The SLEEC Framework for Normative Requirements Engineering

Pedro Ribeiro[1], Radu Calinescu[1], Ana Cavalcanti[1], Marsha Chechik[2], Sinem Getir Yaman[1], Lina Marsso[3], Isobel Standen[4], and Beverley Townsend[5]

[1] Department of Computer Science, University of York, UK
{pedro.ribeiro,radu.calinescu,ana.cavalcanti,sinem.getir.yaman}@york.ac.uk
[2] University of Toronto, Toronto, Canada
marsha.chechik@cs.toronto.edu
[3] Polytechnique Montréal, Montréal, Canada
lina.marsso@polymtl.ca
[4] Department of Philosophy, University of York, UK
isobel.standen@york.ac.uk
[5] York Law School, University of York, UK
bev.townsend@york.ac.uk

Abstract. Autonomous agents are increasingly deployed in sensitive, human-centric domains—such as healthcare, assistive care, and emergency response—where their decision-making must align with complex human norms. These translate into Social, Legal, Ethical, Empathetic, and Cultural (SLEEC) requirements that are often nuanced and context-dependent, challenging traditional software engineering paradigms. Our tutorial paper presents a comprehensive, tool-supported methodology for managing the SLEEC requirements lifecycle, covering elicitation, well-formedness validation, and conformance verification of software design models against SLEEC requirements. We demonstrate the use of our methodology and associated tools through application to a robot-assisted dressing system, providing a guide for researchers and engineers to bridge the gap between abstract human norms and verifiable system designs.

Keywords: Normative requirements · Autonomous agents · Formal verification and validation · SLEEC

1 Introduction

Rapid technological advances, particularly in Artificial Intelligence (AI), enable the development of applications offering significant societal and economic benefits. However, the deployment of the increasingly sophisticated AI-enabled systems underpinning many of these applications—ranging from cyber-physical to agentic systems—comes with significant normative implications of a social, legal, ethical, empathetic, and cultural (SLEEC) nature [4, 16].

The importance and complexity of these implications are particularly evident in domains where AI-enabled systems interact with vulnerable users, such as health and assistive care [21]. An assistive-dressing robot [26], for instance,

must be aware of and navigate complex normative trade-offs, e.g., to ensure both user privacy by closing curtains during dressing, and user autonomy if asked to keep them open by an anxious user. In healthcare, chatbots provide medical advice and machine learning (ML) solutions perform diagnostics, raising concerns regarding accountability and fairness. Beyond care, collaborative robots assist human operators with dangerous manufacturing tasks, and self-driving cars navigate with varying levels of autonomy in public spaces. Each case poses SLEEC challenges; for example, the collaborative robot must balance operational efficiency with the psychological wellbeing of its human partner, while autonomous vehicles must make ethically charged decisions in uncertain environments.

There is wide awareness of the need to address these normative concerns, reflected in numerous international initiatives aimed at ensuring the responsible development and deployment of AI-enabled systems. These efforts span a broad spectrum, from high-level ethical frameworks and principles, such as the UNESCO Recommendation on the Ethics of AI [25], to more concrete standards and legislative instruments like the IEEE 7000 series [15] and the EU AI Act [6]. Nevertheless, a significant gap exists between these useful but general normative documents, which are discussing application-independent issues, and the definition of a concrete set of well-formed, unambiguous and verifiable normative requirements for a concrete AI-enabled system [20, 23]. Furthermore, verifying that a system's design actually complies with such requirements remains a challenge for its developers and regulators [1, 12, 14].

Our tutorial paper presents an integrated suite of tool-supported methods designed to address this gap. It has been developed over almost a decade via an international collaboration [7, 9–12, 17, 22, 23], and has already sparked the interest of multiple research groups. In presenting this tutorial, we benefit from the experience of having presented tutorial sessions on SLEEC requirements engineering across several major conferences in Software Engineering.

The paper is organised as follows. In Section 2, we introduce our definition of normative requirements, and the rule-based language we use to capture them, which has a mathematical semantics [12]. Section 3 describes our collaborative process for eliciting well-formed SLEEC rules, which is based on formal verification tools, yet accessible for use by teams of multi-disciplinary stakeholders. In Section 4, we illustrate our notion of conformance used for the formal verification of designs against SLEEC rules. We conclude with a brief summary and provide pointers to additional material on the SLEEC framework in Section 5.

2 The SLEEC paradigm

2.1 Normative requirements

In this tutorial, we distinguish between high-level normative principles (such as autonomy, non-maleficence, and privacy) and the specific normative requirements (i.e., *SLEEC rules*) that operationalise them. While principles provide abstract evaluative standards covered extensively by existing international guidelines and regulations [6, 15, 25], SLEEC rules formulate the explicit constraints

```
def_start // Capabilities
  event CurtainOpenRqt // Triggers and responses
  event CurtainsOpened
  event DressingAbandoned
  event DressingComplete
  event DressingStarted
  event HealthChecked
  event RefuseRequest
  event RetryAgreed
  event SupportCalled
  event UserFallen
  measure assentToSupportCalls : boolean // Measures
  measure emergency : boolean
  measure roomTemperature : numeric
  measure userUnderDressed : boolean
  measure userDistressed : scale(low, medium, high)
  constant MAX_RESPONSE_TIME = 60 // Constants
def_end
rule_start
 Rule1 when CurtainOpenRqt then CurtainsOpened within 60 seconds
       unless userUnderDressed then RefuseRequest within 30 seconds
       unless userDistressed > medium then CurtainsOpened within 60 seconds

 Rule2 when DressingStarted and userUnderDressed
            then DressingComplete within 2 minutes
       unless roomTemperature < 19 then DressingComplete within 90 seconds
       unless roomTemperature < 17 then DressingComplete within 60 seconds

 Rule3 when UserFallen then HealthChecked within 30 seconds
       otherwise SupportCalled within MAX_RESPONSE_TIME seconds

 Rule4 when DressingAbandoned then RetryAgreed within 2 minutes
       otherwise { SupportCalled unless not assentToSupportCalls }
rule_end
```

Listing 1.1: Sample SLEEC specification for the robot-assisted dressing system

that an autonomous agent must satisfy to comply with these principles within a given context—and are significantly more difficult to elicit, formalise, and verify.

In our methodology, SLEEC rules are treated as non-functional requirements that augment an agent's functional specifications. Functional requirements define what an agent should achieve, whereas SLEEC rules constrain the admissible state space and action selection process. Specifically, these rules limit an agent's degrees of freedom by filtering out plans that violate normative constraints, ensuring the agent executes a normatively compliant plan from the possible alternatives [23]. In the next section, we describe the language used to specify these rules as formally verifiable properties.

2.2 The SLEEC language

The SLEEC language, introduced in [12], is a domain-specific language (DSL) for specifying normative requirements for autonomous agents. Co-developed by an interdisciplinary team of computer scientists, engineers, ethicists, lawyers,

roboticists, and social scientists, the language supports the creation of SLEEC specifications comprising two parts: a definition block, and a rule block.

Definition block. This block declares the capabilities of the agent that have normative relevance because they:

1. raise normative concerns, leading to normative requirements;
2. provide information about the status of the world to support decisions;
3. provide facilities for the agent to react in a norm-sensitive way.

For example, in the robot assistive dressing scenario considered in our paper, the robot's capability to start dressing a user raises a modesty concern. The robot also has a capability to measure room temperature to ensure that decisions regarding when to undress the user consider their (thermal) comfort. Finally, the robot has a capability to close the curtains, so that it can act to protect the user's modesty when asked to start dressing. The process presented in the next section describes how we can iteratively elicit rules based on available and suggested capabilities.

Listing 1.1 illustrates these concepts in a SLEEC specification, showing how agent capabilities are represented by either `event`s or `measure`s. For instance, the `event UserFallen` corresponds to the agent detecting that its user has fallen, while the `measure userDistressed` corresponds to the robot's capability to estimate a user's distress level on a discrete `scale` (`low`, `medium` or `high`). Together, these two constructs are used to provide the *vocabulary* for the rule block, allowing normative rules to be defined over the agent's capabilities.

Rule block. This block contains the SLEEC rules for the autonomous agent, specified over the `event`s and `measure`s defined in the previous block. These rules encode the normative requirements of the agent by mapping environmental triggers to required agent responses. A SLEEC rule has the basic form

id `when` **triggerEvent** [`and` **triggerGuard**] `then` **response** [`within` **time**]

where:

- **id** is a unique rule identifier;
- **triggerEvent** is an `event` that must be monitored by the agent;
- **triggerGuard** is an optional Boolean expression over the agent's `measure`s;
- **response** is an `event` specifying the action that the agent shall perform when the **triggerEvent** occurs and, if specified, the **triggerGuard** is `true`;
- if provided, **time** specifies a deadline for the **response** to happen.

For example, `Rule1` from Listing 1.1 applies when the event `CurtainOpenRqt` occurs, and `Rule2` applies when the event `DressingStarted` occurs and, additionally, the Boolean measure `userUnderDressed` is `true`.

