Inferring Declarative Requirements Specifications from Operational Scenarios

Axel van Lamsweerde, *Member*, *IEEE*, and Laurent Willemet

Abstract—Scenarios are increasingly recognized as an effective means for eliciting, validating, and documenting software requirements. This paper concentrates on the use of scenarios for requirements elicitation and explores the process of inferring formal specifications of goals and requirements from scenario descriptions. Scenarios are considered here as typical examples of system usage; they are provided in terms of sequences of interaction steps between the intended software and its environment. Such scenarios are in general partial, procedural, and leave required properties about the intended system implicit. In the end such properties need to be stated in explicit, declarative terms for consistency/completeness analysis to be carried out.

A formal method is proposed for supporting the process of inferring specifications of system goals and requirements inductively from interaction scenarios provided by stakeholders. The method is based on a learning algorithm that takes scenarios as examples/counterexamples and generates a set of goal specifications in temporal logic that covers all positive scenarios while excluding all negative ones.

The output language in which goals and requirements are specified is the KAOS goal-based specification language. The paper also discusses how the scenario-based inference of goal specifications is integrated in the KAOS methodology for goal-based requirements engineering. In particular, the benefits of inferring declarative specifications of goals from operational scenarios are demonstrated by examples of formal analysis at the goal level, including conflict analysis, obstacle analysis, the inference of higher-level goals, and the derivation of alternative scenarios that better achieve the underlying goals.

Index Terms—Scenario-based requirements elicitation, inductive inference of specifications, goal-oriented requirements engineering, specification refinement and analysis, lightweight formal methods.

1 Introduction

Requirements engineering (RE) is concerned with the elicitation of high-level goals to be achieved by the envisioned system, the refinement of such goals and their operationalization into specifications of services and constraints, and the assignment of responsibilities for the resulting requirements to agents such as humans, devices, and software.

One frequent problem requirements engineers are faced with is that stakeholders may have difficulties expressing goals and requirements in abstracto. Typical scenarios of using the hypothetical system are sometimes easier to get in the first place than some goals that can be made explicit only after deeper understanding of the system has been gained. This fact has been recognized in cognitive studies on human problem solving [4] and in research on inquiry-based RE [58]. We experienced it during the elicitation of several systems, including a well-known RE benchmark [12], [42], [14]. A recent study on a broader scale has confirmed scenarios as important design artifacts that are used for a variety of purposes, in particular in cases when abstract modeling fails [72].

A *scenario* is defined in this paper as a temporal sequence of interaction events among different agents in the restricted context of achieving some implicit purpose(s). The agents are either environmental or software agents that form the *composite* system to be developed [19], [22]. A scenario captures just one particular, fragmentary instance of behavior of such a system; it is *internal* or *external* dependent on whether the agents are all software agents or involve environmental agents as well. We will sometimes make a distinction between *positive* and *negative* scenarios. The former describe desired behaviors whereas the latter describe undesirable ones.

Our definition of scenarios agrees with the one adopted by popular object-oriented modeling techniques such as [68]; it also agrees with the kind of preliminary description used in several industrial projects we have been involved in and briefly discussed in the paper. Such scenarios have strengths and weaknesses in supporting the requirements engineering process.

- On the positive side, they induce an informal, narrative, and concrete style of description that focuses on the dynamic aspects of hypothetical software-environment interactions. They can therefore be deployed easily by multiple stakeholders with different background to build a shared view of the "visible part of the iceberg." Scenarios can be used for a wide variety of purposes in the requirements engineering lifecycle—notably, to elicit requirements in hypothetical situations [2], [58]; to help identify exceptional cases [59]; to populate more abstract artifacts such as conceptual models [68], [67], business rules [66], or glossaries [72]; to validate requirements in conjunction with prototyping [70], animation [18], or plan generation tools [1], [22]; to reason about usability during

• *A. van Lamsweerde and L. Willemet are with the Département d'Ingénierie Informatique, Université Catholique de Louvain, Place Sainte Barbe 2, B-1348 Louvain-la-Neuve, Belgium. E-mail: {avl, lw}@info.ucl.ac.be.*

Manuscript received 12 Jan. 1998; revised 25 Aug. 1998.
Recommended for acceptance by R. Kurki-Suonio and M. Jarke.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 107462.

system development [9]; to generate acceptance test cases [32]; to structure requirements through user-oriented decomposition for subsequent work assignment [72]; to support evolution [47]; to explain, illustrate and document requirements throughout the software lifecycle; etc.

- On the downside, scenarios are inherently *partial*; they raise a coverage problem similar to test cases, making it impossible to verify the absence of errors such as, e.g., incompleteness with respect to stated goals [73], [13] or conflicts among goals/requirements [62], [45]. Instance-level trace descriptions also raise the *combinatorial explosion* problem inherent to the enumeration of combinations of individual behaviors. Scenarios in the form of interaction sequences are also *procedural*, thus introducing risks of overspecification—such as, e.g., too strict dependencies between interaction events. The description of interaction sequences between the software and its environment forces decisions about the *precise boundary* between them which may be premature in the early stages of requirements elicitation. Last but not least, scenarios leave required properties about the intended system *implicit*, in the same way as safety/liveness properties are implicit in a program trace; in the end such properties need to be made explicit in order to support analysis, negotiation/commitment, subsequent implementation, and evolution.

The objective of this paper is to address those deficiencies by means of a formal method for generating explicit, declarative, type-level requirements from operational, instance-level scenarios in which such requirements are implicit, so as to obtain formal specifications amenable to goal-based completeness analysis, conflict detection/resolution, exploration of alternative system proposals, scenario improvement, and specification refinement. To give a concrete example, this method when applied to scenarios of user-ATM interaction will generate formal assertions capturing declarative requirements such as

> "every card inserted shall be returned unless the password entered remains invalid after three attempts."

To do this, our method takes scenarios as examples/counterexamples of intended system behavior, and inductively infers a set of candidate goals/requirements that cover all example scenarios and exclude all counterexample scenarios. The basic technical hint is to generate temporal logic formulas whose logical models include/exclude the temporal sequences given as positive/negative scenarios. The procedure roughly consists of the following steps repeated for each scenario provided:

- map the events from the interaction sequence to operations (possibly after having filtered out irrelevant events);
- generate state predicates along the interaction sequence from conditions that define the elementary state transitions produced by such operations in the domain;
- inductively infer candidate temporal logic assertions that are satisfied by the interaction sequence annotated with such state predicates;
- integrate the assertions thereby obtained with those previously obtained from the scenarios considered before, by use of coverage/exclusion rules.

Unlike deductive inference, inductive inference is not sound; the candidate assertions generated must, therefore, be checked/refined by the requirements engineer for adequacy.

It is important to note that the scenarios we start from are *not* requirements, even though they frequently appear in preliminary material for requirements elicitation—much the same way as examples of input-output pairs are not specifications of a program; program traces do not describe an algorithm; process traces do not describe a process model; etc. When we talk about requirements elicitation in this paper, we refer to the process of acquiring declarative specifications of system goals and requirements from preliminary material such as interview transcripts, documentation available about the application domain, reports about problematic issues with the existing system and opportunities for a new one, etc. The scenarios provided are assumed to be found in such preliminary material; we do not cover the process of eliciting and elaborating such scenarios—see, e.g., [68], [34], [61], [64] for guidelines and research efforts in that direction.

Scenarios are expressed here as event trace diagrams [68], a notation chosen for its popularity and its simplicity. The generated specifications are expressed in the KAOS goal-based language [11], [42] so that a number of formal reasoning techniques can be subsequently applied to the specifications obtained—such as goal-based refinement and completeness checking [13], obstacle analysis [44], conflict detection/resolution [45], formal operationalization [11], and design derivation [20]. Other reasoning techniques may be applied as well to the goals inferred, such as qualitative reasoning to determine the degree to which high-level goals are satisficed/denied by lower-level goals/requirements [54], or requirement/assumption monitoring for the reconciliation of specifications and runtime behavior [21].

The paper is organized as follows. Section 2 provides some necessary background material on the KAOS methodology for goal-oriented RE. Section 3 discusses the integration of goal-based and scenario-based processes for requirements elicitation and validation. Section 4 then presents our goal inference procedure by detailing the above steps and illustrating them piecewise on a simple ongoing example, namely, the ATM system borrowed from [68]. Section 5 shows a complete run of the goal inference procedure on a nontrivial exemplar, namely, the Lift system [50]. Section 6 then discusses what the inferred formal specifications can be used for, by showing various types of formal analysis at the goal level that could not be carried out at the scenario level. Section 7 reviews related efforts in scenario-based RE and Section 8 concludes by discussing the current status of this work together with future plans; the latter include the implementation of a tool to support this method and its application to the most complex scenarios we have seen in the industrial projects we are involved in.

2 Goal-Based RE with KAOS

The KAOS methodology is aimed at supporting the whole process of requirements elaboration—from the high-level

goals to be achieved to the requirements, objects and operations to be assigned to the various agents in the composite system. Thus WHY, WHO, and WHEN questions are addressed in addition to the usual WHAT questions addressed by standard specification techniques.

The methodology comprises a specification language, an elaboration method, and meta-level knowledge used for local guidance during method enactment. Hereafter we introduce some of the features that will be used later in the paper; see [11], [42], [13], [15] for details.

2.1 The Specification Language

The KAOS language provides constructs for capturing various types of concepts that appear during requirements elaboration.

2.1.1 The Underlying Ontology

The following types of concepts will be used in the sequel.

- *Object.* An object is a thing of interest in the composite system whose instances may evolve from state to state. It is in general specified in a more specialized way—as an *entity, relationship,* or *event* dependent on whether the object is an autonomous, subordinate, or instantaneous object, respectively. Objects are characterized by attributes and invariant assertions. Inheritance is of course supported.
- *Operation.* An operation is an input-output relation over objects; operation applications define state transitions. Operations are characterized by pre-, post-, and trigger-conditions. A distinction is made between *domain* pre/postconditions, which capture the elementary state transitions defined by operation applications in the domain, and *required* pre/postconditions, which capture additional conditions that need to strengthen the domain conditions in order to ensure that the requirements are met.
- *Agent.* An agent is another kind of object which acts as processor for some operations. An agent *performs* an operation if it is effectively allocated to it; the agent *monitors/controls* an object if the states of the object are observable/controllable by it. Agents can be humans, devices, programs, etc.
- *Goal.* A goal is an objective the composite system should meet. *AND-refinement* links relate a goal to a set of subgoals (called refinement); this means that satisfying all subgoals in the refinement is a sufficient condition for satisfying the goal. *OR-refinement* links relate a goal to an alternative set of refinements; this means that satisfying one of the refinements is a sufficient condition for satisfying the goal. The goal refinement structure for a given system can be represented by an AND/OR directed acyclic graph. Goals *concern* the objects they refer to. They may *conflict* with each other [45].
- *Requisite, requirement,* and *assumption.* A *requisite* is a goal that can be formulated in terms of states controllable by some individual agent. Goals must be eventually AND/OR *refined* into requisites assignable to individual agents. Requisites in turn are AND/OR *operationalized* by operations and objects through strengthenings of their domain pre/postconditions and invariants, respectively, and through trigger conditions. Alternative ways of assigning responsible agents to a requisite are captured through AND/OR *responsibility* links; the actual assignment of an agent to the operations that operationalize the requisite is captured in the corresponding *performance* links. A *requirement* is a requisite assigned to a software agent; an *assumption* is a requisite assigned to an environmental agent. Unlike requirements, assumptions cannot be enforced in general [21], [44].

2.1.2 Language Constructs

Each construct in the KAOS language has a two-level generic structure: an outer semantic net layer for *declaring* a concept, its attributes and its various links to other concepts [8]; an inner formal assertion layer for *formally defining* the concept. The declaration level is used for conceptual modeling (through a concrete graphical syntax), requirements traceability (through semantic net navigation) and specification reuse (through queries) [15]. The assertion level is optional and used for formal reasoning [11], [13], [44], [45].

The generic structure of a KAOS construct is instantiated to specific types of links and assertion languages according to the specific type of the concept being specified. For example, consider the following goal specification for an ATM system:

Goal *Avoid* [IllegalAccessToAccount]
  InstanceOf SecurityGoal
  Concerns ATM, Card, Account
  InformalDef *An ATM should never give access to an account through a card unless the password entered to the ATM is correct*
  RefinedTo PassWdAsked, PassWdEntered, PassWdChecked, AccessDecisionMade
  FormalDef $\forall$ atm: ATM, c: Card, ac: Account
    $\neg$ GivesAccess(atm, c, ac)
    W [LinkedTo(c, ac) $\wedge$ OKPassWd(c, atm)]

The declaration part of this specification introduces a concept of type "goal," named *Avoid* [IllegalAccessToAccount], prohibiting some property from ever holding ("Avoid" verb), concerned with maintaining secure access to objects by agents ("SecurityGoal" category), referring to objects such as ATM or Account, refined into four subgoals, and defined by some informal statement. (The semantic net layer is represented in textual form in this paper for reasons of space limitations; the reader may refer to [15] to see what the alternative graphical concrete syntax looks like.)

The assertion defining this goal formally is written in a real-time temporal logic inspired from [40]. The following classical operators for temporal referencing are used [48]:

o	(in the next state)	•	(in the previous state)
$\Diamond$	(some time in the future)	$\blacklozenge$	(some time in the past)
$\Box$	(always in the future)	$\blacksquare$	(always in the past)
$\mathcal{W}$	(always in the future *unless*)	U	(always in the future *until*)

Formal assertions are interpreted over historical sequences of states. Each assertion is in general satisfied by some sequences and falsified by some other sequences. The notation:

(H, i) $\models$ P

is used to express that assertion P is satisfied by history H at time position i ($i \in T$), where T denotes a linear temporal structure assumed to be discrete in this paper. The semantics of the above temporal operators is then defined as usual [48], e.g.,

$(H, i) \models \circ P$ iff $(H, next(i)) \models P$

$(H, i) \models \Diamond P$ iff $(H, j) \models P$ for some $j \geq i$

$(H, i) \models \Box P$ iff $(H, j) \models P$ for all $j \geq i$

$(H, i) \models P \mathcal{U} Q$ iff there exists a $j \geq i$ such that $(H, j) \models Q$ and for every k, $i \leq k < j$, $(H, k) \models P$

$(H, i) \models P \mathcal{W} Q$ iff $(H, i) \models P \mathcal{U} Q$ or $(H, i) \models \Box P$

Note that $\Box P$ amounts to $P \mathcal{W}$ **false**. We will also use the logical connectives $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implies), $\leftrightarrow$ (equivalent), $\Rightarrow$ (entails), $\Leftrightarrow$ (congruent), with

$P \Rightarrow Q$ iff $\Box(P \rightarrow Q)$,

$P \Leftrightarrow Q$ iff $\Box(P \leftrightarrow Q)$

(Note the implicit $\Box$-operator in every entailment.)

