Technical Report: Modeling rates of change and aggregations in runtime Goal Models
April, 2022

Rebecca Morgan[1][0000−0002−3347−7425],
Simon Pulawski[2],
Matt Selway[1][0000−0001−6220−6352],
Wolfgang Mayer[1][0000−0002−2154−2269],
Georg Grossmann[1][0000−0003−4415−2228],
Markus Stumptner[1][0000−0002−7125−3289],
Aditya Ghose[2][0000−0002−6175−8726], and
Ross Kyprianou[3][0000−0002−8873−4417]

[1] Industrial AI Research Centre, UniSA STEM, University of South Australia, Adelaide, SA, 5000, Australia
{first}.{lastname}@unisa.edu.au
https://www.unisa.edu.au/research/industrial-ai/
[2] Decision Systems Lab, University of Wollongong, Wollongong, NSW, 2500, Australia
aditya@uow.edu.au
[3] Defence Science and Technology Group, Edinburgh, SA, 5111, Australia
ross.kyprianou@defence.gov.au

Abstract. Achieving real-time agility and adaptation with respect to changing requirements in IT infrastructure can pose a complex challenge. We explore a goal-oriented approach to managing this complexity. We argue that a goal-oriented perspective can form an effective basis for devising and deploying responses to changed requirements in real-time. We offer an extended vocabulary of goal types, specifically by presenting two novel conceptions: differential goals and integral goals, which we formalize in both linear-time and branching-time settings. We then illustrate the working of the approach by presenting a detailed scenario of adaptation in a Kubernetes setting, in the face of a DDoS attack.

Keywords: goal-modeling · self-adaptive systems · micro-services, context-awareness

1 Introduction

Managing complex information systems deployed on cloud platforms remains a challenge in dynamic environments where systems must respond to changes in demand, failures, attacks, or evolving organizational requirements. Despite automated monitoring and adaptation mechanisms provided by cloud platforms such as Kubernetes, oversight of the adaptation remains a human responsibility necessitating a high degree of situational awareness [11].

Situational awareness [8] is an emergent property resulting from system interactions with the environment and provides the environmental context for making optimal decisions regarding system adaptations to meet operational goals. Similarly, the human operator's situational awareness arises from knowing how the system's automated adaptations to the changing environment are performing with respect to goals. This is often impeded by the overwhelming volume of detail collected in dashboards visualizing sensor information, such as resource utilization and event statistics.

Enhancing situational awareness of both autonomous agents and human operators to support effective decision-making requires an understanding of how changes in the environment (manifested in low-level sensor readings) impact higher-level system goals. Lifting such low-level information and actions into goal-based representations from which higher-level goals can be composed is essential in determining the impact on, and planning of, system goals.

We hypothesize that a goal-oriented lens provides the appropriate set of abstractions to improve situational awareness and resulting human/computer decision making that can be layered over the top of existing complex systems, for example, to facilitate the management of monitoring and adaptation of complex IT infrastructure deployments. We offer a goal-oriented approach to presenting system state and controlling system adaptation, offering the advantage that goals are well-suited for human understanding while leveraging well-established reasoning mechanisms to transform higher-level goals into executable

adaptation plans [1,15]. We use goals in two modes: goal-level sensing, i.e., using goal-based abstractions (probes) for building situation awareness models; and goal-based actuation planning, i.e., refining overall adaptation goals into executable plans for achieving the targeted adaptation (via effectors).

To this end, we extend the repertoire of available goal types discussed in the literature with two new categories (differential and integral goals) and show how goals can be refined into executable adaptations using probes and effectors as abstractions to the underlying Kubernetes platform. The resulting goal-drivenapproach is illustrated on a case study drawn from the cybersecurity domain.

This paper is organized as follows. A discussion of related work is provided in Section 2, followed by the introduction of differential and integral goals in Section 3. The case study exemplifying the goal model and reasoning is presented in Section 4. The Kubernetes framework is described in Section 4.1 and the goal model is presented in Section 4.2. A reasoning example with temporal goals is presented in Section 4.3 and a prototype implementation of a goal reasoner utilizing the novel goal types is described in Section 4.4. We conclude in Section 5.

2 Related Work

Goal-modeling is used by in a variety of frameworks for different purposes, such as requirements engineering. However, we are particularly interested in goal-modeling for runtime frameworks such as those used by self-adaptive systems, as the proposed goal types are particularly applicable in such situations.

Goal-oriented frameworks: There have been many goal-oriented frameworks and methodologies proposed in the past. Notably, most of the goal-related work assumes that goals are used for requirements specification and are mainly design-time artifacts, with the Tropos4AS and GoalD approaches being the exception. Like these two approaches, our approach retains goal models and instances in the runtime environment.

Tropos for Adaptive Systems (Tropos4AS) is an agent-oriented framework for engineering adaptive systems based on i* and following the Tropos engineering methodology [13]. The Tropos4AS modeling language features an extended *goal model* with run-time satisfaction criteria, an *environment model* representing external elements which affect goal satisfaction and a *failure model*, representing unwanted states and associated recovery plans. While our approach shares many modeling similarities, unlike Tropos4AS, we do not use an explicit separate category of failure models, instead dealing with failure as an outcome of goal reasoning. In particular, we define new types differential and integral goals supported by *contexts*. Also, the Tropos4A implementation focuses on code generation for prototype implementation while we are aiming at a service interface to provide adaptive capabilities to an existing deployed system.

The KAOS goal-driven specification methodology has been extended with run-time event-monitoring for reconciling system requirements and run-time behavior system [9]. Similar to our approach, a goal modeling language is proposed which allows to specify reconciliation tactics like shifting to an alternative system design. However, this is a design-time approach, whereas our approach is applicable at run time to an existing system such as Kubernetes. Later work [10] applied a goal-driven approach for system reliability and security which is closely related to our example, though still looking at a design-time perspective. In the latest iteration [4], goals have been extended to operating as runtime entities that are monitored, predicting goal satisfaction rates and adapting the goal model to adjust to the varying obstacle probabilities to maximise these success rates. Detailed probabilistic obstacle models in a formalism mirroring the goal models are used for success estimates and applying countermeasures for improving the overall success rate. Similar principles have been applied to so-called Awareness Requirements in the body of work leading to [3]. In comparison, we use a unified goal model with the purpose of enabling adaptation even within a single highly structured domain process that may be subject to multiple iterations of goal and action adaptations within a single problem instance. As such, we extend the kinds of goals that can be expressed, which could be combined with such goal success/adaptation monitoring approaches described above.

