Runtime Goal Models

Fabiano Dalpiaz
University of Toronto
Canada
dalpiaz@cs.toronto.edu

Alexander Borgida
Rutgers University
United States
borgida@cs.rutgers.edu

Jennifer Horkoff, John Mylopoulos
University of Trento
Italy
{horkoff, jm}@disi.unitn.it

***Abstract*—Goal models capture stakeholder requirements for a system-to-be, but also circumscribe a space of alternative specifications for fulfilling these requirements. Recent proposals for self-adaptive software systems rely on variants of goal models to support monitoring and adaptation functions. In such cases, goal models serve as mechanisms in terms of which systems reflect upon their requirements during their operation. We argue that existing proposals for using goal models at runtime are using design artifacts for purposes they were not intended, i.e., for reasoning about runtime system behavior. In this paper, we propose a conceptual distinction between Design-time Goal Models (DGMs)—used to design a system—and Runtime Goal Models (RGMs)—used to analyze a system's runtime behavior with respect to its requirements. RGMs extend DGMs with additional state, behavioral and historical information about the fulfillment of goals. We propose a syntactic structure for RGMs, a method for deriving them from DGMs, and runtime algorithms that support their monitoring.**

***Keywords*—Runtime goal models; Requirements at runtime; Goal reasoning; Self-adaptive systems.**

I. Introduction

The past decade has seen a dramatic rise in interest in software systems that monitor their environment and, if necessary, adapt in order to continue to fulfill their requirements. In the context of Requirements Engineering (RE), such work goes back to the seminal contributions by Fickas and Feather [1], [2], followed by several comprehensive proposals for automated monitoring of requirements [3], [4], [5], based on different languages and formalisms (including the Event Calculus and variants of Temporal Logics).

The pivotal role of requirements during the post-deployment phase of a system's lifecycle has recently been emphasized by foundational research on requirements at runtime [6], [7], and self-adaptive systems [8]. The challenge for us is to create an artifact from requirements that supports (i) analysis to determine whether system operations are in accordance with its specification; and, when needed, (ii) exploration of alternative system configurations that restore normal operations and continuing fulfillment of requirements. In this paper we address the first challenge, showing how to create a runtime artifact from requirements allowing for monitoring and diagnosis.

Some existing approaches (e.g., [5], [9], [10]) rely on early requirements models, often variants of *i**goal models [11], which are insufficiently detailed to effectively express (un)desired system behavior. Early requirements models talk about stakeholder goals and needed functionality for the system-to-be (e.g., sending e-mail), and can play a key role in communicating between various stakeholders by abstracting details that are irrelevant at this level. Requirement models are also useful for choosing among design alternatives.

However, at runtime, system behavior is characterized by *events occurring in the world*, related to goal *instances* (e.g., an e-mail has been sent from X to Y for purpose Z at some point in time). Some approaches (e.g., [4], [12], [13]) adopt low-level specification languages, resembling programming calculi, to express behavior. These are easy to monitor, for they capture actual program behavior, but are difficult to trace back to original stakeholder requirements.

We provide a framework for bridging the gap between design-time goal models and runtime behavior. Starting with an early requirements model representing stakeholder goals (*Design-time Goal Model* or *DGM*), we refine it with additional behavioral detail about how goals are to be achieved. Specifically, we add constraints on valid orderings for pursuing subgoals, thereby creating a *Runtime Goal Model* (*RGM*). Although this information helps us to express additional desired runtime behavior, this model is still at the class level, while we really need to reason over multiple goal instances.

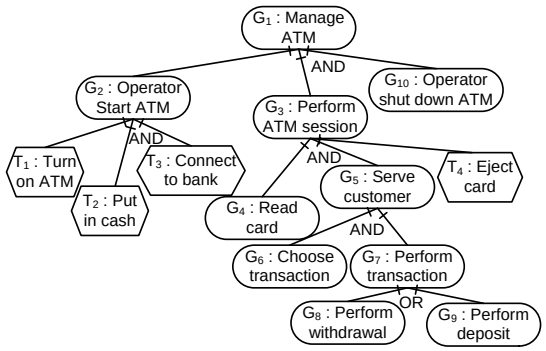

Fig. 1. A partial DGM for an ATM system, adapted from [5]. Goals (oval shapes) are refined via AND/OR refinement links to tasks (hexagons), i.e., functionalities for the system-to-be

We elucidate the need to distinguish between classes and instances using the partial DGM for an Automated Teller Machine (ATM) system (Figure 1) adapted from [5]. The model is informative at the class level, as it conveys information about required functionality for the system-to-be. However, when used at runtime, it does not explain when and how many instances of the goals and tasks in the model need to be created, nor how many have been created. For example, putting in cash (task T_2) may be unnecessary for starting the ATM (achieving goal G_2), if enough money is already available. In such situation, no instance of T_2 is needed to achieve

an instance of G_2. Moreover, between the start and the shut down of the ATM (goals G_2 and G_{10}), many instances of G_3 may be achieved to serve different customers. The model does not include this information. Previous work (see Section VI) has used isomorphic copies of a DGM at runtime, thereby restricting the space of behaviors that are consistent with the given requirements model. For instance, the isomorphism assumption between goal classes and instances does not allow the behaviors noted above for the ATM goal model.

The runtime behavior for goal instances can be reported via a "trace", given in terms of events for leaf-goals/tasks, (e.g., $\langle\, goal_1.start, goal_1.succeed \,\rangle$ where $\langle\, goal_1 \,\rangle$ is an instance of some goal class G), representing a particular run of a system intending to make progress toward system requirements. To connect instance-level traces and class-level RGMs, we introduce *Runtime Goal Instance* structures (*RGI*) that allow us to reason over runtime behavior relative to design-time concepts. Specifically, RGIs support reasoning about (i) current goal instances, (ii) their behavior (e.g., relative ordering of instance events), and (iii) the state in which each instance is currently in (being pursued? successfully completed?). From this, one can answer more general questions for supporting monitoring and adaptation, e.g., "How many instances of class G_3 are there? How many have successfully completed? How many have failed?".

Specifically, the contributions of this paper are as follows:

1) Based on goal-oriented requirements models, we study the relationship among Design-time Goal Models, their enriched version (Runtime Goal Models), and Runtime Goal Instances.
2) We devise a method for enriching a DGM to obtain an RGM. Our method ensures that the RGM developed is consistent with its DGM counterpart.
3) We propose reasoning mechanisms to project the execution of a running system—represented by its trace—on Runtime Goal Instances for a given RGM. These mechanisms enable monitoring of the state of multiple goal instances.
4) We evaluate our approach by illustrating how the events in a possible trace of a meeting scheduler lead to different updates in the RGI for the system.

Organization. Section II motivates our work with the help of the meeting scheduler scenario. Section III presents our method for deriving RGMs from DGMs. Section IV introduces our reasoning mechanisms. Section V shows different updates in an RGI through several runtime scenarios. Section VI contrasts our approach to related work, while Section VII discusses our approach and presents future directions.

II. Preliminaries, with motivating examples

Goal models such as *i** [11] and KAOS [14] capture high level stakeholder requirements, and refinement/influence relationships, indicating what combinations of tasks can achieve desired goals. In this paper, we refer to such goal models as Design-time Goal Models (DGMs), to indicate their role in exploring and analyzing *design* alternatives, each consisting of a collection of tasks through which root-level goals can be fulfilled. We illustrate DGMs on the meeting scheduler exemplar [15] (Section II-A), then point out their limitations in supporting runtime system monitoring and adaptation and suggest extensions (Section II-B).

A. DGM for a meeting scheduler

Figure 2 shows a partial DGM for a meeting scheduler. We use a simple version of goal models [16] that includes functional (hard) goals only (no softgoals), AND/OR refinements, and influence/contribution relationships. This language, formalized in Definition 1, suffices to highlight the different concerns that arise between design-time and runtime goal models. We do not consider softgoals in this paper. Understanding their role at runtime is part of our future work.