To accommodate situations where a **response** may be infeasible within the required time, the `otherwise` construct can be used to specify an alternative response. In `Rule3`, the **response** requires the occurrence of the `event HealthChecked` in 30 seconds, but provides an alternative to have `SupportCalled` if there is a timeout.

Importantly, a rule can be followed by one or more *defeaters* [5], introduced by the `unless` keyword, and specifying circumstances that preempt the original **response** and may provide an alternative. The general form of a defeater is

`unless` **defeaterGuard** [`then` **defeaterResponse**]

where:

- **defeaterGuard** is a Boolean expression over the agent's `measure`s;
- if provided, **defeaterResponse** represents an `event` specifying an alternative action for the agent.

If a SLEEC rule includes defeaters, and any **defeaterGuard** from its defeaters is `true`, then—instead of responding as specified by the base rule **response**—the agent shall act as specified by the outermost defeater whose **defeaterGuard** holds, performing its **defeaterResponse** if present, or ignoring the rule altogether otherwise.

The use of defeaters is illustrated by two rules from the SLEEC specification in Listing 1.1. In `Rule1`, the first `unless` specifies a defeater that preempts the response `CurtainsOpened` if the `measure` `userUnderDressed` is `true`, while a second defeater takes precedence over both the base-rule response *and* the first defeater if the `measure` `userDistressed` is greater than `medium` (i.e., `high`). Additionally, in `Rule4`, curly brackets are used to specify a defeater that applies to the alternative response of `SupportCalled` defined by the `otherwise` construct.

The elicitation of SLEEC rules is supported by a family of tools [7, 11] that provide validation, first through parsing and type-checking, and, second, through the ability to identify and diagnose conflicts and redundancies between rules by exploiting the formal semantics of SLEEC as described later in Section 3. Fig. 1 depicts one of these tools, SLEEC-TK, with a subset of rules from Listing 1.1 being edited in a file named `tutorial.sleec` whose content is validated as the user modifies the SLEEC specification. In this case, an error is shown in the pane at the bottom, indicating that `CurtainsOpenRqt` is mistyped. Related, syntactic elements can be clicked on to look up their definition, as indicated by the hand cursor and the highlighted capability `CurtainsOpened`. Instructions on how to install SLEEC-TK are available online[6], with a set of example projects that can be imported in the tool available from its repository[7].

SLEEC semantics. Rules are given semantics using two distinct, but complementary, formal approaches, to support detection of conflicts, redundancies, and other well-formedness issues, as well as conformance verification of a design against a set of SLEEC rules. In the first approach, the behaviour of SLEEC rules is captured using the process algebra *tock*-CSP [3], a discrete-timed variant of CSP [19] supported by the model checker FDR [13] for formal verification. In the second approach, the semantics are enhanced using SAT-based reasoning grounded in first-order logic with relational objects (FOL*) [8], providing more comprehensive checks for requirements validation, including various types of conflicts, situational conflicts, redundancy, restrictiveness, and insufficiency [7].

[6] https://github.com/UoY-RoboStar/SLEEC-TK
[7] https://github.com/UoY-RoboStar/SLEEC-TK/tree/main/examples

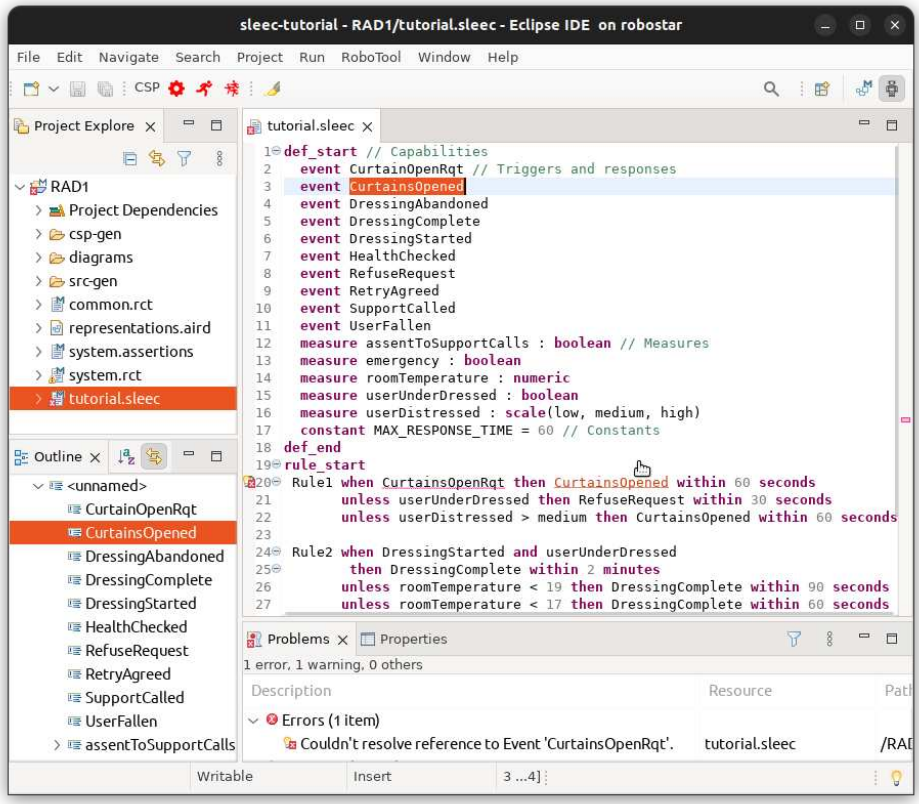


Fig. 1: SLEEC-TK screenshot showing a subset of the SLEEC rules from Listing 1.1 being edited within the file `tutorial.sleec`, which is opened in the top-right pane. The bottom-righ pane summarises syntactic and type errors.

3 Eliciting and validating SLEEC rules

As illustrated in Fig. 2 our SLEEC framework employs a three-stage normative requirements engineering workflow. In stage (i), a team of *SLEEC experts* comprising multi-disciplinary stakeholders, identify normative principles, capabilities and measures relevant for the autonomous agent under development—all captured in a SLEEC document. Through the use of our framework's tools (e.g., see Fig. 1), the SLEEC experts ensure that the rules are correctly formulated, that is, they are syntactically correct and use the available capabilities as specified. The elicitation stage includes consulting a broad range of agent stakeholders, as well as normative documents, such as regulations and standards.

In stage (ii), rules are validated to identify any conflicts, redundancies and other well-formedness issues using the two complementary approaches mentioned earlier, with feedback provided to SLEEC experts for iterative revision of the proposed rules, until a set of valid rules is obtained. In Section 3.1, we describe

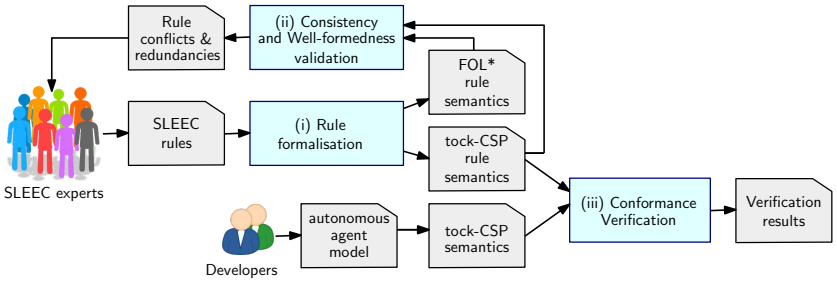


Fig. 2: Elicitation, validation and verification workflow of normative requirements

our framework's tool-supported techniques for pair-wise validation of rules based on their *tock*-CSP semantics [12], while in Section 3.2 we present our FOL*-based techniques and tool for detecting other well-formedness issues across a set of rules. Finally, once a valid set of SLEEC rules is obtained, the autonomous agent model provided by its developers is verified for compliance with the SLEEC rules in stage (iii). This verification process is detailed in Section 4.

3.1 Pair-wise consistency validation

As mentioned, the second stage in our workflow involves validating SLEEC rules to identify any pair of rules that are conflicting or redundant. To enable this validation, each SLEEC rule's semantics is formally defined as a CSP process [19] that specifies a pattern of interaction over CSP events capturing reading of measures, events (i.e., rule triggers and responses), and the passage of time as marked by *tock*. These patterns can be characterised mathematically as a set of traces, that is, finite sequences of CSP events.