Applications in service environments: Pereira et al. [14] proposed the ATMOSPHERE/TMA platform implementing the MAPE-K control loop for cloud systems and supported by a distributed monitoring system. The MAPE-K components are designed as microservices deployable in container-based systems, e.g., Kubernetes. The usage example focuses on maintaining trustworthiness w.r.t. system/service performance scores, but mention is made that TMA can be used with other metrics, such as dependability, privacy, security. In contrast to our work, it does not address how goals are incorporated or can evolve over time to address a changing environment. Alkhabbas et al.'s [2] work on deploying self-adaptive IoT systems focuses on environments which are (partly) unknown, e.g., it could be a cloud, edge, or hybrid edge-cloud model. It is not clear to which extent the work can deal with changes at runtime.

Other goal-driven adaptation frameworks: Sykes et al. [16] proposed a 3-layer model for adaptable software architecture which focuses on task synthesis from high-level goals. In the *Goal Management* layer, reactive plans are generated from domain models and goals expressed in temporal logic. These plans are interpreted and component configurations are generated in the *Change Management* layer, with the configuration changes applied in the *Component* layer. This approach is limited in the ability to express structural constraints, e.g., resource competition, and did not support dynamic replanning or feedback loops. Mendonça et al.[12] propose a modeling and analysis framework for contextual failures and dependable system requirements, aligning concepts of dependability and failure classification to the requirements of a Contextual Goal Model (CGM). This approach can be coupled to self-adaptation mechanisms at design-time to support the use of dependability criteria. Rodrigues et al. [15] describe *GoalD*, a goal-driven framework for autonomous resource deployment in heterogeneous computing environments, operating in two stages: offline activities performed in preparation for deployment, and online deployment adaptation activities for the runtime environment. The latter are performed by a MAPE-K based manager. GoalD incorporates a runtime framework and algorithms for synthesizing and updating system goal models.

3 Goals for Change and Aggregation

The Kubernetes use case motivates a novel taxonomy of goals. These new goal types are not currently supported even in sophisticated temporal logics such as CTL and LTL or by the most sophisticated model checkers on offer (including probabilistic model checkers such as PRISM). The two new goal types are *Differential goals* and *Integral goals*.

Differential goals A number of the use case scenarios involve goals that involve gradual ramping down or ramping up provisioning. All of these are statements about *differentials* or rates of change. In terms of the underlying semantics of this extension to the goal language, on every path, between some specified lower and upper bound (specified using the UNTIL and UNLESS temporal operators, as one possibility), the rate of change of some numeric variable should be equal to, no more than, or no less than some value. This will permit the specification of the *rate* at which we want to ramp up or ramp down provisioning. In other words, we are making the notion of "gradually" changing something concrete by qualifying it with numbers.

We use $D(F, start, end)$ *op* k to denote that the rate of change of function F satisfies the inequality *op* with constant k at the end of the interval between the first state where the condition *start* is true and the first state where condition *end* is true. As an example, consider the requirement that storage capacity be restored at a rate no faster than 2GB per minute. This would be formalized as: $D(StorageCapacity) \leq 2$.

Integral goals In a similar spirit to specifying goals in terms of differentials, we will also want to specify goals in terms of definite integrals, e.g., the total duration of downtime between some lower and upper bound, should be equal to (or no less than or no greater than) some value. The semantics over paths would be similar, i.e., we would count the number of states satisfying a given property.

We use $I(F, start, end)$ *op* k to denote that the definite integral of the function F satisfies the inequality *op* with constant k in all states between the first state where the condition *start* is true and the first state where condition *end* is true. As an example, consider the requirement that the total downtime for the IP-Telephony service should not exceed 2 hours between the point when a DOS attack is detected and the point where full restoration of normal operations occurs. We use the boolean state variable $notAvailable(IPTelephony)$ to denote the non-availability of the IP-Telephony service. We assume that state transitions are uniformly spaced over time. The *start* condition is denoted by *attackDetected*. The *end* condition is denoted by *operationsRestored*. The property would then be formalized as $I(notAvailable(IPTelephony), attackDetected, operationsRestored) \leq 2$.

Extensions In the following, we present an intuitive formalization in a linear time setting as an extension to LTL (henceforth referred to as Ext-LTL), and then a formalization in a branching time setting as an extension to CTL (henceforth referred to as Ext-CTL).

We consider Ext-LTL first. Let H be the state history. Given s_i in H, let s_m and s_n be states in that history such that $m \leq i \leq n$ such that s_m is the first state where *start* is true, s_n is the first state after s_i where *end* is true.

Then $(H, s_i) \models D(F, start, end)$ *op* k iff $\mid F_n - F_m \mid / \mid n - m \mid$ *op* k. (Here we use F_j to denote the value of F at state s_j, for any j.)

Also, $(H, s_i) \models I(F, start, end)$ *op* k iff $\#S_F$ *op* k where $\#S_F$ represents the count of the number of states between the *start* and *end* conditions where F is *true*. Alternatively $\#S_F = |\{s_j \in H \mid m \leq j \leq n, s_j \models F\}|$

Note that, for simplicity, this considers only boolean variables, states are observed at each time increment, and the constraints are defined with a corresponding time unit. If there is not a one-to-one correspondence then the formalization is more involved but, ultimately, straightforward. Similarly, the formalization can be extended to non-boolean variables.

Ext-CTL expressions for integral and differential goals would involve appending A and E in front of the corresponding Ext-LTL expressions, with the usual semantics involving universal and existential quantification over paths.

Variations on the general categories of differential and integral goals can also be posed. One variation of the general formulation of the differential and integral goals discussed above is the category of *sliding-window* differential and integral goals. Building on the integral goal example above, consider a goal where we want to ensure that the total service downtime should not exceed 2 hours in the preceding 24 hours. The property would then be formalized as $I^{-}(notAvailable(IPTelephony), 24) \leq 2$ (where the dash superscript denotes the sliding window version of the integral goal and here we are counting the number of times when *notAvailable(IPTelephony)* is true). In general, a sliding-window integral goal is written as $I^{-}(F, w)$ *op* k and is equivalent to the integral goal $I(F, t_0 - w, t_0)$ *op* k where the sliding window has width w time units ending at time t_0 (often taken to be now). Here, we have extended the concept of *condition* from being only associated with system states to also include events, including the passage of time. Sliding differential goals are defined similarly:
$D^{-}(F, w)$ *op* k means $D(F, t_0 - w, t_0)$ *op* k.

