwindSpeed*). If *v1windSpeed* is the first literal *light*, then it is guaranteed to be less than or equal to *v2windSpeed*, no matter the value of *v2windSpeed*. If, however *v1windSpeed* is *moderate*, then the inequality holds if *v2windSpeed* is not *light*, since it is then at least *moderate* as well. Finally, if *v1windSpeed* is *strong*, then the inequality holds if, and only if, *v2windSpeed* 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*. $\square$

The recursive definition of $\llbracket _ \rrbracket_{DS}$ is given by two equations. For a single definition *def*, the semantics is given by another function $\llbracket _ \rrbracket_D$. 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 $\llbracket \text{def} \rrbracket_D$ to capture the semantics of *def*, followed by the CSP declarations defined by a recursive application of $\llbracket _ \rrbracket_{DS}$ to *defS*. We do not consider the empty list of defi-

```

channel BatteryCritical
channel CameraStart
channel SoundAlarm
channel GoHome
channel personNearby : Bool
channel temperature : Int
channel windSpeed : STwindSpeed
datatype STwindSpeed = light | moderate | strong
STlewindSpeed(v1windSpeed, v2windSpeed) =
    if v1windSpeed == light
    then true
    else ( if v1windSpeed == moderate
           then (v2windSpeed  $\notin$  {light})
           else v2windSpeed == strong )
ALARM_DEADLINE = 3

```

Figure 2: CSP declarations for the definitions in Listing 1

nitions, 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 $\llbracket _ \rrbracket_D$ consider each form of **definition** in turn. We assume that identifiers in SLEEC satisfy the usual lexical restrictions adopted in CSP, so that, for example, events and measures are represented by

```

Rule2 = TriggerRule2 ; MonitoringRule2 ; Rule2
TriggerRule2 =
  let MTrigger = 0 ◀ (personNearby?vpersonNearby →
    if vpersonNearby == true
    then Skip
    else TriggerRule2)
  within CameraStart → MTrigger
  □
  SoundAlarm → TriggerRule2
MonitoringRule2 = 2 ◀ (SoundAlarm → Skip)

```

Figure 3: Semantics of Rule2 in Listing 2

channels of the same name. For a **constant**, we assume that a value is given. The type used in the declaration of a measure is given by the semantic function $\llbracket _ \rrbracket_{\mathbf{T}}$ whose arguments are the type $\mathbf{T}$ and the identifier $\mathbf{mID}$ of the measure. The equations defining $\llbracket _ \rrbracket_{\mathbf{T}}$ for **boolean** and **numeric** are straightforward. For a **scale** type, we use the name of the measure in defining the corresponding CSP declarations.

For simplicity, we use an informal notation to represent a **scale** type with n parameters sp_1 to sp_n , namely, $\text{scale}(sp_1, \dots, sp_n)$. 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 $\mathbf{mID}$, 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 $\llbracket _ \rrbracket_{\mathbf{RS}}$ for a list of rules is similar to that of $\llbracket _ \rrbracket_{\mathbf{DS}}$, but is based on the semantic function $\llbracket _ \rrbracket_{\mathbf{R}}$ for a **rule**. The semantics of each rule 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 Figure 3. The behaviour of the process *Rule2* is initially defined by that of *TriggerRule2*. When the trigger of Rule2 is observed, *TriggerRule2* terminates (via the **Skip** from the conditional statement), and the process *MonitoringRule2* takes over. When the response happens, *MonitoringRule2* terminates and *Rule2* recurses. $\square$

In Table 2, the definition of $\llbracket _ \rrbracket_{\mathbf{R}}$ uses the identifier $\mathbf{rID}$ 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 $\llbracket _ \rrbracket_{\mathbf{TG}}$ whose argu-

ments 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 $\alpha_{\mathbf{E}}(\text{resp})$. The first continuation process determines the behaviour when the trigger happens. In the definition of $\llbracket _ \rrbracket_{\mathbf{R}}$, 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 $\llbracket _ \rrbracket_{\mathbf{R}}$, 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 $\llbracket _ \rrbracket_{\mathbf{RDS}}$. The first argument is the **response** that is to be monitored. The subsequent arguments are needed because a **defeater** may void the monitoring, which then needs to check the trigger again. The extra arguments are the **trigger** and the alphabet of events used in the response. The final argument of $\llbracket _ \rrbracket_{\mathbf{RDS}}$ is a continuation process, which in the definition of $\llbracket _ \rrbracket_{\mathbf{R}}$ is *Monitoring*.

We define $\llbracket _ \rrbracket_{\mathbf{TG}}$ next, and $\llbracket _ \rrbracket_{\mathbf{RDS}}$ 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 $\llbracket _ \rrbracket_{\mathbf{TG}}$ is given in Table 3. For a **trigger** that has just an event $\mathbf{eID}$, the process is a synchronisation on that event followed by the argument process $\mathbf{sp}$ 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 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 $\llbracket \mathbf{eID} \text{ and } \mathbf{mBE}, \mathbf{AR}, \mathbf{sp}, \mathbf{fp} \rrbracket_{\mathbf{TG}}$, we have the synchronisation on $\mathbf{eID}$ is followed by a process *MTrigger*, defined in the **let** clause using a semantic function $\llbracket _ \rrbracket_{\mathbf{ME}}$.

Example 4.4. As shown in Figure 3, 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. $\square$

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.

$\llbracket \text{eID}, \text{AR}, \text{sp}, \text{fp} \rrbracket_{\text{TG}}$	$= \text{eID} \rightarrow \text{sp} \square (\square \text{e} : \text{AR} \bullet \text{e} \rightarrow \text{fp})$
$\llbracket \text{eID and mBE}, \text{AR}, \text{sp}, \text{fp} \rrbracket_{\text{TG}}$	$= \text{let } M\text{Trigger} = \llbracket \alpha_{\text{ME}}(\text{mBE}), \text{mBE}, \text{sp}, \text{fp} \rrbracket_{\text{ME}}$ $\quad \text{within } \text{eID} \rightarrow M\text{Trigger} \square (\square \text{e} : \text{AR} \bullet \text{e} \rightarrow \text{fp})$
$\llbracket \langle \rangle, \text{mBE}, \text{sp}, \text{fp} \rrbracket_{\text{ME}}$	$= \text{if norm}(\text{mBE}) \text{ then } \text{sp} \text{ else } \text{fp}$
$\llbracket \langle \text{mID} \rangle \wedge \text{mIDs}, \text{mBE}, \text{sp}, \text{fp} \rrbracket_{\text{ME}} = 0$	$\blacktriangleleft (\text{mID}?v\text{mID} \rightarrow \llbracket \text{mIDs}, \text{mBE}[v\text{mID}/\text{mID}], \text{sp}, \text{fp} \rrbracket_{\text{ME}})$

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

The inductive definition of $\llbracket _ \rrbracket_{\text{ME}}$ 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 *v***mID** 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 $\langle \text{mID} \rangle \wedge \text{mIDs}$, the definition of $\llbracket _ \rrbracket_{\text{ME}}$ defines a process that reads the value of **mID** and records it into a local variable named *v***mID**. To make that urgent, it uses the operator $\blacktriangleleft$ with deadline 0 over a process that starts with the communication **mID**?*v***mID**. The behaviour that follows is defined by the process characterised by a recursive application of $\llbracket _ \rrbracket_{\text{ME}}$.