To handle real requirements we often need to introduce real-time restrictions. We, therefore, introduce bounded versions of the above temporal operators in the style advocated by [40], such as

$\Diamond_{\leq d}$ (some time in the future within deadline *d*)

$\Box_{\leq d}$ (always in the future up to deadline *d*)

The semantics of the real-time operators is then defined accordingly, e.g.,

$(H, i) \models \Diamond_{\leq d} P$ iff $(H, j) \models P$ for some $j \geq i$ with $dist(i, j) \leq d$

$(H, i) \models \Box_{<d} P$ iff $(H, j) \models P$ for all $j \geq i$ such that $dist(i, j) < d$

where *dist:* $T \times T \rightarrow D$ is a temporal distance function from the temporal structure T to a metric domain D. A frequent choice is

T: the set of naturals

D: {d | there exists a natural *n* such that $d = n \times u$}, where *u* denotes some chosen time unit

dist(i, j): $| j - i | \times u$

Multiple units can be used (e.g., second, day, week); they are implicitly converted into some smallest unit. For example, the subgoal PassWdEntered above could be formally specified by the assertion

$\forall$ atm: ATM, c: Card

PwdAsked(atm, c) $\Rightarrow \Diamond_{\leq d}$ PwdEntered(c, atm)

In the assertion formalizing the goal *Avoid* [IllegalAccessToAccount] above, the predicate OKPassWd(c, atm) means that, in the current state, an instance of the OKPassWd relationship links variables c and atm of sort Card and ATM, respectively. The OKPassWd relationship, ATM agent, and Card entity are declared in other sections of the specification, e.g.,

Agent ATM
  CapableOf AskPassWd, CheckPassWd, DeliverCash, ...
  Has CashAvailable, CashGiven: *Amount-$*

Relationship OKPassWd
  Links Card {**card** 0:N}, ATM {**card** 0:N}
  DomInvar $\forall$ c: Card, atm: ATM
    OKPassWd (c, atm) $\Leftrightarrow$
    ($\exists$u: User, x: Nat) [Types (u, atm, x, c) $\wedge$ c.PassWd = x]

In the declaration of ATM, CashAvailable is declared as an attribute of ATM whose values are in the range *Amount-$.*

As mentioned earlier, operations are specified formally by pre- and postconditions in the state-based tradition [26], [60], e.g.,

Operation DeliverCash
  Input Card {**arg**: c}, Amount-$ {**arg**: amount};
  Output CashDelivered
  PerformedBy ATM {**arg**: atm}
  DomPre $\neg$ CashDelivered(c, atm, amount)
  DomPost CashDelivered(c, atm, amount)

It is important to note that the invariant defining the OKPassWd relationship is *not* a requirement, but a *domain description* in the sense of [35], [36], [74]; it specifies what password correctness does precisely mean in the domain. The pre- and postcondition of the operation DeliverCash above are domain descriptions as well; they capture corresponding elementary state transitions in the domain, namely, from a state where the amount of cash is not delivered to a state where the amount of cash is delivered (for some amount and card specified among the arguments of the operation).

The software *requirements* are found in the terminal goals assigned to software agents (e.g., the goal *Avoid* [IllegalAccessToAccount] assigned to the ATM agent), and in the additional pre / postconditions that need to strengthen the corresponding domain pre- and postcondition in order to ensure all such goals [11], [42], e.g.,

Operation DeliverCash
  ...
  RequiredPre for *Avoid* [IllegalAccessToAccount]:
    OKPassWd (c, atm)
  RequiredPre for *Maintain* [LimitedAdvance]:
    amount $\leq$ c.Limit

The distinction between domain conditions and requirements is very important to the method presented in this paper; it is similar to the distinction between indicative and optative properties in [36] and between NAT and REQ in Parnas' 4-variable model [57].

2.2 The Elaboration Method

Fig. 1 outlines the major steps that may be followed to systematically elaborate KAOS specifications from high-level goals. (Section 3 will discuss how scenario-based elicitation and validation enter into this process model.)

- *Goal elaboration.* Elaborate the goal AND / OR structure by defining goals and their refinement / conflict links until assignable requisites are reached. The process of identifying goals, defining them precisely, and relating them through positive / negative contribution links is in general a combination of top-down and bottom-up subprocesses [42]; offspring goals are identified by asking *HOW* questions about goals already identified whereas parent goals are identified by asking *WHY* questions about goals and operational requirements already identified.
- *Object capture.* Identify the objects involved in goal formulations, define their conceptual links, and describe their domain properties by invariants.

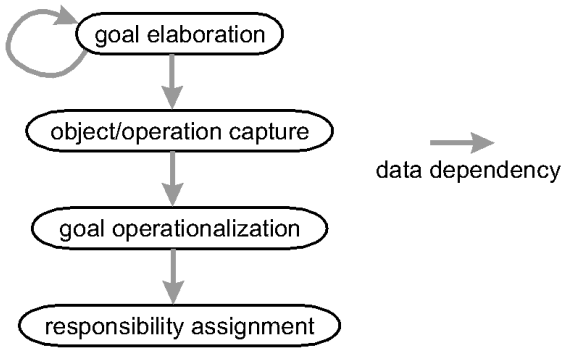

Fig. 1. Goal-based requirements elaboration.

- *Operation capture.* Identify object state transitions that are meaningful to the goals. Goal formulations refer to desired or forbidden states that are reachable by state transitions; the latter correspond to applications of operations. The principle is to specify such state transitions as domain pre- and postconditions of operations thereby identified, and to identify agents that could have those operations among their capabilities.
- *Operationalization.* Derive strengthenings on operation pre/postconditions and on object invariants in order to ensure that all requisites are met. Formal derivation rules are available to support the operationalization process [11].
- *Responsibility assignment.* Identify alternative responsibilities for requisites; make decisions among refinement, operationalization, and responsibility alternatives—with process-level objectives such as: reduce costs, increase reliability, avoid overloading agents, resolve conflicts; assign the operations to agents that can commit to guaranteeing the requisites in the alternatives selected. The boundary between the system and its environment is obtained as a result of this process, and the various requisites become requirements or assumptions depending on the assignment made.

The steps above are ordered by data dependencies; they may be running concurrently, with possible backtracking at every step.

2.3 Using Meta-Level Knowledge

At each step of the goal-driven method, domain-independent knowledge can be used for local guidance and validation in the elaboration process.

- A rich taxonomy of goals, objects and operations is provided together with rules to be observed when specifying concepts of the corresponding subtype. We give a few examples of such taxonomies.
  - Goals are classified by pattern of temporal behavior they require:

    Achieve: $P \Rightarrow \Diamond Q$ or *Cease:* $P \Rightarrow \Diamond \neg Q$

    Maintain: $P \Rightarrow Q \mathcal{W} R$ or *Avoid:* $P \Rightarrow \neg Q \mathcal{W} R$
  - Goals are also classified by category of requirements they will drive with respect to the agents concerned (e.g., SatisfactionGoals are functional goals concerned with satisfying agent requests; InformationGoals are goals concerned with keeping agents informed about object states; SecurityGoals are goals concerned with maintaining secure access to objects by agents; other categories include SafetyGoals, AccuracyGoals, etc.) These categories refine well-known functional and nonfunctional goal categories [38], [54].

  Such taxonomies are associated with heuristic rules that may guide the elaboration process, e.g.,
    - SafetyGoals are AvoidGoals to be refined in HardRequirements;
    - ConfidentialityGoals are AvoidGoals on *Knows* predicates.
- Tactics capture heuristics for driving the elaboration or for selecting among alternatives, e.g.,
  - Refine goals so as to reduce the number of agents involved in the achievement of each subgoal;
  - Favor goal refinements that introduce less conflicts.

Goal verbs such as *Achieve/Maintain* and categories such as *Satisfaction/Information* are language keywords that allow users to specify more information at the declaration level; for example, the declaration *Avoid* [IllegalAccessToAccount] allows specifiers to state in a lightweight way that the situation named IllegalAccessToAccount should never hold, without entering into the temporal logic level. (We avoid the classical safety/liveness terminology here to avoid confusions with SafetyGoals.)

To conclude this short overview of KAOS, we would like to draw the reader's attention on the complementarity between the outer semiformal layer and the inner formal layer. At the semantic net level, the user builds her requirements model in terms of concepts whose meaning is annotated in InformalDef attributes; the latter are placeholders for the designation of objects and operations [74] and for the informal formulation of goals, requirements, assumptions and domain descriptions. At the optional formal assertion level, more advanced users may make such formulations more precise, fixing problems inherent to informality [52], and apply various forms of formal reasoning, e.g., for goal refinement and exploration of alternatives [13], requirements derivation from goals [11], obstacle analysis [44], conflict analysis [45], requirements/assumptions monitoring [21], or inference of goal specifications from scenarios as shown in this paper. Our experience with various industrial projects in which KAOS was used reveals that the semiformal semantic net layer is easily accessible to industrial users; the formal assertion layer proved effective after the results of formal analysis by trained users were propagated back to the semiformal semantic net layer.

3 Integrating Scenario-Based and Goal-Based RE

The objective of this section is to put our goal inference method into perspective by: 1) briefly reviewing some experimental facts grounding some of the principles, assumptions and choices made in the paper, 2) introducing

the notation used to capture scenarios as interaction sequences, and 3) discussing the intertwining of goal-based and scenario-based processes for requirements elicitation and validation.

3.1 Experience with Scenarios

Recent studies have shown that scenarios are of widespread use in the requirements engineering process [72]. Our own experience suggests that scenarios play a decisive role in eliciting, validating, and explaining requirements. In particular, scenarios are frequently used for requirements elicitation. For example, a study of eight independent elicitation sessions for a meeting scheduler system showed that the eight potential users interviewed all used fragmentary, instance-level scenarios to illustrate features they would expect from such a system, with an average of 3.6 scenarios per subject [12]. This limited study also revealed that negative scenarios are sometimes a natural way of drawing the interviewer's attention on undesirable behaviors in exceptional situations.

The CEDITI technology transfer institute we are working with has been contractually involved in various industrial RE projects ranging over a wide variety of domains, including a phone system on TV cable, a complex copyright/royalty distribution system, a system to support the emergency service in a major hospital, a complex system to support good delivery to retailers, and an air traffic control system. Scenarios were found in all those projects. They had commonalities and differences.

- All of them were used as preliminary material from interviews and existing documents to start the building of the domain and requirements models.
- All of them were initially obtained in the form of narrative text describing *instance-level* behavior of agents; sentences were articulated by connectives indicating *temporal sequencing* (such as "when", "then", "after", "until"), e.g.,

  "*When* a new CNT is installed on the network, it will synchronize itself on the RFLT signals by finding ... The CNT will *then* look for its UAC and wait *until* it receives ... *When* the RFLT detects this access transmission, it shall measure ..."
- All of them were found to be *partial*. For example, the admission of patients to the emergency service of an hospital was described by scenarios covering a few typical cases, such as the victim of a car accident, the victim of an accident at school, or someone deceasing during admission; such scenarios were by no means intended to be exhaustive.
- The temporal sequences were in general fairly small—typically, below 10 interaction events; one notable exception was a scenario describing a complete flight history and comprising 92 interaction events from flight plan filing up to aircraft taxying to gate.
- The elaboration of scenarios seemed not too difficult in domains where event sequencing is somewhat built-in (e.g., telecommunication protocols, flight tracking); there were situations however where scenario elaboration was difficult—e.g., in the copyright/royalty distribution system an analogy with train freight dispatching through a network of stations had to be used to get agreement from all parties concerned.
- In many of the narrative sequences we have seen, the goals and requirements underlying them were *implicit* and not really visible in the first place.

Those various observations motivated our efforts to support the process of eliciting declarative specifications of goals from interaction sequences among agents in which goals are left implicit. The formal method presented in this paper can be seen as a systematization of an informal process followed in a number of cases. This method does not tackle two problems upstream to the inference of goal specifications, namely,

- the elaboration of narrative temporal sequences of interactions,
- the translation of such narratives into the diagrammatic notation used as input language to our inference procedure.

Guidelines and research efforts to tackle those two problems can be found in, e.g., [68], [34], [61], [64].

3.2 Representing Scenarios

A scenario was defined in Section 1 as a temporal sequence of interaction events among different agents in the restricted context of achieving some implicit purpose(s).

To represent such scenarios we use the popular event trace diagram notation [68]. Fig. 2 shows a scenario for an ATM system. Each vertical line is the *time line* of one agent instance, with time progressing from top to bottom. The label of a time line specifies the type of the corresponding agent

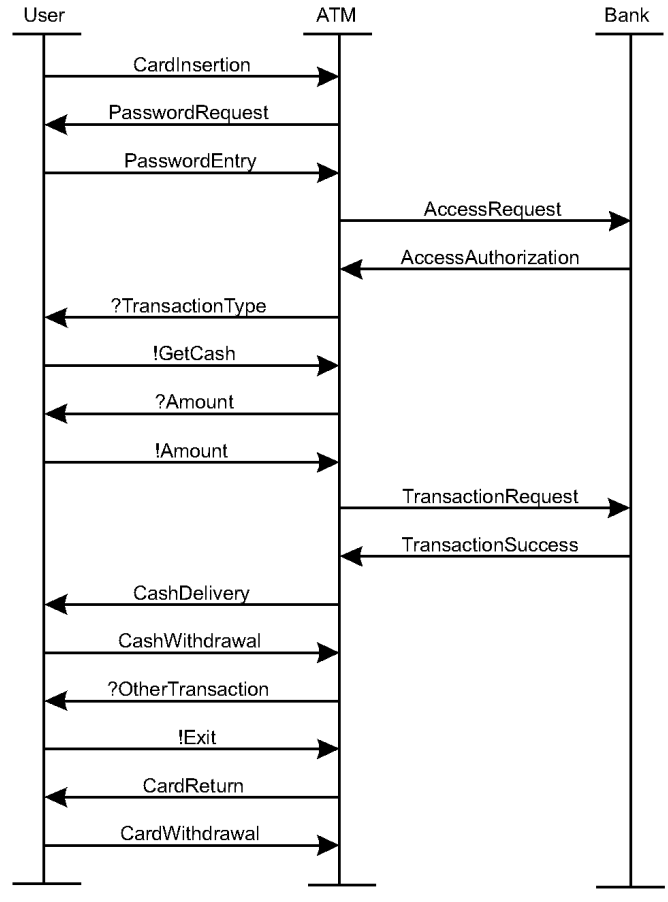

Fig. 2. Event trace diagram for the ATM exemplar.

instance. An arrow from a source line to a target line means that the source agent *generates* an event to be *perceived* by the target agent. The label of an arrow specifies the type of the corresponding event instance. Attributes can be attached to events to capture communication of information.

In the KAOS context, an arrow denotes a shared phenomenon between two agent instances [35], [36], [74], corresponding to the application of a KAOS operation performed by the source agent and observed by the target agent.

We have chosen this semiformal notation here because it is well-known, very simple, and widely used. There are many more sophisticated representation schemes that are richer in expressive power and structuring mechanisms supported, e.g., type-level languages based on regular expressions [11], decision trees [32], statecharts [25], annotated use cases [61], etc. A simpler, widely used graphical language was felt more appropriate for visualizing instance-level interaction sequences from the temporal narratives obtained during the early stages of requirements elicitation; it turns out that this language is also quite natural for depicting models of temporal logic specifications for a reactive system.

Note that event traces cannot be really considered as requirements, much the same way as input-output pair examples are not specifications or program traces are not algorithm descriptions. They convey partial information at the instance level in procedural form; they leave desired properties such as safety / liveness requirements implicit.

3.3 Intertwining of Goal-Based and Scenario-Based RE

Fig. 3 shows a process model that integrates both goal-based and scenario-based processes for requirements elicitation and validation. (As in Fig. 1, the arrows indicate data dependencies.)