For simplicity, we assume the window width is given in integer units of time – in our example, hours – matching the state interval; again, if there is not a 1-to-1 correspondence then the formalization is more involved but straightforward.

4 Cybersecurity Use Case and Goal Model

In this paper, we use the cybersecurity context to illustrate goal-based adaptation with proactive situational awareness. For example, consider our aim to be providing responsive and reliable web services to as many users as possible, without making unnecessary or expensive demands on resources.

Ideally, these goals are achieved by applying a management policy that dynamically scales (up or down) the services in response to actual or predicted demand. To maintain reliability, new instances are created or services are restarted in response to failures, and requests are rerouted to alternatives during the fail-over period.

Under normal usage conditions, such a policy would appear to meet our goals, where demand scales predictably and failures are infrequent. However, during anomalous events such as a Distributed Denial of Service (DDoS) attack the system would scale up the services in response to the increased demand. Since resources are finite, the scale up will eventually stop, the DDoS having overwhelmed the available capacity, ultimately leading to total system failure.

To handle such situations, we desire the system to have sufficient situational awareness capability to recognize the anomalous event and autonomously change its policies in response. That is, once an attack is detected, the policy change prioritises the goal of protecting the integrity of the system over the goal of reliably serving users' requests. This may be achieved by the system adapting to filter some of the adversarial requests aiming to compromise the system (e.g., by changing network policies to block regions, sequester services, etc.). Once the system detects the adversarial event is over, rather than immediately return to "business as usual", the system may apply policies enabling a gradual restoration of services to allow system administrators to investigate if any systems, data, etc., were compromised during the attack. Therefore, the system must apply its situational awareness of recent events to gradually adapt back to normality.

We first discuss the Kubernetes environment, the setting for the example, in Section 4.1. The goal model for the use case is detailed in Section 4.2.

4.1 The Kubernetes framework

Kubernetes manages *clusters* of *nodes* on which applications are deployed and run[4]. Nodes may be physical or virtual machines and comprise a number of *pods* that it manages. Each pod is a collection of *containers*, where each container is a running application defined by a *container image* providing the packaged software, its dependencies, core configuration—i.e., everything necessary for the application

[4] https://kubernetes.io/docs/

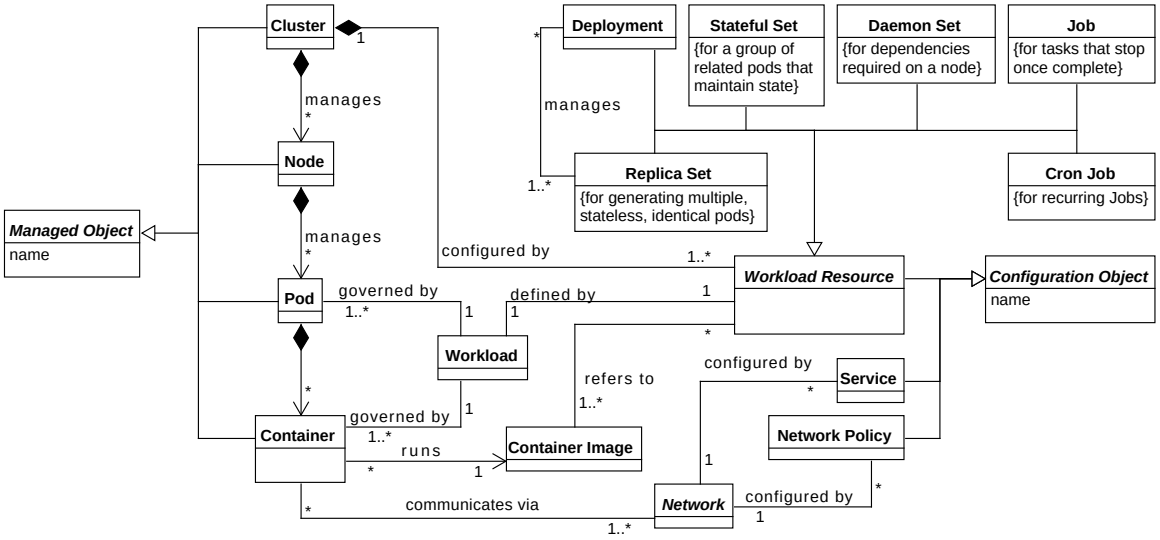

Fig. 1. Model of Kubernetes showing physical structure and configuration overlays

to run. Containers within a pod can communicate with one another via loopback network, but cannot communicate with containers in other pods (unless enabled through configuration).

Kubernetes supports basic self-healing, i.e., restarting failed containers and rescheduling pods on new nodes if a node fails, manual and automatic (resource-based) scaling, managed upgrades and rollbacks of applications/containers. However, the information and decision-making is low-level, so we raise the basic functionality into goal-based framework through probes that can query Kubernetes state, monitoring, etc., and effectors, which can take actions on the Kubernetes configuration. Raising the information and decision making to goal level provides additional control over the framework based on the higher-level goals and its situational awareness.

The model of Kubernetes displayed in Figure 1 represents the physical structure and configuration overlay of the Kubernetes architecture. To maintain simplicity for the time being, the model does not explicitly represent the control plane and its components, nor the kubelet and proxy components of the nodes.

On top of the physical structure, Kubernetes overlays configuration 'objects' which represent the physical elements plus more abstract-concepts that may influence or determine the running physical structure. Such objects include *Workloads* (*Deployment*, *Job*, *StatefulSet*, *DaemonSet*), which describe specific configurations of applications/containers and how they are deployed across pods and nodes, *Services* and *NetworkPolicies*, which describe how applications (as a logical group of pods) can be accessed from outside of the cluster, *Volumes*, which describe mechanisms for data storage including persistent and non-persistent storage, and *ConfigMaps* for providing application specific configuration.

Probes and effectors for Kubernetes are defined through interactions with the Kubernetes components, such as kubelet and kube-apiserver, and the APIs that they expose. Metrics such as pod health status and number of requests are collected from kubelets and made available through the metrics server endpoint, which exposes them in the kube-apiserver through the metrics API. Effectors cause changes in Kubernetes environments by triggering actions of the built-in controllers through the exposed APIs of the kube-apiserver. In addition, they may update and commit configuration changes to file, allowing the Kubernetes machinery to pick up the change and enact it.