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

If the first argument of $\llbracket _ \rrbracket_{\text{ME}}$ 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, $\llbracket _ \rrbracket_{\text{ME}}$ 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 extra 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 $\llbracket _ \rrbracket_{\text{RDS}}$ for **response** definitions is specified in Table 4. The semantics of a **response** en-

closed 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 $\llbracket _ \rrbracket_{\text{C}}$.

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 Figure 3, captures the time constraint in the response. It requires that *SoundAlarm* takes place within 2 time units. In this case, we have the assumption that each *tock* represents the passage of 1 second. $\square$

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 millisecond, 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 minute” should be normalised to 60.

The definition of $\llbracket _ \rrbracket_{\text{C}}$ has one equation for each possible form of constraint. If it is just an event, the constraint process defined by $\llbracket _ \rrbracket_{\text{C}}$ 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 $\blacktriangleleft$. 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.

If the possibility of a timeout is considered, via an **otherwise** clause, instead of a $\blacktriangleleft$, we use a timed interrupt (Δ_d) to specify that, if the budget $\text{norm}(\text{v}, \text{tU})$ 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.

$$(\text{SoundAlarm} \rightarrow \text{Skip}) \Delta_2 (\text{GoHome} \rightarrow \text{Skip})$$

If after 2 seconds, for whatever reason, the alarm cannot be sounded, then *GoHome* is required. $\square$

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`.

$\llbracket \text{const}, \text{trig}, \text{ARDS}, \text{mp} \rrbracket_{\text{RDS}}$	$= \llbracket \text{const}, \text{trig}, \text{ARDS}, \text{mp} \rrbracket_{\text{C}}$
$\llbracket \text{const } dfts, \text{trig}, \text{ARDS}, \text{mp} \rrbracket_{\text{RDS}}$	$= \text{let } \llbracket (\text{const}) \wedge dfts \upharpoonright_{\text{RP}}, \text{trig}, \text{ARDS}, \text{mp}, 1 \rrbracket_{\text{LRDS}}$ $\quad \text{within } \llbracket \alpha_{\text{ME}}(dfts), dfts, \#dfts + 1 \rrbracket_{\text{CDS}}$
$\llbracket \text{elD}, \text{trig}, \text{ARDS}, \text{mp} \rrbracket_{\text{C}}$	$= \text{elD} \rightarrow \text{Skip}$
$\llbracket \text{elD within } v \text{ tU}, \text{trig}, \text{ARDS}, \text{mp} \rrbracket_{\text{C}}$	$= \text{norm}(v, \text{tU}) \blacktriangleleft (\text{elD} \rightarrow \text{Skip})$
$\llbracket \text{elD within } v \text{ tU otherwise } \text{resp}, \text{trig}, \text{ARDS}, \text{mp} \rrbracket_{\text{C}}$	$= (\text{elD} \rightarrow \text{Skip}) \triangleleft_{\text{norm}(v, \text{tU})} (\llbracket \text{resp}, \text{trig}, \text{ARDS}, \text{mp} \rrbracket_{\text{RDS}})$
$\llbracket \text{not elD within } v \text{ tU}, \text{trig}, \text{ARDS}, \text{mp} \rrbracket_{\text{C}}$	$= \text{Wait}(\text{norm}(v, \text{tU}))$
$\llbracket \langle \text{resp} \rangle, \text{trig}, \text{AR}, \text{mp}, n \rrbracket_{\text{LRDS}}$	$= \text{Monitoring}_n = \llbracket \text{resp}, \text{trig}, \text{AR}, \text{mp} \rrbracket_{\text{RDS}}, \text{ provided } \text{resp} \neq \text{NoRep}$
$\llbracket \langle \text{NoRep} \rangle, \text{trig}, \text{AR}, \text{mp}, n \rrbracket_{\text{LRDS}}$	$= \text{Monitoring}_n = \llbracket \text{trig}, \text{AR}, \text{mp}, \text{Monitoring}_n \rrbracket_{\text{TG}}$ $\quad \square$ $\quad (\square e : \text{AR} \bullet e \rightarrow \text{Monitoring}_n)$
$\llbracket \langle \text{resp} \rangle \wedge \text{resps}, \text{trig}, \text{AR}, \text{mp}, n \rrbracket_{\text{LRDS}}$	$= \llbracket \langle \text{resp} \rangle, \text{trig}, \text{AR}, \text{mp}, n \rrbracket_{\text{LRDS}}$ $\quad \llbracket \text{resps}, \text{trig}, \text{AR}, \text{mp}, n + 1 \rrbracket_{\text{LRDS}}$
$\llbracket \langle \rangle, dfts, n \rrbracket_{\text{CDS}}$	$= \llbracket dfts, \text{Monitoring}_1, n \rrbracket_{\text{EDS}}$
$\llbracket \langle \text{mID} \rangle \wedge \text{mIDs}, dfts, n \rrbracket_{\text{CDS}}$	$= 0 \blacktriangleleft (\text{mID} ? \text{vmID} \rightarrow \llbracket \text{mIDs}, dfts[\text{vmID}/\text{mID}], n \rrbracket_{\text{CDS}})$
$\llbracket \text{unless mBE}, \text{fp}, n \rrbracket_{\text{EDS}}$	$= \text{if } \text{norm}(\text{mBE}) \text{ then } \text{Monitoring}_n \text{ else fp}$
$\llbracket \text{unless mBE then } \text{resp}, \text{fp}, n \rrbracket_{\text{EDS}}$	$= \text{if } \text{norm}(\text{mBE}) \text{ then } \text{Monitoring}_n \text{ else fp}$
$\llbracket dfts \text{ dft}, \text{fp}, n \rrbracket_{\text{EDS}}$	$= \llbracket \text{dft}, \llbracket dfts, \text{fp}, n - 1 \rrbracket_{\text{EDS}}, n \rrbracket_{\text{EDS}}$

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

If a response has one or more defeaters, $\llbracket - \rrbracket_{\text{RDS}}$ 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
  Monitoring1 = SoundAlarm → Skip
  Monitoring2 = GoHome → Skip
within
  0 ◀ (personNearby? vpersonNearby →
    if vpersonNearby == true
    then Monitoring2 else Monitoring1)

```

The constraint in the **then** clause and the response in the **unless** clause are captured by local processes *Monitoring1* and *Monitoring2*. In the **within** clause, after reading the relevant measures, the process chooses a local *Monitoring* process based on the **unless** condition. $\square$

The local *Monitoring* processes are defined by $\llbracket - \rrbracket_{\text{LRDS}}$, which takes as argument a list containing the constraint of the response, and the responses in the defeaters *dfts*. We use $dfts \upharpoonright_{\text{RP}}$ to represent the list of those responses. For an **unless** defeater without a response, we get **NoRep**.