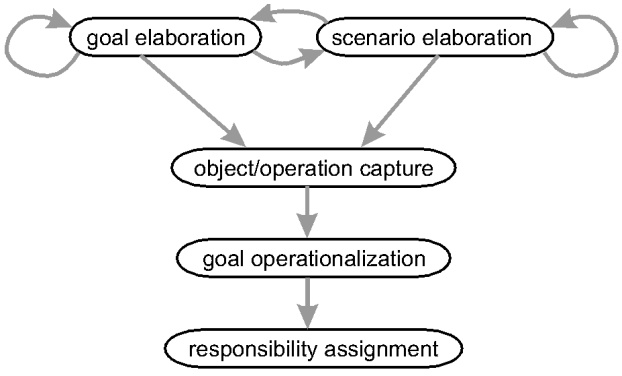

Fig. 3. Integrating goal-based and scenario-based RE.

The RE process typically starts with interviews and analysis of available documentation to find out what the domain is, what the problematic issues with the existing situation are, and what the opportunities for a new system might be. Goals and scenarios are elicited in parallel from this raw material together with domain descriptions, organizational policies, and operational choices whose rationale is generally implicit.

Goals are refined top-down (through HOW questions) and abstracted bottom-up (through WHY questions); this gives rise to new subgoals and supergoals that populate the goal AND / OR graph (see the left circle arrow in Fig. 3).

In parallel, scenarios are refined, translated into event trace diagrams, and possibly cleaned up (see the right circle arrow in Fig. 3; Section 4.2 will further discuss the issue of scenario clean up).

Goal elaboration and scenario elaboration are intertwined processes; a concrete scenario description may prompt the elicitation of the specifications of the goals underlying it (see the right-to-left arrow in Fig. 3); conversely, a goal specification may prompt the elaboration of scenario descriptions to illustrate or validate it (see the left-to-right arrow in Fig. 3).

The paper's main focus is on the upper right-to-left, goal elicitation arrow in Fig. 3. Sections 4 and 5 present and illustrate a formal method to support that arrow. The objective there is to obtain from scenarios additional goal specifications that were not obtained from the goal refinement / abstraction processes triggered by the goals initially identified from interviews and existing documentation.

The additional goal specifications obtained from scenarios may be brand new; they may cover specifications already found by the goal elaboration process (in which case the elicitation just reduces to some form of validation); they may also be conflicting with specifications found by the goal elaboration process. In the latter case such conflicts have to be detected and resolved (see Section 6.3) which will result in new versions of goals and scenarios.

The additional goal specifications obtained from scenarios may in turn trigger new goal elaboration processes (see the left circle arrow again in Fig. 3). In particular, more abstract goals are obtained bottom-up either informally, by asking WHY questions about the goals obtained from scenarios, or formally, by using formal refinement / abstraction patterns bottom-up (see Section 6.1). In doing so one may find better alternatives to achieve the more abstract goals which in turn results in better subgoals and corresponding scenarios (see the left-to-right arrow again in Fig. 3, and Section 6.4). Formal obstacle analysis can also be applied to the goal specifications obtained from scenarios to generate new goals together with corresponding new scenarios (see the left-to-right arrow again in Fig. 3, and Section 6.2).

The lower part of Fig. 3 shows that relevant objects and operations are identified and defined from the goal formulations, as outlined in Section 2.2, but also from the scenarios elaborated, see Sections 4.3 and 4.4.

Goal operationalization and responsibility assignment then proceed as explained in Section 2.2.

Fig. 3 does not show backtracking steps that may take place in the overall process. For example, the goal operationalization step may require revisiting some of the scenarios to make them consistent with the choices made there; likewise, different software-environment boundaries may be chosen during the responsibility assignment step, which may result in updated versions of the interaction scenarios initially elaborated.

4 The Goal Inference Procedure

The procedure for inferring explicit, declarative goal specifications inductively from scenarios is specified as follows:

GIVEN a set of positive and negative scenarios described by event trace diagrams,

FIND a conjunctive set of goal specifications taking the form

$$P \Rightarrow \Diamond Q \quad \text{(Achieve/Cease)}$$
$$P \Rightarrow Q \mathcal{W} R \quad \text{(Maintain/Avoid)}$$

that covers all positive scenarios and excludes all negative ones.

A positive scenario describes a desired system behavior whereas a negative scenario describes an undesirable one. A conjunctive set of specifications will be said to be *admissible* with respect to a set of positive / negative scenarios if it covers all positive ones and excludes all negative ones [53], [41].

We first give an overall description of the procedure for inferring an admissible set of specifications for all scenarios provided; the various steps of this procedure are detailed next and illustrated on our ongoing ATM example.

4.1 Overview of the Goal Inference Procedure

As discussed before, the procedure assumes that each scenario is provided in the form of an event trace diagram. The procedure iterates over such diagrams; at each iteration it inductively infers temporal logic formulas satisfied by the current diagram, and integrates the result to the conjunctive set of formulas obtained at previous iterations so as to make the resulting set of formulas cover / exclude the current positive / negative scenario as well. The procedure can be summarized as follows.

For each scenario *ET* provided **do**
begin

1. Clean up *ET*;
2. Map the events in *ET* onto corresponding operations;
3. Generate state predicates along *ET's* time lines from the domain pre/postconditions of these operations;
4. Inductively infer temporal logic formulas that are satisfied by *ET* annotated with these state predicates:
   - For each time line, collect progress and invariance properties as ground facts that causally relate state predicates along that line:
     - an event generated along the line produces a state transition described by the following progress fact:

       $$\text{PRE-State} \rightarrow \mathbf{o}\ \text{POST-State}$$

     - a condition *R* remaining true along the line from a state transition characterized by *ST* up to a point after which a condition *N* becomes true produces the following invariance fact:

       $$R \wedge ST \rightarrow (R \mathcal{W} N)$$

   - Generalize these facts over time and over instances of agents/objects to obtain quantified goal specifications taking the form:

     $$\text{PRE-State} \Rightarrow \Diamond \text{POST-State} \quad (\textit{Achieve/Cease})$$
     $$R \wedge ST \Rightarrow (R \mathcal{W} N) \quad (\textit{Maintain/Avoid})$$
5. Integrate these goal specifications to the admissible conjunctive set of specifications obtained at previous iterations so as to keep the resulting set admissible, using coverage/exclusion rules.
6. Submit the resulting specifications to the requirements engineer for approval/refinement;
7. Elicit or infer supergoals and subgoals from the goals obtained

end

The various steps of this iterative procedure are now detailed.

4.2 Step 1: Cleaning Up Scenarios

Scenarios provided by multiple stakeholders may be quite complex [72]. The same one-shot scenario may address different, unrelated concerns; irrelevant events sometimes enter the picture; scenarios elicited from different viewpoints may have different granularities of interaction. The more complex a scenario is, the more candidate goal specifications will be generated by the procedure that are likely to be *irrelevant to some specific concern*.

It is, therefore, often helpful to clean up scenarios in order to facilitate the subsequent steps of goal inference. We propose three clean up rules here; a richer set of such rules is subject to future work.

Event aggregation. A sequence of consecutive interaction events of the same type but with different attribute values, between the same two agents and in the same direction, can be aggregated into one single event whose attribute value is the sequence of original attribute values.

Fig. 4 illustrates the use of this rule on the ATM example.

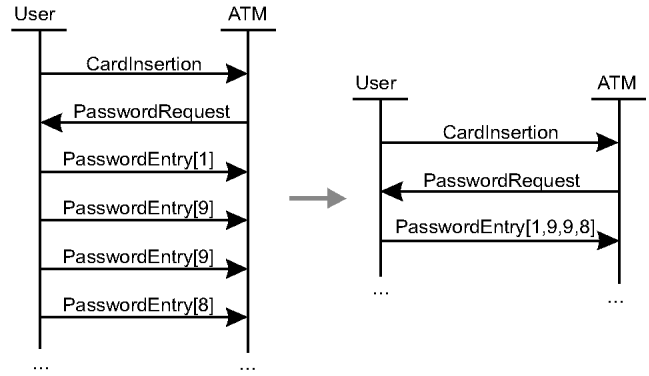

Fig. 4. Event aggregation.

Elimination of nonshared phenomena. It is often the case that potential users of the intended system provide scenarios that describe a hypothetical sequence of events they would be involved in when using the system; among such events there may be some irrelevant ones that do not correspond to interactions with a software agent. The rule here is to eliminate all events generated by the environment that cannot (or need not) be observed by a software agent [74].

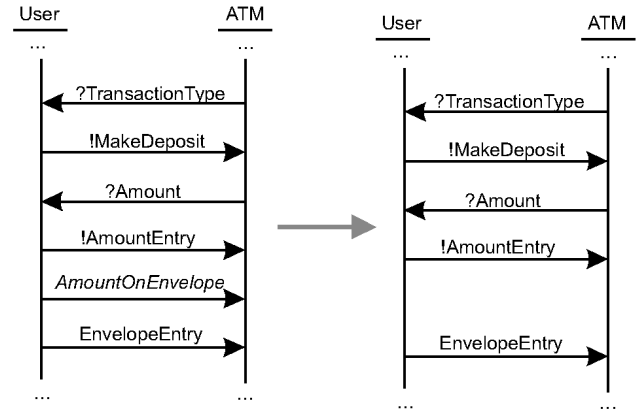

Fig. 5. Eliminating nonshared phenomena.

Fig. 5 illustrates the use of this rule on the ATM example. The event of printing the deposit amount on the envelope to be entered is eliminated because the ATM cannot observe this event even though the event has to occur in the environment.

Scope restriction. The principle here is to remove from the scenario all interaction events that are not relevant to some aspect of interest. We use the goal categories provided by KAOS to make this principle more precise (see Section 2.3). A scenario *perspective* according to some goal category is the subsequence of all interaction events in this scenario that are relevant to this category.

The concept of perspective introduced here is somewhat similar to the notion of perspective [46], viewpoint [56] or view [43]; the main difference is that a scenario perspective here is not associated with a process-level stakeholder but with a product-level goal category.

Given some complex scenario one may thereby extract its SatisfactionGoal perspective that shows the sequence of interaction events to satisfy an agent's request, its InformationGoal perspective that shows the sequence of interaction events to keep an agent informed about object states, etc. Fig. 6 shows the SecurityGoal perspective for the ATM scenario given in Fig. 2.

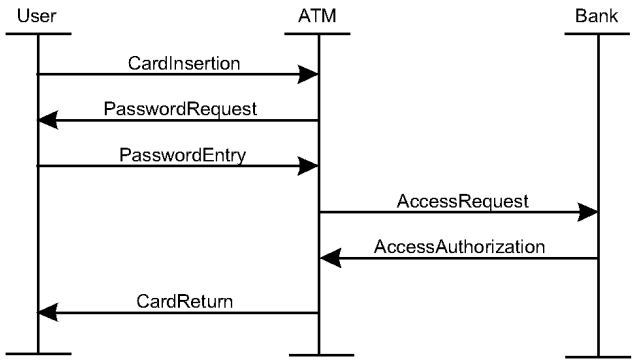

Fig. 6. Scope restricion: the *SecurityGoal* perspective.

4.3 Step 2: Mapping Interaction Events to Operations

Given an event trace diagram, possibly pruned after application of clean up rules, the next step is to introduce for each interaction event in the diagram the operation which the event is an application of. This step comprises three substeps for each event:

- Identify the operation whose applications define the event type.
- Make the operation's inputs and outputs explicit and introduce typed variables to name them; such inputs / outputs generally include attributes of the event type, and the source and target agents involved in the interaction.
- Get the operation's *domain* pre- and postcondition from knowledge available about the domain, or by elicitation from domain experts. As a result, the operation associated with the event is defined by a pair of formal assertions together with their logical interpretation specified in corresponding InformalDef attributes.

In the ATM example, a CardInsertion event will produce an operation InsertCard with instance variables c and atm of sort Card and ATM, respectively; the corresponding domain pre- and postcondition are:

DomPre: ¬ CardInserted(c, atm)
  InformalDef: card *c* is not inserted in ATM *atm*
DomPost: CardInserted(c, atm)
  InformalDef: card *c* is inserted in ATM *atm*

Note again that the domain pre- / postconditions only capture the elementary state transition associated with the operation in the domain, as explained in Section 2.1.2; they do *not* capture the additional conditions that need to strengthen them so as to ensure the goals / requirements on the system. Steps 4 and 5 below are aimed to infer the goals / requirements from which such strengthenings are derived by operationalization (see Fig. 3 and Section 2.2).

The following lexical conventions will be used throughout the paper for choosing names for event types, operations, and domain predicates: an event type will be denoted by a noun; an operation whose applications define an event type will be denoted by a verb with the same root as this event type; and a predicate capturing the corresponding event occurrence will be denoted by a past participle with the same root. In the examples below, the logical interpretations for formal operations and predicates named according to this scheme will be provided only when they are not straighforward.

4.4 Step 3: Generating State Predicates Along Time Lines

The objective of this step is to determine the exact state of each agent instance before and after each interaction.

A *state predicate* attached to some point on a time line is an assertion that captures the state of the corresponding agent at that point. We will represent state predicates by *condition lists* annotating the event trace diagram at the corresponding points. The conditions in a list are implicitly connected by conjunction; a list at some point of a time line captures the state of the corresponding agent at that point. (Condition lists have been used for a long time in STRIPS-like planning systems [55].)

Fig. 7 shows condition lists for the ATM scenario restricted to the security perspective in Fig. 6.

Condition lists are built in a systematic way for each time line from the initial state of this line, the domain pre / postconditions of operations corresponding to the events along the line, and the insertion / deletion of conditions so as to meet the *frame* and *observability* axioms. We first discuss these axioms before explaining the process of generating condition lists.

Frame axiom. A condition holding at some point on a time line remains subsequently true along this line until an interaction event is found whose operation has among its effects to make it false.

The frame axiom essentially says that every condition P that is true at some time point is down propagated to subsequent time points until an operation is applied at some later time point whose effect is to retract it, that is, whose domain postcondition implies $\neg P$. (For a discussion of the frame problem in requirements specification, see [7]).

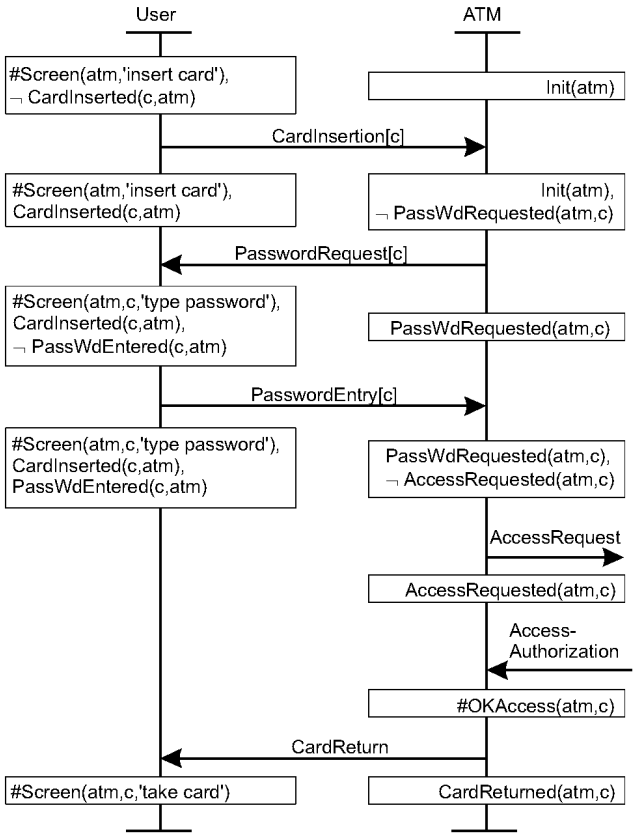

Fig. 7. ATM scenario annotated with condition lists.