4.2 Goal model

The goal model is based on well-established notions of goal decomposition following the i* framework [17], complemented by dependency and inhibition relationships among goals as defined in Tropos4AS [13]. To account for situational goals which may apply in one context but not another, we employ and extend the notion of a goal life cycle, introduced in [7], with the goal model providing an explicit context structure. Three major categories of goals are available. *Achieve-goals* are satisfied once, in specified conditions, *Maintain-goals* are satisfied when specified conditions are maintained over a period of time, while *Perform-goals* or (Manual Goals) are satisfied through execution of associated activities. Goal relationships are expressed in terms of goal *sequence* and *inhibition*, expressing run-time precedence between goals.

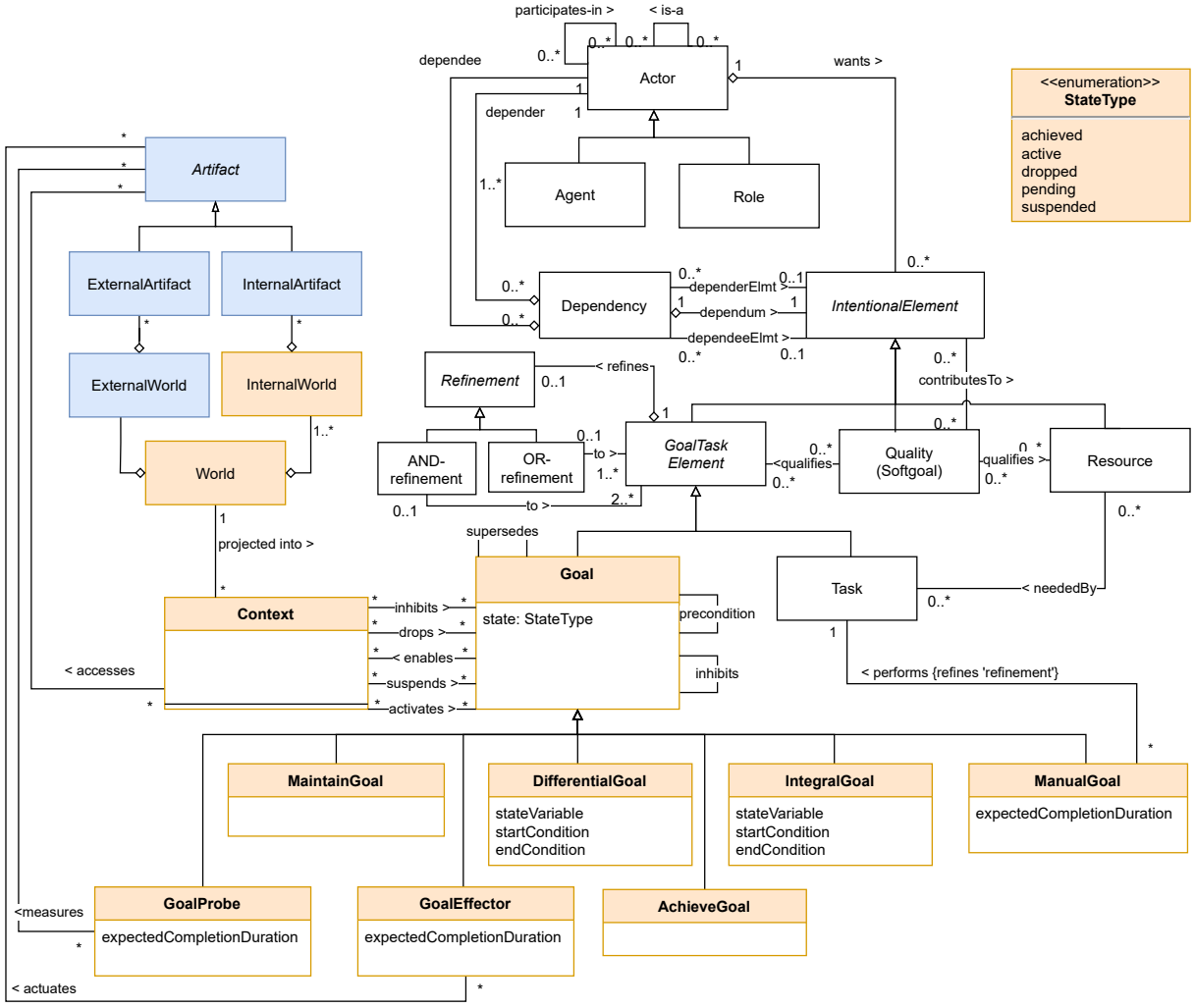

Fig. 2. Goal Meta-model: core i* on the left, extensions including Contexts, Differential and Integral Goals on the right.

The goal life cycle we have chosen, which differs somewhat from [7], centers on the *Active* state, with the *Suspended* state permitting the goal to be temporarily removed from consideration to be later reactivated, and the *Dropped* state represents the goal being discarded. The *trigger* relationship results in the achievement of a goal initiating the transition to a different context. In each context, a different set of goals may apply. Upon transition, goals can be re-activated, suspended, dropped, and maintenance goals can be carried over to the target context (unless they are dropped). Goals can be equipped with domain-specific attributes representing, e.g., costs and delays associated with the goal. For clarity of presentation, we omit the details of cost and temporal consideration in the goal reasoning of the following scenario. We further rely on the notion of *probes* for sensing system state and *effectors* for manipulating system state [5]. Probes and effectors at the goal level facilitate the reasoning process, while their concrete implementation is conceptually represented by invocation of platform services.