```

Rule4_a when CameraStart then SoundAlarm
        unless personNearby then GoHome

```

Listing 4: Simpler version of Rule4

This is a special response defined in the semantics just for the purposes of simplifying the semantic rules. The additional arguments of $\llbracket - \rrbracket_{\text{LRDS}}$ are the extra arguments of $\llbracket - \rrbracket_{\text{RDS}}$, and a counter for the *Monitoring* processes used to define their names. In the definition of $\llbracket - \rrbracket_{\text{RDS}}$, we define the (initial) value of the counter as 1.

The definition of $\llbracket - \rrbracket_{\text{LRDS}}$ has two equations for a singleton list of responses, and a third equation for a list $\langle \text{resp} \rangle \wedge \text{resps}$ 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 $\llbracket - \rrbracket_{\text{RDS}}$. For the special response **NoRep**, we need to consider the trigger.

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 *Monitoring3* 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

```
Monitoring1 = SoundAlarm → Skip
Monitoring2 = GoHome → Skip
Monitoring3 = CameraStart → MonitoringRule4
              □ SoundAlarm → Monitoring3
              □ GoHome → Monitoring3
```

within

```
0 ◀ (personNearby? vpersonNearby →
    0 ◀ (temperature? vpersonNearby →
        if vtemperature > 35 then Monitoring3
        else (if vpersonNearby == true
            then Monitoring2 else Monitoring1)))
```

Monitoring3 corresponds to the **unless** clause for the condition **temperature > 35**, which does not have an associated response. The meaning in this case is that the rule imposes no restrictions. So, all CSP events used in the rule semantics need to be allowed. In the example, the *Monitoring3* process needs to offer the choice ($\square$) of all CSP events corresponding to the events used in *Rule4*. For the event in the trigger, its occurrence leads to the outer *MonitoringRule4* process taking over. For all other events, *Monitoring3* simply recurses: *Monitoring3* does not block these events, but ignores them by just proceeding.

As illustrated, the definition of a local *Monitoring* process for an **unless** clause without a response, that is, with a response **NoRep**, requires information about the trigger of the overall rule, its alphabet, and the overall *Monitoring* process. These are the extra arguments of $\llbracket - \rrbracket_C$ and $\llbracket - \rrbracket_{LRDS}$.

For a response **NoRep**, the local process *Monitoring_n* defined by $\llbracket - \rrbracket_{LRDS}$, when applied to a counter value *n*, is specified as a choice over a trigger process characterised by $\llbracket - \rrbracket_{TG}$ and a choice of events *e* from the alphabet **AR** of the rule given as argument. In every choice, such an event *e* is followed by a recursion. In our example above, **AR** contains *SoundAlarm* and *GoHome*. The arguments of $\llbracket - \rrbracket_{TG}$ are the trigger, the process **mp**, providing the continuation in case the trigger occurs, and the process *Monitoring_n*, the continuation if the trigger does not occurs. We recall that $\llbracket - \rrbracket_R$ defines **mp** as the *Monitoring* process for the rule.

The process in the **within** clause of a response process (as defined by $\llbracket - \rrbracket_{RDS}$) is specified by a $\llbracket - \rrbracket_{CDS}$ function. Its arguments are the alphabet $\alpha_{ME}(dfts)$ 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 $\alpha_{ME}(dfts)$ is similar to that of $\alpha_{ME}(mBE)$, but applies to a list of defeaters, considering the Boolean expressions that they use.

The inductive definition of $\llbracket - \rrbracket_{CDS}$ is simple. For a list of measures $\langle mID \rangle \wedge mIDs$, the process inputs the values *vmID* or the measure urgently and then behaves as the process defined by $\llbracket - \rrbracket_{CDS}$ for *mIDs*. In the defeaters used

as argument for the recursive application of $\llbracket - \rrbracket_{CDS}$, 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 $\llbracket - \rrbracket_{EDS}$. This is again an inductive definition, whose arguments are the list of defeaters, the local *Monitoring1* process that applies when no defeater in the list does, and the number of defeaters in that list. We recall that *Monitoring1* is the process that captures the behaviour of the overall constraint of the rule (see definitions of $\llbracket - \rrbracket_{RDS}$ and $\llbracket - \rrbracket_{LRDS}$). If the list of defeaters *dfts dft* has more than one defeater, the result is the application of $\llbracket - \rrbracket_{EDS}$ to the last defeater, whose monitoring process is given by a recursive application of $\llbracket - \rrbracket_{EDS}$ 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. If 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 $\llbracket - \rrbracket_{EDS}$ to define the conditional in Example 4.8.