For example, the state predicate CardInserted(c, atm) on the User time line in Fig. 7 is down propagated until the time point where the CardReturn event is generated by the ATM agent, because the domain postcondition of the operation ReturnCard corresponding to the latter event is CardReturned(atm, c) which implies ¬ CardInserted(c, atm).

Observability axiom. Every interaction event generated by a source agent must be observable by the corresponding target agent.

This axiom essentially says that every interaction event must be a shared phenomenon observable by both parties [74]. It requires the postcondition of the operation corresponding to the event generated by the source agent to be consistently *reified* in terms of interface objects / attributes that are observable by the target agent. Consistency means here that the local postcondition and the reified version need to be equivalent.

For example, the PasswordRequest event generated by the source ATM agent must be observable by the target User agent; this requires the reification of the local postcondition PassWdRequested(atm, c) into a reified version such as

#Screen(atm, c, 'type passwd')

for observability of this postcondition by the User agent; the local and observable versions of the postcondition need to be equivalent:

#Screen(atm, c, 'type passwd') ⇔ PassWdRequested(atm, c)

Similarly, the AccessAuthorization event generated by the source Bank agent must be observable by the target ATM agent; this requires the reification of the postcondition AccessAuthorized(atm, c) local to the Bank agent into a reified counterpart such as

#OKAccess(atm, c),

observable by the ATM agent (see the bottom of the ATM time line in Fig. 7), with

#OKAccess(atm, c) ⇔ AccessAuthorized(atm, c)

Other reified conditions in Fig. 7 are bound by equivalences such as

#Screen(atm, 'insert card') ⇔ Init(atm)
#Screen(atm, c, 'take card') ⇔ CardReturned(atm, c)

Reified versions of the local postconditions CardInserted(c, atm) and PassWdEntered(c, atm) on the User line are left implicit on the ATM line in Fig. 7 for sake of clarity; they are defined by similar equivalences:

#CardInserted(c, atm) ⇔ CardInserted(c, atm)
#PassWdEntered(c, atm) ⇔ PassWdEntered(c, atm)

The introduction of reified conditions is part of the condition list generation process as we explain it now.

Generating condition lists. The condition lists along an agent's time line are produced systematically as follows.

- The condition list before the first arrow contains the predicate restricting the agent's initial state, to be given by the scenario provider (e.g., Ready(atm) for the ATM agent and the reified counterpart #Screen(atm, 'insert card') for the User agent).
- A condition list *before* an *outgoing* arrow is extended with the domain precondition corresponding to this arrow.
- A condition list *after* an *outgoing* arrow is extended with the domain postcondition corresponding to this arrow; the conditions negated by this postcondition are removed from the list (if any; see the frame axiom).
- A condition list *after* an *ingoing* arrow is extended with the reified counterpart of the domain postcondition corresponding to this arrow on the source agent's line (see the observability axiom); the conditions negated by this extension are removed from the list (if any).

The simple, frequent case of retraction from a condition list occurs when the new condition to be added is the syntactic negation of one already in the list. This happens throughout the two lines in Fig. 7.

Sometimes laws of the form P ⇒ ¬Q have to be found in the available domain theory for the retraction of Q to take place when P is added to the list; e.g., domain laws like

PassWdRequested(atm, c) ⇒ ¬ Init(atm)
CardReturned(atm, c) ⇒ ¬ #OKAccess(atm, c)
#Screen(atm, c, 'take card') ⇒ ¬ CardInserted(c, atm)

are responsible for the deletion of the conditions Init(atm), #OKAccess(atm, c), and CardInserted(c, atm), respectively, when the conditions on the left-hand side of these laws are added to the corresponding condition lists (see Fig. 7).

Removing redundant information. Domain laws can also be used to simplify state predicates containing redundant subformulas. The following *simplification rule* is frequently used:

if a state predicate has the form $P \wedge P1$
and there is a domain law of the form $P \Rightarrow P1$
then this state predicate can be simplified to P

This rule was used on the ATM line in Fig. 7 to remove the condition PassWdRequested(atm, c) when AccessRequested(atm, c) is added to the list; the law used is

$$\text{AccessRequested}(\text{atm}, \text{c}) \Rightarrow \text{PassWdRequested}(\text{atm}, \text{c})$$

The same simplification rule was used to remove the precondition $\neg$CardReturned(atm, c) from the state predicate before the CardReturn event; the law used is

$$\#\text{CardInserted}(\text{c}, \text{atm}) \Rightarrow \neg\, \text{CardReturned}(\text{atm}, \text{c})$$

4.5 Step 4: Inferring Temporal Logic Assertions from a Single Scenario

An event trace diagram represents a temporal sequence of interaction events. On the other hand, a temporal logic assertion is satisfied by temporal sequences of states (see Section 2.1.2). The problem is thus to find out meaningful candidate temporal logic assertions that are *satisfied* by the annotated agent lines in the event trace diagram considered. "Meaningful" means here that the assertions should capture the operational goals underpinning the interactions among agents. Such goals are requirements or assumptions dependent on whether the satisfying temporal line corresponds to a software or an environmental agent, respectively (see Section 2.1.1).

The process of inferring temporal logic assertions from a single event trace diagram has two steps: 1) for each time line, progress and invariance properties about the state transitions along that line are collected as ground facts that causally relate conditions from condition lists, and 2) these facts are then generalized over time and over instances of agents and objects.

4.5.1 Collecting Facts about Agent Behavior

A state transition along an agent's time line occurs when an operation is performed by the agent, that is, when an outgoing arrow is found along the line. The outgoing arrows along an agent's time line are, therefore, considered successively.

Progress properties. A *progress* property is obtained by causally linking the agent's states before and after an outgoing arrow. Let oa denote the current outgoing arrow; let PRE-list_{oa} and POST-list_{oa} denote the condition lists right before and right after oa, respectively. The corresponding progress property capturing the state transition is obtained simply by the fact:

$$\text{PRE-list}_{oa} \rightarrow \mathbf{o}\ \text{POST-list}_{oa}$$

where "$\rightarrow$" denotes the logical implication in the current state and "**o**" denotes the "next" temporal operator (see Section 2.1.2).

Let us consider the agent line in Fig. 7 that will give rise to requirements, that is, the ATM line. The first outgoing arrow corresponds to the PasswordRequest interaction event. We obtain the following fact:

$$\#\text{CardInserted}(\text{c}, \text{atm}) \wedge \text{Init}(\text{atm}) \wedge \neg\, \text{PassWdRequested}(\text{atm}, \text{c}) \rightarrow \mathbf{o}\ [\text{PassWdRequested}(\text{atm}, \text{c}) \wedge \#\text{CardInserted}(\text{c}, \text{atm})]$$

(In this fact, #CardInserted(c, atm) is the reified version on the ATM side of the postcondition CardInserted(c, atm) local to the User line and resulting from the previous CardInsertion ingoing arrow.)

For the CardReturn outgoing arrow we similarly obtain:

$$\#\text{CardInserted}(\text{c}, \text{atm}) \wedge \#\text{OKAccess}(\text{atm}, \text{c}) \rightarrow \mathbf{o}\ \text{CardReturned}(\text{atm}, \text{c})$$

It is worth noticing that progress properties have a stimulus-response pattern. The condition lists above can be rewritten as

PRE-list_{oa}: $\quad E_{oa} \wedge C_{oa} \wedge \text{Pre}_{oa}$
POST-list_{oa}: $\quad C_{oa} \wedge \text{Post}_{oa}$

with

- E_{oa}: reified postcondition of the operation corresponding to the last *ingoing* event preceding the outgoing arrow oa on the agent's line (E_{oa} captures the occurrence of this "stimulus" event),
- Pre_{oa}: domain precondition of the "response" operation corresponding to oa,
- Post_{oa}: domain postcondition of the "response" operation corresponding to oa,
- C_{oa}: other conditions from the agent's condition list left unchanged by the response operation corresponding to oa (C_{oa} captures the agent's "context" in which the stimulus-response interaction takes place).

A factual progress property then takes the stimulus-response form

$$E_{oa} \wedge C_{oa} \wedge \text{Pre}_{oa} \rightarrow \mathbf{o}\ (C_{oa} \wedge \text{Post}_{oa})$$

What the above reformulation captures is the *only* change in the agent's state that results from the specific response to the input stimulus.

Invariance properties. An *invariance* property is obtained by finding a state transition along the agent's line at which some condition in the agent's condition list is true and remains subsequently true along the time line up to some point. The following fact is collected in such a situation:

$$R \wedge ST \rightarrow (R\ W\ N)$$

where

- *ST* is the condition becoming true as a result of the state transition,
- *R* is the condition remaining true up to some point on the agent's line,
- *N* is the agent's state predicate at the subsequent point where *R* is no longer true,
- *W* is the "unless" temporal operator (see Section 2.1.2).

The following invariance property is thereby obtained from the ATM line:

$$\#\text{CardInserted}(\text{c}, \text{atm}) \rightarrow \#\text{CardInserted}(\text{c}, \text{atm})\ W\ \text{CardReturned}(\text{atm}, \text{c})$$

For a richer ATM scenario that would include the initial selection of language lg by the User agent, one would similarly obtain the fact that the language does not change during the entire transaction:

#LanguageSelected(c, atm, lg)
→ #LanguageSelected(c, atm, lg) *W* Init(atm)

4.5.2 Generalizing Facts into Quantified Assertions

Progress/invariance facts along each time line are generalized as follows.

Generalization over time. The various facts collected so far refer to some specific current state and some specific subsequent ones. The first generalization step is, therefore, twofold.

- All assertions are generalized to hold over *any* state; implications "→" are replaced by entailments "⇒" everywhere (recall that P ⇒ Q iff □(P → Q)).
- Progress assertions are generalized so as to refer not necessarily to the next state but to some future one; the "**o**" operator is replaced by the "◊" operator in every such assertion. The reason for this generalization is that other scenarios to be integrated with the one under consideration may refer to intermediate interaction events and states taking place between the occurrence of a stimulus and the occurrence of the corresponding response. Requiring that every response occurs immediately in the next state is thus too restrictive (and in general unachievable) in view of other interactions that may arise in-between from other scenarios.

As a result of this first kind of generalization, two classes of goal specifications are obtained for each time line:

- *Achieve/Cease* goal specifications taking the form

$$\text{PRE-list}_{oa} \Rightarrow \Diamond\ \text{POST-list}_{oa}$$

- *Maintain/Avoid* goal specifications taking the form

$$R \wedge ST \Rightarrow (R\ W\ N)$$

Generalization over instances. The specifications generalized over time are still ground assertions about specific instances of agents and objects. The next step is to generalize them by introduction of quantifiers over instance variables.

All ground assertions are implications of the form

$$A[u] \Rightarrow Q[u,v]$$

where u denotes the variables occurring in the antecedent *A* and v denotes the variables occurring in the consequent *Q* but not in the antecedent *A*. The standard way of generalizing such formulas is to take the form

∀ u:
A[u] ⇒ ∃ v: *Q*[u, v] (*standard generalization*)

that is, the assertion is universally quantified over all instance variables u occurring in the antecedent; its consequent is existentially quantified over the instance variables v that do not occur in the antecedent (existential quantifiers ∃v may of course jump over subformulas in *Q* that contain no occurrences of v). While it is reasonable to generalize by stating that any u involved in *A* is involved in *Q*, it would be much too strong to state that any possible v is unrestrictedly involved in *Q*.

Back to the ATM example, the goal specifications obtained by generalization over time and over instances from the facts collected in Section 4.5.1 include

∀c: Card, atm: ATM
#CardInserted(c, atm) ∧ #OKAccess(atm, c)
⇒ ◊ CardReturned (atm, c)

and

∀ c: Card atm: ATM, lg: AtmLanguage
#LanguageSelected(c, atm, lg)
⇒ #LanguageSelected(c, atm, lg) *W* Init(atm)

(Examples of existential quantification over variables not occurring in the antecedent will appear in Section 5.)

The above generalization rule may be too strong for a specific class of *Achieve/Cease* goal specifications, namely, goals that capture the *first response after a stimulus* in which the identity of the agent to respond is *not* specified. Such goals have the general form:

E[...] ∧ C[..., ra, ...] ∧ Pre[..., ra, ...]
⇒ ◊ (C[..., ra, ...] ∧ Post[..., ra, ...])

where the instance variable ra for the responding agent does *not* occur in the stimulus predicate *E* corresponding to some ingoing arrow but occurs in the context condition *C* and in the domain *Pre*/*Post* of the first outgoing arrow after this stimulus.

For such goals the standard generalization rule would yield

∀... ∀ ra:
E[...] ∧ C[..., ra ...] ∧ Pre[..., ra ...]
⇒ ◊ (C[..., ra, ...] ∧ Post[..., ra, ...])

Given that the stimulus predicate *E* does not mention which target agent should respond, this specification asserts that every agent in a state satisfying C ∧ Pre should eventually respond (e.g., every server in some required state should respond to a client request that does not mention a server identity). Such a requirement is most often much too strong.

A first alternative generalization is to weaken the consequent by introducing an existential quantification over the responding agent:

∀... ∀ ra :
E[...] ∧ C[..., ra, ...] ∧ Pre[..., ra, ...]
⇒ ◊ ∃ ra': (C[..., ra', ...] ∧ Post [..., ra', ...])

that is,

∀... :
E[...] ∧ ∃ ra: (C[..., ra, ...] ∧ Pre[..., ra, ...])
⇒ ◊ ∃ ra': (C[..., ra', ...] ∧ Post [..., ra', ...])
(*weakened consequent generalization*)

This specification now asserts that if there is some target agent in some state required to respond to the stimulus then there should eventually be a response by some target agent. The first response thus includes the determination of an appropriate responding agent left unspecified in the stimulus.

A second alternative generalization is to weaken the above specification further by strengthening its antecedent:

∀... :
E[...] ∧ ∀ ra: (C[..., ra, ...] ∧ Pre[..., ra, ...])
⇒ ◊ ∃ ra': (C[...,ra,'...] ∧ Post[..., ra', ...])

(*strengthened antecedent generalization*)

The latter specification asserts that if every agent is in some state required to respond to the stimulus then there should eventually be a response by some target agent.

Which generalization alternative to choose may depend on the specific problem at hand. We have been unable so far to state a precise selection heuristics; it seems that the strengthened antecedent generalization is often needed when the context condition *C* asserts a *negative* fact. In any case the requirements engineer has to assess whether the generalization may or not lead to a too strong or too weak requirement (see the validation step 6 below).