A sketch of the goal meta-model is shown in Figure 2 in the context of i* concepts (specifically iStar 2.0 [6]), shown in white, and which incorporates aspects of the Tropos4AS methodology which relate to the environmental context (shown in blue). The proposed extensions (shown in orange) are predominantly refinements of the i* goal concept, covering the three major categories of goal as described in the previous section. In addition, *Goal Probes* and *Goal Effectors*, which measure properties of or actuate behaviours in the internal and external world, respectively, are associated with the *Artifact* concept from Tropos4AS, which represents the non-autonomous things, states, etc., of the world that do not possess autonomous behaviour. *Artifacts* are separated into *internal* and *external* artifacts, the latter comprising the *External World*. We extend this representation with general representation of *Internal World*, vis-á-vis *External World*, and an overall *World* being the composition of an *External World* and a multitude of *Internal*

Table 1. Goal Summary (excl. manual goals)

ID	Type	Goal Description
G1	Int.	Services shall be available for no less than 1425 min in any 24hr interval (i.e., 99%)
G1.1	Maint.	Monitor availability of services
G1.2	Achieve.	Establish availability of a service
G2	Int.	Limit number of pod allocation events to no more than 5 over the baseline (e.g., 3) while we have 2 nodes.
G2.1	Achieve.	Reduce resource usage (to prevent need to allocate a new pod)
G3	Diff.	Planning constraint limits resource alloc. change (no. pods) to within 35% (i.e., 2 pods) per 5min
G4	Achieve.	Detect DDoS attack; indicated by monitoring events
G6	Achieve.	Mitigate attack
G6.1	Achieve.	Filter attack
G6.2	Maint.	Prevent restart of failed services
G6.3	Maint.	Monitor logs for suspicious activity
G8	Achieve.	Restore normal operations
G8.1	Diff.	Redirect requests to original; no more than 20% of service instances (i.e., 1 instance) per 5 mins
P1	Probe	Active Probing of service accessibility every 5 min (performed by Kubernetes)
P2	Probe	Monitor Response Times (monitoring and recording by Kubernetes components)
E1	Effector	Restart Failed Instance (triggered in Kubernetes based on restart policy)
E2	Effector	Spawn Instance (triggered in Kubernetes based on configuration change)
P3/P4	Probe	Monitor Resources (as for P2)
E3	Effector	Stop Instance; (temporarily) stops a container instance using Kubernetes API
P5/P6	Probe	(D)DoS Alert raised by IDS
E4	Effector	Redirect Routes (updates network for pods/ containers via Kubernetes configuration)
E5	Effector	Restore Route; as E4 only undoes filtering

Note: The manual goals G5 and G7 are not listed for brevity.

*World*s (possibly representing each Actor's internal state, for example). It is the totality of the world that is then projected onto the environmental *Context* under which the goals operate as the contexts required for situational awareness and adaptation of decision making may cross boundaries, factoring in not only external state but also the internal states of autonomous agents (at minimum, the agent performing the goal reasoning).

We show the use of the novel differential- and integral goals in a cyber-situational awareness scenario where a system is affected by a DDOS attack and must respond. The response is governed by general cybersecurity response plans, which in this case include five phases: normal, where the system performs normally; analysis, where the nature and effects of a detected attack are analyzed; mitigation, where the response measures are taken to mitigate the effects of the attack; learning, where the outcomes of the defense measures are assessed; and restoration, where the system attack has been mitigated and the system is returned to normal operations. The different phases map directly to contexts in the goal model.

In each context, different goals are relevant, which is shown in Figure 3 including the goals, their decomposition, and relations. Table 1 lists the goals in brief. Achievement and maintenance goals are represented as green shapes: optional adornments identify differential and integral goals. Probes and effectors are purple subgoals. Subgoals whose achievement establishes the preconditions for their parent goals are annotated with a yellow precondition symbol.

In the *normal operations context*, the system is concerned with monitoring and maintaining its ability to deliver services (goal G1). This integral goal measures the services functioning as expected (as assessed by probing or measuring service response times) over a period and takes remedial action if services are thought to be affected in terms of their ability to respond within a set time threshold, in which case the system can restart failed instances or spawn additional instances to increase the capacity of the system. At the same time, goal G2 ensures that costs are maintained within a pre-set budget profile by reducing provisioned resources and preventing the spawning of new instances. This integral goal aggregates the resource usage over a period and takes remedial action if the number (or equivalently, cost) of resources exceed acceptable thresholds. Differential goal G3 ensures that the rate of change in resource provisioning remains at acceptable levels to avoid catastrophic downscaling and undesirable upscaling due to mispredictions of resource demand. This goal constrains the actions triggered by G1 and G2. The system shall detect DDoS attack and trigger an appropriate response (goal G4). We abstract from the details of the detection mechanisms for G4 and instead rely on intrusion detection and monitoring systems