```
[[ unless vpersonNearby then GoHome
   unless vtemperature > 35,
   Monitoring1, 3 ]]EDS
=
[[ unless vtemperature > 35,
   [[unless vpersonNearby then GoHome, Monitoring1, 2]]EDS,
   3 ]]EDS
=
[[ unless vtemperature > 35,
   ( if vpersonNearby == true
     then Monitoring2 else Monitoring1 ),
   3 ]]EDS
=
if vtemperature > 35
then Monitoring3
else ( if vpersonNearby == true
      then Monitoring2 else Monitoring1 )
```

Additional examples are provided in the next sections. $\square$

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 conflict needs to be resolved. In Section 5.1, we present an approach that uses the semantics of our rules presented above to detect conflicts, and in Section 5.3, we present redundancy checks. Finally, in Section 5.3, we discuss the verification of an agent model against a set of SLEEC rules.

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 $\llbracket r1, r2 \rrbracket_{CP}$, which defines the conjunction process $\text{idCC}(r1, r2)$ for the rules $r1$ and $r2$. In the **within** clause of this definition, we compose the processes $\text{id}(r1)$ and $\text{id}(r2)$ (which define the semantics of $r1$ and $r2$) in parallel ($\llbracket \dots \rrbracket$), synchronising on their common alphabet of events, i.e., on the intersection alphabets $\alpha_E(r1) \cap \alpha_E(r2)$ of their alphabets.

An additional parallel process Env captures the environment in which the rules are considered, by recording the values of the measures for sharing between $\text{id}(r1)$ and $\text{id}(r2)$. 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 $\alpha_M(r1, r2)$ of $r1$ and $r2$. These processes input the value of the measure e when a rule first requires that measure ($e?x$). They then record the value x input as a parameter for another process $VEnv_e$, which outputs x ($e!x$) 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 $\llbracket r1, r2 \rrbracket_{CP}$, 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 $\llbracket r1, r2 \rrbracket_{CP}$, there is a trace t_2 of $\llbracket r1, r2 \rrbracket_{CP}$ after t_1 that contains at least one *tock* and at least one event different from *tock*.

Table 5: Conjunction of rules $r1$ and $r2$

$\llbracket r1, r2 \rrbracket_{CP} =$	$\text{idCC}(r1, r2) = \text{let}$
	$Env = \llbracket e : \alpha_M(r1, r2) \bullet Env_e$
	$Env_e = e?x \rightarrow VEnv_e(x)$
	$VEnv_e(x) = e!x \rightarrow VEnv_e(x)$
	within
	$(\text{id}(r1) \llbracket \alpha_E(r1) \cap \alpha_E(r2) \rrbracket \text{id}(r2))$
	$\llbracket \alpha_M(r1, r2) \rrbracket$
	Env

With this definition, we require that, at no point, enforcing both rules, as defined by the process $\llbracket r1, r2 \rrbracket_{CP}$, 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 $\llbracket r1, r2 \rrbracket_{CP}$. 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 an assertion based on our mechanisation of a timed-deadlock freedom check in the context of tock-CSP that is inspired by work in [26].

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 Figure 4 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 $Env_{\text{temperature}}$ 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*. In this case, we are identifying a time unit with 1s. So, the counterexample, indicates that if RuleA is triggered, and after 13s, Rule3 is triggered, then, after another 47s, we have a deadlock, as RuleA requires *GoHome* to take place, but Rule3 forbids it. $\square$

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

```

RuleARule3 = let Envtemperature = temperature?x → VEnvtemperature(x)
              VEnvtemperature(x) = temperature!x → VEnvtemperature(x)
              Env = Envtemperature
              within(RuleA || { GoHome } || Rule3) || { temperature } || Env

```

Figure 4: Conjunction process for RuleA in Listing 5 and Rule3 in Listing 2.

```

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

```

Listing 5: Conflicting rule for a firefighter robot

```

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

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. $\square$

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 superfluous 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 .

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 $\text{id}(r1)$, there is a trace of t_2 of $\llbracket r1, r2 \rrbracket_{\text{CP}}$, such that, $t_1 \upharpoonright \alpha_E(r1) = t_2 \upharpoonright \alpha_E(r1, r2)$.

First of all, we observe that 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 $\alpha_E(r1)$ 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 $t_2 \upharpoonright \alpha_E(r1, r2)$ characterise the restrictions of $r1$ and $r2$. If every behaviour allowed by $r1$ is also allowed by $r1$ and $r2$, then $r2$ imposes no additional restrictions, and it is, therefore, redundant.

The mechanisation of this check is direct, since trace inclusion in CSP corresponds to refinement, and ignoring events can be captured using the hiding CSP operator ($\upharpoonright$).

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. $\square$

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. Next, however, we consider autonomous agent conformance to SLEEC rules.

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, if that. 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 feasi-

ble. The treatment of more complex data types provided by measures, however, are 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 by means of refinement in tock-CSP. To that end, we assume the existence of a tock-CSP model for the SUV. Such models can be generated automatically from other design models, or can be devised manually by system developers with CSP modelling expertise. Here, we consider examples where a RoboChart [15] model for the agent is available and used as a basis to generate a tock-CSP model SUV automatically.

The notion of conformance $r \models_{TT} SUV$ that we adopt is defined below. In words, it corresponds to traces refinement in tock-CSP, where the specification is defined in terms of the process $\llbracket r \rrbracket_R$ 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. (In the definition of redundancy from Section 5.2, this is ensured by the conjunction process.)

Definition 5.3. An SUV conforms to a rule r , written $r \models_{TT} SUV$, 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 $\llbracket r \rrbracket_R$ such that: (1) $t_1 \upharpoonright \alpha_E(r) = t_2 \upharpoonright \alpha_E(r)$ and; (2) for every event e of $\alpha_E(r)$ in a position i of these traces, for every measure m in $\alpha_M(r)$, 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. The value v of a measure m at i in a trace t is that in the last event $m.v$ before the occurrence of the i -th event of $\alpha_E(r)$ in t . Conformance requires that, if a rule reads a measure, a conforming SUV must read that measure as well. Moreover, when checking conformance, we consider traces based on the same values for those measures.

As mentioned earlier, the mechanisation of conformance checking is based on refinement, but the specification is a weakening of $\llbracket r \rrbracket_R$, 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 Figure 5, we show a RoboChart interface **Capabilities** that declares those capabilities of the firefighter UAV that we 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 (**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 local 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 this event 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 that requires the wind not to be **strong** ($wind \neq windScale::strong$, where **windScale** is the enumeration type of **wind** defined on the left in Figure 5), and the temperature to be high ($temp > 35$), 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**. Every 1s, this state machine reads that measure and records it in the variable **person**. Depending on whether there is such a person or not, it raises the event **SoundAlarm**. A transition out of **Recording** ensures that, when the **BatteryCritical** is signalled, the UAV goes back to base by raising the event **GoHome**. The guard ensures that the amount of time since the state **Recording** has been entered ($sinceEntry(Recording)$) is greater than 0, so that the check for the presence of a person is carried out before returning.