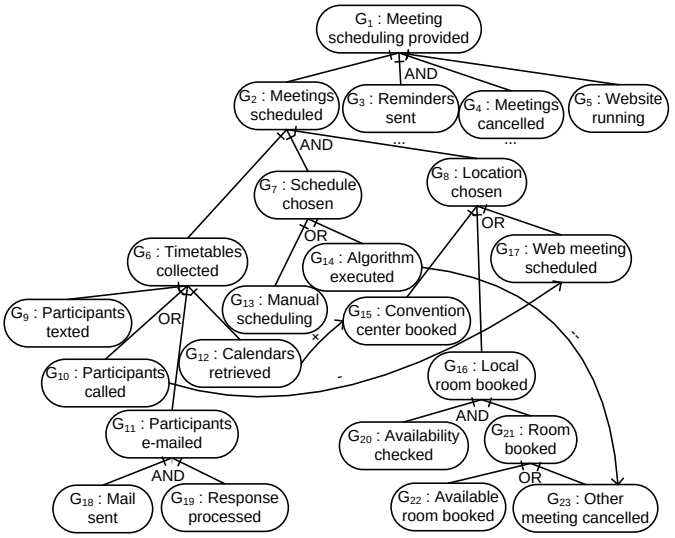

Fig. 2. A partial DGM for a meeting scheduler

In Figure 2, the top-level goal is to provide meeting scheduling services (G_1). To do so, the system has to schedule meetings (G_2), send reminders (G_3), cancel meetings (G_4), and ensure these functionalities are provided via a running website (G_5). Meetings are scheduled by collecting participant timetables (G_6), choosing a schedule (G_7), and choosing a location (G_8). There are different ways for collecting timetables: sending text messages to mobile phones (G_9), calling participants by phone (G_{10}), e-mailing them (G_{11}), or retrieving their online calendar (G_{12}). Some goals influence other goals. For example, retrieving calendars influences positively the booking of a convention center, for it eases matching the participants' calendars with room availability. This model is a useful tool for analysts to explore alternative functionalities for fulfilling the top-level goal, also for understanding influences between goals. This model intentionally abstracts away certain details, such as the order in which subgoals are to be fulfilled (e.g., G_2 should occur before G_4), or repetitions (e.g., G_{10} may have to be repeated a few times until contact is made).

Definition 1 (DGM)**.** *A Design-time Goal Model (DGM) is a directed graph, $(\mathcal{G}, \mathcal{R})$, where $\mathcal{G}$ is a set of goals (nodes), and $\mathcal{R}$ is the set of edges. $\mathcal{R}$ is partitioned into two subsets: Refinements and Influences edges. The latter are labeled with a strength value that indicates to what extent the fulfillment of one goal contributes to the fulfillment of another: strongly positive (++), weakly positive (+), weakly negative (-), strongly negative (--). The goal nodes $\mathcal{G}$ may belong in one of two*

disjoint groups: AND-nodes and OR-nodes. An AND-node (OR-node) is considered fulfilled if all nodes (at least one node) with refinement edges pointing to it are (is) fulfilled.

We require that 1) there be exactly one root node, and 2) every node have at most one incoming refinement edge and there are no refinement cycles (i.e., refinements form a forest, not a graph).

Leaf-level (unrefined) goals are called tasks. □

B. Monitoring meeting schedulers at runtime

We argue that DGMs are in some ways too abstract to enable runtime monitoring of requirements. We illustrate their limitations through examples from Figure 2, aiming to answer questions about the relationship between the behavior of an executing system and its DGM.

The first and most basic question at runtime is $\mathbf{Q}_1$: "Is observed behavior compliant with the system specification?" A follow-up question is $\mathbf{Q}_2$: "How does system behavior relate to the fulfillment of stakeholder (root-level) goals?" If some goals are violated, one may ask $\mathbf{Q}_3$: "Can the system switch to an alternative behavior to restore fulfillment?" These questions can be projected over temporal frames. For example, $\mathbf{Q}_4$: "What is the percentage of success for a given goal during the last month?", or $\mathbf{Q}_5$: "What is the trend for failure of a given goal in the last week?" These questions constitute the bread-and-butter of feedback loops for adaptive systems.

In this paper, we are primarily concerned with $\mathbf{Q}_1$ and $\mathbf{Q}_2$. The key to answer these questions is to observe that there is a conceptual link between DGMs and system behavior, in terms of *task classes and their instances*:

- A DGM provides a high-level *specification* for a system in terms of alternative sets of tasks that the system should implement to fulfill the top-level goal. Tasks, and goals in general, are expressed at the class level, e.g., "G_{18} : Mail Sent", "G_{19} : Response Processed".
- The execution of a system defines a trace of events involving task instances. We assume that these events form a total order that we call a **task trace**. The sequence $\tau_1 = \langle\, g_{18}.start, g_{18}.succeed, g_{19}.start \,\rangle$ is a task trace where task instance g_{18} of class G_{18} starts and subsequently succeeds (indicating that an individual e-mail was sent, e.g., to Bob for the executive board meeting next Friday), and then task instance g_{19} of class G_{19} starts (Bob's response is being processed).

The basic question $\mathbf{Q}_1$ can then be refined to: "Does a given task trace, consisting of events concerning task instances, comply with a DGM specification, which is about task/goal classes?" Although we can map task instances to design classes, key information concerning the runtime traces cannot be mapped to existing DGMs. Such information may distinguish between system success or failure. Specifically, DGMs do not express information concerning behavior (ordering) or possibilities for multiple instances.

Problem 1 (Behavior)**.** *The events concerning individual tasks may be interleaved in a trace. For instance, $\tau_2 = \langle\, g_{18}.start, g_{19}.start, g_{18}.succeed \,\rangle$ is a possible trace. However, DGMs do not impose any ordering among subgoals/tasks. Notice that, in τ_2, the response to an e-mail is processed before the original e-mail was sent! We would like to proscribe this.* □

Problem 2 (Multiple instances)**.** *Multiple instances of the same task class may appear in a single trace. For example, we may expect instances $g_{18\text{-}1}$ and $g_{18\text{-}2}$ of class "G_{18} : Mail Sent" to occur, one for each participant in the meeting. However, DGMs do not indicate if multiple instances of the same class can occur in one trace, or how many instances can be pursued concurrently.* □

Our solution to these two problems consists of enriching DGMs with annotations that describe behavior, e.g., the possible ways through which combinations of tasks can lead to the fulfillment of root-level goals, but without providing the full details of a program. The outcome of this enrichment process (described in Section III) is a Runtime Goal Model (RGM). RGMs enable more granular specifications of system behavior. However, they are still at the class level, while runtime traces are at the instance level.

Problem 3 (Goal state and life-cycle)**.** *Task and goal instances are inherently stateful. Inspired by prior work in goal models [17], [18], we capture this by viewing instances as having a* **history** *of state transitions ("events")—as in Figure 3—that lead to the current state. RGMs are not expressive enough to represent and reason about multiple stateful goal and task instances, because they consist of classes, not instances. For example, given trace $\tau_3 = \langle\, g_{18\text{-}1}.start, g_{18\text{-}1}.susp, g_{18\text{-}2}.start \,\rangle$, how would one show the existence of two instances of G_{18}, the former in state* waiting*, and the latter in state* running*?* □

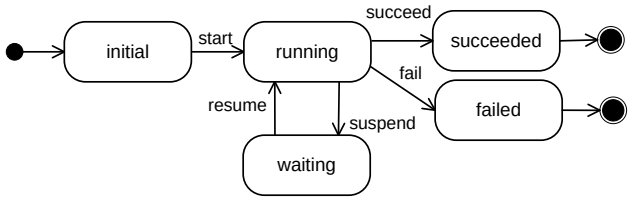

Fig. 3. State diagram for goal instances

We abbreviate event names $g.suspend$, $g.resume$ and $g.succeed$ to $g.susp$, $g.resm$ and $g.succ$. We use predicates initial?, failed?, running?, waiting?, succeeded? to determine what state a particular goal instance is in.

To overcome the third problem, we introduce a runtime artifact for representing and reasoning about requirements, which we call a Runtime Goal Instance (RGI). An RGI is a tree structure whose nodes are goal and task instances:

- An RGM can be considered as the *schema* for a collection of RGIs: the goal instances in the RGI are structured according to the refinement relations of the corresponding RGM. For example, an RGI for trace $\langle\, g_{18}.start \,\rangle$ would have a node for task instance g_{18}, and its parent would be an instance of goal G_{11}, because G_{11} is refined to G_{18} in the corresponding design-time goal model.

- RGIs are not necessarily isomorphic to RGMs, since they support multiple instances of the same class. For example, given trace $\langle\, g_{18\text{-}1}.start, g_{18\text{-}2}.start \,\rangle$, there would be two nodes for instances of G_{18} in the RGI, and they could share the same parent.
- Unlike nodes in DGMs and RGMs, nodes in an RGI keep track of the state reached by the task or goal instance. For instance, given trace $\langle\, g_{18}.start, g_{18}.succeed \,\rangle$, the node for g_{18} would be in state succeeded.