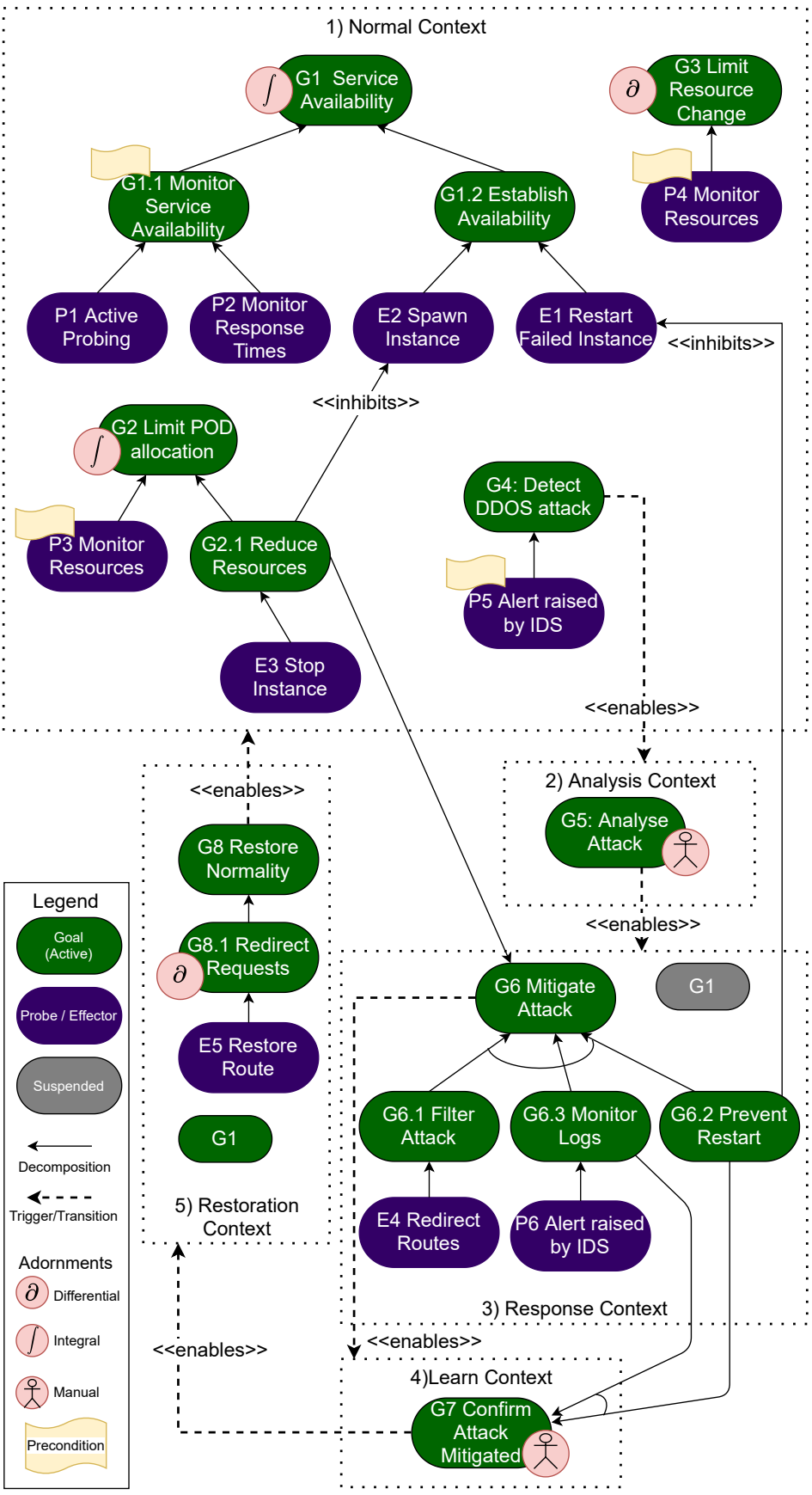

Fig. 3. Goal Model for DDoS Scenario

(IDSs) to detect potential attacks. Once an attack has been detected, the response is initiated, and the system exits the normal context and enters the analysis context.

In the *Analysis Context*, the goal of analyzing the nature of the attack (G5) is added to the active goals. These activities are predominantly manual tasks aiming to identify the nature of the attack, its sources, and impacts on the system. Here, the goal model can provide situational awareness about the progress of these tasks to the team, and support activities through collecting and extracting relevant information. These technical support tasks are not shown in the goal model for brevity. Once the nature of the attack is understood, the response context is triggered. For discussion purposes, it is assumed that the attack is a request flood, where the system is overloaded by a huge number of requests from external sources.

Upon entering the *Response Context*, goal G1 is suspended to prevent uncontrolled growth of resources to serve the spurious requests. Goal G6 is initiated, which aims to implement measures to mitigate the effects of the attack. We consider three possible measures: implementing network-level filters to remove as many bogus requests as possible (G6.1), to disable automatic fail-over mechanisms that can degrade the system performance further due to many restarts of assumed failed services (G6.2), implement monitoring tasks to assess if any other attacks may be masked by the DDoS attack (G6.3). Once mitigation activities are implemented, the learning phase of the response protocol begins.

The *Learning Context* involves predominantly manually controlled activities to assess the success of the response measures and to confirm that the attack has been mitigated (G7). Akin to the analysis phase, the technical monitoring and information collection tasks initiated by the goal model are omitted for brevity. Once the attack has subsided, the system enters the restoration phase to return it to normal operations.

In the *Restoration Context*, goal G1 is reinstated, and the actions taken during mitigation must be undone. Goal G8 is activated to govern the gradual return to normality. This differential goal ensures that parts of the system are made available incrementally (for example by service, by data center, or by zone) by controlling the rate at which services are reinstated. As services need to return to normal operation, goals G6.2 and G6.3 are dropped, preventing goal conflicts and allowing G6 to be achieved—Goals G6 and G6.1 were already achieved upon executing the filtering with the maintenance of G6.2 and G6.3 transferring to G7. Once G8 is achieved, the system reenters the normal context and resumes operation.

4.3 Reasoning Example with Temporal Goals

Consider that each state is characterized by the variables:

nodes: The number of nodes in the Kubernetes cluster. Constant 2 in the example.
pods: The number of pods allocated to nodes.
avail$_s$**:** Boolean indicating if service s meets its QoS requirements.
filtered$_{s,i}$**:** Boolean indicating if requests to instance i of service s have been blocked.
time: Time stamp at which the state is captured.

For simplicity, we consider only a single service s with multiple instances in the example; the subscripts on the variables are omitted.

Let the Initial state be described as follows (3 instances of the service):

nodes: 2;
pods: 3;
avail: T;
filtered: [F,F,F]

The differential and integral goals are defined below.

G1: Availability Service availability to be 99% or 1425 minutes in each 1440 minute (24hr) interval.

$$I^{-}(avail, 1440) \geq 1425$$

where avail denotes that the service is available.

G2: Limit pod allocations Limit number of pod allocation events to no more than 5 over baseline (e.g. 3) while there are 2 nodes.

$$I(allocation, \psi_s, \psi_e) \leq 5$$

where $\psi_s = (nodes = 2) \wedge X^{-1}(nodes \neq 2)$, $\psi_e = (nodes = 2) \wedge X(nodes \neq 2)$, and X^{-1} and X denote the LTL predecessor and successor operators, respectively. Variable *allocation* is a boolean variable indicating that a change in allocation has occurred in a state.

G3: Limit resource allocation change Limit resource allocation change (number of pods) to within 35% (i.e. 2 pods) per 5 min. We assume that the resource usage is measured by the number of pods for the example.

$$D^{-}(pods, 5) \leq 2$$

where pods denotes the number of pods allocated to nodes.

G8.1: Redirect requests to original Limit service instance request redirects (restoration) to no more than 20% (i.e. 1 instance) per 5 minute window.

$$D^{-}(|\{f \in filtered \,|\, f = F\}|, 5) \leq 1$$

which operates over the count of $filtered$ stare variables that are 'F', i.e., not filtering the requests.

Suppose i is the current state in H. States with index $\leq i$ make up the actual history of the system, whereas states with index $> i$ are predictions of future state.

We can evaluate goal conditions at i and, if goal conditions are violated, use a lookahead approach to infer actions. Suppose the planner considers an action a to repair the goal. We can use a model to predict the future trajectory if a was applied in state i, and then evaluate the goal conditions to see if that resolves the issue. Select the action that achieves the best results (in some notion of "best"). Ideally, no goal condition is violated, but some metrics quantifying the deviation from normality could be applied for soft goals etc. An example event sequence for the reasoning is illustrated in Table 2.