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, tock
SoundAlarm, 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 as *light* and 36, respectively. With that, in the state **Recording**, the camera is started, when there is a **personNearby**. So, after 1 s (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 minutes. Indeed, the projection of this UAV trace to

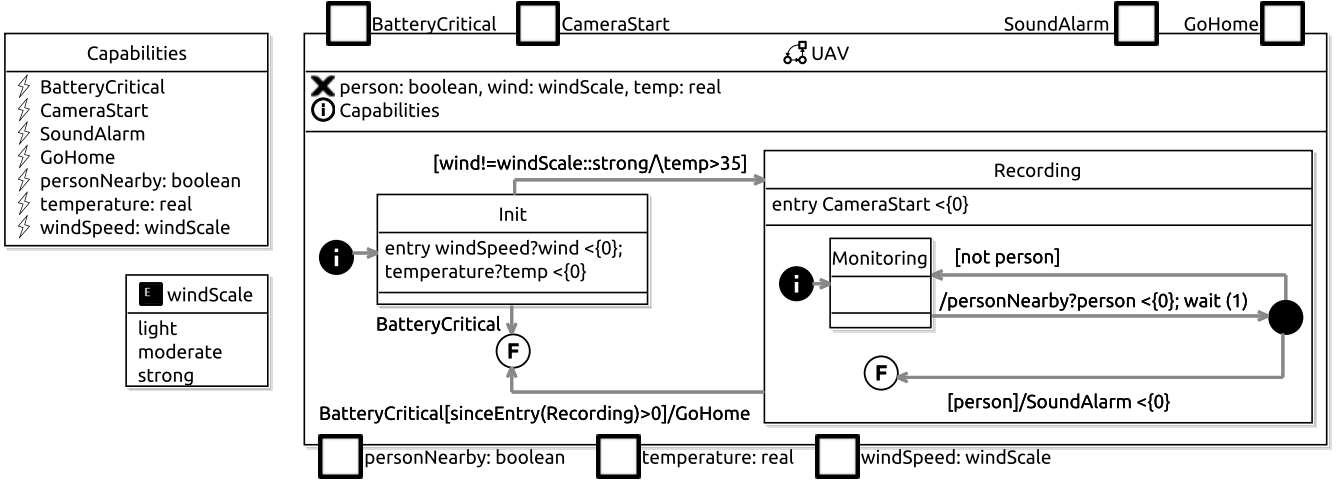


Figure 5: Sketch of RoboChart model for a simple firefighter UAV

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. $\square$

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 support 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 [27]. In particular, theorem proving is promising, and amenable to automation if we use automatically generated semantics, like that of SLEEC.

6. Evaluation: Tool support and additional case study

In this section, we present our efforts to validate our work beyond the firefighter UAV case study presented as a running example in the previous sections. In Section 6.1, we present a mechanisation of the semantics in Section 4 to support editing of SLEEC rules and automatic generation of tock-CSP scripts. The close relationship between the definition of the semantics and its mechanisation provides validation for the work via evidence that there are enough definitions, and that they produce valid tock-CSP processes. Section 6.2 presents another case study, namely, an assistive dressing robot. The SLEEC rules for this example have been developed in collaboration with SLEEC experts, and we have used our tool to validate the rules.

In addition, we have verified a design of that robot with respect to our rules.

6.1. SLEEC tool

We have implemented the SLEEC syntax (Figure 1) in Eclipse with approximately 120 lines of Xtext [28] code. The translation of SLEEC documents to tock-CSP is based on the definitions presented in Tables 2–4 and is implemented in Xtend (approximately 700 lines of code) [29], a lightweight version of Java.

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 [25]. We tested a wide range of different rule structures from Tables 2–4 specified in our SLEEC language. All code and the models are publicly available [30].

For the semantics, we translate from the trace-based definitions of conflict, redundancy, and conformance, to refinement checks via a mechanisation of *tock-CSP* [13] for verification using the CSP model-checker FDR [16]. 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.

Figure 6 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 $\{0..35\}$.

6.2. Robot Assistive Application

We present here our second case study, a robotic assistive dressing (RAD) system tasked with aiding a physically impaired user with dressing adapted from the solution presented in [31]. A secondary function of RAD is to monitor

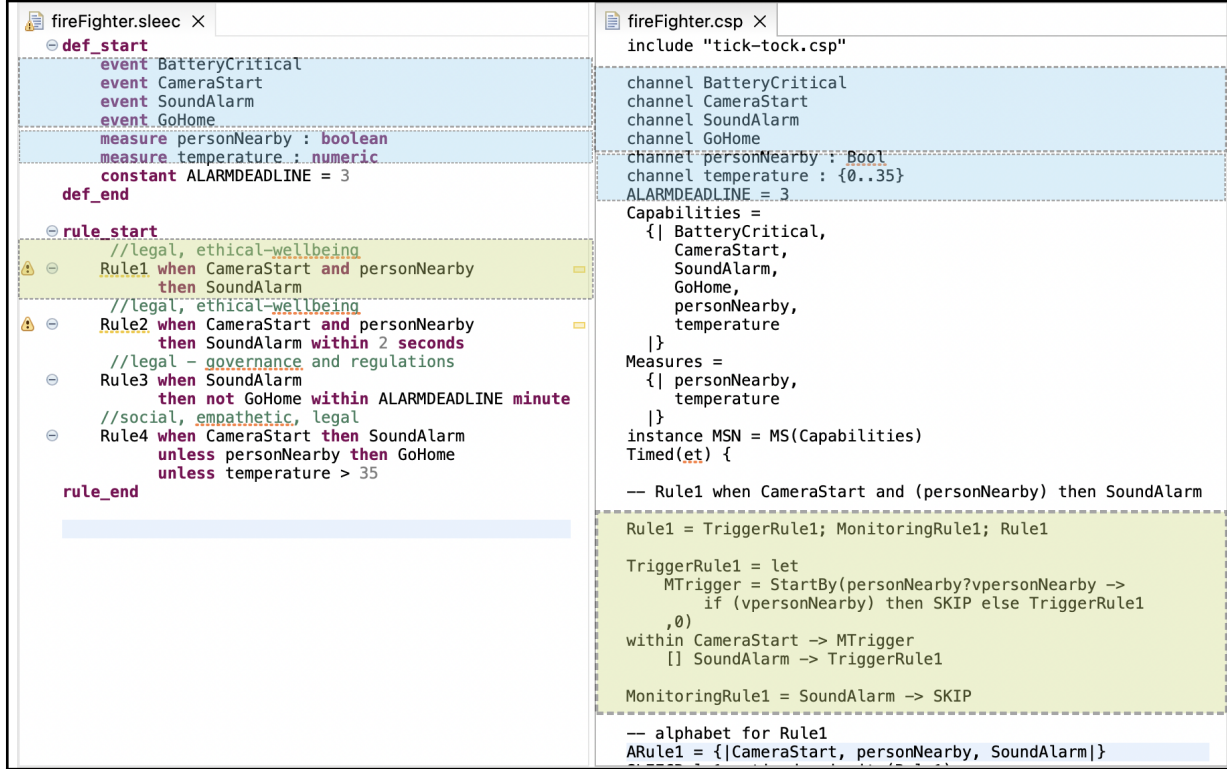


Figure 6: Tool for editing and supporting reasoning about SLEEC rules

the health of the 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 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/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 state machines relevant to the discussion here are *DressingService* and *MonitoringService*; they are depicted in Figure 7. *DressingService* specifies the software for dressing of users, and interacts with the platform via events such as *DressingStarted* and *DressingAbandoned*. *MonitoringService* mediates the opening of a room's curtains, and can call support if needed. In what follows, we describe each state machine in further detail.