The semantics of a SLEEC rule encompasses two phases. First, there is a monitoring phase before a trigger occurs, where events used in the responses of a rule are unconstrained, followed by a response phase, if any responses are applicable. The response phase is followed again by the monitoring phase. So, following a trigger event, the measures used in the expression of the trigger and defeaters, if any, are read immediately. Then, the trigger's expression, if it exists, is evaluated. For example, the following *CSP trace* (i.e., sequence of interactions including SLEEC events, readings of measures, and the passage of discrete time as captured by `tock` events) can be observed for `Rule1` in Listing 1.1:

Trace 1 for `Rule1`

```
CurtainOpenRqt, userUnderDressed.false, userDistressed.medium, tock,
CurtainsOpened
```

In this scenario, the trigger event `CurtainOpenRqt` is observed, followed by the reading of the measures `userUnderDressed` and `userDistressed`, with values `false` and `medium`, respectively, indicating that neither *defeater* applies, followed by one `tock` event indicating the passage of one second, and finally the response event `CurtainsOpened` is observed. A complete account of the *tock*-CSP semantics of SLEEC specifications can be found in [12].

```
rule_start
 RuleA when CurtainOpenRqt then CurtainsOpened within 2 minutes
       unless userUnderDressed then RefuseRequest within 3 seconds
       unless userDistressed > medium then CurtainsOpened within 6 seconds

 RuleB when CurtainOpenRqt and userUnderDressed then
       not CurtainsOpened within 60 seconds

 RuleC when CurtainOpenRqt and userUnderDressed then
       not CurtainsOpened within 120 seconds
rule_end
```

Listing 1.2: Example of a set of rules with conflicts and redundancies.

Given this semantics, conflicts arise for rules whose restrictions cannot be jointly satisfied, whereas a redundancy indicates that a rule can be discarded as the restrictions it imposes are covered by another rule. We define both notions more precisely in what follows and illustrate how our tools can be used to identify and diagnose them.

Conflicts. Two rules are conflicting if there is a scenario where both apply but impose restrictions that cannot be simultaneously satisfied. For example, both `RuleA` and `RuleB` from Listing 1.2 have the event `CurtainOpenRqt` as a trigger. Consider the scenario where a user asks for the curtains to be opened, is underdressed, and is exhibiting a `high` distress level. In these circumstances, there is a conflict between the requirement to open the curtains `within 6 seconds` (`RuleA`), as the user is highly distressed, and to not open them for `60 seconds` (`RuleB`) for modesty reasons. Stakeholders can be guided by our tools to automatically identify such inconsistencies and revise their rules accordingly.

Checking with SLEEC-TK. The first tool at our disposal is SLEEC-TK [12], which implements the calculation of the *tock*-CSP [3] semantics for SLEEC. Checking for conflicts between `RuleA` and `RuleB` of Listing 1.2 in SLEEC-TK invokes model checking with FDR in the background, and results in the following counterexample being generated as a CSP trace.

Trace 2 showing conflict between `RuleA` and `RuleB`

```
CurtainOpenRqt, userUnderDressed.true, userUnderDressed.true,
userDistressed.high, tock, tock, tock, tock, tock, tock
```

This trace specifies a scenario where after the occurrence of the trigger event `CurtainOpenRqt`, the measures `userUnderDressed` and `userDistressed` are read, with values `true` and `high`, respectively, followed by the passage of 6 seconds, as indicated by the six `tock`s, leading to a conflict. We observe that measures are read more than once with the same value in a conflicting trace, allowing for independent reading of measures by two rules. The trace suggests that resolving the conflict requires relaxing the deadline of `RuleA` or adding defeaters to `RuleB`.

Process-algebraic notion of conflict. In what follows, we explain how conflicts can be diagnosed for a pair of rules based on their traces. For example, the following trace can be observed for `RuleB`:

Trace 3 for RuleB

```
CurtainOpenRqt, userUnderDressed.false, CurtainOpenRqt
```

Here, the measure `userUnderDressed` is read as being `false` after the triggering event `CurtainOpenRqt`, the response does not apply, and so the rule can be triggered again. On the other hand, if the trigger's expression evaluates to true, or there is none, the measures associated with defeaters, if any, are evaluated following the reverse order in which they are written. For example, for `RuleA`, the expression `userDistressed > medium` is evaluated first, and only when it is false is the expression for the next defeater (`unless userUnderDressed`) evaluated. So, the following trace can be observed for `RuleA`, noting that both measures are read at the same time, though in an order unrelated to that of evaluation:

Trace 4 for RuleA

```
CurtainOpenRqt, userUnderDressed.true, userDistressed.low, RefuseRequest
```

In this scenario, the second defeater (`unless userDistressed > medium`) does not apply, but the first defeater (`unless userUnderDressed`) does, and so the response `RefuseRequest` can be observed afterwards, with a maximum of 3 `tock`s in between corresponding to the imposed deadline (`within 3 seconds`).

Our notion of conflict for a pair of rules r1 and r2, adapted from [12] is recast below in terms of traces, where a joint trace captures the behaviour of two rules, with triggers and responses performed in agreement and the value of measures read being the same.

Definition 1. *Rules* r1 *and* r2 *are conflict free, if, and only if, for every joint trace* t_1 *of* r1 *and* r2*, there is a joint trace* t_2 *of* r1 *and* r2 *after* t_1 *that contains at least one event and at least one event different from tock.*

In other words, conflict-freedom requires that for every joint trace t_1, there is an extension t_2 of t_1 that can make timed progress for both rules, namely by having at least one event and a non-*tock* event. This ensures that enforcing both rules does never lead to a deadlock (where no further events are possible), or to a scenario in which there is no deadlock, but only passage of time can be observed. The latter is a *timed deadlock*, where time can progress via *tock*, but no other event is possible.

The mechanisation of our notion of conflict [12] relies on composing the *tock*-CSP semantics of the two rules under examination in parallel, synchronising on their common alphabets, that encompass trigger and response events, while ensuring uniform passage of time by also synchronising on *tock*s. Without loss of generality, this composition is defined in a context which ensures that the values of measures do not change over time, and that the values of measures read by both rules are the same, though they can be read independently, i.e., in a different order. A conflict arises if, and only if, the composition has a (timed) deadlock, with a counter-example trace identifying a scenario where no further joint progress is possible. Our tool automatically produces assertions for identifying the presence of such deadlocks using the CSP model checker FDR [13].

Redundancies. A pair of non-conflicting rules that have overlapping alphabets may be redundant. Informally, a rule is redundant if all restrictions it imposes are covered by another rule. For example, `RuleB` and `RuleC` from Listing 1.2 have the same alphabet and are not conflicting; however, `RuleB` imposes a restriction on `not CurtainsOpened` for `60 seconds`, which is weaker than the restriction imposed by `RuleC` for a longer period of `120 seconds`, so `RuleB` is redundant with respect to `RuleC`. Therefore, `RuleB` can be removed from the set of normative requirements, as an implementation that satisfies `RuleC` will also satisfy `RuleB`.

Checking with SLEEC-TK. Checking for redundancies with SLEEC-TK is performed via model checking with FDR. For each pair of SLEEC rules `r1` and `r2` with overlapping alphabets, the tool considers checks for redundancies in both directions, that is, whether `r1` is redundant with respect to `r2` and vice-versa.

Unlike for conflict checking, a counterexample trace is only produced when a rule is not redundant, corresponding to a scenario not constrained by the other rule being checked. For example, in the case of `RuleB` and `RuleC` in Listing 1.2, checking whether `RuleC` is redundant with respect to `RuleB` produces the trace:

Trace 5 showing redundancy of `RuleC` with respec to `RuleB`

```
userUnderDressed.true, CurtainOpenRqt, ...
(userUnderDressed.false, tock)^60 ..., CurtainOpenRqt
```

This trace corresponds to a scenario where the measure `userUnderDressed` is initially read as `true`, the trigger `CurtainOpenRqt` happens, followed by 60 occurrences of reading `userUnderDressed` as `false` and `tock`, with the trigger happening again afterwards. Here, the trace indicates that `RuleB` may apply again `60 seconds` after being triggered, yet `RuleC` continues to require that the response `CurtainsOpened` should not happen for the full `120 seconds` from the initial trigger. On the other hand, checking whether `RuleB` is redundant with respect to `RuleC` produces no trace, indicating that the restrictions imposed by `RuleB` are fully covered by `RuleC`. Next, we elaborate on the formal notion of redundancies based on traces.

Process-algebraic notion of redundancies. Our definition of redundancy [12] considers a pair of conflict-free rules and is also based on their traces. It is reproduced below, where $t \restriction E$ denotes the trace obtained from t by removing all events that are not in the set E, and $\alpha_{\mathsf{E}}(\mathsf{r})$ is the alphabet of the rule r, that is, the set of CSP events capturing the rule's trigger and responses.

Definition 2. *For conflict-free rules* $\mathsf{r1}$ *and* $\mathsf{r2}$*, rule* $\mathsf{r2}$ *is redundant with respect to* $\mathsf{r1}$ *if, and only if, for every trace* t_1 *of* $\mathsf{r1}$*, there is a trace* t_2 *of* $\mathsf{r2}$*, such that, (1)* $t_1 \restriction \alpha_{\mathsf{E}}(\mathsf{r}_1) = t_2 \restriction \alpha_{\mathsf{E}}(\mathsf{r}_1)$*, and (2) for every event* e *of* $\alpha_{\mathsf{E}}(\mathsf{r}_1)$ *in a position* i *of these traces, for every measure* m *in* $\alpha_{\mathsf{M}}(\mathsf{r}_1)$*, the value of* m *recorded in* t_1 *and* t_2 *at position* i *are the same.*

In words, we characterise a rule $\mathsf{r2}$ as redundant if every trace of $\mathsf{r1}$, with events restricted to triggers and responses of just $\mathsf{r1}$, via $t_1 \restriction \alpha_{\mathsf{E}}(\mathsf{r}_1)$, can also be performed by $\mathsf{r2}$ when restricted to the same events (1), and the traces agree on the value

of measures recorded in relation to a trigger or response event (2). This is so that reading of measures can be performed in different orders, but their values must be consistent when comparing two rules for redundancy. In summary, if every behaviour allowed by r1 is also allowed by r2, then r2 imposes no additional restrictions, and so is redundant.