In Section IV, we will formalize RGIs and provide algorithms to derive and update them from the events in a task trace.

III. From design- to runtime goal models

We propose a process that analysts can carry out for enriching DGMs in order to obtain RGMs:

1) The analyst decides that some alternatives will not be implemented (Section III-A). This activity corresponds to the traditional design-time usage of goal models (e.g., [19], [20]).
2) Annotations concerning runtime behavior are added to specify sequencing and cardinality constraints among goal and task instances (Section III-B).
3) Some influence relationships are converted to temporal constraints on runtime traces (Section III-C).

We are introducing a *refinement* process for goal models. The activities in the process restrict the space of admitted behaviors that a system can exhibit while remaining compliant with its requirements. Moreover, no additional alternative is added during the refinement process. We provide a definition and example of an RGM in Section III-D, illustrating the results of the enrichment process using our running example.

Notation. Identifiers with a leading mathematical calligraphic typeset letter are sets (e.g., $\mathcal{G}$ is a set of goals). Identifiers with a leading uppercase letter are classes (e.g., G is a goal class). Identifiers with a leading lowercase letter are instances (e.g., g is a goal instance).

Terminology. Henceforth, a *trace* is a sequence of events concerning task and goal instances, while a *task trace* involves task instances only. Given a trace τ and an instance g, let $\mathsf{filterHistory}(g, \tau)$ be a function that returns the ordered subsequence of τ consisting of events of the form $g.\_\_$. A **trace τ is valid** if and only if for all instances c appearing in it, $filterHistory(c, \tau)$ causes only valid transitions in the diagram in Figure 3. Thus, $\langle\, g.start, g.succ, g'.start \,\rangle$ is valid, while $\langle\, g.succ, g.start \,\rangle$ and $\langle\, g.start, g.succ, g.start \,\rangle$ are not.

A. Making choices at design-time

Choosing among alternative requirements is an important part of an RE process. Such choices rely on the decision-making skills and the expertise of analysts, which compare various alternatives, in consultation with end-users. Our process suggests, as a preprocessing stage of DGMs, the use of traditional analysis techniques [21] to make choices and rule out some alternatives. Henceforth, we assume that this has been done, and that the DGM we are given reflects this pruning.

Example 1 (Design-time decisions)**.** *The analyst examines the identified partial negative influence relation between G_{10} and G_{17}: calling participants to obtain their timetables makes it more difficult to arrange a web meeting, for the meeting organizer software has no way to retrieve their calendars in digital format. The stakeholders suggest excluding the option of web scheduling a meeting (G_{17}). Moreover, the stakeholders do not like the idea of sending texts to their employees, due to recent privacy concerns in the organization. Thus, goal G_9 is removed too.* □

B. Adding annotations for constraining runtime traces

We have argued earlier for the need to analyze traces of individual tasks/goals, and events involving them. We present here a set of enrichments to a DGM that prescribe allowable behaviors of the system-to-be at runtime. We introduce "goal expressions", which allow (restricted) specification of the runtime behavior of a system in terms of how different system tasks and goals interact one with another.

Definition 2 (Goal expression)**.** *A goal expression is a formula E over a set of goal class identifiers that adheres to the syntax (akin to regular expressions) in the first column of Table I.* □

Each such expression denotes a set of well-formed traces, according to column three of Table I, extending the standard approach for regular expressions. The '$\frown$' operator used in Table I is concatenation of sequences; e.g., $\langle\, t_1.start \,\rangle \frown \langle\, t_1.succ \,\rangle$ equals $\langle\, t_1.start, t_1.succ \,\rangle$. The '$\odot$' operator is interleaving (shuffling) of sequences. A sequence Z is a shuffle of two sequences X and Y if and only if there is a way to partition the symbols of Z into two strings (preserving their relative order) to produce X and Y. For example, $\langle\, ev_1, ev_2, ev_3, ev_4 \,\rangle$ is a shuffle of $\langle\, ev_1, ev_3 \,\rangle$ and $\langle\, ev_2, ev_4 \,\rangle$. Moreover, since we are mostly interested in the successful completion/achievement of tasks/goals, we abbreviate G^{succ} to G in goal expressions.

As part of building an RGM, given a DGM, we associate with each non-leaf goal, a goal expression $\mathsf{annot}(G)$—called its *goal annotation*—that is intended to describe, for any instance g of G, the expected behavior of the instances of subgoals of G. For example, $\mathsf{annot}(G)$=$G_1; G_2$ describes that one instance of G_1 shall occur successfully, followed by one successful instance of G_2.

We require that every annotation complies with the following rules:

- A_1. Each subgoal appears in at least one allowed trace. Annotations are added to a pruned goal tree wherein all tasks have to be implemented by the system. If a subgoal appears in no trace, then a kept design alternative (Section III-A) will never be used by the system.
- A_2. Annotations should not allow for empty traces. An empty trace is trivially satisfied. Therefore, the refined goal would be satisfied without doing any action, which contradicts the refinement to subgoals or tasks;
- A_3. The annotations for AND-refined goals should allow for at least one trace of children instances where all subgoals appear. This rule preserves the intuitive meaning of refining an entity to its constituent parts;

TABLE I. SYNTAX AND DENOTATION OF GOAL EXPRESSIONS

Expression E	Meaning	Denotation $\mathcal{L}(E)$ as set of traces
$skip$	Do nothing	$\{\langle\,\rangle\}$
$E_1\ ;\ E_2$	Sequential occurrence	$\{u^\frown w \mid u \in \mathcal{L}(E_1),\ w \in \mathcal{L}(E_2)\}$
$E_1 \mid E_2$	Alternation (exclusive choice)	$\mathcal{L}(E_1) \cup \mathcal{L}(E_2)$
$opt(E)$	E is optional	$\mathcal{L}(skip \mid E)$
E^+	One or more sequential occurrences of E (iterated concatenation)	$\mathcal{L}(E \mid E;E \mid (E;E);E \mid \dots)$
G^{succ}	An instance of G starts and terminates in state succeeded	$\bigcup_{g \in G} \mathcal{L}(\langle\, g.start\,\rangle; opt(\langle\, g.susp, g.resm\,\rangle^+); \langle\, g.succ\,\rangle)$
G^{fail}	An instance of G starts and terminates in state failed	$\bigcup_{g \in G} \mathcal{L}(\langle\, g.start\,\rangle; opt(\langle\, g.susp, g.resm\,\rangle^+); \langle\, g.fail\,\rangle)$
$try(G)?E_1 : E_2$	If an instance of G succeeds, an E_1; otherwise, an E_2	$\mathcal{L}(G^{succ};E_1) \cup \mathcal{L}(G^{fail};E_2)$
$E_1\ \#\ E_2$	Interleaved occurrence of E_1 and E_2	$\{u \odot w \mid u \in \mathcal{L}(E_1),\ w \in \mathcal{L}(E_2)\}$
$E^\#$	One or more instances of E occurring concurrently (iterated shuffle)	$\mathcal{L}(E \mid E\#E \mid (E\#E)\#E \mid \dots)$

Below, we illustrate our goal annotations through examples.

Two fundamental system behaviors are sequentiality and interleaving (see Problem 1). Our language supports them through the ';' and '#' operators, respectively.

Example 2 (Sequencing and Shuffle)**.** *Goal G_2 (meeting scheduled) in Figure 2 is refined to G_6 (timetables collected), G_7 (schedule chosen), and G_8 (location chosen). A possible goal annotation for G_2 may specify that an instance of G_6 shall be achieved first, followed by (;) the interleaved fulfillment (#) of schedule allocation and location choice. Formally,* $\mathsf{annot}(G_2) = G_6;(G_7\#G_8)$. □

Our language supports alternatives (annotation '|'), to express that the system can exhibit different behaviors. Moreover, it also supports an annotation, *try*, in the spirit of if-then-else. It specifies that the system should try to achieve a goal and, depending on the success of failure, different behaviors are expected.