In *DressingService* the state machine 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 the behaviour is given by a call to an operation *Dress* 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. In *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* is enabled and 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 that 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 under dressed nor highly distressed, then

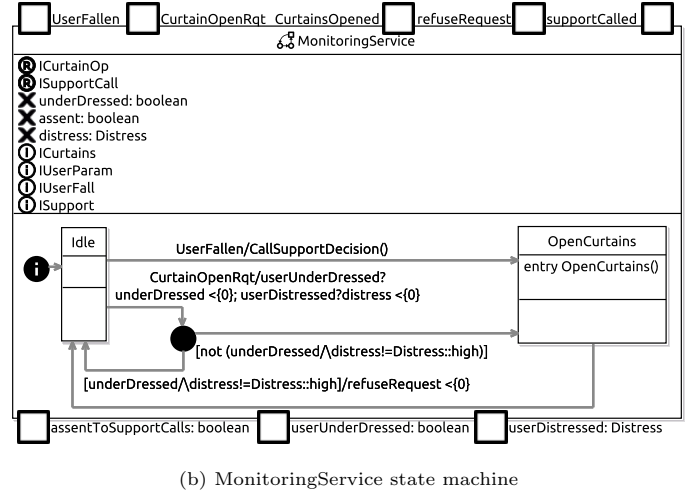
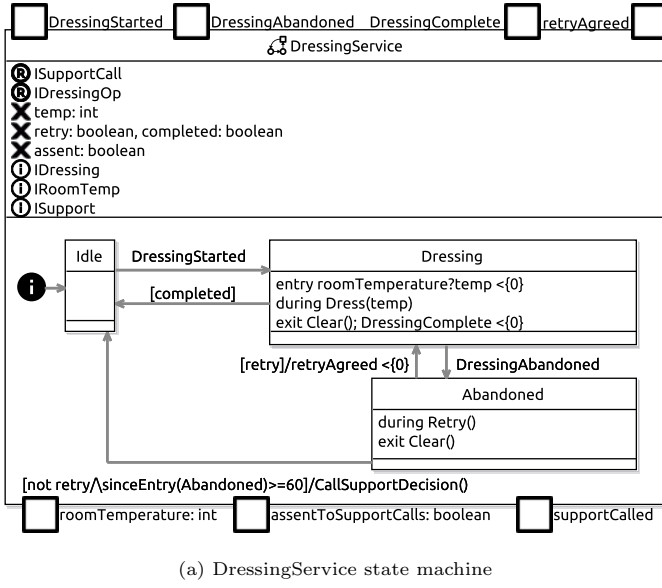


Figure 7: RoboChart state machines of the RAD application

the curtains are opened, and otherwise the request is refused as indicated by the output event **refuseRequest**.

Robotic assistive dressing raises multiple SLEEC concerns. We show in Table 6 four rules defined by experts; their motivation, in terms of SLEEC principles that should be followed and why, is indicated in the table.

Rule1 is concerned with the time taken by a dressing episode; at most 2 minutes 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. Finally, if the dressing is abandoned, there should be an attempt to retry, and, eventually the support must be called. Again, however, as required by Rule4, the user’s assent should be gained beforehand.

Only Rule3 and Rule4 have overlapping alphabets of events: both refer to **SupportCalled**. So, using our SLEEC tool from Section 6.1 we need to check them for conflict first. As there is none, we need to check whether either of them is redundant. They are not, so we can check next whether the design conforms to all rules.

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 Figure 8. In this version, the design engineer has followed an extra requirement to call the support in no more than 1 minute, if the user falls and there is consent. So, in **CallSupportDecision()** there is a deadline on **SupportCalled**, which flags the return from a call to an operation **CallSupport()** of the platform. With this deadline, we require that any actions

involved in implementing **CallSupport()**, such as establishing a phone connection and dialling, are completed within 1 minute. Since **CallSupport()** is a platform operation, it is asynchronous and does not block.

The extra requirement is incompatible with the requirement to satisfy both Rule3 and Rule4, because Rule4 requires a delay of 2 minutes before calling support in case dressing is abandoned. There is no conflict between Rule3 and Rule4, because Rule3 does not impose a deadline on calling support. The extra requirement, however, creates a conflict. The counterexample, shown below, reveals the issue. Specifically, running the FDR model checker on the tock-CSP semantics generated by our SLEEC tool produces the trace:

*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 deadline requires **SupportCalled** to take place in 60 seconds. 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 and requirements stakeholders. A few outcomes may be possible. A domain expert may agree that a one-minute deadline is too strict, and, in this case, the design may be changed. If **DressingAbandoned** happens before **SupportCalled**, RAD

Table 6: SLEEC rules for the RAD system

Rule id	SLEEC Specification	SLEEC principle	Implication
Rule1	<pre> when DressingStarted then DressingComplete within 2 minutes unless roomTemperature < 19 then DressingComplete within 90 seconds unless roomTemperature < 17 then DressingComplete within 60 seconds </pre>	empathetic ethical	promotes and supports user well-being
Rule2	<pre> when CurtainOpenRqt then CurtainsOpened within 60 seconds unless userUnderDressed then RefuseRequest within 30 seconds unless userDistressed > medium then CurtainsOpened within 60 seconds </pre>	cultural empathetic	respect for privacy and cultural sensitivity
Rule3	<pre> when UserFallen then SupportCalled unless not assentToSupportCalls </pre>	legal ethical social	respect for autonomy and preventing harm
Rule4	<pre> when DressingAbandoned then {RetryAgreed within 2 minutes otherwise {SupportCalled unless not assentToSupportCalls}} </pre>	legal ethical	promoting user beneficence and respecting autonomy

may consider whether the user is well enough to agree to a retry, in spite of having fallen. In either case, `SupportCalled` occurs after two minutes, either because the `RetryAgreed` does not happen, or as a later response to `UserFallen`.

If the solution agreed is to comply with the shorter time in the case of a fall, this can be captured by considering the RAD capabilities to call support in the case of a fall and to call support when dressing is not possible as distinct. In this case, we can have two different events representing two types of call to support. In fact, this may represent the fact that the information to be passed on to the support team in the different cases is different. We may also have different support teams to deal with a fall, and with a difficulty to get dressed.