The case of an unspecified responding agent cannot be illustrated in our ATM ongoing example as any interaction scenario always involves a specific ATM agent. We will illustrate it in Section 5 in the scenario of a passenger calling for a lift; in such a case it does not make sense to specify which lift should be called, and the response to the stimulus will include the determination of one responding lift among all possible ones. The first generalization rule will be applied to obtain the requirement that if there is a lift at the passenger's floor then there shall be a lift opening its doors at that floor; the second generalization rule will be used to obtain the requirement that if there is no lift at the passenger's floor or moving towards it then there shall be a lift moving to that floor.

To conclude this section, we mention two heuristics that can be used to infer additional goal specifications from a single scenario.

Dataflow heuristics. If the occurrence of an interaction event, formalized by a domain postcondition *C*, consumes an attribute produced by the occurrence of a previous interaction event, formalized by a domain postcondition *P*, then the following assertion may be inferred from the scenario:

$$\neg P \wedge \neg C \Rightarrow (\neg C \; W \; P),$$

that is, no consumption is allowed unless a production occurs. For the ATM scenario in Fig. 2, one would thereby obtain the assertion that no cash be delivered unless the requested amount is provided:

$$\forall \text{ c: Card, atm: ATM}$$
$$\neg \text{ \#AmountProvided(c, atm)} \wedge \neg \text{ CashDelivered(atm, c)}$$
$$\Rightarrow \neg \text{ CashDelivered(atm, c)} \; W \; \text{\#AmountProvided(c, atm)}$$

Asserting that every consumption requires a production involves an existential quantification over productions; if *P* contains occurrences of an instance variable referring to a producer agent different from the consumer agent, then these occurrences are existentially quantified (see examples in Section 5).

Strengthened precondition heuristics. If a state predicate *S* holds right before the generation of an interaction event, formalized by a domain postcondition *E*, and is different from the corresponding domain precondition *D*, then *S* might be interpreted as an additional necessary condition to the occurrence of this event:

$$E \Rightarrow \bullet (D \wedge S)$$

For the ATM scenario in Fig. 7, one would thereby obtain the following assertion:

$$\forall \text{ c: Card, atm: ATM}$$
$$\text{CardReturned(atm, c)}$$
$$\Rightarrow \bullet [\neg \text{ CardReturned(atm, c)} \wedge \textit{\#OKAccess(atm, c)}]$$

4.6 Step 5: Integration in the Admissible Set of Specifications

The specifications obtained in step 4 of the goal inference procedure cover the scenario currently considered. They now need to be integrated into the conjunctive set of specifications admissible for the scenarios previously considered, so that the resulting set of specifications remains admissible; this means that the new conjunctive set of specifications must cover (exclude) all the positive (negative) scenarios considered so far, including the current one.

Let *G* denote the conjunctive set of goal specifications admissible for all scenarios previously considered, and let *F* denote the conjunctive set of specifications covering the current scenario *S*.

Integrating a positive scenario. The logical models of *G* are temporal sequences of states capturing behaviors that are so far considered desirable in the envisioned system. Requiring *G* to additionally cover *S* amounts to remove from *G*'s set of models all behaviors that are incompatible with *S*, that is, all behaviors that do not satisfy the specification *F* covering *S*. For example, requiring *G* to cover the scenario of the card being returned in case the access has been authorized requires that all behaviors in which the card is not returned in such a case be removed. The integration of a positive scenario is therefore achieved by intersecting *G*'s and *F*'s sets of models, that is, by conjoining *G* and *F*. The new conjunctive set admissible for all scenarios considered so far is thus simply $G \cup F$.

Instead of just connecting all assertions from *G* and *F* by an implicit conjunction, it may be desirable to "merge" assertions from *G* and *F* that share some common prefix and have the same pattern (*Achieve/Cease* or *Maintain/Avoid*). Such assertions correspond to scenarios starting with a common episode, that is, a common temporal subsequence of interaction events. For the ATM Security perspective, the normal CardReturn scenario in Fig. 6 and the exceptional CardSwallow scenario will share an initial subsequence of events starting with CardInsertion and ending with AccessRequest. For the more extensive GetCash scenario in Fig. 2, the companion ShowBalance scenario will share an initial subsequence of events starting with CardInsertion and ending with ?TransactionType. Common initial subsequences will also be found for scenarios about account transfers, deposits, cash requests exceeding card limit, cancellation requests, too slow data entry, expired cards, cash unavailable, etc.

Let *g* and *f* denote two specifications from *G* and *F* having the same goal pattern and obtained at steps 4 from scenarios sharing a common starting episode. Such specifications are obtained by syntactic search through *G* and *F* to find out pairs of assertions whose antecedent shares a longest prefix conjunct, say *P*, in which the variables are quantified the same way; *P* captures some common state predicate characterizing the common episode.

The integration of *g* and *f* is then achieved by means of the following *coverage rules:*

$$\frac{g:\ P \wedge G1 \Rightarrow \Diamond G2, \quad f:\ P \wedge F1 \Rightarrow \Diamond F2}{g \wedge f:\ P \Rightarrow [(G1 \rightarrow \Diamond G2) \wedge (F1 \rightarrow \Diamond F2)]}$$

$$\frac{g:\ P \wedge G1 \Rightarrow (G2\ W\ G3), \quad f:\ P \wedge F1 \Rightarrow (F2\ W\ F3)}{g \wedge f:\ P \Rightarrow [(G1 \rightarrow (G2\ W\ G3)) \wedge (F1 \rightarrow (F2\ W\ F3))]}$$

Note that the inner implication in the conclusion part of these rules is not an entailment. For the first rule, for example, an inner entailment would prescribe the eventual reaching of the target predicate *G2* from any future state in which *G1* holds, without *P* necessarily holding, which is not what the premise of the rule requires (recall that $P \Rightarrow Q$ means $\Box(P \rightarrow Q)$). The soundness of the two rules above can easily be proved using the proof theory of temporal logic.

Also note that the case where no *g* and *f* are found with a common prefix conjunct in their antecedent corresponds to the particular case where *P* = **true** in the rules above; the integration of scenarios with no common episode is achieved just by conjunction of their goal assertions.

Let us illustrate the use of coverage rules in our ATM example. Suppose that the new ShowBalance scenario has to be integrated with the GetCash scenario shown in Fig. 2; one of the goal specifications inferred for the latter at a previous iteration of the procedure is

$$\begin{aligned}&\forall\ c: Card,\ atm: ATM\\ &\#CardInserted(c, atm) \wedge \#OKAccess(atm, c)\\ &\wedge \#TransactTypeEntered(c, atm, \text{'getCash'})\\ &\wedge \neg AmountAsked(atm, c)\\ &\Rightarrow \Diamond AmountAsked(atm, c)\end{aligned}$$

The following goal specification is found to share a longest common prefix among those covering the ShowBalance scenario:

$$\begin{aligned}&\forall\ c: Card,\ atm: ATM\\ &\#CardInserted(c, atm) \wedge \#OKAccess(atm, c)\\ &\wedge \#TransactTypeEntered(c, atm, \text{'showBalance'})\\ &\wedge \neg BalanceDisplayed(atm, c)\\ &\Rightarrow \Diamond BalanceDisplayed(atm, c)\end{aligned}$$

The coverage rule then yields the following integrated specification:

$$\begin{aligned}&\forall\ c: Card,\ atm: ATM\\ &\#CardInserted(c, atm) \wedge \#OKAccess(atm, c)\\ &\Rightarrow [\#TransactTypeEntered(c, atm, \text{'getCash'})\\ &\quad \wedge \neg AmountAsked(atm, c)\\ &\quad\ \rightarrow \Diamond AmountAsked(atm, c)\\ &\quad \wedge \#TransactTypeEntered(c, atm, \text{'showBalance'})\\ &\quad \wedge \neg BalanceDisplayed(atm\ c)\\ &\quad\ \rightarrow \Diamond BalanceDisplayed(atm, c)]\end{aligned}$$

Integrating a negative scenario. Requiring the admissible conjunctive set *G* to exclude the new negative scenario *S* amounts to remove from *G*'s set of models all behaviors that are compatible with *S*, that is, all behaviors that satisfy the specification *F* covering *S*. For example, requiring *G* to exclude the scenario of the card being returned after three unsuccessful password trials requires that all behaviors in which the card is returned in such a case be removed. The integration of a negative scenario is therefore achieved by conjoining *G* and $\neg F$ which yields a global specification taking the form

$$\bigwedge_i g_i \wedge (\bigvee_k \neg f_k)$$

where the only $\neg f_k$ remaining are those whose conjunction with all g_i in *G* does not yield **false**. The remaining assertions f_k precisely capture what is undesirable in the negative scenario.

Let *f* denote such an undesirable assertion remaining in *F*, and let *g* denote a longest common prefix assertion from *G*. The integration of *g* and *f* is then achieved by means of *exclusion rules* such as

$$\frac{g:\ P \wedge G1 \Rightarrow \Diamond G2, \quad f:\ P \wedge F1 \Rightarrow \Diamond (C_T \wedge Post_T)}{g \wedge \neg f:\ P \Rightarrow [(G1 \rightarrow \Diamond G2) \wedge (F1 \rightarrow (C_T \wedge \neg Post_T)\ W\ F3))]}$$

In this rule, *P* denotes the longest common prefix conjunct, $Post_T$ denotes the domain postcondition resulting from the undesirable event, C_T denotes the conjunction of all other conditions in the condition list remaining unchanged by the undesirable state transition, and *F3* denotes a domain-dependent condition until which $\neg Post_T$ must remain permanently true. Indeed, we want to precisely exclude those state transitions in the context C_T that would result in $Post_T$. The default case F3 = **false** corresponds to requiring $\neg Post_T$ to hold forever; this could be too restrictive in some cases, and a weaker *F3* may be elicited instead (see the examples below).

In case the negative scenario is covered by a *Maintain/Avoid* goal assertion, the following exclusion rule may be used:

$$\frac{g:\ P \wedge G1 \Rightarrow (G2\ W\ G3), \quad f:\ P \wedge F1 \Rightarrow \Box F2}{g \wedge \neg f:\ P \Rightarrow [(G1 \rightarrow (G2\ W\ G3)) \wedge (F1 \rightarrow \Diamond \neg F2)]}$$

Back to the ATM example again, suppose that the scenario of the card being returned after three unsuccessful password trials has been given as negative scenario. For this scenario step 4 of the goal inference procedure will generate the following assertion covering it:

$$\begin{aligned}&\forall c: Card,\ atm: ATM\\ &\#CardInserted(c, atm) \wedge \#KO\text{-}PassWd(atm, c, 3)\\ &\Rightarrow \Diamond CardReturned(atm, c)\end{aligned}$$

The following goal assertion was inferred in Section 4.5 for the related CardReturn scenario:

$$\begin{aligned}&\forall c: Card,\ atm: ATM\\ &\#CardInserted(c, atm) \wedge\ \#OKAccess(atm, c)\\ &\Rightarrow \Diamond CardReturned(atm, c)\end{aligned}$$

The application of the first exclusion rule above to these two assertions yields the following integrated goal assertion to cover the previous positive scenario and exclude the current negative one:

$$\begin{aligned}&\forall c: Card,\ atm: ATM\\ &\#CardInserted\ (c, atm)\\ &\Rightarrow [\ \#OKAccess(atm, c) \rightarrow \Diamond CardReturned(atm, c)\\ &\quad \wedge \#KO\text{-}PassWd(atm, c, 3) \rightarrow \neg CardReturned(atm, c)\ W\ F3)]\end{aligned}$$

In this case, the requirement of never returning the card might be too strong, and the requirements engineer should elicit the condition *F3* for returning the card at some point.

Note that the role of negative scenarios is not to infer brand new specifications but rather to avoid overgeneralizations from positive scenarios. The inductive learning of new

specifications from negative examples won't make much sense in the context of requirements specification, because a Closed World assumption doesn't make sense in this context (everything that is not explicitly excluded is not necessarily admissible). Rather than a fairly large number of negative scenarios enumerating prohibited behaviors, a limited number of well-chosen ones is expected to be provided as warnings about specific exceptional situations [12], [59].

The goal inference procedure could be simplified by turning negative scenarios into positive ones. Instead of a negative scenario capturing some undesired behavior in some specific situation, the stakeholder(s) providing the scenario might be asked to provide an explicit positive scenario instead in order to capture an alternative desirable behavior in that specific situation. Experience however suggests that negative scenarios are often provided in the first place when exceptional situations need to be pointed out.

Note finally that our procedure is conceptually biased when no additional domain knowledge is used; like any learning technique, the admissible set of goal specifications inferred is bound to the vocabulary used to describe the scenarios given as examples [41].

4.7 Steps 6 and 7: Validation and Further Goal Elicitation

Unlike deduction, inductive inference is not necessarily sound; what is inferred is not necessarily true in the interpretations of interest. The inferred assertions need therefore to be validated. In our context, this means that the goals, requirements and assumptions obtained at each iteration over a new scenario need to be submitted to the requirements engineer and the stakeholders in order to check their adequacy and adjust them if necessary.

As already mentioned in Section 4.5.2, a first validation task is to check whether the generalized goal specifications are not too strong or too weak.

A frequent adjustment is the *temporal strengthening* of Achieve/Cease goals. The specifications obtained so far take the form

$$P \Rightarrow \Diamond Q$$

Real-time restrictions most often need to be put on them, leading to specifications taking the form

$$P \Rightarrow \Diamond_{\leq d} Q$$

Examples of temporally strengthened goals obtained at this stage include the following requirement on the ATM side:

$\forall$c: Card, atm: ATM
#CardInserted(c, atm) $\wedge$ Init(atm) $\wedge \neg$PassWdRequested(atm, c)
$\Rightarrow \Diamond_{\leq \delta}$ [PassWdRequested(atm, c) $\wedge$ #CardInserted(c, atm)]

and the following assumption on the User side:

$\forall$c: Card, atm: ATM
#Screen(atm, c, 'type passwd') $\wedge \neg$ PassWdEntered(c, atm)
$\Rightarrow \Diamond_{\leq \varepsilon}$ PassWdEntered(c, atm)

Once the goals inferred from scenarios have been checked and possibly adjusted, new goals can be elicited from them.

A first, informal technique consists in asking WHY questions about the inferred goal to elicit more abstract supergoals, and HOW questions to find out more concrete subgoals and companion missing goals.

In our ongoing ATM example, a WHY question about the real-time goals above would lead to the elicitation of higher-level goals such as *Achieve*[PromptService] and *Achieve*[ServiceAvailable]. Coming back to the requirement limiting repeated password trials, a WHY question will lead to the goal *Avoid*[PassWordSearched]. A new WHY question about the latter will lead to the goal *Avoid*[IllegalAccessToAccount].

An alternative, formal technique consists in using formal goal refinement/abstraction patterns [13] bottom-up to obtain more abstract supergoals, and top-down to obtain more concrete subgoals and companion missing goals. We come back to this formal technique in Section 6.1 below.

5 A Complete Example: The Lift System

The purpose of this section is to show a complete run of the goal inference procedure on a nontrivial benchmark. We have chosen the Lift exemplar [50] for that purpose because it involves some less trivial treatment of quantifiers. Some conclusions and prospects for automated support of this method will be discussed from there.

Fig. 8 shows a first scenario of interaction between a passenger and a lift. This scenario has been given as a positive one by stakeholders and represented by an event trace diagram by the requirements engineer.