Example 3 (Alternatives and Try)**.** *Take the subgoals of G_6 that were not pruned in Section III-A: calendars can be collected by calling participants (G_{10}), e-mailing them (G_{11}), and retrieving their calendars online (G_{12}). One may specify that the system should either send e-mails (G_{11}) or retrieve online calendars (G_{12}). Moreover, if e-mails do not work out due to some reason, participants are called (G_{10}). Formally,* $\mathsf{annot}(G_6) = (try(G_{11})\ ?\ skip\ :\ G_{10}) \mid G_{12}$. □

Sometimes goals are refined in a way that subgoals could have more than one instance. Our language supports multiple instantiation (see Problem 2) through iterated concatenation ('E^+')—multiple instances of E in sequence—, iterated shuffle ('$E^\#$')—multiple instances of E interleaved—, and the optional operator ($opt(E)$)—zero or one instance of E.

Example 4 (Multiple instances)**.** *Consider goal G_{11} (Participants e-mailed). Its subgoals mail sent (G_{18}) and response processed (G_{19}) can be related through iterated shuffle, i.e., they have to be repeated multiple times (for each participant), and their instances can be interleaved (i.e., there is no need to wait for a response before contacting another participant). Moreover, the system shall accommodate the fact that a response may not arrive (hence, G_{19} is optional). Formally,* $\mathsf{annot}(G_{11}) = (G_{18};\ opt(G_{19}))^\#$. □

C. From influence relationships to temporal constraints

There is no consensus on how influence/contribution relationships shall be used at runtime. Some approaches (e.g., [10], [18]) use contributions to softgoals in order to choose among alternative sets of tasks. However, in DGMs, influence relations can apply to (hard) goals and tasks too.

We propose that the analyst decides which influence relationships are retained at runtime, and has to interpret these relationships as two types of temporal constraints:

- $\mathsf{requires}(G_1, G_2)$ is obtained from a positive influence from G_2 to G_1; it indicates that an instance of G_1 can occur only if an instance of G_2 has already occurred.
- $\mathsf{prevents}(G_1, G_2)$ is obtained from a negative influence from G_1 to G_2; it indicates that the occurrence of an instance of G_1 prevents that of an instance of G_2.

Since an RGI can contain many instances of an individual RGM goal class, we need to specify between which instances the constraint applies to. We want to ensure that only related goal instances, i.e., being pursued in the same *scope*, are affected. To do that, given a constraint between G_1 and G_2, we determine the lowest common ancestor G of the two goal classes in the RGM, and we define that the constraint applies only to instances of G_1 and G_2 in the subtree of the same instance of G. We illustrate this intuition in the following examples.

Example 5 (Requires)**.** *We can map* $\mathsf{influences}(G_{12}, G_{15}, +)$ *to* $\mathsf{requires}(G_{15}, G_{12})$*, indicating that if the system tries to book a convention center, it must have already attempted to retrieve the participants' online calendars. The lowest common ancestor of G_{12} and G_{15} is G_2 (meetings scheduled). Thus, the constraint applies only to convention center bookings and calendars retrievals that relate to the same instance of G_2 (i.e., to the same meeting scheduling).* □

Example 6 (Prevents)**.** *The relation* $\mathsf{influences}(G_{14}, G_{23}, --)$ *can be mapped to* $\mathsf{prevents}(G_{14}, G_{23})$*, indicating that if the system tries to execute the algorithm, it should not attempt to cancel another meeting afterwards. Again, the lowest common ancestor is G_2, and, thus, the constraint applies in the context of the scheduling of one individual meeting.* □

D. Runtime Goal Model

An RGM is a refined DGM wherein all goals are to be fulfilled by the system, and a set of annotations specifies how goal instances may be sequentialized at runtime. In an RGM, all refined goals are provided with an annotation using the language introduced in Section III-B, and requires and prevents relationships are added as explained in Section III-C. Figure 4 shows an RGM for the DGM of Figure 2.

Definition 3 (RGM)**.** *Given a DGM $M = (\mathcal{G}, \mathcal{R})$, an RGM for M, written as R_M, is a 4-tuple $(M, \mathsf{annot}(), \mathsf{requires}(), \mathsf{prevents}())$ where*

- $\mathsf{annot}()$ *is a total function from $\{ G \mid G \in \mathcal{G}, G \in$ AND_nodes $\cup$ OR_nodes $\}$ to goal expressions over $\mathcal{G}$ such that $\mathsf{annot}(G)$ complies with the rules A_1–A_3 in Section III-B;*
- $\mathsf{requires}(G_1, G_2)$ *is true only if* $\mathsf{influences}(G_2, G_1, s) \in M$*, for $s \in \{+, ++\}$;*
- $\mathsf{prevents}(G_1, G_2)$ *is true only if* $\mathsf{influences}(G_1, G_2, s) \in M$ *with $s \in \{-, --\}$.* □

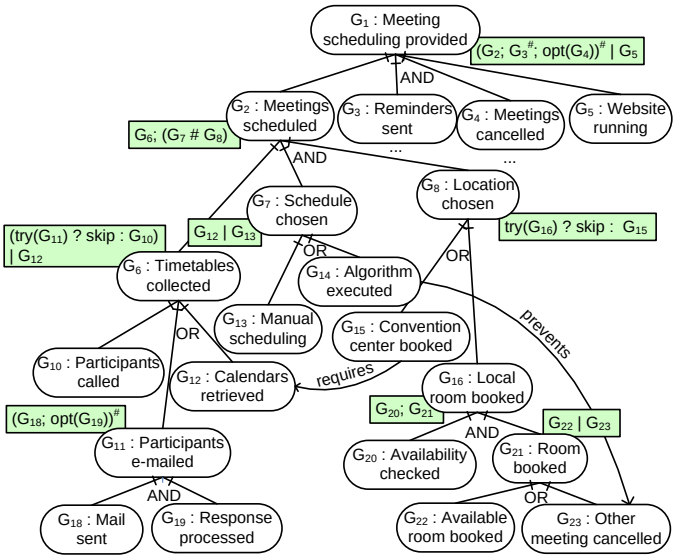

Fig. 4. Visualization of an RGM based on the DGM of Figure 2

IV. System diagnosis with RGMs and RGIs

We provide algorithms to interpret a task trace from a running system in terms of its goals. Guided by a given RGM as input, our algorithms construct an RGI from the task trace in an *incremental* manner. Being incremental means that, given an RGI and an event, the RGI is updated without re-processing the events that have occurred before the latest. Before presenting our algorithms in Section IV-B, we formalize RGIs (which we motivated and informally described in Section II) in Section IV-A.

A. Runtime Goal Instances (RGIs)

Intuitively, given a task trace τ and an RGM, an RGI for them is a tree whose interior nodes are instantiations of goals in the RGM which "explain" why tasks are being carried out in terms of the original DGM. The DGM loosely restricts the kinds of instances that can appear as children of g in the tree: they must be instances of the refinements of g's goal type. The annotations in the RGM further constrain the histories of children, and how they are intertwined. Thus, we need to associate with a node a trace of its children's behavior (this cannot be reconstructed from the history of each individual child, because we allow interleaving). Formally:

Definition 4 (RGI)**.** *A tree i_M is an RGI for an RGM $R_M = (M, \mathsf{annot}(), \dots)$ if, for every node g in i_M:*

- $\mathsf{id}(g)$ *is a unique goal/task instance identifier;*
- $\mathsf{type}(g)$ *is a goal/task class in M;*
- $\mathsf{children}(g)$ *is the sequence (possibly empty) of the child nodes of g in the tree;*
- $\mathsf{state}(g)$ *is a state in the lifecycle diagram in Figure 3;*
- $\mathsf{chnTrace}(g)$ *is a trace of events involving only the instances in* $\mathsf{children}(g)$*;*
- *the root of the tree i_M has type the root goal of M.*

An RGI i_M is said to be **valid** *if for every node g in it:*

1) $\mathsf{type}(g)$ *is the goal or task class in M of which g is an instance;*
2) $\mathsf{chnTrace}(g)$ *must be a valid trace (see Section III);*
3) *all the instances c in* $\mathsf{children}(g)$ *must appear in at least one event in* $\mathsf{chnTrace}(g)$*;*
4) *for every node g' in* $\mathsf{children}(g)$*,* $\mathsf{type}(g')$ *appears in* $\mathsf{annot}(\mathsf{type}(g))$*;*
5) $\mathsf{chnTrace}(g)$ *is an initial subsequence of some trace in* $\mathcal{L}(\mathsf{annot}(\mathsf{type}(g)))$*;*
6) *for every node g' in* $\mathsf{children}(g)$*, the history* $\mathsf{filterHistory}(g', \mathsf{chnTrace}(g))$ *must lead to* $\mathsf{state}(g')$ *according to the transition diagram in Figure 3.* □

B. Interpreting task traces on an RGI

We provide algorithms to *construct and update an RGI* based on a task trace τ and a specific RGM R_M. Our algorithms propagate the successive events in τ in a bottom-up fashion. The output (an RGI) is the artifact that enables answering questions $\mathbf{Q}_1$ and $\mathbf{Q}_2$ in Section II.