7. Related work

A commonality in normative themes found in many recently developed artificial intelligence ethics and guidance instruments inform a ‘normative core’ of a principled approach to development, deployment, and adoption [32] [33] [34] of agents whose behaviour relies on use of artificial intelligence techniques. Significant work has been done in the development of autonomous systems from the perspective of normative ideas [35, 7] including work around transparency [36], explainability, and accountability [4]. Another research perspective develops a data-driven personalised tool, based on the moral choices of the user [9]. Our SLEEC framework, however, is concerned with the problem of the operationalisation of such norms [1] [37]

and defines a formalisation and an automated process for validating and verifying rules that capture these norms.

There exists significant research on the development and verification of autonomous systems [38][39][40]. Most of the 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 requirements of the agents.

Concerns with verification regarding some level of ethical constraints and legal aspects [41] have been recently studied [5, 6] and investigated from a verification perspective [7], although not from the perspective of operationalisation of these requirements. Robots are formally verified in [7] with an action selection of the robot controller which evaluates the outcomes of actions using simulation and prediction, and makes selection using a safety/ethical logic [5]. Bremer et al. [5] present a technique for verification of transparency and ethical concerns using the belief–desire–intention model and a simulation module to obtain ethical rules. This line of work is complementary to our SLEEC framework, as our focus is not on the identification of rules. Furthermore, unlike our SLEEC framework, these approaches do not provide a notation dedicated to the encoding of SLEEC-related concerns as requirements.

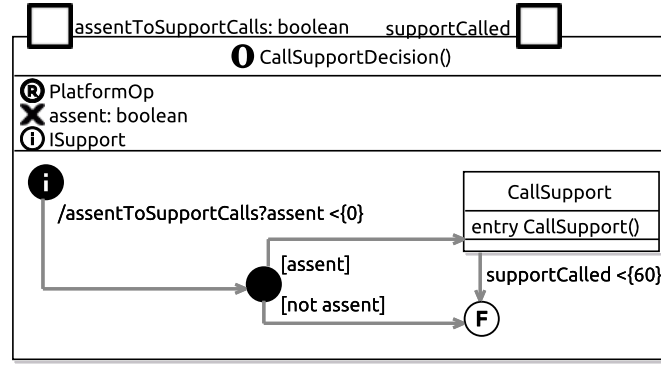


Figure 8: RoboChart model for CallSupportDecision

8. Conclusion

We introduced a tool-supported framework for the end-to-end specification, consistency validation and verification of social, legal, ethical, empathetic and cultural requirements for autonomous agents. The framework supports the specification of these requirements as SLEEC rules formalised in a domain-specific language grounded in defeasible logic [11, 12], and their translation into tock-CSP for redundancy and conflict checking, and for verifying autonomous agent compliance with SLEEC requirements. By enabling the operationalisation of SLEEC requirements for autonomous agents, our framework complements the significant international efforts to define ethical principles for AI and autonomous systems, e.g. [33, 34, 42], and our own recent work to elicit SLEEC requirements for autonomous agents by starting from relevant normative principles and stakeholder needs [1].

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 will improve the integration of the tools used by the framework, e.g., through automating the invocation of the FDR model checker used to verify autonomous agent compliance with SLEEC requirements. Third, we intend to augment our tool support with a module that converts the counterexample traces produced by the model checker into error messages that are easier to understand for framework users who do not have formal methods expertise. Finally, we plan to continue to evaluate the SLEEC framework in additional case studies from different application domains, and with a larger number of SLEEC experts (lawyers, ethicists, domain experts, regulators, sociologists) in order to identify and fix any remaining applicability and usability issues, and to assess the scalability and generalisability of the framework further.

In the longer term, we will consider also extending the framework with two additional methods. The former

method is needed for 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 latter method will further support this evolution by focusing on the online synthesis of SLEEC-compliant adaptation plans for autonomous agents.

Acknowledgements

The work described in this paper was funded by the EPSRC project EP/V026747/1 ‘UKRI Trustworthy Autonomous Systems Node in Resilience’. The work of Ana Cavalcanti is funded by the Royal Academy of Engineering grant CiET1718/45 and ‘UKRI TAS Verifiability Node EP/V026801/1’. The work of Pedro Ribeiro is funded by EPSRC RoboTest EP/R025479/1.

References

- [1] B. Townsend, C. Paterson, T. T. Arvind, G. Nemirovsky, R. Calinescu, A. Cavalcanti, I. Habli, A. Thomas, From pluralistic normative principles to autonomous-agent rules, *Minds and Machines* 32 (4) (2022) 683–715. doi:10.1007/s11023-022-09614-w.
- [2] J. M. Wing, Trustworthy AI, *Communications of the ACM* 64 (10) (2021) 64–71.
- [3] J. H. Moor, The nature, importance, and difficulty of machine ethics, *IEEE intelligent systems* 21 (4) (2006) 18–21.
- [4] P. Inverardi, The Challenge of Human Dignity in the Era of Autonomous Systems, Springer International Publishing, Cham, 2022, Ch. 4, pp. 25–29. doi:10.1007/978-3-030-86144-5_4.
- [5] P. Bremner, L. Dennis, M. Fisher, A. Winfield, On proactive, transparent, and verifiable ethical reasoning for robots, *Proceedings of the IEEE* 107 (2019) 1–21. doi:10.1109/JPROC.2019.2898267.
- [6] L. Dennis, M. Fisher, M. Slavkovik, M. Webster, Formal verification of ethical choices in autonomous systems, *Robotics and Autonomous Systems* 77 (2016) 1–14. doi:https://doi.org/10.1016/j.robot.2015.11.012
- [7] L. Dennis, M. Fisher, A. F. T. Winfield, Towards verifiably ethical robot behaviour, *ArXiv abs/1504.03592* (2015).
- [8] N. Boltz, L. Sterz, C. Gerking, O. Raabe, A model-based framework for simplified collaboration of legal and software experts in data protection assessments, in: *INFORMATIK, Lecture Notes in Informatics, Gesellschaft fuer Informatik*. Ed.: D. Demmler, Vol. P-326, 2022, pp. 521–532. doi:10.18420/inf2022_44.