4.4 Goal reasoner implementation

To demonstrate our approach we have developed a prototype implementation of a goal reasoner that utilizes the novel differential and integral goals to reason about the new goal states of our automated system. The goal reasoner takes as input an *instantiated goal model*, i.e. a goal model for which goal states have been assigned to each goal, and a *state history*, a set of *state variables* and their corresponding values at particular points in time. The values of state variables are extracted from the environment using *probes*, which are an element in the goal model. The goal reasoner returns a new instance of the goal model for which the goal states and the active contexts have been updated given the state histories. The implementation has been made available.[5]

The new goal model also encodes information on which *effectors* to activate or suspend, consequently providing information on how the automated system utilizing the goal reasoner should act on the environment it is operating in. The model similarly provides information on which probes to activate or suspend so that state variable measurements can be recorded for the subsequent reasoning cycle.

In the implementation, the values of state variables at particular points in time take the form of logical predicates `at(`$Predicate$`,`$Time$`)` where $Predicate$ is the predicate containing the state variable and its value, and $Time$ is the time at which the predicate was recorded.

For each differential or integral goal, a number of attributes must be provided to define the goal within the goal reasoner: `stateVar` specifies the predicate which contains a goal's relevant state variable. `start` specifies the starting time at which the state variable's value becomes relevant to the goal (setting this to `-inf` specifies that the values are relevant immediately). `finish` specifies the final time which a value is relevant to the goal (this can be set to `inf` to specify that it is a maintenance goal going on forever). `intervalSize` specifies the window size for summing states or measuring rates of change (this can be set to `inf` to specify a goal has a static window). `atLeast` specifies the lower bound on the rate of change or aggregation value being computed. `atMost` specifies the upper bound on the computed value, `frequency` specifies how often the value of a state variable must be known. In this paper we have assumed evenly distributed time steps and assume that state variable values are known in each state, so frequency does not affect our example in this paper; we have included it for the purpose of generality.

Each time the goal reasoner is called, it checks each goal to see if its satisfaction of failure condition has been reached given the known values of state variables at each time step. This information is used to update the goal states of each goal, which then triggers the context phase. The context phase determines which contexts should be activated given the new goal states in the goal model. Once new contexts have been activated, goal states are updated again given the new contexts. Finally, the new goal model can be returned.

Our implementation contains two examples that showcase the functionality of the goal reasoner, one where no goals were found to have failed and context shifting occurs (found on the branch **main**), and

[5] https://github.com/DSL-UOW/ER2022-Differential-Integral-Goal-Reasoner

Table 2. Example Event Sequence

Event	Context	Active Goals	Unsatisfied Goals	Action	n	p	a	filtered
(1) Initial state	Normal	G1*, G2*, G3, G4	G4	NONE	2	3	T	F,F,F
(2) Predict G1.1 failure i+1	Normal	G1*, G2*, G3, G4	G4, G1.1/G1	Spawn Instance: #4	2	4	F	F,F,F,F
(3) Predict G1.1 failure i+2	Normal	G1*, G2*, G3, G4	G4, G1.1/G1	Spawn Instance: #5 (limit reached G3)	2	5	F	F,F,F,F,F
(4) Detect DOS attack (P5)	Normal	G1*, G2*, G3, G5	G5	Context Change: Analyze (activate G5)	2	5	T	F,F,F,F,F
(5) G5 Achieved	Analyse	G1*, G2*, G3	$\emptyset$	Context Change: Response (activate G6)	2	5	T	F,F,F,F,F
(6) G1 Suspended, G6 added (Skip to Restore Context)	Response	G2*, G3, G6*	G6,G6.1	Filter/sequester all instances	2	5	F	T,T,T,T,T
(7) G1 reactivated, G6.2 dropped	Restore	G1*, G2*, G3, G8*	G8.1/G8	Restore: #1, #2 (limited by G8.1)	2	5	F	F,F,T,T,T
(8) 5 mins later (G8.1 rate limit)	Response	G1*, G2*, G3, G8*	G81./G8	Restore: #3, #4 (limited by G8.1)	2	5	T	F,F,F,F,T
(9) 5 mins later (G8.1 rate limit)	Response	G1*, G2*, G3, G8*	G81./G8	Restore: #5	2	5	T	F,F,F,F,F
(10) G8.1/G8 Achieved	Response	G1*, G2*, G3	$\emptyset$	Context Change: Normal (activate G4)	2	5	T	F,F,F,F,F
(11) Workload Dropped	Normal	G1*, G2*, G3, G4	G2.1/G2	Stop Instances: #4,#5	2	3	T	F,F,F,F,F

Gn* indicates inclusive of child goals; n = **nodes**; p = **pods**; a = **avail**

one where the integral goal $G1$ is determined to have failed and the program exits (found on the branch **failure-example**). These examples are given in the **main** and **failure-example** branches respectively.

The goal model is initially uninitialized and all goals are set to the goal state *inactive*. When running the query `?- run.` in branch **main** the goal model is initialized according to the current context specified i.e. the *Normal Context* specified by the `currentContext(normalC)` predicate. When initializing this context, the goal reasoner activates all the goals specified by the predicate `activates(normalC,`*Goal*`)` by recursively activating goals that stem from the specified root goal. This can be observed by all goals in the *Normal Context* being changed from *inactive* to *active*.

Next, each active differential and integral goal is checked using the state histories provided and is found to have not failed resulting in output specifying the goals remaining active. Next, all active goals are checked for whether their success condition is true so that they can be marked active. $P5$ is changed to *achieved* because the condition `alertRaisedByIDS` is true in the current timestep. At this point, if any goals have failed the program will have aborted. The achievement of goals is then propagated so that goals that are achieved by other goals that have just been achieved are also marked as achieved. During this step, $G4$ is marked as achieved because $P5$ has been achieved. Because $G4$ enables the *Analysis Context*, the new context is activated in the same way the *Normal Context* was initialized. $G4$ and $P5$ are changed from *achieved* to *active* as their achievement triggers a new context and must be reset. This concludes the first timestep.

When running the query `?- run.` in the **failure-example** branch, the procedure runs in much the same way as the **main** branch with the exception that when $G1$ is checked it fails because it's state history does not satisfy the goal criteria. The program aborts shortly after because a goal has failed.