We propose deterministic algorithms, which interpret the events in τ in terms of a unique instance i_M of R_M. In general, our annotations may lead to different interpretations of a given τ. For example:

- Given $\mathsf{annot}(G) = G_2^+; G_2^+$, and given a trace $\langle\, g_2.start,\ g_2.succ,\ g_2'.start,\ g_2'.succ,\ g_2''.start,\ g_2''.succ \,\rangle$, we cannot tell if g_2' belongs in the first or second G_2^+ in $\mathsf{annot}(G)$.
- Given $\mathsf{annot}(G) = (G_1 \# G_1)$, $\mathsf{annot}(G_1) = G_2; G_2$, and given a trace $\langle\, g_2.start, g_2'.start \,\rangle$, we cannot tell if g_2, g_2' belong to the same instance of G_1 or not.

The issue is well-known in AI plan recognition, and solutions exist [22]. We assume that proper disambiguation strategies are adopted to (i) identify a unique parent for a given task or goal instance, and (ii) determine the termination

TABLE II. STATE PROPAGATION FOR AN OCCURRENCE e OF A COMPOSITE GOAL EXPRESSION E, STARTING FROM ITS SUBEXPRESSIONS

Rule	e.event	$\mathbf{E = E_1; E_2}$	$\mathbf{E = E_1 \# E_2}$	$\mathbf{E = E_1 \mid E_2}$	$\mathbf{E = (E_1)^+}$	$\mathbf{E = (E_1)^\#}$
R_1	*succ*	The trace [..., event] is a final string in the language defined by the expression				
R_2	*fail*	The trace [..., event] is not recognized by the language defined by the expression				
R_3	*start*	$e_1.start$	$e_1.start$ or $e_2.start$	$e_1.start$ or $e_2.start$	$e_1.start$	$e_1.start$
R_4	*susp*	$e_1.susp$ or $e_2.susp$ or $e_1.succ$	([not running?(e_2)] $e_1.susp$ or $e_1.succ$) or ([not running?(e_1)] $e_2.susp$ or $e_2.succ$)	$e_1.susp$ or $e_2.susp$	$e_1.susp$ or $e_1.succ$	$[\mid j.$ not running?($e_{1\text{-}j}$)$\mid = 1]$ $e_{1\text{-}i}.susp$ or $e_{1\text{-}i}.succ$
R_5	*resm*	$e_1.resm$ or $e_2.resm$ or $e_2.start$	$e_1.resm$ or $e_1.start$ or $e_2.resm$ or $e_2.start$	$e_1.resm$ or $e_2.resm$	$e_1.resm$ or $e_1.start$	$e_1.resm$ or $e_1.start$

of constructs that involve multiple instances (E^+ and $E^\#$), i.e., if more occurrences are expected or not.

In the following, Algorithms 2–5 have as implicit parameters the RGM *rgm*, the RGI *rgi*, and the task trace τ.

RGI monitoring loop (Algorithm 1). This algorithm is launched when the system is deployed, and it coordinates the interpretation of retrieved events (lines 3–4) in terms of an RGI. The algorithm terminates when the root goal of the RGI is in a final state (succeeded or failed). After adding the retrieved event *ev* to the trace (line 5), it checks if *ev* is valid (i.e., if it updates a task instance in accordance with the transition diagram in Figure 3). If the trace is invalid, a notification is sent (either to the analyst or to the system itself), and the event is ignored (lines 7–8). Notice that one may define an alternative monitoring loop that throws an exception and terminates. Choosing on how to react to unexpected inputs depends on domain-specific factors. If the trace is valid (line 9), the latest event is processed by PROCESSEVENT (Algorithm 2).

Algorithm 1 Requirements monitoring loop

```
REQMONITORLOOP(RGM rgm)
1  RGI rgi ← ∅
2  TaskTrace τ ← ⟨ ⟩
3  while state(GETROOT(rgi)) ∉ {succeeded, failed}
4  do Event ev = (id, G, act) ← WAITNEXTEVENT()
5     τ ← τ^ev
6     if not ISEVENTVALID(id, act, rgi)
7        then NOTIFY(invalid_event, τ, rgi, ev)
8             continue
9     PROCESSEVENT(ev)
```

Preliminary event processing (Algorithm 2). This makes a preliminary analysis of an event, then redirects further analysis to other algorithms. Each event consists of the task/goal instance identifier (*id*), the type of the instance (*G*), and the occurred action (*act*). If a corresponding goal instance g already exists in the RGI (line 1–2), state(g) is updated. Then, the event is checked against the temporal constraints expressed in the RGM (line 3). The event is propagated to the parent (lines 4–6) by calling PROPAGUP (Algorithm 4). Line 5 handles root goals, which require no further up-propagation. If no corresponding goal instance exists in the RGI, BUILDTREE (Algorithm 3) creates a goal instance with identifier *id*, and adds the ancestors in the tree, if needed.

Bottom-up propagation rules (Table II). Our algorithms construct an RGI in a bottom-up fashion. The events in the task trace define the task instances in the RGI. Then, these events affect other goal instances, following the path from leafs to root in the RGM. The propagation rules define when and how an event that affects a component of a composite expression affects an instance e of the composite expression. Our algorithms start from a valid chnTrace(), and determine how the latest event in the trace affects the composite expression. The rules are formalized in Table II, are briefly explained below, and are illustrated in Section V:

Algorithm 2 Processing a goal event

```
PROCESSEVENT(Event ev = (id, G, act))
1  if ∃g ∈ rgi. type(g) = G ∧ id(g) = id
2     then state(g) ← GETTRANSITIONTARGET(act)
3          CHECKTEMPCONSTRS(ev)
4          parent ← GETPARENTINST(g, rgi)
5          if parent = NIL then return
6          PROPAGUP(gev, parent)
7     else GoalInst gi ← NEWGOALINST(id, G, ∅, running, ⟨ ⟩)
8          BUILDTREE(gi)
```

- R_1. e.*succ* if the trace is a final string in $\mathcal{L}(E)$;
- R_2. e.*fail* if the trace is not an initial substring in $\mathcal{L}(E)$;
- R_3. e.*start* as soon as a subexpression instance starts;
- R_4. e.*susp* when there is no running subexpression instance; either they are all waiting, or one instance is succeeded, and the following has not started yet;
- R_5. e.*resm* when a a component of the subexpression is resumed or started (if all components are terminated, but the expression is not completed yet).

Constructing an RGI (Algorithm 3). This constructs an RGI tree from a goal instance g: it adds g (line 1) and its ancestors to the RGI, until it finds an existing ancestor or it reaches a root goal. The algorithm looks up the parent goal class in the RGM (line 3). If such class does not exist, the algorithm has encountered the root (line 4), it checks temporal constraints on event $g.start$, and returns. Otherwise, the algorithm looks for the parent of g in the RGI (line 7). If this parent *g-par* exists, g is added as a child of *g-par* (line 9), and the event $g.start$ is propagated bottom-up by Algorithm 4 (line 10). If the RGI contains no parent for g, a goal instance for the parent is created, by generating a unique identifier, specifying g as a child, and $g.start$ as children trace (line 11–13). Then, BUILDTREE is recursively called on such parent (line 14). If an instance of G' cannot start with g (R_3 in Table II is evaluated on annot(G')), then PROCESSEVENT is

Algorithm 3 Constructing an RGI tree

```
BUILDTREE(GoalInst g)
 1  rgi ← rgi ∪ {g}
 2  Event ev ← (id(g), type(g), start)
 3  G' ← GETPARENTCLASS(G, rgm)
 4  if G' = NIL
 5     then CHECKTEMPCONSTRS(ev)
 6          return
 7  GoalInst g-par ← GETPARENTINST(g, rgi)
 8  if g-par ≠ NIL
 9     then children(g-par) ← children(g-par)^g
10          PROPAGUP(ev, g-par)
11     else id' ← GENIDFORPARENT(g, G')
12          GoalInst gi ← NEWGOALINST(id', G', ⟨ g ⟩,
13                                    running, ⟨ ev ⟩)
14          BUILDTREE(gi)
15          if not CANSTARTWITH(annot(G'), ev)  // Rule R3
16             then PROCESSEVENT((id', G', fail))
17  CHECKTEMPCONSTRS(ev)
```

called on a failure event for *id'* (lines 15–16). Lastly, temporal constraint are checked (line 17).