The mechanisation of redundancy checking is stated in terms of traces refinement [19] between the *tock*-CSP semantics of a pair of rules, r1 and r2, in a context where all measures used by either rule are read, once at the beginning of every trace, and after every *tock*, before any other events, encoding condition (2). Similarly, the context also encodes condition (1) by ensuring that events not in $\alpha_{\mathsf{E}}(\mathsf{r}_2)$ are unrestricted when comparing the refinement of r1's semantics in context against that of r2. This is equivalent to the formulation in Definition 2.

In summary, the traces of the process that characterises a rule identify the behaviours allowed by the rule. So the smaller that set of traces, as captured by the notion of traces refinement, the more restrictive is that rule. This concludes our discussion of pair-wise consistency validation of rules. In what follows we discuss set-wise validation of rules.

3.2 Set-wise well-formedness validation

Pairwise conflict and redundancy checks provide important guarantees, but they do not capture all *well-formedness issues* (WFIs) that arise from the *combined* effect of multiple rules. To support the identification of these additional WFIs, our SLEEC framework includes a second validation tool, LEGOS-SLEEC, whose functionality augments that of SLEEC-TK.

LEGOS-SLEEC[8] enables *set-wise verification* of SLEEC rulesets [7], allowing stakeholders to reason about the global behaviour induced by a set of normative rules. Specifically, the tool supports the analysis of five classes of WFIs: (i) *vacuous conflicts*, where a rule can never be triggered without violating the ruleset; (ii) *situational conflicts*, where inconsistencies arise only under specific contextual conditions; (iii) *set-wise redundancies*, where a rule does not restrict behaviour beyond what is already enforced; (iv) *insufficiencies*, where normatively undesirable behaviours remain possible despite the rules; and (v) cases of *over-restrictiveness*, where desirable or intended behaviours are inadvertently ruled out. Together, these checks allow stakeholders to assess whether a SLEEC ruleset is coherent, sufficiently protective, and aligned with its intended purpose.

For each WFI, LEGOS-SLEEC provides: (i) a formal check at the level of the full ruleset, and (ii) an explanation either as a witness trace (when the property is satisfiable), or as a causal unsatisfiability proof (when the property is impossible).

Trace semantics (rule sets). LEGOS-SLEEC adopts a trace-based semantics. Given a SLEEC specification over a set of events E and a set of measures M, a

[8] The LEGOS-SLEEC tool and instructions for its installation are available online at https://zenodo.org/records/14714867.

Fig. 3: Examples of traces for the ALMI robot.

trace represents a *finite* sequence of *states*

$$\sigma = (\mathcal{E}_1, \mathbb{M}_1, \delta_1), (\mathcal{E}_2, \mathbb{M}_2, \delta_2), \ldots (\mathcal{E}_n, \mathbb{M}_n, \delta_n),$$

where, for every time point $i \in [1, n]$

1. $\mathcal{E}_i \in E$ is a set of events that occur at time point i;
2. $\mathbb{M}_i : M \to \mathbb{N}$ assigns every measure in M to a concrete value at time point i (with appropriate mapping of Boolean measures to $\{0, 1\}$, scale measures with three values to $\{0, 1, 2\}$, etc.), and
3. $\delta_i : \mathbb{N}$ captures the time value of time point i (e.g., the second time point can have the value 30, for 30 seconds).

We assume that the time values in the trace are strictly increasing (i.e., $\delta_i < \delta_{i+1}$ for every $i \in [1, n-1]$). Given a measure assignment $\mathbb{M}_i$ and a term t, let $\mathbb{M}_i(t)$ denote the result of substituting every measure symbol m in t with $\mathbb{M}_i(m)$. Since $\mathbb{M}_i(t)$ does not contain free variables, the substitution results in a natural number. Similarly, given a proposition p, we say that $\mathbb{M}_i \models p$ if p is evaluated to $\top$ after substituting every term t with $\mathbb{M}_i(t)$, where $\top$ stands for Boolean true and $\bot$ for false.

Example. Consider the trace σ_1 shown in Fig. 3, corresponding to $(\mathcal{E}_1, \mathbb{M}_1, \delta_1)$, where $\mathcal{E}_1 = \texttt{HumanOnFloor}$, $\mathbb{M}_1(\texttt{humanAssents}) = \top$, and $\delta_1 = 1$. While the trace σ_2 shown in Fig. 3, corresponds to $(\mathcal{E}_1, \mathbb{M}_1, \delta_1), (\mathcal{E}_2, \mathbb{M}_2, \delta_2)$, where $\mathcal{E}_1 = \texttt{HumanOnFloor}$, $\mathbb{M}_1(\texttt{humanAssents}) = \bot$, $\delta_1 = 1$, $\mathcal{E}_2 = \texttt{CallEmergencyServices}$, $\mathbb{M}_2(\texttt{humanAssents}) = \bot$, and $\delta_2 = 200$.

Trace fulfilment. A trace σ *fulfils* an obligation by evaluating whether the required event occurrence (or non-occurrence) holds within the designated time frame starting from the triggering time point.

- A *positive obligation* ("*e within t*") is fulfilled if the specified event e occurs within the designated time window of length t from the starting time point.
- A *negative obligation* ("*not e within t*") is fulfilled if the specified event e does *not* occur at any time throughout the designated time window.
- A *conditional obligation* ("$p \Rightarrow ob$") is fulfilled if either the precondition p does not hold at the triggering time point, or the subsequent obligation ob is fulfilled.
- An *obligation chain* ("cob_1^+ *otherwise* $cob_2^+ \ldots$ cob_m") is fulfilled if the first obligation is fulfilled; otherwise, if it is violated, the subsequent obligation in the chain must be fulfilled from the violation point onward.

A rule r is fulfilled if, *whenever its conditions hold at any time point*, the associated obligation (or obligation chain) is fulfilled from that time point onward.

Example. Consider the negative obligation

```
then not CallEmergencyServices within 600 seconds
```

of rule R1 in Listing 1.3. We analyze the traces σ_1, σ_2, and σ_3 shown in Fig. 3. In σ_2 and σ_3, the triggering event HumanOnFloor occurs at $\delta_1 = 1$ and the measure humanAssents is $\perp$, thus activating the obligation. In σ_2, CallEmergencyServices occurs at $\delta_2 = 200$. Since $200 - 1 = 199 \leq 600$, the event occurs within the forbidden 600-second window; hence, the negative obligation is *violated*. In σ_3, CallEmergencyServices occurs at $\delta_2 = 610$. Since $610 - 1 = 609 > 600$, the event occurs outside the 600-second window; therefore, the negative obligation is *fulfilled*.

Rule set fulfilment. A trace σ fulfils a ruleset Rules if it fulfils every rule in the set. A SLEEC ruleset Rules induces a language of accepted behaviours, denoted $L(\mathsf{Rules})$, defined as: $L(\mathsf{Rules}) = \{\, \sigma \mid \forall\, r \in \mathsf{Rules},\ \sigma \models r \,\}$. That is, $L(\mathsf{Rules})$ is the largest set of traces such that every trace in the set respects every rule in Rules. For example, if $\mathsf{Rules} = \{R_1, R_2, R_3\}$, then: $L(\mathsf{Rules}) = L(R_1) \cap L(R_2) \cap L(R_3)$. This set-based view enables uniform definitions of all well-formedness issues (WFI) in terms of emptiness or non-emptiness of language intersections, and supports diagnostics in the form of concrete witness traces (counter-examples) or unsatisfiability explanations. In the following, we present each class of well-formedness issue.

Vacuous conflicts. A rule is *vacuously conflicting* when triggering it in any feasible situation necessarily leads to a violation of the ruleset. Intuitively, such a rule is unachievable, that is, its trigger can never be realised without contradicting other rules.

Trace-based definition. Let Rules be a ruleset and let r be a rule in Rules with trigger $Trig(r)$. The rule r is vacuously conflicting (w.r.t. Rules) if no trace in $L(\mathsf{Rules})$ contains an occurrence of its trigger under satisfied trigger conditions. Equivalently, $L(\mathsf{Rules}) \cap L(Trig(r)) = \emptyset$, where $L(Trig(r))$ denotes traces in which $Trig(r)$ can occur with its associated measures satisfying the trigger predicate.