In the second timestep, each of the differential and integral goals in the first timestep are still active and so they are checked again including their new state history entry at the current timestep. Goals are checked for achievement again and because the predicate `attackAnalysed` is true, $G5$ is marked as achieved and the *Response Context* is activated in the same way as before. In addition to this, *Reponse Context* also suspends $G1$ and so it and all of its subgoals are marked as *suspended* in the same way that they were activated.

The branch **main** contains an example of a goal model and state history for which no goals have failed. Each active integral and differential goal (those in the initial *Normal Context*) are checked against the state histories by considering each goal's parameters. For example, the parameters for goal *G1: Service Availability* are as follows:

```
stateVar(g1,servicesAvailable,1).
start(g1,0).
finish(g1,inf).
intervalSize(g1,1440).
atLeast(g1,285).
atMost(g1,inf).
frequency(g1,5).
```

State histories take the `at(Predicate,Time)`. An example of state history predicates pertaining to goal $G1$ are shown below:

```
at(servicesAvailable(1),15).
at(servicesAvailable(0),20).
at(servicesAvailable(1),25).
```

These state histories are used to determine the success, failure or nofail conditions of differential and integral goals.

Because there is no failure when checking each goal, the goal model in branch **main** switches contexts from *Normal Context* to *Analysis Context* when probe *P5: Alert raised by IDS* is determined to have been achieved because its success condition `alertRaisedByIDS` was found to be true at the current timestep. In the subsequent time step, *G5: Analyse Attack* was determined to be successful due to its success condition `attackAnalysed` being true and so the *Response Context* was activated.

Each time a context is activated, some new goals are activated and some old goals may be suspended. These are specified using the predicates `activates(Context,Goal).` and `suspends(Context,Goal).` respectively. The activation and suspension of each goal when shifting context can be observed in the output of running the program in branch **main**.

When running the program in branch **failure-example**, it can be observed that the program exits due to goal $G1$ failing. This is a consequence of providing a different state history where the service is considered unavailable in time steps 0-25 given the following predicates:

```
at(servicesAvailable(0),0).
at(servicesAvailable(0),5).
at(servicesAvailable(0),10).
at(servicesAvailable(0),15).
at(servicesAvailable(0),20).
at(servicesAvailable(0),25).
```

After summing the total number of time steps where the service is available it is determined that the service was unavailable for too long given the integral goal's parameters and so the goal fails.

5 Conclusions

We presented a goal-oriented approach to IT infrastructure agility in the context of the Kubernetes cloud platform and showed that the goal-based approach can help maintain situational awareness of intended and actual system operations through high-level goals aligned with human abstractions. We introduced differential and integral goals to capture goals related to rate of evolution and aggregate measurements derived from the system operations over time and showed how these goals can inform the goal reasoning and goal refinement process. Mappings of elements in the goal models to elements in the underlying execution platform facilitated the synthesis of concrete executable adaptation and monitoring activities. We intend to study the characteristics of the proposed approach in more detail on additional case studies on the Kubernetes platform as part of future work.

References

1. Ali, R., Dalpiaz, F., Giorgini, P.: A goal-based framework for contextual requirements modeling and analysis. Requir. Eng. **15**(4), 439–458 (2010)
2. Alkhabbas, F., Murturi, I., Spalazzese, R., Davidsson, P., Dustdar, S.: A Goal-Driven Approach for Deploying Self-Adaptive IoT Systems. In: Proc. of ICSA. pp. 146–156. IEEE (2020), https://doi.org/10.1109/ICSA47634.2020.00022
3. Angelopoulos, K., Souza, V.E.S., Mylopoulos, J.: Dealing with multiple failures in Zanshin: a control-theoretic approach. In: Proc. SEAMS. pp. 165–174. ACM (2014)
4. Cailliau, A., van Lamsweerde, A.: Runtime Monitoring and Resolution of Probabilistic Obstacles to System Goals. ACM TAAS **14**(1), 3:1–3:40 (2019)
5. Cheng, S.W.: Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation. Ph.D. thesis, CMU School of Computer Science (2008), CMU-ISR-08-113
6. Dalpiaz, F., Franch, X., Horkoff, J.: iStar 2.0 language guide (2016, v3). https://doi.org/10.48550/ARXIV.1605.07767
7. Dastani, M., Riemsdijk, M., Winikoff, M.: Rich goal types in agent programming. In: Proc. of AAMAS. pp. 405–412. IFAAMAS (2011)
8. Endsley, M.R.: Toward a Theory of Situation Awareness in Dynamic Systems. Human Factors **37**, 32–64 (1995)
9. Feather, M., Fickas, S., van Lamsweerde, A., Ponsard, C.: Reconciling system requirements and runtime behavior. In: Proc. of Int. Workshop on Software Specification and Design (IWSSD). pp. 50–59. IEEE (1998)
10. van Lamsweerde, A.: Engineering requirements for system reliability and security. In: Software Systems Reliability and Security, vol. 9, pp. 196–238. IOS Press (2007)
11. Li, N., Cámara, J., Garlan, D., Schmerl, B.R., Jin, Z.: Hey! Preparing Humans to do Tasks in Self-adaptive Systems. In: Proc. SEAMS. pp. 48–58. IEEE (2021)
12. Mendonça, D.F., Ali, R., Rodrigues, G.N.: Modelling and analysing contextual failures for dependability requirements. In: Proc. SEAMS. pp. 55–64. ACM (2014)
13. Morandini, M., Penserini, L., Perini, A., Marchetto, A.: Engineering requirements for adaptive systems. Requir. Eng. **22**(1), 77–103 (2017)
14. Pereira, J.D., Silva, R., Antunes, N., Silva, J.L.M., de França, B., Moraes, R., Vieira, M.: A platform to enable self-adaptive cloud applications using trustworthiness properties. In: Proc. SEAMS. pp. 71–77. ACM (2020)
15. Rodrigues, G.S., et al: GoalD: A Goal-Driven deployment framework for dynamic and heterogeneous computing environments. Inf. Softw. Technol. **111**, 159–176 (2019)
16. Sykes, D., Heaven, W., Magee, J., Kramer, J.: From goals to components: a combined approach to self-management. In: Proc. SEAMS. pp. 1–8. ACM (2008)
17. Yu, E.: Modelling Strategic Relationships for Process Reengineering. Ph.D. thesis, University of Toronto (1995)