Bottom-up event propagation (Algorithm 4). This enacts the propagation rules R_1–R_5 in Table II, by evaluating if the latest event added to chnTrace(g) affects the goal instance g. First, the children trace is updated by adding the event. Then, if g was already in a final state, the algorithm returns (line 2). Otherwise, the rules of Table II are verified (lines 5–16). The rules for success and failure are examined first, then those for suspension and resume. The rule for *start* is used by Algorithm 3, which creates goal instances in state running. If no rule is applicable, the state of g is not affected (line 17). Otherwise, the proper event to reach the new state of g is processed (lines 18–19).

Algorithm 4 Bottom-up propagation of an event

```
PROPAGUP(Event ev, GoalInst g)
 1  chnTrace(g) ← chnTrace(g)^ev
 2  if state(g) ∈ {succeeded, failed} then return
 3  State newState ← NIL
 4  Annotation ann-g ← annot(type(g))
 5  if ISFINALSTRINGOF(chnTrace(g), ann-g)
 6     then newState ← succeeded  // Rule R1
 7     else if NOTINITSTRINGOF(chnTrace(g), ann-g)
 8             then newState ← failed  // Rule R2
 9  if newState = NIL
10     then switch state(g)
11             case running :
12                  if CANSUSP(state(g), chnTrace(g), ann-g)
13                     then newState ← waiting  // Rule R4
14             case waiting :
15                  if CANRESM(state(g), chnTrace(g), ann-g)
16                     then newState ← running  // Rule R5
17  if newState = NIL then return
18  Event ev' ← GETEVENTBETWEEN(state(g), newState)
19  PROCESSEVENT(ev')
```

Verifying temporal constraints (Algorithm 5). This checks if an occurred event violates any of the temporal constraints defined in the RGM, and it notifies all detected violations. For each temporal constraint *tc* in the RGM (line 3), the algorithm determines the lowest common ancestor *lca-c*, and gets the instance of *lca-c* that is an ancestor of g in the RGI (lines 4–5). If *tc* is of type requires (lines 7–10), and the processed event concerns a goal/task instance whose type matches the first argument of *tc*, then the algorithm verifies that there are no instances of the second argument class in the subtree rooted in *lca-i*. A similar process applies for prevents constraints (lines 11–14). Finally, violations for constraints are notified (line 15).

Algorithm 5 Checking temporal constraints

```
CHECKTEMPCONSTRS(Event ev = (id, G, act))
 1  TempConstraint { } constrs ← GETTEMPCONSTRAINTS(rgm)
 2  ConstrViolation { } viol ← ∅
 3  for each tc = (type, arg1, arg2) ∈ constrs
 4  do GoalClass lca-c ← LOWESTCOMMANC(arg1, arg2)
 5     GoalInst lca-i ← GETANCESTOROFTYPE(G, lca-c)
 6     switch type
 7        case requires :
 8             if G ≠ arg1 then continue
 9             if ∅ = GETISTANCESINSUBTREE(arg2, lca-i)
10                then viol ← viol ∪ (tc, ∅, G)
11        case prevents :
12             if G ≠ arg2 then continue
13             if arg1-i ← GETISTANCESINSUBTREE(arg1, lca-i)
14                then viol ← viol ∪ (tc, arg1-i, G)
15  for each v ∈ viol do NOTIFY(constr_viol, τ, rgi, ev, v)
```

V. EVALUATION THROUGH SCENARIOS

We illustrate our algorithms through incremental scenarios of a running meeting scheduler whose RGM is that in Figure 4. We summarize each scenario, then describe how the algorithms from Section IV-B are applied to update the RGI as the task trace τ unfolds. To keep our illustrations readable, we focus on the RGM subtree rooted in G_2.

Scenario 1 (Start)**.** *From trace $\tau_1 = \langle\, g_{18\text{-}1}.start \,\rangle$, a task instance of class G_{18} (Mail sent) is created, indicating that the system has started sending an e-mail. Since the RGI is initially empty, this event cascades up, leading to the RGI rooted in $g_{2\text{-}1}$ in Figure 5. All task/goal instances in RGI are in state* running. □

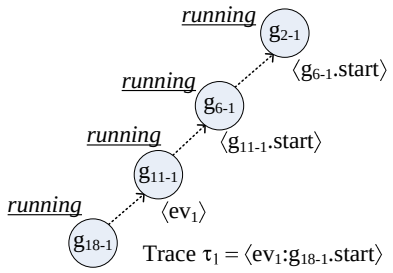

Fig. 5. RGI for the RGM of Figure 4 for the task trace τ_1. Each node is labeled with identifier and type ($g_{1\text{-}2}$ indicates instance 2 of goal class G_1), and it is annotated with the current state (underlined, in italic) and the children trace (between angular brackets ⟨ ⟩)

We explain how our algorithms process τ_1 to obtain the RGI in Figure 5. Algorithm 1 retrieves event ev_1 (line 4) and adds it to τ. The event is valid, and is thus processed by Algorithm 2. There is no node for $g_{18\text{-}1}$ in *rgi*, so BUILDTREE is called. Algorithm 3 adds $g_{18\text{-}1}$ to *rgi* in state running. Since there is no parent for $g_{18\text{-}1}$ in *rgi*, a goal instance $g_{11\text{-}1}$ is

created in state running, with $g_{18\text{-}1}$ as a child, and with children trace $\langle\, g_{18\text{-}1}.start \,\rangle$, and Algorithm 3 is recursively invoked on $g_{11\text{-}1}$. This process is repeated till the root goal $g_{2\text{-}1}$ is created. All goal instances are set in state running by Algorithm 3, line 12. None of the *start* events violates temporal constraints.

Scenario 2 (Failure)**.** *From trace τ_2, a second instance of G_{18} ($g_{18\text{-}2}$) starts (a new e-mail is being sent), and fails. This failure implies a failure in $g_{11\text{-}1}$ too, while $g_{6\text{-}1}$ is still in state* running*, since the RGM supports other alternatives for collecting timetables. The root goal $g_{2\text{-}1}$ remains in state* running *too. This scenario is depicted in Figure 6.* □

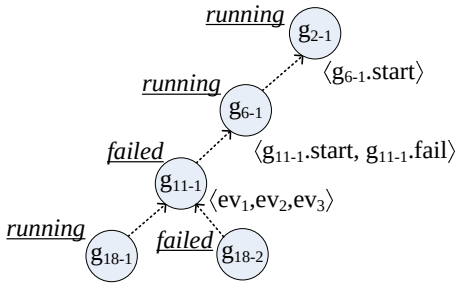

Fig. 6. RGI for the RGM of Figure 4 for the task trace τ_2

Algorithm 1 processes event ev_2 in τ_2. The processing of ev_2 leads to the creation of a node $g_{18\text{-}2}$ in *rgi*. This time, however, algorithm BUILDTREE (called by PROCESSEVENT) finds a parent goal instance for $g_{18\text{-}2}$ in *rgi*: $g_{11\text{-}1}$. Therefore, $g_{18\text{-}2}$ is added to children($g_{11\text{-}1}$), and PROPAGUP is called to propagate the state of $g_{18\text{-}2}$ to $g_{11\text{-}1}$. The state of $g_{11\text{-}1}$ (and of its ancestors) is unaffected by ev_2.

Event ev_3 leads to an update of the already existing task instance state($g_{18\text{-}2}$) through PROCESSEVENT. PROPAGUP is called on $g_{18\text{-}2}.fail$; chnTrace($g_{11\text{-}1}$) is $\langle\, ev_1, ev_2, ev_3 \,\rangle$, which is not an initial substring of annot(G_{11})= $(G_{18}; opt(G_{19}))^{\#}$. Thus, $g_{11\text{-}1}$ switches to state failed. The parent $g_{6\text{-}1}$ is not affected, because annot(G_6)= $(try(G_{11})\ ?\ skip\ :\ G_{10})\mid G_{12}$ supports the failure of instances of G_{11} through the *try* annotation. An instance of G_{10} is now expected.