Example. Let Rules consist of the rules in Listing 1.3. Rules R6 and R7 together imply that whenever SmokeDetectorAlarm occurs, CallEmergencyServices must occur within 7 minutes (2 + 5 minutes). This follows *transitively*: R6 requires that SmokeDetectorAlarm triggers NotifyCaregiver within 2 minutes, and the occurrence of NotifyCaregiver in turn triggers R7, which requires the response CallEmergencyServices within 5 additional minutes. Thus, the response of one rule (R6) activates the trigger of another (R7), propagating the obligation. However, R8 prohibits the response CallEmergencyServices within 10 minutes after SmokeDetectorAlarm. Since 7 minutes $\leq$ 10 minutes, any trace that triggers SmokeDetectorAlarm necessarily violates either R7 or R8. Therefore, no trace in $L(\mathsf{Rules})$ can contain a feasible occurrence of SmokeDetectorAlarm satisfying these rules. Hence, R6 (and symmetrically R8) is vacuously conflicting, as $L(\mathsf{Rules}) \cap L(Trig(\mathsf{R6})) = \emptyset$.

```
rule_start
 R1 when HumanOnFloor and not humanAssents then
    not CallEmergencyServices within 600 seconds

 R2 when OpenCurtainRequest and (not underDressed) then OpenCurtain within 30
     seconds

 R3 when SmokeDetectorAlarm then CallEmergencyServices within 300 seconds

 R4 when DressingStarted and (roomTemperature < MIN_TEMP)
    and userUnderDressed then DressingComplete within 1 minute

 R5 when SmokeDetectorAlarm then CallEmergencyServices within 7 minutes

 R6 when SmokeDetectorAlarm then NotifyCaregiver within 2 minutes

 R7 when NotifyCaregiver then CallEmergencyServices within 5 minutes

 R8 when SmokeDetectorAlarm then not CallEmergencyServices within 10 minutes
rule_end
```

Listing 1.3: Example of a set of rules with well-formedness issues.

Checking with LEGOS-SLEEC. The tool checks whether $L(\mathsf{Rules}) \cap L(Trig(r))$ is empty and, if so, produces a causal unsatisfiability proof pinpointing why the trigger cannot occur without violating the ruleset, as shown below.

Diagnostic for Vacuous Conflicting SLEEC Rules

Conflicting SLEEC rule:
R8: when SmokeDetectorAlarm then not CallEmergencyServices within 10 min.

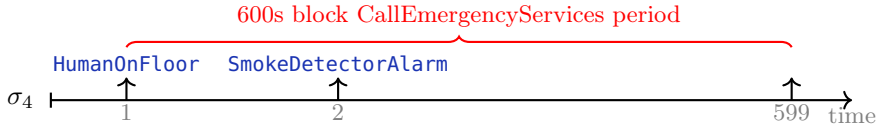
Because of the following SLEEC rules:
R6: when SmokeDetectorAlarm then NotifyCaregiver within 2 min.
R7: when NotifyCaregiver then CallEmergencyServices within 5 min.

Situational conflicts. A *situational conflict* exists when there is *at least one feasible situation* (trace prefix) in which the ruleset can be triggered into an inconsistent state, i.e., a history of events and measures, always leads to a conflict with some other rules in the future. Unlike vacuous conflicts, these are *witnessed* by a feasible trace.

Trace-based definition. A ruleset Rules has a situational conflict if there exists a trace σ that is consistent with the ruleset up to some point, but cannot be extended to a full trace in $L(\mathsf{Rules})$ once certain triggers occur under certain measure valuations. Operationally, LEGOS-SLEEC reports a witness trace demonstrating a feasible path to inconsistency.

Example. Consider the rules `R1` and `R3` shown in Listing 1.3. `R1` conflicts with `R3` in a situation where the event `HumanOnFloor` occurs without human consent for calling emergency services (`not humanAssents`), while `SmokeDetectorAlarm`

is also triggered. This results in R3 blocking the response of R1, as illustrated below in trace σ_4.



Checking with LEGOS-SLEEC. The tool produces a concrete witness trace (scenario) that exhibits the conflict (shown below), which can be used as guidance during repair (e.g., adding defeaters, introducing priorities, or refining triggers).

Diagnostic for Situational Conflict

Situational conflict under situation:
at time 1: HumanOnFloor
at time 1: Measure (humanAssents = false)
at time 2: SmokeDetectorAlarm
at time 2: blocked_CallEmergencyServices

Conflicting SLEEC rule:
R3: when SmokeDetectorAlarm then CallEmergencyServices within 300 sec.

Because of the following SLEEC rule:
R1: when HumanOnFloor and not humanAssents then
not CallEmergencyServices within 600 sec.

Set-wise redundancy. A rule is *set-wise redundant* if removing it does not change what the ruleset allows the system to do; that is, it imposes no additional restriction beyond the remaining rules.

Trace-based definition. Given Rules and a rule $r \in$ Rules, the rule r is set-wise redundant if $L(\text{Rules}) = L(\text{Rules} \setminus \{r\})$. Equivalently, $L(\text{Rules}) \subseteq L(\text{Rules} \setminus \{r\})$ always holds (by monotonicity), and redundancy holds when also $L(\text{Rules}\setminus\{r\}) \subseteq L(\text{Rules})$.

Example. Consider rules R5, R6, and R7. Together, R6 and R7 imply that whenever SmokeDetectorAlarm occurs, CallEmergencyServices must occur within 7 minutes. Rule R5 explicitly requires that when SmokeDetectorAlarm occurs, CallEmergencyServices must occur within 7 minutes. Since this obligation is already enforced by the combination of R6 and R7, adding R5 does not further restrict the accepted behaviour of the ruleset. Hence, R5 is redundant, that is: $L(\{\text{R6},\text{R7},\text{R5}\}) = L(\{\text{R6},\text{R7}\})$.

Checking with LEGOS-SLEEC. LEGOS-SLEEC checks redundancy at the ruleset level and provides an explanation (reproduced below) showing why the removed rule does not exclude any additional traces beyond those already excluded by the remaining rules.

```
concern_start
 C1 when SmokeDetectorAlarm and ((not userDisablesAlarm) or alarmRestarts) then
      not CallEmergencyServices within 1 minute
concern_end
purpose_start
 P1 when HumanOnFloor and (userUnconscious and (not humanAssents)) then
     CallEmergencyServices within 5 minutes
purpose_end
```

Listing 1.4: Example of SLEEC concerns and purpose.

Diagnostic for Set-wise Redundant SLEEC Rules

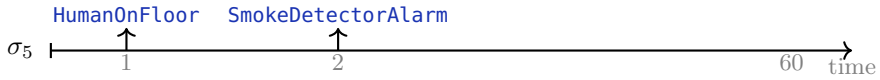
Redundant SLEEC rule:
R3 when SmokeDetectorAlarm then CallEmergencyServices within 300 seconds

Because of the following SLEEC rules:
R6: when SmokeDetectorAlarm then NotifyCaregiver within 2 minutes
R7: when NotifyCaregiver then CallEmergencyServices within 5 minutes

Insufficiency. A set of rules are insufficient if adhering to those rules can still allow executing an undesirable behavior,5which we call *concern*. Concerns capture undesirable behaviours that must be avoided (e.g., hazards, violations, or harms). Concerns are specified in a SLEEC-like form (such as `C1` shown in Listing 1.4), and are treated as sets of traces $L(c)$ representing bad behaviours.

Trace-based definition. A ruleset Rules is *insufficient* for preventing a concern c iff $L(\mathsf{Rules}) \cap L(c) \neq \emptyset$. That is, there exists at least one feasible trace that satisfies all rules yet realises the undesirable behaviour.

Example. Consider the concern `C1` shown in Listing 1.4. `C1` can arise even when all the rules in Listing 1.3 are respected, as illustrated below in σ_5.



Checking with LEGOS-SLEEC. The tool searches for a witness trace satisfying both the ruleset and the concern. The returned trace is a concrete diagnostic scenario that guides strengthening the ruleset (e.g., adding missing obligations or refining defeaters).

Diagnostic for Insufficiency

Concern raised:
C1 when SmokeDetectorAlarm and ((not userDisablesAlarm) or alarmRestarts) then not CallEmergencyServices within 1 minute
under the situation:
at time 1: HumanOnFloor
at time 1: Measure (humanAssents = false, userDisablesAlarm = false, alarmRestarts = false)
at time 2: SmokeDetectorAlarm
at time 2: Measure (userDisablesAlarm = false, alarmRestarts = false)

Over-restrictiveness A ruleset is *overly restrictive* if it prevents the realisation of desirable or intended behaviours, even though these behaviours are consistent with the system's purpose. Purposes capture functional goals or intended outcomes of the system and are specified (such as `P1` shown in Listing 1.4), similarly to concerns, as sets of traces $L(p)$ representing desirable behaviours.

Trace-based definition. A ruleset Rules is *overly restrictive* with respect to a purpose p iff $L(\mathsf{Rules}) \cap L(p) = \emptyset$. That is, no feasible trace satisfying all rules realises the intended behaviour.