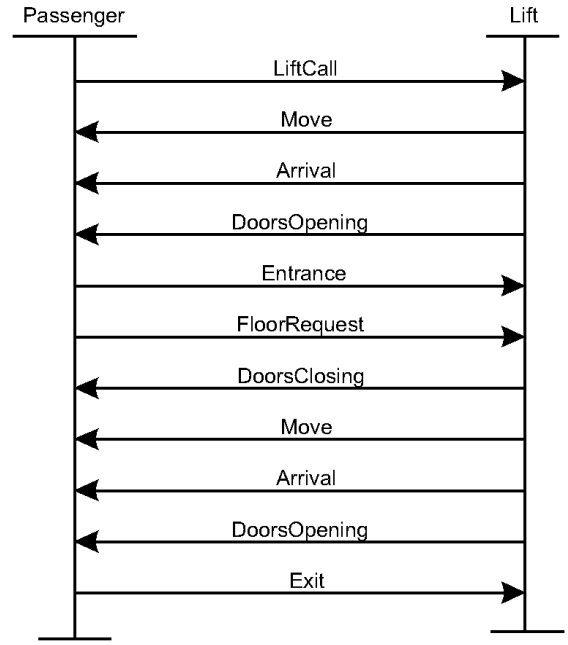

Fig. 8. Positive scenario for the lift exemplar.

5.1 Step 1: Cleaning Up Scenarios

There is no sequence of consecutive interaction events of the same type; the *event aggregation* rule is therefore not applicable.

We will assume that there is no wish here to *restrict the scope* of the scenario to some specific concern. Alternatively, the requirements engineer might restrict the scope of the scenario in Fig. 8 to the SafetyGoal perspective by retaining only those interactions concerned with the Passenger's safety during transportation.

The interaction events in Fig. 8 are then reviewed successively by the requirements engineer to check whether each of them corresponds to a shared phenomenon between Passenger and Lift instances. The only questionable events are Entrance and Exit. At first sight it seems that those events could be dropped since there is no apparent need for the lift to observe the passenger's entrance/exit to/from the lift. After checking with stakeholders it appears however that there are expectations on the system to manage lift overloads. An additional sensor device has therefore to be foreseen which will make the Entrance and Exit events observable by the Lift. These events are therefore not removed from the scenario.

5.2 Step 2: Mapping Interaction Events to Operations

For each interaction event an operation is identified whose application corresponds to it, e.g.,

LiftCall yields the operation CallLift,
Move yields the operation MoveTo,
FloorRequest yields the operation RequestFloor, etc.

The requirements engineer identifies the inputs and outputs of each such operation and introduces typed variables to name them. Beside names for agent instances, names for attributes of the corresponding event type may need to be introduced among inputs/outputs. The above operations become

CallLift(p, f)	p: Passenger, f: Floor
MoveTo(l, f)	l: Lift, f: Floor
RequestFloor(p, l, f)	p: Passenger, l: Lift, f: Floor

The event types in the event trace diagram are then decorated with the corresponding variables introduced (see the arrow labels in Fig. 9).

The domain pre/postconditions capturing the elementary state transition produced by the operation are elicited or retrieved from domain knowledge. The following conditions are obtained:

CallLift(p, f) :	**DomPre**: ¬ LiftCalled(p,f) **DomPost**: LiftCalled(p,f)
MoveTo(l, f) :	**DomPre**: ¬ Moving(l,f) **DomPost**: Moving(l,f)
Arrive(l, f) :	**DomPre**: ¬ LiftAt(l,f) **DomPost**: LiftAt(l,f)
OpenDoors(l) :	**DomPre**: l.Doors = 'closed' **DomPost**: l .Doors = 'open'
CloseDoors(l) :	**DomPre**: l.Doors = 'open' **DomPost**: l.Doors = 'closed'
Enter(p, l) :	**DomPre**: ¬ In(p,l) **DomPost**: In(p,l)
Exit(p, l) :	**DomPre**: In(p,l) **DomPost**: ¬ In(p,l)
RequestFloor(p, l, f):	**DomPre**: ¬ FloorRequested(p,l,f) **DomPost**: FloorRequested(p,l,f)

The logical interpretation of the predicate LiftAt(l, f) above is: "lift l is stopped at floor f."

5.3 Step 3: Generating State Predicates Along Time Lines

Fig. 9 shows the result of applying the condition list generation procedure to each time line. The reified versions of

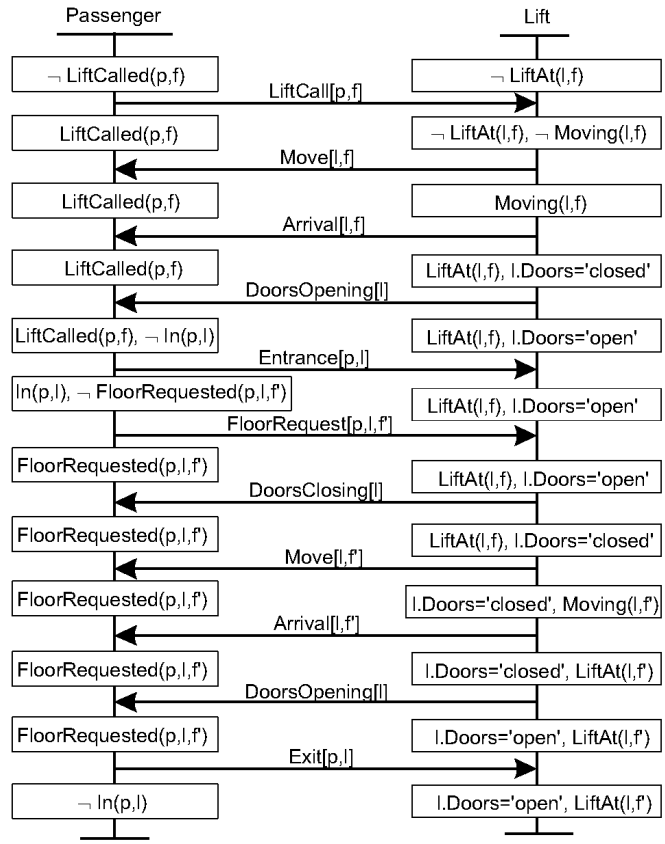

Fig. 9. Annotated event trace diagram for the lift exemplar.

conditions propagated along ingoing arrows are not shown there for sake of clarity. For observability from the Passenger line of state transitions along the Lift line, the reified conditions are defined by

#Moving(l,f) ⇔ Moving(l,f)
#LiftAt(l,f) ⇔ LiftAt(l,f)

In practice, those reified versions will be made concrete through interface objects and predicates such as:

#Moving(l,f): l.Light(f) = 'on' ∧ f.Light = 'on'
#LiftAt(l,f): l.Bell(/) = 'on' ;

we don't go further into such user interface details here.

For observability from the Lift line of state transitions along the Passenger line, the reified conditions are defined by

#LiftCalled(p,f) ⇔ LiftCalled(p,f)
#In (p, l) ⇔ In (p, l)
#FloorRequested (p, l, f') ⇔ FloorRequested (p, l, f')

The condition lists generated successively from the initial state for the Lift line are:

1. ¬ LiftAt(l,f)
2. ¬ LiftAt(l,f), ¬ Moving(l,f), #LiftCalled(p,f)
3. Moving(l,f), #LiftCalled(p,f)

In the list above, the condition ¬ Moving(l, f) was retracted because the condition Moving(l f) was inserted; the redundant condition ¬ LiftAt(l, f) was removed by use of the simplification rule and the domain law

Moving(l,f) ⇒ ¬ LiftAt(l,f)

After the Arrival event the generated condition list is

4. LiftAt(l, f), l.Doors = 'closed,' #LiftCalled(p, f)

The condition Moving(l, f) was retracted there because of the contraposed version of the domain law above. Continuing from there we get

5. LiftAt(l, f), l.Doors = 'open,' #LiftCalled(p, f)

where the condition l.Doors = 'closed' was retracted because the condition l.Doors='open' was inserted. After generation of the Entrance event from the Passenger line we get

6. LiftAt(l, f), l.Doors = 'open,' #In(p, l)

In the list above, the condition #LiftCalled(p,f) was retracted because of the domain law

$$In(p, l) \Rightarrow \neg\, LiftCalled(p, f)$$

After the next, FloorRequest ingoing arrow on the Lift line we get

7. LiftAt(l, f), l.Doors = 'open,' #FloorRequested(p, l, f')

The redundant condition #In(p, l) was removed from this list by application of the simplification rule and the domain law

$$FloorRequested(p, l, f') \Rightarrow In(p, l)$$

From the next, DoorsClosing outgoing arrow on the Lift line the remaining condition lists are successively:

8. LiftAt(l, f), l.Doors = 'closed,' #FloorRequested(p, l, f')
9. l.Doors = 'closed', Moving(l, f'), #FloorRequested(p, l, f')
10. l.Doors = 'closed', LiftAt(l, f'), #FloorRequested(p, l, f')
11. l.Doors = 'open', LiftAt(l, f'), #FloorRequested(p, l, f')
12. l.Doors = 'open', LiftAt(l, f'), ¬ #In(p, l)

5.4 Step 4: Inferring Temporal Logic Assertions from a Single Scenario

Collecting facts about agent behavior. The following ground facts are successively collected along the Lift time line.

1) *Progress facts:*

2-3. $\neg$ LiftAt(l, f) $\wedge$ $\neg$ Moving(l, f) $\wedge$ #LiftCalled(p, f)
$\rightarrow$ **o** [Moving(l, f) $\wedge$ #LiftCalled(p, f)]

3-4. Moving(l, f) $\wedge$ #LiftCalled(p, f)
$\rightarrow$ **o** [LiftAt(l, f) $\wedge$ l.Doors = 'closed' $\wedge$ #LiftCalled(p, f)]

4-5. LiftAt(l, f) $\wedge$ l.Doors = 'closed' $\wedge$ #LiftCalled(p, f)
$\rightarrow$ **o** [LiftAt(l, f) $\wedge$ l.Doors = 'open' $\wedge$ #LiftCalled(p, f)]

7-8 LiftAt(l, f) $\wedge$ l.Doors = 'open' $\wedge$ #FloorRequested(p, l, f')
$\rightarrow$ **o** [LiftAt(l, f) $\wedge$ l.Doors = 'closed'
$\wedge$ #FloorRequested(p, l, f')]

8-9. LiftAt(l, f) $\wedge$ l.Doors = 'closed' $\wedge$ #FloorRequested(p, l, f')
$\rightarrow$ **o** [l.Doors = 'closed' $\wedge$ Moving(l, f')
$\wedge$ #FloorRequested(p, l, f')]

9-10. l.Doors='closed' $\wedge$ Moving(l, f') $\wedge$ #FloorRequested(p, l, f')
$\rightarrow$ **o** [l.Doors = 'closed' $\wedge$ LiftAt(l, f')
$\wedge$ #FloorRequested(p, l, f')]

10-11. l.Doors ='closed' $\wedge$ LiftAt(l, f') $\wedge$ #FloorRequested(p, l, f')
$\rightarrow$ **o** [l.Doors = 'open' $\wedge$ LiftAt(l, f')
$\wedge$ #FloorRequested(p, l, f')]

2) *Invariance facts*:

At state transition resulting in

ST: l.Doors = 'open'

the predicate

R: LiftAt(l, f)

is true and remains true along the Lift's time line up to some point; at the subsequent point where *R* is no longer true the state predicate

N: l.Doors='closed' $\wedge$ Moving(l, f') $\wedge$ #FloorRequested(p, l, f')

is true. Hence the invariance fact

LiftAt(l, f) $\wedge$ l.Doors = 'open'
$\rightarrow$ LiftAt(l, f) *W*
[l.Doors='closed' $\wedge$ Moving(l, f') $\wedge$ #FloorRequested(p, l, f')]

which states that the lift at floor f with doors open remains permanently at that floor unless it moves to the requested floor with its doors being closed.

At state transition resulting in

ST: Moving (l, f')

the predicate

R: l.Doors = 'closed'

is true and remains true along the Lift's time line up to some point; at the subsequent point where *R* is no longer true the state predicate

N: l.Doors='open' $\wedge$ LiftAt(l, f') $\wedge$ #FloorRequested(p, l, f')

is true. Hence the invariance fact

l.Doors = 'closed' $\wedge$ Moving(l, f')
$\rightarrow$ l.Doors= 'closed' *W*
[l.Doors='open' $\wedge$ LiftAt(l, f') $\wedge$ #FloorRequested(p, l, f')]

which states that the lift moving with doors closed keeps its doors permanently closed unless the lift is at its destination floor.

Generalizing facts into quantified assertions. The rules in Section 4.5.2 for generalizing facts over time and over agent / object instances yield the following goal specifications.

Achieve/Cease goals:

2-3. $\forall$p: Passenger, f: Floor
#LiftCalled(p, f) $\wedge$ ($\forall$l: Lift) ($\neg$ LiftAt(l, f) $\wedge$ $\neg$ Moving(l, f))
$\Rightarrow$ $\Diamond$ [($\exists$l: Lift) Moving(l, f) $\wedge$ #LiftCalled(p, f)]

3-4. $\forall$ p: Passenger, f: Floor, l: Lift
#LiftCalled(p, f) $\wedge$ Moving(l, f)
$\Rightarrow$ $\Diamond$ [LiftAt(l, f) $\wedge$ l.Doors = 'closed' $\wedge$ #LiftCalled(p, f)]

4-5.. $\forall$ p: Passenger, f: Floor, l: Lift
#LiftCalled(p, f) $\wedge$ LiftAt(l, f) $\wedge$ l.Doors = 'closed'
$\Rightarrow$ $\Diamond$ [LiftAt(l, f) $\wedge$ l.Doors = 'open' $\wedge$ #LiftCalled(p, f)]

7–8. $\forall$ p: Passenger, f, f': Floor, l: Lift
#FloorRequested(p, l, f') $\wedge$ LiftAt(l, f) $\wedge$ l.Doors = 'open'
$\Rightarrow$ $\Diamond$ [LiftAt(l, f) $\wedge$ l.Doors = 'closed'
$\wedge$ #FloorRequested(p, l, f')]

8–9. $\forall$ p: Passenger, f, f': Floor, l: Lift
#FloorRequested(p, l, f') $\wedge$ LiftAt(l, f) $\wedge$ l.Doors = 'closed'
$\Rightarrow$ $\Diamond$ [l.Doors = 'closed' $\wedge$ Moving(l, f')
$\wedge$ #FloorRequested(p, l, f')]

9–10. $\forall$ p: Passenger, f': Floor, l: Lift
#FloorRequested(p, l, f') $\wedge$ l.Doors = 'closed' $\wedge$ Moving(l, f')
$\Rightarrow$ $\Diamond$ [l.Doors = 'closed' $\wedge$ LiftAt(l, f')
$\wedge$ #FloorRequested(p, l, f')]

10–11. $\forall$ p: Passenger, f': Floor, l: Lift
#FloorRequested(p, l, f') $\wedge$ l.Doors = 'closed' $\wedge$ LiftAt(l, f')
$\Rightarrow \Diamond$ [l.Doors = 'open' $\wedge$ LiftAt(l, f')
$\wedge$ #FloorRequested(p, l, f')]

Note that the strengthened antecedent generalization rule was chosen for the goal 2-3; the latter corresponds to the first response to a stimulus LiftCall[p, f] in which no target Lift agent is specified to respond to the stimulus. The resulting assertion adequately prescribes a responding lift move if there is no lift at the passenger's floor or moving towards it. The weakened consequent generalization rule would have produced a too strong requirement here, namely, that a responding move occurs if there is at least one lift not at the passenger's floor and not moving towards it. Also note that the standard generalization rule has been applied to the goal 7-8 corresponding to the first response to a stimulus FloorRequest[p, l, f'], because there the stimulus specifies the agent l to respond.