Scenario 3 (Alternative)**.** *From trace τ_3, the system uses an alternative function and successfully calls participants (ev_4, ev_5). In turn, this leads* state($g_{6\text{-}1}$) *to* succeeded*. The corresponding RGI is depicted in Figure 7.* □

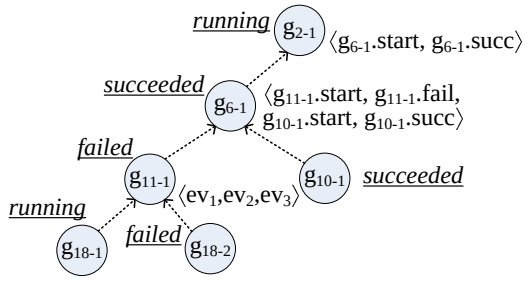

Fig. 7. RGI for the RGM of Figure 4 for the task trace τ_3

Let us skip the processing of ev_4, which is analogous to ev_2. The success of $g_{10\text{-}1}$ (ev_5), instead, leads to chnTrace($g_{6\text{-}1}$) as in Figure 7. This is recognized by Algorithm 4 as a final string for annot(G_6)= $(try(G_{11})\ ?\ skip\ :\ G_{10})\mid G_{12}$. Thus, event $g_{6\text{-}1}.succ$ leads to state($g_{6\text{-}1}$)=succeeded. PROPAGUP is called on the parent $g_{2\text{-}1}$. The state of $g_{2\text{-}1}$ is not affected, because annot(G_2)= $G_6; (G_7 \# G_8)$ requires the interleaved execution of G_7 and G_8 to be completed.

Scenario 4 (Suspension)**.** *From trace τ, events ev_6 and ev_7 indicate that a manual meeting scheduling ($g_{13\text{-}1}$) starts and is suspended. ev_7 affects all ancestors of $g_{13\text{-}1}$, as the only active task in the system was $g_{13\text{-}1}$. See Figure 8.* □

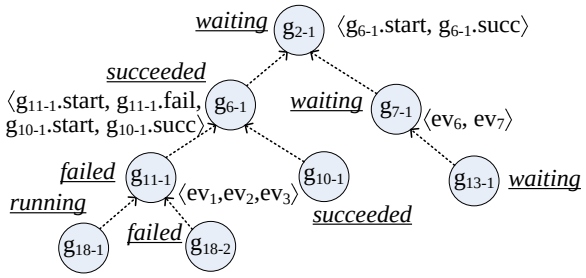

Fig. 8. RGI for the RGM of Figure 4 for the task trace τ_4

Event ev_6 leads to the creation of a node for $g_{13\text{-}1}$ (Algorithm 2 processes the event and calls BUILDTREE). In turn, BUILDTREE generates a node for $g_{7\text{-}1}$, and links it to the root $g_{2\text{-}1}$. Event ev_7 suspends $g_{13\text{-}1}$. Then, PROPAGUP propagates the event suspension to produce the state waiting till the root. Indeed, annot(G_2)= $G_6; (G_7 \# G_8)$, an instance of G_6 has terminated successfully, and, so far, the interleaving of G_7 and G_8 has only one instance ($g_{7\text{-}1}$), which has been suspended.

Analyzing an RGI. We can now answer $\mathbf{Q}_1$ by simply looking at the state of an RGI's root (failed means that the trace violates the specification). In our scenarios, the trace complies with the specification. We can answer $\mathbf{Q}_2$ by investigating the state of subgoals (e.g., state($g_{11\text{-}1}$) is failed). Answering $\mathbf{Q}_3$—is there an alternative behavior?—requires checking if the failure of a task instance leads the root goal instance to state failed (in that case, there is no possible alternative). In our scenarios, the failure of $g_{18\text{-}2}$ did not result in the failure of the root, as an alternative task was executed to achieve $g_{6\text{-}1}$. The framework needs extensions to answer more sophisticated questions, including $\mathbf{Q}_4$ and $\mathbf{Q}_5$.

Proof-of-concept. We have implemented a Prolog program that determines if a trace is recognized by an annotation. This implementation is essentially an extended regular expression recognizer that supports our annotation language. Recognition is a key building block of our proposal. The implementation has helped us to refine our goal annotations and to understand the subtleties of constructing RGIs from traces. This implementation is a first step towards the creation of a full-fledged framework for creating, updating, and querying RGIs.

VI. Related work

We review some approaches that use goal models at runtime for monitoring or planning/guiding system behavior as well as methods to reason about enriched goal models.

Operational goals in KAOS can be used for runtime requirements monitoring [23]. Cheng et al. [24] propose a method to introduce requirements about adaption into KAOS goal models [14] using threat analysis to identify those goals that should be mitigated by introducing adaptive functions. Fuzzy goals [25] go beyond KAOS's crisp goals, tolerating small deviations (e.g., small delays in deadlines). While these approaches are useful at design-time for eliciting and specifying requirements about adaptation, the resulting artifacts do not explicitly refer to stateful goal instances that can be monitored from system traces.

Different authors [26], [27] suggest using *i** goal models to support the design of high-variability systems. The system chooses among existing options by evaluating the strategy that provides the best contribution to softgoals. Welsh et al. [9] also include the notion of *claim* (taken from the NFR framework [28]) to represent the assumptions that modelers makes when indicating the strength of a contribution from a task or hardgoal to a softgoal. These claims are monitored at runtime and, if proven false, they can be reconsidered (e.g., the contribution may be removed or its strength may be inverted). Assumptions in requirements models have also been studied by Ali et al. [29]. These works choose an adequate level of abstraction (goals) for modeling adaptive systems. However, they fail to distinguish between goal classes from goal instances, and provide no foundations for dealing with goal instances as stateful entities.

Awareness requirements [30] are requirements about the state of other requirements. They have been proposed for designing adaptive systems that fulfill requirements such as "Requirement R must never fail three times in a row". Awareness requirements do use runtime goal models, but thus far have only considered single isomorphic instances of DGMs. Such an approach can benefit from our proposal.

Morandini et al. [10] show how detailed-design goal models can be mapped to rational agent programming languages. In subsequent work [17], they investigate the life-cycle of goals at runtime. Our annotation language resembles the expressions in [12], which support mapping goal models to agent specifications in ConGolog. However, they do not consider stateful goals. We have been inspired by these approaches, but go beyond by proposing a method to derive RGMs from high-level DGMs, and runtime mechanisms that combine annotations and state inference.

DeLoach and Miller [31] differentiate between goal classes and instances for adaptive systems, and they rightly observe that instances have an associated state. Similar observations arose in our previous work [32], [33], [18], where we implicitly differentiated between classes and instances by using parametric goals (different instances differ in the actual values assigned to the parameters). In this paper we go beyond by providing an annotation language for expressing how goal instances shall be interleaved, also introduce RGI artifacts that represent the behavior of the system in terms of its requirements.

Artificial Intelligence planners have been used to find satisfactory system behavior during design-time. Liaskos et al. [13] enrich goal models with precedence annotation and preferences, determining the most suitable behavior based on this. Sykes et al. [34] map goals to components, and combine the actions that these components can do via planners. Although DGMs are enriched with behavioral information, such extensions are not as expressive as our annotations in Section III-B. Moreover, these approaches are aimed at ensuring system design allows for plans which sufficiently achieve goals and preferences, and are not aimed for runtime monitoring or analysis.

In Section II, we have shown how our problem differs from design-time reasoning on DGMs [16], [21], which works at the class level and takes a set of goals as input, as opposed to a sequential trace of events. Other approaches perform reasoning on DGMs extended with behavioral information. KAOS goal models use temporal logics to describe behavior. They have been mapped to labeled transition systems [35], [36] to support consistency checking, identification of implicit requirements, and animation. A different approach enriches goal models with scenarios, to obtain verifiable modal transition systems [37]. However, the focus of these approaches is mainly on formal verification, as opposed to constructing a runtime requirements object that supports questions such as $\mathbf{Q}_1$.

VII. Discussion and outlook

We have proposed a modeling language for runtime goal models, to be used for monitoring and diagnosis of adaptive software systems. Our proposal introduces supports goal models at a class and instance level, also associates state, behavioral and historical information to goal instances. It also supports reasoning about the state and behavior of a system relative to its requirements.