Example. Consider a system purpose `P1` shown in Listing 1.4, stating that when a person is detected on the floor and is unconscious without explicitly assenting, emergency services must be called within 5 minutes. However, consider the ruleset shown in Listing 1.3, which contains `R1`, a rule prohibiting calling emergency services within 600 seconds whenever a person is on the floor and has not assented. Since 600 seconds corresponds to 10 minutes, this prohibition directly blocks the intended 5-minute call in all feasible traces. As a result, the ruleset becomes overly restrictive with respect to this purpose.

Checking with LEGOS-SLEEC. LEGOS-SLEEC checks whether the intersection $L(\mathsf{Rules}) \cap L(p)$ is empty. If so, the tool produces a causal unsatisfiability explanation identifying the subset of rules responsible for blocking the intended behaviour, thereby supporting rule relaxation or refinement.

Diagnostic for Restrictive SLEEC Rules

Blocked system purpose:
P1 when HumanOnFloor and (userUnconscious and (not humanAssents))
then CallEmergencyServices within 5 minutes

Because of the following SLEEC rules:
R1 when HumanOnFloor and not humanAssents then
not CallEmergencyServices within 600 seconds

4 Conformance verification of design models

The implementation of a SLEEC-aware system must conform to the relevant SLEEC rules. We now show, via an example, what is required of a conforming implementation. In Section 4.1, we present a design for an assistive-dressing robot described in a diagrammatic domain-specific language called RoboChart [18]. This is a language for defining control software for robotic applications. Via the example, we illustrate the different roles of events and measures in considering conformance. Section 4.2 describes how we can use our tool to check that a RoboChart design satisfies a set of SLEEC rules. Finally, in Section 4.3, we present our mathematical notion of conformance based on CSP refinement.

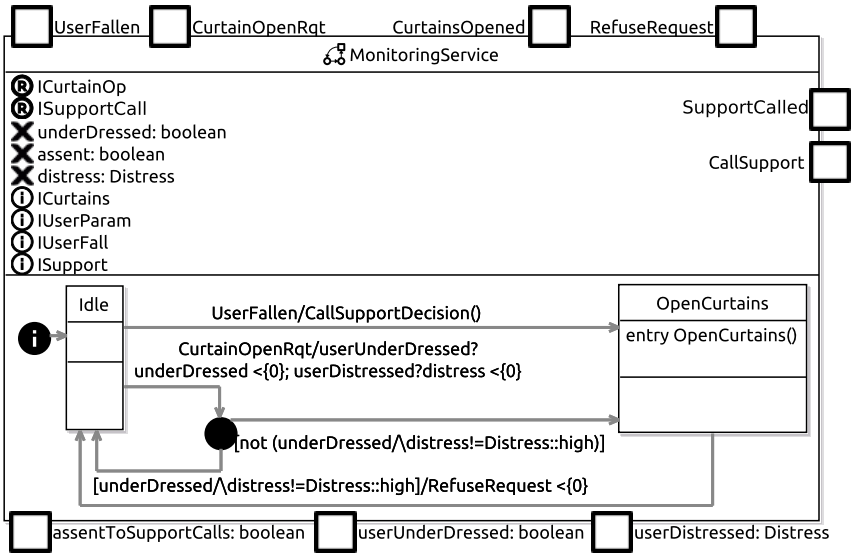


Fig. 4: MonitoringService state machine for the assistive-dressing robot

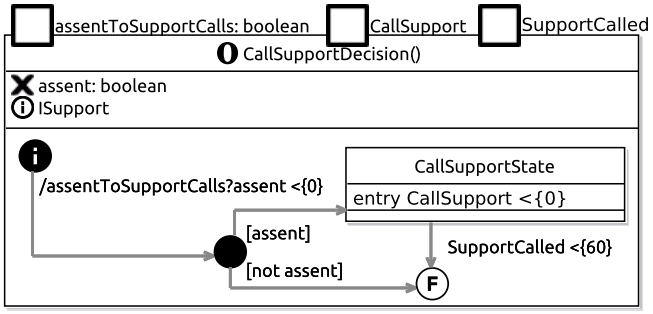


Fig. 5: CallSupportDecision operation

4.1 Design

A RoboChart model defines behaviour based on a collection of (parallel) state machines and neural network components grouped to define a RoboChart component called module. For our example, we present in Figs. 4, 5, and 6 three RoboChart state machines for part of the software of an assistive-dressing robot. The machine in Fig. 4, called MonitoringService, defines a monitor that accepts requests to open the curtains (via a capability CurtainOpenRqt) and UserFallen signals. Around the box of the machine, all capabilities used in this agent are indicated by small squares. At the top half of the box, above the actual machine, a context of declarations defines these capabilities and local variables.

At the start, indicated by the black circle with an i inside, MonitoringService is in the state Idle. In that state, two outgoing transitions define behaviour when

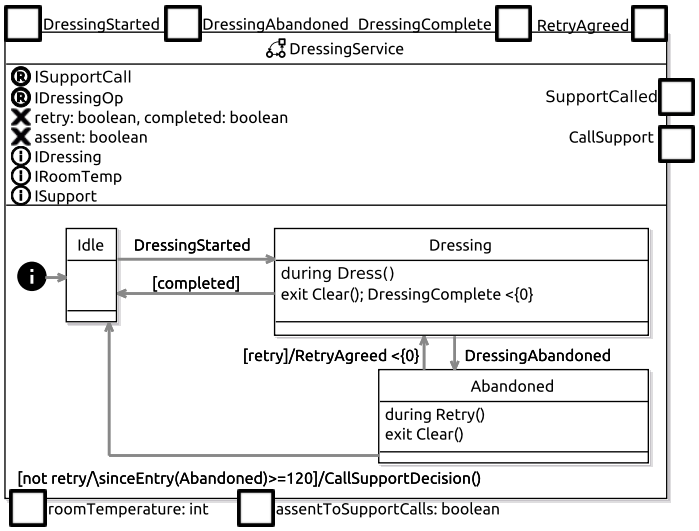


Fig. 6: DressingService state machine for the assistive-dressing robot

a CurtainOpenRqt and when UserFallen is signalled. For UserFallen, the outgoing transition calls a software operation CallSupportDecision(). This operation is defined by another state machine presented in Fig. 5.

In CallSupportDecision(), at the start, the measure assentToSupportCalls is read and the result is recorded in the variable assent. With <{0}, we indicate that the reading of the measure is immediate, since all measures are assumed to be always readily available. A junction, represented by a black circle, defines behaviour when there is assent and when there is not (not assent). If there is, CallSupportDecision() moves to the CallSupport state. If there is not, CallSupportDecision() moves to the final state: white circle with an F.

In CallSupport, the entry action raises the event CallSupport immediately. That capability establishes a communication with the support team, with an input via supportCalled indicating when the contact is successful, before CallSupportDecision() terminates. The deadline of 60 time units (seconds) requires that any actions involved in implementing the CallSupport capabilities, such as establishing a phone connection and dialling, are completed within that time.

Finally, the state machine DressingService in Fig. 6 deals with the interactions between the human and the robot during dressing. In the Idle state, a request to start the dressing via the capability represented by an event DressingStarted is accepted. At that point, DressingService moves to a state Dressing, where the actual dressing is carried out via an operation Dressing A result can be signalled via a variable completed, when DressingService goes back to Idle, of via an event DressingAbandoned, when DressingService transitions to the state Abandoned. Upon exitting of Dressing, a Clear() operation resets the robot and signals that the dressing is completed, via the event DressingComplete.

In Abandoned, the operation Retry() negotiates an attempt to try again. Success is recorded in a variable retry and signalled via RetryAgreed. If that does not happen, and more than 120 time units (corresponding to two minutes) have gone by, CallSupportDecision() is called, before DressingService moves back to Idle.

RoboChart has a tool, called RoboTool[9], which supports modelling via editing and validation facilities. RoboTool also includes support for verification that RoboChart models conforms to a SLEEC rule, with such functionality also available as part of SLEEC-TK. In the next sections, we present this tool and the technique that it implements. The complete RoboChart model for our robot is available in our repository of supplementary material[10].

4.2 Verification

Using our technique, we get confirmation that Rule1 in Listing 1.1, for instance, is satisfied by our design of assistive-dressing robot. Fig. 7 shows a screenshot of RoboTool showing the result of the verification.

At the top of Fig. 7, we show the RoboTool editing panel, where the state machine MonitoringService in Fig. 4 is visible. This is the part of our RoboChart model that enforces `Rule1`. Underneath, the editing panel we show a few lines of a file highlighted on the left panel. It is called rad.assertions, and defines a document using a simple controlled natural language to specify verification checks of SLEEC rules. The first check shown is introduced by a clause `timed sleec assertion`. This clause gives a name to the check, A1, and then the name of the RoboChart package, RAD, and module, Software, defining a design for the control software of our robot. MonitoringService is part of that module. Finally the assertion states that `RAD::Software conforms to Rule1`. We recall that `Rule1` is in the file tutorial.sleec that is in the Eclipse project.