Maintain/Avoid goals:

$\forall$ l: Lift, f: Floor
LiftAt(l, f) $\wedge$ l.Doors = 'open'
$\Rightarrow$ LiftAt(l, f) W
[l.Doors = 'closed' $\wedge$ $\exists$ f': Floor, p: Passenger
Moving(l, f') $\wedge$ #FloorRequested(p, l, f')]

$\forall$ l: Lift, f': Floor,
l.Doors = 'closed' $\wedge$ Moving(l, f')
$\Rightarrow$ l.Doors = 'closed' W
[l.Doors = 'open' $\wedge$ LiftAt(l, f') $\wedge$ $\exists$ p: Passenger
#FloorRequested(p, l, f')]

Note that the standard generalization rule produced an existential quantification over f' in the first *Maintain* goal, because this variable does not occur in the antecedent of that goal, but a universal quantification over f' in the second *Maintain* goal, because f' there appears in the antecedent.

All goal specifications above have been inferred from the Lift line; they correspond therefore to *requirements* to be achieved by the lift control software. The following goal specifications, inferred similarly from the Passenger line, yield *assumptions*:

$\forall$ p: Passenger, f: Floor, l: Lift
LiftCalled(p, f) $\wedge \neg$ In(p, l) $\wedge$ #LiftAt(l, f) $\wedge$ l.Doors = 'open'
$\Rightarrow \Diamond$ [In(p, l) $\wedge$ #LiftAt(l, f) $\wedge$ l.Doors = 'open'
$\wedge$ ($\exists$ f': Floor) $\neg$ FloorRequested(p, l, f')]

$\forall$ p: Passenger, f, f': Floor, l: Lift
In(p, l) $\wedge \neg$ FloorRequested(p, l, f') $\wedge$ #LiftAt(l, f) $\wedge$ l.Doors = 'open'
$\Rightarrow \Diamond$ [FloorRequested(p, l, f') $\wedge$ #LiftAt(l, f) $\wedge$ l.Doors = 'open']

The first assumption states that a passenger who called a lift will get into it when the lift is stopped at her floor with the doors open; the second assumption states that a passenger who entered a lift will indicate her destination floor while the lift is stopped with the doors open.

Finally, we can use the dataflow heuristics from Section 4.5.2 to generate more specifications.

There is a producer-consumer relationship between the LiftCall[p, f] event and the Arrival[l, f] event; the latter consumes the attribute f produced by the former. We thus obtain:

$\forall$ l: Lift, f: Floor
$\neg$($\exists$ p: Passenger)#LiftCalled(p, f) $\wedge \neg$ LiftAt(l, f)
$\Rightarrow$ [$\neg$ LiftAt(l, f) W ($\exists$ p: Passenger)#LiftCalled(p, f)]

Note the existential quantification over the producer agent, as explained in Section 4.5.2. There is a similar producer-consumer relationship between the FloorRequest[p, l, f'] event and the Arrival[l, f'] event; the latter consumes the attribute f' produced by the former. We thus obtain:

$\forall$ l: Lift, f': Floor
$\neg$($\exists$ p: Passenger)#FloorRequested(p, l, f') $\wedge \neg$ LiftAt(l, f')
$\Rightarrow$ [$\neg$ LiftAt(l, f') W ($\exists$ p: Passenger)#FloorRequested(p, l, f')]

These specifications assert that unnecessary lift moves should be avoided; the first states that a lift should move to some floor only if called from that floor whereas the second states that a lift should move to some floor only if requested by an inside passenger. As we will see in Section 6.3, conflict analysis applied to these declarative specifications reveals that they are divergent [45]; the conflict will be resolved by a straightforward weakening to cover both of them.

5.5 Step 5: Integration in the Set of Admissible Specifications

Suppose that a second positive scenario is given in which the lift is already at the floor where the passenger issues a lift call (see Fig. 10).

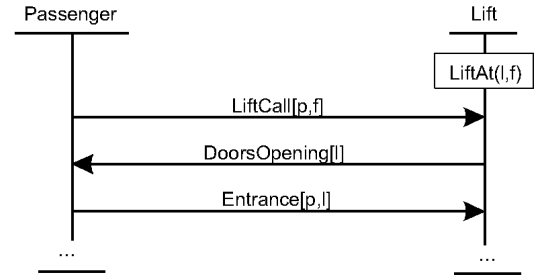

Fig. 10. Next positive scenario for the lift exemplar.

The initial state on the Lift line is now LiftAt(l,f); step 4 will directly infer

$\forall$ p: Passenger, f: Floor, l: Lift
#LiftCalled(p, f) $\wedge$ LiftAt(l, f) $\wedge$ l.Doors = 'closed'
$\Rightarrow \Diamond$ [LiftAt(l, f) $\wedge$ l.Doors = 'open' $\wedge$ #LiftCalled(p, f)]

The procedure when trying to match assertions to find out longest common prefixes for the application of a coverage rule will find that this specification is already among those inferred from the previous scenario. (Alternatively, one might introduce a preliminary stage of event trace analysis to discover scenario overlaps—see discussion below.)

Suppose now that a negative scenario is provided next, which resembles the normal scenario in Fig. 8 except that a LiftOverload event is generated by the Lift agent between the Entrance and FloorRequest events—in other words, a counter-scenario is given in which the lift moves on in spite of being overloaded.

The goal specifications derived from this negative scenario by step 4 include:

f : ∀ l: Lift, p: Passenger, f, f': Floor
LiftAt(l, f) ∧ l.Doors = 'open' ∧ #FloorRequested(p, l, f')
∧ *Overloaded (l*)
⇒ ◊ [LiftAt(l, f) ∧ l.Doors = 'closed'
∧ #FloorRequested(p, l, f')]

(Note that the reified version #Overloaded(l) on the Passenger's side will typically correspond to an alarm ringing.)

The longest common prefix found for applying the exclusion rule will correspond to an initial subsequence of events starting with LiftCall and ending with Entrance. This prefix is found in the goal specification 7-8 inferred from the first positive scenario:

g: ∀ l: Lift, p: Passenger, f, f': Floor
#FloorRequested(p, l, f') ∧ LiftAt(l, f) ∧ l.Doors = 'open'
⇒ ◊ [LiftAt (l, f) ∧ l.Doors = 'closed'
∧ #FloorRequested(p, l, f')]

The integration by means of the first exclusion rule in Section 4.6 yields the goal specification

∀ l: Lift, p: Passenger, f, f': Floor
LiftAt(l, f) ∧ l.Doors = 'open' ∧ #FloorRequested(p, l, f')
⇒[◊ (LiftAt(l ,f) ∧ l.Doors ='closed' ∧ #FloorRequested(p, l, f'))
∧ Overloaded(l) → (LiftAt(l, f) ∧ l.Doors ≠ 'closed') *W* F3]

In case of lift overload, the specification above states that the lift must remain at the same floor with the doors remaining open, unless something happens. One obvious choice for *F3* here is ¬ In(p, l), that is, the passenger who caused the overload by entering the lift should leave it in order to get the doors closed and allow the lift to behave as required by the rest of the specification.

5.6 Steps 6 and 7: Validation and Further Elicitation of Goals

Validation and elicitation of more abstract goals proceed in parallel with steps 4 and 5. The various specifications inferred from these steps are reviewed for adequacy. In particular, the assertions produced by the generalization rules at step 4 are checked to see if the result is adequate, too strong or too weak. For example, the wrong selection of the weakened consequent generalization rule to produce a stronger version for the *Achieve* goal 2-3 above should have been detected thereby.

The scope of the temporal operators are also checked in the various specifications obtained to see whether the latter are not too weak (in case of ◊-operators) or to strong (in case of □-operators). Real-time constraints would thereby be superimposed for the various responses along the Lift line. More abstract and more concrete goals can be elicited informally by asking WHY and HOW questions about the goal specifications inferred. For example, WHY questions about the goals *Avoid*[LiftOverloaded] and *Maintain*[DoorsClosedWhileMoving] inferred above will lead to higher-level Safety goals concerning Passenger agents.

More abstract goals can also be obtained formally by application of formal *abstraction patterns* to the fine-grained goal specifications above, e.g., to derive a more coarse-grained goal specification that aggregates the *Achieve* goals 2-3, 3-4, and 4-5. We will illustrate this in Section 6.1 below.

5.7 Discussion

For an agent line with *M* ingoing events and *N* outgoing events in a single scenario, the goal inference procedure will generate *M + N* condition lists, *N Achieve/Cease* goal specifications, and at most *N Maintain/Avoid* goal specifications (in general this will be much less as invariant conditions are unlikely to be found from every state transition). The admissible set of specifications does not expand with as many specifications at each iteration over a new scenario; specifications produced from common event subsequences are not added as they are already there, and the specifications corresponding to the last, different interaction in longest common subsequences are merged.

An obvious optimization here is to avoid at each iteration recomputing condition lists and reinferring specifications that were already computed/inferred before. To achieve this, a preliminary step at each iteration could detect common subsequences of interaction events between the current scenario and the previous ones so that only new condition lists and specifications are recomputed; the latter correspond to the last state transition before such subsequences differ.

As far as requirements are concerned, only time lines that correspond to software agents need to be considered. It may be helpful, however, to generate the assumptions from time lines associated with environmental agents in order to check whether such assumptions are realistic, that is, whether they are likely to be met by environmental agents as requisites for achieving higher-level goals. If this is not the case, another responsibility assignment should be made (see Fig. 3 and Section 2.2).

Since every outgoing event produces an *Achieve/Cease* goal specification, fine-grained scenarios will correspondingly produce fine-grained goal specifications. For a long, complex scenario such fine-grained specifications are not necessarily the ones one may wish to work with. There are several ways to obtain more abstract and/or more coarse-grained specifications. The first one is to apply event aggregation and scope restriction rules as discussed in Section 4.2. A second one, not explored in this paper, is to aggregate recurring subsequences of interaction events into *interaction chunks*, corresponding to macro-operators in the planning sense [55]. A third one is to apply *formal abstraction patterns* to generate parent goals from the fine-grained goals generated by the procedure; the latter are then seen as milestone subgoals to achieve the parent goal. We discuss this technique in Section 6.1 below.

The complete run of our method on the lift system exemplar should help visualizing what level of tool support can be provided. The outer loop of the procedure would be under control of the tool. After having asked the requirements engineer (RER) for a new event trace diagram, the tool (T) suggests applying the clean up rules available to the scenario entered by RER (*step 1*); while event aggregation can be done automatically, scope restriction and elimination of nonshared phenomena must be under control of RER. T then generates the abstract syntax tree for the possibly simplified scenario from its diagrammatic representation. Listing the proposed operation names produced from the corresponding events, T asks RER to identify and supply

names for operation inputs/outputs and for their corresponding type. T then asks for the domain pre-/postconditions of these operations or retrieves them from the domain knowledge base (*step 2*). The initial state along each agent line must also be provided by RER. From there T automatically generates all condition lists along each line and decorates the abstract syntax tree with the annotations generated (*step 3*). In this process, the retraction of a condition that gets negated is easily achieved by tree matching; for retraction of implied conditions and removal of redundant conditions T needs domain laws to be supplied by RER if not available in the domain knowledge base; the simplification of condition lists is then achieved by deletion of matching subtrees. The progress and invariance facts along each time line are then automatically collected by T from the annotated abstract syntax tree (*step 4.1*). Generalization over time and over instances is performed next by T (*step 4.2*); in case of a stimulus predicate not referencing the target responding agent, RER is asked to select the appropriate generalization rule to be applied by T. Integration into the admissible set of specifications using the coverage/exlusion rules is then performed by T which searches for largest common prefix subtrees (*step 5*). RER is then asked to validate the goal specifications generated and to adapt them, if necessary, using the structural editor available. If requested, T then generates more abstract specifications by application of abstraction patterns whose leaves match specifications corresponding to successive interactions (*step 6*).

Our effort so far has been concentrated on method development and experimentation rather than tool development; we do not expect any major implementation problem as the current GRAIL/KAOS environment [15] has powerful facilities for generating abstract syntax trees from graphical input, for generating related information from such trees via attribute grammar mechanisms, and for object base querying and retrieval.

6 Goal-Level Analysis of Specifications Inferred from Scenarios

The purpose of this section is to show what the inferred formal specifications can be used for, by presenting various types of formal analysis at the goal level that could not be performed at the scenario level. The benefits of inferring temporal logic specifications of goals from operational scenarios are demonstrated by examples of formal conflict analysis, obstacle analysis, the inference of more abstract goals, and the derivation of alternative scenarios that better achieve the underlying goals.

6.1 Eliciting New Goals by Goal Abstraction Patterns

In [13], a formal technique is described to elaborate goal refinements in a systematic way for a number of frequent goal assertion patterns. The general principle is to reuse formal refinement patterns defined in a pattern library, and proved correct once for all using the proof theory of temporal logic [48] and available tools such as STeP [49]. The selection of a pattern whose root matches the goal to be refined generates instantiated subgoals that are formally guaranteed to achieve this goal.

Fig. 11 gives an example of such a pattern. Milestone predicates Q1, ..., Qn, Q are introduced; if each of them eventually holds from the previous milestone then the parent goal is guaranteed to be achieved.

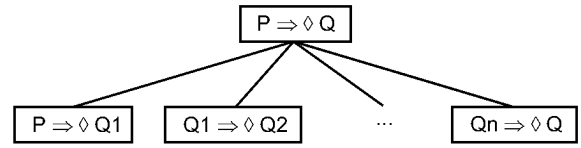

Fig. 11. The milestone-driven refinement/abstraction pattern.

Such a tree can be considered *bottom-up* instead of top-down; we then get an *abstraction pattern*. The selection of a pattern whose leaves match a set of goals to be abstracted generates an instantiated parent goal which is formally guaranteed to be achieved by the more concrete subgoals.

In particular, the milestone-driven abstraction pattern is especially suitable for generating more coarse-grained goal specifications from fine-grained ones inferred from successive events generated along an agent line.

We illustrate this on the following goal specifications that were inferred in Section 5:

2-3. $\forall$ p: Passenger, f: Floor
$\#LiftCalled(p,f) \wedge (\forall$ l: Lift$)(\neg LiftAt(l,f) \wedge \neg Moving(l,f))$
$\Rightarrow \Diamond [(\exists$ l: Lift$) Moving(l,f) \wedge \#LiftCalled(p,f)]$

3-4. $\forall$ p: Passenger, f: Floor, l: Lift
$\#LiftCalled(p,f) \wedge Moving(l,f)$
$\Rightarrow \Diamond [LiftAt(l,f) \wedge l.Doors = \text{'closed'} \wedge \#LiftCalled(p,f)]$

4-5. $\forall$ p: Passenger, f: Floor, l: Lift
$\#LiftCalled(p,f) \wedge LiftAt(l,f) \wedge l.Doors = \text{'closed'}$
$\Rightarrow \Diamond [LiftAt(l,f) \wedge l.Doors = \text{'open'} \wedge \#LiftCalled(p,f)]$

These three fine-grained specifications match the leaves of a more general version of the milestone-driven pattern given in Fig. 12 (the pattern in Fig. 11 is obtained by taking R = **true** in Fig. 12). The leaf nodes of this pattern are instantiated as follows:

P: $(\forall$ l: Lift$)(\neg LiftAt(l,f) \wedge \neg Moving(l,f))$
R: $\#LiftCalled(p,f)$
Q1: $(\exists$ l: Lift$) Moving(l,f)$
Q2: $LiftAt(l,f) \wedge l.Doors = \text{'closed'}$
Q: $LiftAt(l,f) \wedge l.Doors = \text{'open'}$

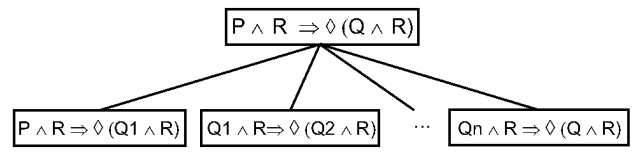

Fig. 12. A general milestone-based abstraction pattern.