For the future, our proof-of-concept implementation needs to be supplemented with a more robust and comprehensive version that can be evaluated for scalability. We also plan to revisit earlier work that uses runtime goal models (e.g., [5], [30], [18]) and revise algorithms for monitoring and diagnosis based on the results presented herein. Another interesting future direction for this research concerns the inclusion of contexts in RGMs, based on our previous work with contextual goal models for adaptive systems [33], [38].

Acknowledgment

This work has been partially supported by the ERC advanced grant 267856 for a project titled "Lucretius: Foundations for Software Evolution" (http://www.lucretius.eu), and by the Natural Sciences and Engineering Research Council (NSERC) of Canada through the Business Intelligence Network.

References

[1] S. Fickas and M. S. Feather, "Requirements Monitoring in Dynamic Environments," in *Proceedings of the 2nd IEEE International Symposium on Requirements Engineering (RE)*. IEEE Computer Society, 1995, pp. 140–147.

[2] D. Cohen, M. S. Feather, K. Narayanaswamy, and S. S. Fickas, "Automatic Monitoring of Software Requirements," in *Proceedings of the 19th International Conference on Software Engineering (ICSE)*. ACM, 1997, pp. 602–603.

[3] K. Mahbub and G. Spanoudakis, "A Framework for Requirements Monitoring of Service Based Systems," in *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC)*. ACM, 2004, pp. 84–93.

[4] W. N. Robinson, "A Requirements Monitoring Framework for Enterprise Systems," *Requirements Engineering*, vol. 11, no. 1, pp. 17–41, 2006.

[5] Y. Wang, S. McIlraith, Y. Yu, and J. Mylopoulos, "Monitoring and Diagnosing Software Requirements," *Automated Software Engineering*, vol. 16, no. 1, pp. 3–35, 2009.

[6] N. Bencomo, J. Whittle, P. Sawyer, A. Finkelstein, and E. Letier, "Requirements Reflection: Requirements as Runtime Entities," in *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 199–202.

[7] J. Kramer and J. Magee, "Self-Managed Systems: an Architectural Challenge," in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2007, pp. 259–268.

[8] B. Cheng, R. de Lemos, P. Inverardi, and J. Magee, *Software Engineering for Self-Adaptive Systems*, ser. LNCS. Springer, 2009, vol. 5525.

[9] K. Welsh, P. Sawyer, and N. Bencomo, "Towards Requirements Aware Systems: Run-time Resolution of Design-time Assumptions," in *Proceedings of the 26th International Conference on Automated Software Engineering (ASE)*. IEEE, 2011, pp. 560–563.

[10] M. Morandini, L. Penserini, and A. Perini, "Towards Goal-oriented Development of Self-Adaptive Systems," in *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2008, pp. 9–16.

[11] E. S.-K. Yu, "Modelling Strategic Relationships for Process Reengineering," Ph.D. dissertation, Univ. of Toronto, 1996.

[12] A. Lapouchnian and Y. Lespérance, "Modeling Mental States in Agent-Oriented Requirements Engineering," in *Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE)*, ser. LNCS, vol. 4001. Springer, 2006, pp. 480–494.

[13] S. Liaskos, S. McIlraith, S. Sohrabi, and J. Mylopoulos, "Integrating Preferences into Goal Models for Requirements Engineering," in *Proceedings of the 18th IEEE International Requirements Engineering Conference (RE)*. IEEE Computer Society, 2010, pp. 135–144.

[14] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-directed Requirements Acquisition," *Science of Computer Programming*, vol. 20, no. 1-2, pp. 3–50, 1993.

[15] A. van Lamsweerde, R. Darimont, and P. Massonet, "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt," in *Proceedings of the 2nd International Symposium on Requirements Engineering (RE)*, 1995, pp. 194–203.

[16] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani, "Reasoning with Goal Models," in *Proceedings of the 21st International Conference on Conceptual Modeling (ER)*, ser. LNCS, vol. 2503, 2002, pp. 167–181.

[17] M. Morandini, L. Penserini, and A. Perini, "Operational Semantics of Goal Models in Adaptive Agents," in *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IFAAMAS, 2009, pp. 129–136.

[18] F. Dalpiaz, P. Giorgini, and J. Mylopoulos, "Adaptive Socio-Technical Systems: a Requirements-driven Approach," *Requirements Engineering*, vol. 18, no. 1, pp. 1–24, 2013.

[19] J. Mylopoulos, L. Chung, and E. Yu, "From Object-Oriented to Goal-Oriented Requirements Analysis," *Communications of the ACM*, vol. 42, no. 1, pp. 31–37, 1999.

[20] A. van Lamsweerde, "Goal-oriented Requirements Engineering: A Guided Tour," in *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE 2001)*, 2001, pp. 249–263.

[21] J. Horkoff and E. Yu, "Comparison and Evaluation of Goal-Oriented Satisfaction Analysis Techniques," *Requirements Engineering*, pp. 1–24, 2013, to appear.

[22] H. Kautz, "A Formal Theory of Plan Recognition," Ph.D. dissertation, Bell Laboratories, 1987.

[23] M. S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behavior," in *Proceedings of the 9th International Workshop on Software Specification and Design (IWSSD)*. IEEE Computer Society, 1998, pp. 50–59.

[24] B. Cheng, P. Sawyer, N. Bencomo, and J. Whittle, "A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty," in *Model Driven Engineering Languages and Systems*, ser. LNCS. Springer, 2009, vol. 5795, pp. 468–483.

[25] L. Baresi, L. Pasquale, and P. Spoletini, "Fuzzy Goals for Requirements-driven Adaptation," in *Proceedings of the 18th International IEEE Requirements Engineering Conference (RE)*. IEEE Computer Society, 2010, pp. 125–134.

[26] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos, and J. C. Leite, "From Goals to High-Variability Software Design," in *Proceedings of the 17th International Syposium on Methodologies for Intelligent Systems (ISMIS)*, ser. LNCS, vol. 4994. Springer, 2008, pp. 1–16.

[27] H. J. Goldsby, P. Sawyer, N. Bencomo, B. H. C. Cheng, and D. Hughes, "Goal-Based Modeling of Dynamically Adaptive System Requirements," in *Proceedings of the 15th Annual IEEE International Conference on the Engineering of Computer-Based Systems (ECBS)*. IEEE Computer Society, 2008, pp. 36–45.

[28] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-functional Requirements in Software Engineering*. Springer, 2000.

[29] R. Ali, F. Dalpiaz, P. Giorgini, and V. E. S. Souza, "Requirements Evolution: From Assumptions to Reality," in *Proceedings of the 16th International Conference on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD)*, ser. LNBIP, vol. 81. Springer, 2011, pp. 372–382.

[30] V. Souza, A. Lapouchnian, W. Robinson, and J. Mylopoulos, "Awareness requirements for adaptive systems," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2011, pp. 60–69.

[31] S. A. DeLoach and M. Miller, "A Goal Model for Adaptive Complex Systems," *International Journal of Computational Intelligence: Theory and Practice*, vol. 5, no. 2, pp. 83–92, 2010.

[32] F. Dalpiaz, P. Giorgini, and J. Mylopoulos, "An Architecture for Requirements-Driven Self-Reconfiguration," in *Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE)*, ser. LNCS, vol. 5565. Springer, 2009, pp. 246–260.

[33] R. Ali, F. Dalpiaz, and P. Giorgini, "A Goal Modeling Framework for Self-Contextualizable Software," in *Proceedings of the 14th International Conference on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD)*, ser. LNBIP, vol. 29. Springer, 2009, pp. 326–338.

[34] D. Sykes, W. Heaven, J. Magee, and J. Kramer, "From Goals to Components: a Combined Approach to Self-Management," in *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2008, pp. 1–8.

[35] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Deriving Event-based Transition Systems from Goal-Oriented Requirements Models," *Automated Software Engineering*, vol. 15, no. 2, pp. 175–206, 2008.

[36] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel, "Deriving non-Zeno Behaviour Models from Goal Models using ILP," *Formal aspects of computing*, vol. 22, no. 3, pp. 217–241, 2010.

[37] S. Uchitel, G. Brunet, and M. Chechik, "Behaviour Model Synthesis from Properties and Scenarios," in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2007, pp. 34–43.

[38] A. Lapouchnian and J. Mylopoulos, "Modeling domain variability in requirements engineering with contexts," in *Proceedings of the 28th International Conference on Conceptual Modeling (ER)*, ser. LNCS. Springer, 2009, vol. 5829, pp. 115–130.