At the bottom of Fig. 7, we show the report that RoboTool generates after checking this assertion using the model checker FDR [13]. At the rightmost column, we see that the result of the verification is true. As said, `Rule1` is enforced by MonitoringService. As explained in the previous section, in its initial state, MonitoringService accepts a CurtainOpenRqt, and then checks the measures userUnderDressed and userDistressed. As dictated by the defeaters of `Rule1`, the results of those checks lead to a call to a software operation OpenCurtains() which effectively opens the curtains, or to a RefuseRequest event. That refusal event happens immediately. Since we reading of measures is also immediate, and calculation of boolean expressions also do not take time, the deadline for a RefuseRequest given in `Rule1` is satisfied. Satisfaction of the other deadlines is ensured by the definition of the operation OpenCurtains() omitted here.

The report in Fig. 7 also shows that `Rule4` is not satisfied. As shown in Fig. 4, the design engineer has specified a behaviour in which support is called in no more than 1 minute, if the user falls and there is consent. Even though this is a reasonable course of action, it means that `Rule4` is not satisfied at least in the scenario described by the trace below, which our tool indicates as possible.

[9] https://robostar.cs.york.ac.uk/robotool/
[10] https://github.com/UoY-RoboStar/SLEEC-TK

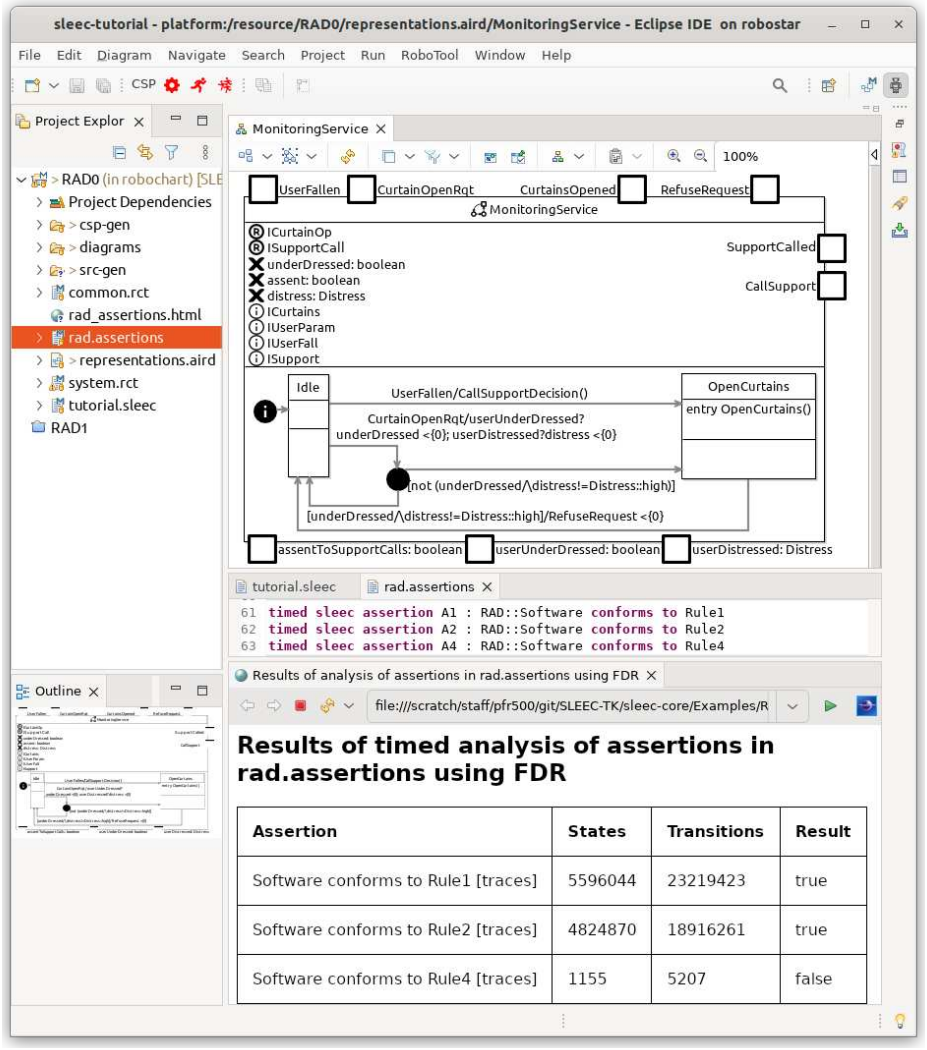


Fig. 7: Verification of `Rule1`, `Rule2` and `Rule4` in Listing 1.1 using RoboTool

Trace 6 as counter-example of non-conformance against `Rule4`

```
assentToSupportCalls.true, DressingStarted.in, UserFallen.in,
DressingAbandoned, SupportCalled
```

The trace indicates that there is assentToSupportCalls, the dressing process has started (DressingStarted), the user has fallen (UserFallen), and the dressing has been abandoned (DressingAbandoned). The issue is that `Rule4` dictates that when dressing is abandoned, there should be a period of 2 minutes in which a protocol for agreeing on a new attempt (RetryAgreed) should take place. The defined

design, however, allows for an immediate call to support, as indicated by the final event SupportCalled in the trace above, because the user has fallen.

Using our tool, we can get these scenario descriptions, which help the identification of at least one situation in which a rule is violated. This may lead to changes in the design, or to revision of rules and other requirements.

In Fig. 7, we note that Rule2 is also satisfied. This rule has two defeaters (see Listing 1.1), both based on the roomTemperature. The motivation is that the temperature should inform how fast the dressing process needs to proceed. The designer, however, has decided to meet Rule2 by always carrying out the dressing as fast as possible. So, Rule2 requires use of the measure roomTemperature to decide on a response, but a correct design does not need to take that measure.

In this case, taking the measure is not needed because there is a response that is satisfactory in all cases predicted in the rule, and that response in chosen. In other situations, it might be because the value of the measure is already available. For example, if our robot were to control the room temperature, it might take that measure routinely, and not need to take it again when the dressing is started.

In the next section, we explain our definition of conformance, which describes formally the set of designs that are correct.

4.3 Conformance

Conformance of a given System Under Verification (SUV) against a SLEEC rule is considered only when a rule is "relevant", that is, the rule only refers to events representing capabilities of the SUV. This is to ensure that the rule is applicable, that is, it can be triggered and respond using capabilities available in the SUV.

Our definition of conformance for an SUV with respect to a SLEEC rule assumes that the SUV can be given a semantics as a *tock*-CSP process. For the RoboChart machines in Figs. 4, 5, and 6, this can be automatically calculated using RoboTool so that conformance can be established by model checking with FDR. It is based on the notion of traces refinement for *tock*-CSP, that ensures that events of the SUV occur in the order and time specified by the rules, respecting time budgets and deadlines. Our notion of conformance assumes that the value of measures do not change in a time unit.

The formal definition is reproduced [12] below, where $[\![\mathsf{r}]\!]_{\mathsf{R}}$ is the *tock*-CSP semantics for a rule r, $\alpha_{\mathsf{E}}(\mathsf{r})$ is the alphabet of the rule, that is, the set of CSP events capturing the rule's trigger and responses, and $\alpha_{\mathsf{M}}(\mathsf{r})$ is the set of measures used by the rule.

Definition 3. *An SUV conforms to a rule* r*, written* $\mathsf{r} \models SUV$*, where SUV is the tock-CSP semantics of SUV, if, and only if, for every trace* t_1 *of the process* $SUV; Stop$*, there is a trace* t_2 *of* $[\![\mathsf{r}]\!]_{\mathsf{R}}$*, such that: (1)* $t_1 \upharpoonright \alpha_{\mathsf{E}}(\mathsf{r}) = t_2 \upharpoonright \alpha_{\mathsf{E}}(\mathsf{r})$ *and; (2) for every event* e *of* $\alpha_{\mathsf{E}}(\mathsf{r})$ *in a position* i *of these traces, for every measure* m *in* $\alpha_{\mathsf{M}}(\mathsf{r})$*, the value of* m *recorded in* t_1 *and* t_2 *at position* i *are the same.*

In words, it corresponds to traces refinement in *tock*-CSP, where the specification is defined in terms of the process $[\![\mathsf{r}]\!]_{\mathsf{R}}$ that defines the semantics of a rule r. Similarly to the check for redundancy, refinement disregards the measures,

however, we require that the values of the measures recorded in the specification and the SUV to be the same.

In the definition above, we consider the CSP process $SUV; Stop$, rather than just SUV, as the process that gives semantics to a rule is non-terminating, whereas the SUV could terminate. Therefore, to avoid flagging a problem with termination, SUV is sequentially composed (;) with the *tock*-CSP process *Stop* that deadlocks, but allows passage of time.

The filtering (1) in our definition of conformance allows an SUV to engage in additional events and read additional measures, as the comparison is based on the alphabet of events $\alpha_{\mathsf{E}}(\mathsf{r})$ and measures $\alpha_{\mathsf{M}}(\mathsf{r})$ of a rule. The mechanisation of conformance adopts a similar pattern to that of redundancy (Definition 2), introducing a context on both CSP processes in the refinement.