- [9] C. Alfieri, P. Inverardi, P. Migliarini, M. Palmiero, Exosoul: Ethical profiling in the digital world, in: S. Schlobach, M. Perez-Ortiz, M. Tielman (Eds.), HHAI, Vol. 354 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2022, pp. 128–142.
- [10] L. Floridi, Soft ethics and the governance of the digital, *Philosophy & Technology* 31 (2018) 1–8. doi:10.1007/s13347-018-0303-9.
- [11] J. F. Horty, *Reasons as defaults*, Oxford University Press, 2012.
- [12] E. N. Zalta, U. Nodelman, C. Allen, R. L. Anderson, Defeasible reasoning, *Stanford encyclopedia of philosophy*, Palo Alto CA: Stanford University press (2005).
- [13] J. Baxter, P. Ribeiro, A. Cavalcanti, Sound reasoning in tock-CSP, *Acta Informatica* 59 (1) (2022) 125–162. doi:10.1007/s00236-020-00394-3.
- [14] C. A. R. Hoare, Communicating sequential processes, *Commun. ACM* 21 (8) (1978) 666–677, doi:10.1145/359576.359585.
- [15] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, J. Woodcock, RoboChart: Modelling and verification of the functional behaviour of robotic applications, *Software & Systems Modeling* 18 (2019) 1–53. doi:10.1007/s10270-018-00710-z.
- [16] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, A. Roscoe, FDR3 – A Modern Refinement Checker for CSP, in: E. Abraham, K. Havelund (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 8413 of *Lecture Notes in Computer Science*, 2014, pp. 187–201.
- [17] O. Alon, S. Rabinovich, C. Fyodorov, J. R. Cauchard, Drones in firefighting: A user-centered design perspective, in: 23rd International Conference on Mobile Human-Computer Interaction, 2021, pp. 1–11.
- [18] A. Cervantes, P. Garcia, C. Herrera, E. Morales, F. Tarriba, E. Tena, H. Ponce, A conceptual design of a firefighter drone, in: 15th International Conference on Electrical Engineering, Computing Science and Automatic Control, IEEE, 2018, pp. 1–5.
- [19] M. S. Innocente, P. Grasso, Self-organising swarms of firefighting drones: Harnessing the power of collective intelligence in decentralised multi-robot systems, *Journal of Computational Science* 34 (2019) 80–101.
- [20] J. Brunero, Reasons and Defeasible Reasoning, *The Philosophical Quarterly* 72 (1) (2021) 41–64. doi:10.1093/pq/pqab013
- [21] A. W. Roscoe, *The Theory and Practice of Concurrency*, Prentice-Hall Series in Computer Science, Prentice-Hall, 1998.
- [22] A. J. R. G. Milner, *Calculus for Synchrony and Asynchrony*, *Theoretical Computer Science* 25 (1983) 267–310.
- [23] R. Milner, *Communicating and Mobile Systems: the π -calculus*, Cambridge University Press, 1999.
- [24] J. A. Bergstra, J. W. Klop, Algebra of communicating processes with abstraction, *Theoretical Computer Science* 37 (1985) 77–121.
- [25] S. Getir Yaman, C. Burholt, M. Jones, R. Calinescu, A. Cavalcanti, Specification and validation of normative rules for autonomous agents, in: 26th International Conference on Fundamental Approaches to Software Engineering, Springer-Verlag, 2023, pp. 241–248. doi:10.1007/978-3-031-30826-0_13.
- [26] A. W. Roscoe, The automated verification of timewise refinement, in: *First Open EIT ICT Labs Workshop on Cyber-Physical Systems Engineering*, 2013.
- [27] A. Cavalcanti, W. Barnett, J. Baxter, G. Carvalho, M. C. Filho, A. Miyazawa, P. Ribeiro, A. Sampaio, *RoboStar Technology: A Robotist’s Toolbox for Combined Proof, Simulation, and Testing*, Springer International Publishing, 2021, Ch. 9, pp. 249–293. doi:10.1007/978-3-030-66494-7_9.
- [28] Eclipse XText, <https://www.eclipse.org/Xtext/>, Accessed: 2023-07-5 (2010).
- [29] Eclipse XTend, <https://www.eclipse.org/xtend/>, Accessed: 2023-07-5 (2010).
- [30] SLEECVAL GitHub repository (2023). URL <https://github.com/SLEEC-project/SLEEC>.
- [31] A. Camilleri, S. Dogramadzi, P. Caleb-Solly, A study on the effects of cognitive overloading and distractions on human movement during robot-assisted dressing, *Frontiers in Robotics and AI* 9 (May 2022).
- [32] A. Jobin, M. Ienca, E. Vayena, The global landscape of AI ethics guidelines, *Nature Machine Intelligence* 1 (9) (2019) 389–399.
- [33] UNESCO, Recommendation on the Ethics of Artificial Intelligence, <https://unesdoc.unesco.org/ark:/48223/pf0000380455>, Accessed: 2022-03-18, Document code: SHS/BIO/REC-AIETHICS/2021 (2021).
- [34] OECD, Recommendation of the Council on Artificial Intelligence, OECD/LEGAL/0449, <http://legalinstruments.oecd.org>, Accessed: 2023-03-27 (2022).
- [35] J. Fjeld, N. Achten, H. Hilligoss, A. Nagy, M. Srikumar, Principled artificial intelligence: Mapping consensus in ethical and rights-based approaches to principles for AI, Berkman Klein Center Research Publication 2020-1 (2020).
- [36] A. Winfield, E. Watson, T. Egawa, E. Barwell, I. Barclay, S. Booth, L. A. Dennis, H. Hastie, A. Hossaini, N. Jacobs, M. Markovic, R. I. Muttram, L. Nadel, I. Naja, J. Olszewska, F. Rajabiyazdi, R. K. Rannow, A. Theodorou, M. A. Underwood, O. von Stryk, R. H. Wortham, IEEE standard for transparency of autonomous systems (2022). doi:10.1109/IEEESTD.2022.9726144
- [37] P. Solanki, J. Grundy, W. Hussain, Operationalising ethics in artificial intelligence for healthcare: a framework for AI developers, *AI Ethics* 3 (1) (2023) 223–240. doi:10.1007/s43681-022-00195-z.
- [38] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, M. Fisher, Formal specification and verification of autonomous robotic systems: A survey, *ACM Comput. Surv.* 52 (5) (sep 2019). doi:10.1145/3342355.
- [39] A. Nordmann, N. Hochgeschwender, S. Wrede, A survey on domain-specific languages in robotics, in: D. Brügali, J. F. Broenink, T. Kroeger, B. A. MacDonald (Eds.), *Simulation, Modeling, and Programming for Autonomous Robots*, Springer International Publishing, 2014, pp. 195–206.
- [40] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, T. Berger, Specification patterns for robotic missions, *IEEE Transactions on Software Engineering* 47 (2019) 2208–2224.
- [41] H. Bhuiyan, F. Olivieri, G. Governatori, M. B. Islam, A. Bond, A. Rakotonirainy, A methodology for encoding regulatory rules, in: *MIREL@JURIX*, 2020, pp. 1–13.
- [42] IEEE Global Initiative for Ethics of Autonomous and Intelligent Systems, Ethically aligned design: A vision for prioritizing well-being with autonomous systems and intelligent systems. version 2 (2019). URL https://standards.ieee.org/wp-content/uploads/import/documents/other/ead_v2.pdf 20