5 Conclusion

SLEEC [23], its language [12], its semantics [8,12], and its techniques and tools [7, 11] provide a powerful basis to study, design, and verify SLEEC properties of autonomous agents. This tutorial paper provides a starting point to understand the in-depth published work on elicitation and operationalisation of SLEEC principles, concerns, and requirements (rules).

As explained here, elicitation of normative requirements, however, is not solely a task for computer scientists. It potentially requires the involvement of lawyers, psychologists, sociologists, philosophers, domain experts, and users, or their representatives. To develop the SLEEC framework, we have collaborated within a multi-disciplinary team to define a notation and devise techniques that are accessible across all these disciplines (therefore, catering for non-technical stakeholders) and across domains. SLEEC provides a common language and structure to discussions, with tool-supported validation and verification based on (counter)examples providing valuable input, and feedback on intermediate versions of a SLEEC ruleset and/or autonomous agent design.

The SLEEC tools are freely available for use by the academic community. SLEEC-TK [11] is released open source under the Eclipse Public Licence (EPL-2.0), while LEGOS-SLEEC [7] is licenced under CC-BY-A.

The SLEEC paradigm has attracted the attention of the community worldwide. Besides our contributions to academic and industrial conferences, other researchers have taken up the work. For example, in [24], the authors have studied SLEEC in the context of Prolog implementations, and the project presented in [2] proposes a fuzzy-logic representation for SLEEC rules. In South Korea, there is an effort to use SLEEC to capture the rules of the road embedded in traffic regulations, contributing to the development of SLEEC-aware monitors for autonomous vehicles.

There is much that still needs to be done. On the elicitation side, a richer language that considers prioritisation of rules [22] may facilitate describing their required application in the case of conflicts. On the engineering side, the issue

of explainability, based on the interpretation of the low-level decisions and actions of an agent in terms of the rules, can have a significant impact on the acceptability of that agent.

We invite colleagues to contribute to our efforts. Our goal is to enable the cost-effective development of trustworthy autonomous agents that can be assured to act according to human values. For that, we need the contribution of scientists and engineers with a wide range of backgrounds.

References

1. Alechina, N., Dastani, M., Logan, B.: Norm specification and verification in multi-agent systems. Journal of Applied Logics **5**(2), 457–489 (2018)
2. Assadi, Z., Inverardi, P.: Fuzzy representation of norms (2026), https://arxiv.org/abs/2601.04249
3. Baxter, J., Ribeiro, P., Cavalcanti, A.L.C.: Sound reasoning in tock-CSP. Acta Informatica **59**, 125–162 (2022). https://doi.org/10.1007/s00236-020-00394-3
4. Boltz, N., Getir Yaman, S., Inverardi, P., de Lemos, R., Landuyt, D.V., Zisman, A.: Human empowerment in self-adaptive socio-technical systems. In: SEAMS'24. pp. 200–206. ACM (2024)
5. Brunero, J.: Reasons and Defeasible Reasoning. The Philosophical Quarterly **72**(1), 41–64 (04 2021). https://doi.org/10.1093/pq/pqab013
6. European Union: Regulation (EU) 2024/1689 of the European Parliament and of the Council of 13 June 2024 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act). Official Journal of the European Union **L 2024/1689** (2024), http://data.europa.eu/eli/reg/2024/1689/oj
7. Feng, N., Marsso, L., Getir Yaman, S., Baatartogtokh, Y., Ayad, R., De Mello, V.O., Townsend, B., Standen, I., Stefanakos, I., Imrie, C., et al.: Analyzing and debugging normative requirements via satisfiability checking. In: 46th IEEE/ACM International Conference on Software Engineering. pp. 1–12 (2024)
8. Feng, N., Marsso, L., Sabetzadeh, M., Chechik, M.: Early verification of legal compliance via bounded satisfiability checking. In: Proceedings of the 34th international conference on Computer Aided Verification (CAV'23), Paris, France. Lecture Notes in Computer Science, Springer (2023)
9. Feng, N., Marsso, L., Yaman, S., Standen, I., Baatartogtokh, Y., Ayad, R., de Mello, V.O., Townsend, B., Bartels, H., Cavalcanti, A., Calinescu, R., Chechik, M.: Normative requirements operationalization with large language models. In: 32nd IEEE International Requirements Engineering Conference. pp. 129–141 (2024). https://doi.org/10.1109/RE59067.2024.00022
10. Feng, N., Marsso, L., Yaman, S.G., Townsend, B., Cavalcanti, A., Calinescu, R., Chechik, M.: Towards a formal framework for normative requirements elicitation. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1776–1780. IEEE (2023)
11. Getir Yaman, S., Ribeiro, P., Burholt, C., Jones, M., Cavalcanti, A., Calinescu, R.: Toolkit for specification, validation and verification of social, legal, ethical, empathetic and cultural requirements for autonomous agents. Science of Computer Programming **236**, 103118 (2024)
12. Getir-Yaman, S., Ribeiro, P., Cavalcanti, A., Calinescu, R., Paterson, C., Townsend, B.A.: Specification, validation and verification of social, legal, ethical,

empathetic and cultural requirements for autonomous agents. Journal of Systems and Software **220**, 112229 (2025). https://doi.org/10.1016/J.JSS.2024.112229
13. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: FDR3 — A Modern Refinement Checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 8413, pp. 187–201 (2014)
14. Hashmi, M., Governatori, G., Wynn, M.T.: Normative requirements for regulatory compliance: An abstract formal framework. Information Systems Frontiers **18**(3), 429–455 (2016)
15. IEEE Standard 7000/2021: Model Process for Addressing Ethical Concerns during System Design (2021), https://ieeexplore.ieee.org/document/9536679
16. Inverardi, P.: The Challenge of Human Dignity in the Era of Autonomous Systems. In: Perspectives on Digital Humanism, pp. 25–29. Springer (2022). https://doi.org/10.1007/978-3-030-86144-5_4
17. Kleijwegt, A., Getir Yaman, S., Calinescu, R.: Tool for supporting debugging and understanding of normative requirements using LLMs. In: 33rd IEEE International Requirements Engineering Conference. pp. 576–579. IEEE (2025)
18. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J., Woodcock, J.: RoboChart: Modelling and verification of the functional behaviour of robotic applications. Software & Systems Modeling **18**, 1–53 (10 2019). https://doi.org/10.1007/s10270-018-00710-z
19. Roscoe, A.W.: Understanding Concurrent Systems. Texts in Computer Science, Springer (2011)
20. Ryan, M., Stahl, B.C.: Artificial intelligence ethics guidelines for developers and users: clarifying their content and normative implications. Journal of Information, Communication and Ethics in Society **19**(1), 61–86 (2021)
21. Tana, C., Siniscalchi, C., Cerundolo, N., Meschi, T., Martelletti, P., Tana, M., Moffa, L., Wells-Gatnik, W., Cipollone, F., Giamberardino, M.: Smart aging: integrating ai into elderly healthcare. BMC Geriatrics (2025), https://doi.org/10.1186/s12877-025-06723-w
22. Townsend, B., Parnell, K.J., Yaman, S.G., Nemirovsky, G., Calinescu, R.: Normative conflict resolution through human–autonomous agent interaction. Journal of Responsible Technology **21**, 100114 (2025). https://doi.org/10.1016/j.jrt.2025.100114
23. Townsend, B., Paterson, C., Arvind, T., Nemirovsky, G., Calinescu, R., Cavalcanti, A., Habli, I., Thomas, A.: From pluralistic normative principles to autonomous-agent rules. Minds and Machines **32**(4), 683–715 (2022)
24. Troquard, N., De Sanctis, M., Inverardi, P., Pelliccione, P., Scoccia, G.L.: Social, legal, ethical, empathetic, and cultural rules: Compilation and reasoning. Proceedings of the AAAI Conference on Artificial Intelligence **38**(20), 22385–22392 (2024). https://doi.org/10.1609/aaai.v38i20.30245
25. UNESCO: Recommendation on the Ethics of Artificial Intelligence (2021), https://unesdoc.unesco.org/ark:/48223/pf0000380455
26. Zhu, J., Gienger, M., Franzese, G., Kober, J.: Do you need a hand? – a bimanual robotic dressing assistance scheme. IEEE Transactions on Robotics **40**, 1906–1919 (2024). https://doi.org/10.1109/TRO.2024.3366008

Tutorial outline

Our tutorial addresses the elicitation, formalisation, validation, and conformance verification of Social, Legal, Ethical, Empathetic, and Cultural (SLEEC) normative requirements for autonomous agents. It is structured as a half-day tutorial comprising the lectures and practical sessions summarised in Table 1.

Table 1: Tutorial schedule

	PART 1: SLEEC rule elicitation & specification
15 min.	Introduction to SLEEC requirements and paradigm (methodology)
25 min.	SLEEC requirements language, illustrate the semantics with scenarios
20 min.	Elicitation
30 min.	Hands on: tool-supported SLEEC rule specification
	Coffee break
	PART 2: SLEEC rule consistency validation & verification
30 min.	SLEEC requirements well-formedness issues detection and resolution
30 min.	SLEEC requirements conformance verification
30 min.	Hands on: tool-supported SLEEC conformance verification