We may, therefore, apply this abstraction pattern and generate the following more coarse-grained goal:

$\forall$ p: Passenger, f: Floor
$\#LiftCalled(p,f) \wedge (\forall$ l: Lift$)(\neg LiftAt(l,f) \wedge \neg Moving(l,f))$
$\Rightarrow \Diamond [(\exists$ l: Lift$)(LiftAt(l,f) \wedge l.Doors = \text{'open'} \wedge \#LiftCalled(p,f))]$

that is,

"if a lift call is issued with no lift at the passenger's floor or moving towards it, there shall be a lift at the floor where the call was issued, with its door open."

The same abstraction pattern applied to the fine-grained goals 7-8 to 10-11 inferred in Section 5 yields the following more coarse-grained goal:

$$\forall p: \text{Passenger}, f, f': \text{Floor}, l: \text{Lift}$$
$$\text{LiftAt}(l, f) \wedge l.\text{Doors} = \text{'open'} \wedge \#\text{FloorRequested}(p, l, f')$$
$$\Rightarrow \Diamond [\text{LiftAt}(l, f') \wedge l.\text{Doors} = \text{'open'} \wedge \#\text{FloorRequested}(p, l, f')]$$

Formal abstraction/refinement patterns may also be used *top-down* to find out missing companion goals, requirements, assumptions and/or scenarios.

To illustrate this on the ATM example again, consider the requirement

$$\forall c: \text{Card}, atm: \text{ATM}, a: \text{Amount}$$
$$\#\text{TransactTypeEntered}(c, atm, \text{'getCash'})$$
$$\wedge \#\text{AmountEntered}(c, atm, a) \wedge a \leq c.\text{Limit}$$
$$\wedge \neg \text{CashDelivered}(atm, c)$$
$$\Rightarrow \Diamond \text{CashDelivered}(atm, c),$$

inferred from the scenario in Fig. 2. Asking a WHY question about this requirement may lead to the higher-level goal *Achieve*[CashDelivered]:

$$\forall c: \text{Card}, atm: \text{ATM}$$
$$\#\text{TransactTypeEntered}(c, atm, \text{'getCash'})$$
$$\wedge \neg \text{CashDelivered}(atm, c)$$
$$\Rightarrow \Diamond \text{CashDelivered}(atm, c)$$

The parent and child goals match the "strengthened antecedent" pattern shown in Fig. 13, with instantiations

$$P: \#\text{TransactTypeEntered}(c, atm, \text{'getCash'}) \wedge \neg \text{CashDelivered}(atm, c)$$
$$Q: \text{CashDelivered}(atm, c)$$
$$R: (\exists a: \text{Amount})\ \#\text{AmountEntered}(c, atm, a) \wedge a \leq c.\text{Limit}$$

The following assertions are thereby generated as *missing* companion subgoals:

$$\forall c: \text{Card}, atm: \text{ATM}$$
$$\#\text{TransactTypeEntered}(c, atm, \text{'getCash'})$$
$$\wedge \neg \text{CashDelivered}(atm, c)$$
$$\Rightarrow \Diamond (\exists a: \text{Amount})\ \#\text{AmountEntered}(c, atm, a) \wedge a \leq c.\text{Limit}$$

and

$$\forall c: \text{Card}, atm: \text{ATM}$$
$$\#\text{TransactTypeEntered}(c, atm, \text{'getCash'})$$
$$\wedge \neg \text{CashDelivered}(atm, c)$$
$$\Rightarrow [\#\text{TransactTypeEntered}(c, atm, \text{'getCash'}) \wedge \neg \text{CashDelivered}(atm, c)]\ W\ \text{CashDelivered}(atm, c)$$

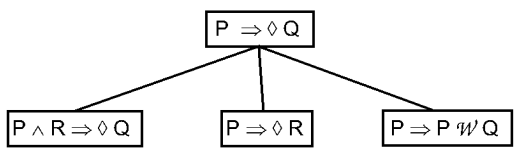

Fig. 13. The "strengthened antecedent" refinement pattern.

The former captures a new goal, with a corresponding scenario, to allow users to re-enter lower amounts than previously asked in order to meet the card's limitation; the latter makes explicit the assumption that users choosing the 'getCash' option keep interested in getting cash up to when cash is delivered.

The inference of explicit, declarative goal specifications from operational scenarios thus allows new goals, requirements and assumptions to be elicited through formal techniques.

6.2 Obstacle Analysis

In [44], formal techniques are presented to systematically generate obstacles that obstruct given goals. Obstacle assertions are formally derived from a goal assertion by

1) regressing the negated goal assertion through the domain theory available in order to find out preconditions for goal obstruction, or
2) applying formal obstruction patterns associated with specific temporal goal patterns in order to build formal fault trees.

Once obstacle assertions are obtained, scenarios are being sought as models of these assertions to check whether the obstacles are satisfiable. The result of this process is a systematic enumeration of obstacles and exceptional scenarios whose resolution leads to more realistic and more complete requirements (see [44] for details).

When goal specifications are inferred from scenarios using our procedure, these obstacle/scenario generation techniques may be used to find out missing requirements and scenarios.

To illustrate this on our ongoing ATM example, consider the goal

$$\forall c: \text{Card}, atm: \text{ATM}$$
$$\#\text{CardInserted}(c, atm) \wedge \neg \text{PassWdRequested}(atm, c)$$
$$\Rightarrow \Diamond [\text{PassWdRequested}(atm, c) \wedge \#\text{CardInserted}(c, atm)],$$

inferred in Section 4 from the scenario in Fig. 2. Looking in the domain theory (or asking domain experts) for necessary conditions for passwords to be requested by an ATM, one may get, e.g.,

$$\text{PassWdRequested}(atm, c) \Rightarrow \text{CardReadable}(atm, c),$$

that is,

$$\neg \text{CardReadable}(atm, c) \Rightarrow \neg \text{PassWdRequested}(atm, c)$$

We regress the negated goal above, that is,

$$\Diamond \exists c: \text{Card}, atm: \text{ATM}$$
$$\#\text{CardInserted}(c, atm)$$
$$\wedge \Box (\#\text{CardInserted}(c, atm) \rightarrow \neg \text{PassWdRequested}(atm, c))$$

through the above domain rule and thereby formally derive the following obstacle to this goal:

$$\Diamond \exists c: \text{Card}, atm: \text{ATM}$$
$$\#\text{CardInserted}(c, atm)$$
$$\wedge \Box (\#\text{CardInserted}(c, atm) \rightarrow \neg \text{CardReadable}(atm, c))$$

The obvious satisfying scenario is a card being inserted and remaining unreadable when inserted. New goals and scenarios have to be elicited therefrom in order to resolve this obstacle (see [44] for resolution techniques).

Other possibly missing goals and scenarios about, e.g., expired cards, cash unavailable, or no paper left anymore for transaction records, would have been derived formally in exactly the same way.

The inference of explicit, declarative goal specifications from operational scenarios thus may ensure more coverage and more robustness through formal obstacle analysis.

6.3 Conflict Analysis

In [45], formal techniques are presented to systematically detect/resolve divergences among goal formulations. A divergence corresponds to the possible occurrence of a boundary condition that makes the set of goal assertions conflicting (that is, logically inconsistent).

Boundary conditions are formally derived by regressing the negation of one of the goal assertions through the domain theory extended with the other goal assertions.

When goal assertions are inferred from scenarios using the procedure described in Section 4, boundary conditions may be sought using these techniques to detect possible divergences among the goals inferred and thus among the scenarios they cover.

To illustrate this, consider the following goal specifications inferred in Section 5:

$$\forall \text{ l: Lift, p: Passenger,, f, f': Floor}$$
$$\#\text{FloorRequested}(p,l,f') \wedge \text{LiftAt}(l,f) \wedge \text{l.Doors} = \text{'open'}$$
$$\Rightarrow \Diamond\, [\text{LiftAt}(l,f) \wedge \text{l.Doors} = \text{'closed'} \wedge \#\text{FloorRequested}(p,l,f')]$$
$$\#\text{FloorRequested}(p,l,f') \wedge \text{LiftAt}(l,f) \wedge \text{l.Doors} = \text{'closed'}$$
$$\Rightarrow \Diamond\, [\text{l.Doors} = \text{'closed'} \wedge \text{LiftAt}(l,f') \wedge \#\text{FloorRequested}(p,l,f')]$$
$$\#\text{FloorRequested}(p,l,f') \wedge \text{LiftAt}(l,f') \wedge \text{l.Doors} = \text{'closed'}$$
$$\Rightarrow \Diamond\, [\text{LiftAt}(l,f') \wedge \text{l.Doors} = \text{'open'} \wedge \#\text{FloorRequested}(p,l,f')]$$

On another hand, consider the negative scenario about the lift being overloaded and the following goal specification that was inferred from it:

$$\text{LiftAt}(l,f) \wedge \text{l.Doors} = \text{'open'} \wedge \#\text{FloorRequested}(p,l,f')$$
$$\wedge\ \text{Overloaded}(l)$$
$$\Rightarrow [(\text{LiftAt}(l,f) \wedge \text{l.Doors} \neq \text{'closed'})\ W\ \neg\, \text{In}(p,l)]$$

The following boundary condition can be formally derived to establish the divergence among those four goals:

$$\exists \text{ l: Lift, p: Passenger, f,' f: Floor}$$
$$\Box\, [\text{LiftAt}(l,f) \wedge \text{l.Doors} = \text{'open'} \wedge \#\text{FloorRequested}(p,l,f')$$
$$\Rightarrow \text{Overloaded}(l)]$$

This boundary condition is satisfied by the "starvation" scenario of a passenger never reaching her destination floor because of the lift getting overloaded at every attempt to enter the lift. Once such a divergence is detected formally, appropriate resolution strategies can be applied, see [45].

The two following goals about avoiding unnecessary lift moves were derived in Section 5 using the dataflow heuristics:

$$\forall \text{ l: Lift, f: Floor}$$
$$\neg\, (\exists\, p\text{: Passenger})\, \#\text{LiftCalled}(p,f) \wedge \neg\, \text{LiftAt}(l,f)$$
$$\Rightarrow [\neg\, \text{LiftAt}(l,f)\ W\ (\exists\, p\text{: Passenger})\#\text{LiftCalled}(p,f)]$$

and

$$\forall \text{ l: Lift, f: Floor}$$
$$\neg\, (\exists\, p\text{: Passenger})\#\text{FloorRequested}(p,l,f) \wedge \neg\, \text{LiftAt}(l,f)$$
$$\Rightarrow [\neg\, \text{LiftAt}(l,f)\ W\ (\exists\, p\text{: Passenger})\#\text{FloorRequested}(p,l,f)]$$

Using the conflict detection techniques in [45] we find that these two *Avoid* goals are pairwise conflicting with the two coarse-grained *Achieve* goals inferred in Section 6.1; one of the boundary conditions derived to make these goals pairwise logically inconsistent is:

$$\exists \text{ l: Lift, f: Floor, p': Passenger}$$
$$\neg\, (\exists\, p\text{: Passenger})\#\text{LiftCalled}(p,f)$$
$$\wedge\ \neg\, \text{LiftAt}(l,f) \wedge \#\text{FloorRequested}(p',l,f)$$

This boundary condition captures a situation of a passenger wishing to go to floor f whereas the lift may not do so because it is not called from that floor. We can then apply the weakening rule for conflict resolution [45] which yields the following conflict-free version for the two *Avoid* goals above:

$$\forall \text{ l: Lift, f: Floor}$$
$$\neg\, \text{LiftAt}(l,f) \wedge \neg\, (\exists\, p\text{: Passenger})\#\text{LiftCalled}(p,f)$$
$$\wedge\ \neg\, (\exists\, p\text{: Passenger})\#\text{FloorRequested}(p,l,f)$$
$$\Rightarrow [\neg\, \text{LiftAt}(l,f)\ W\ \exists\, p\text{: Passenger}$$
$$(\#\text{LiftCalled}(p,f) \vee \#\text{FloorRequested}(p,l,f))]$$

The inference of explicit, declarative goal specifications from operational scenarios thus allows formal conflict analysis to take place.

6.4 Goal-Based Identification of Better Scenarios

Scenarios may sometimes contain a great deal of overspecification with respect to their underlying goals left implicit. Sometimes they even prescribe some sequencing of events that is not the best one for achieving those implicit goals. In such cases, the inference of explicit, declarative goal formulations for subsequent reasoning may result in finding out better scenarios to achieve them.

To illustrate this, let us come back again to the goal Achieve[CardProperlyHandled] whose declarative formulation was inferred in Section 4.6:

$$\forall c\text{: Card, atm: ATM}$$
$$\#\text{CardInserted (c, atm)}$$
$$\Rightarrow\ \#\text{OKAccess}(atm,c) \rightarrow \Diamond\, \text{CardReturned}(atm,c)$$
$$\wedge\ \#\textit{KO-PassWd}(atm, c, 3) \rightarrow \neg \text{CardReturned}(atm,c)\ W\ \text{F3})]$$

A WHY question about this goal leads to the higher-level goals *Achieve[RepeatedUse]* and *Avoid[IllegalUse]*. IllegalUse may result from situations where the card has been forgotten at the ATM. It then turns out that there is a better scenario to enforce both goals than the scenario in Fig. 2 (taken from [68]). A piece of domain knowledge, based on empirical evidence, tells us that people accessing an ATM in order to get cash tend to concentrate on their main goal of getting cash—to the point that they sometimes forget about the rest of the transaction once their primary goal is achieved, that is, getting their receipt and taking their card back. A better way to achieve the goals Achieve[RepeatedUse] and Avoid[IllegalUse] is, therefore, to require the ATM to deliver cash only when the card has been withdrawn, that is, to make the interaction events CardReturn and CardWithdrawal precede the interaction events CashDelivery and CashWithdrawal in Fig. 2. (This scenario is in fact implemented in many ATM systems.)

The strengthened precondition heuristics discussed in Section 4.5 may also help detecting unnecessary sequencing in scenarios provided. If the postcondition of the operation associated with some event is not required to strengthen the domain precondition of the operation associated with the

event just following, then the sequencing is not required; there is no logical producer-consumer dependency between the postcondition of the former and the precondition of the latter. This heuristics may be used to detect the unnecessary sequencing of the FloorRequest and DoorsClosing events in the Lift scenario given in Fig. 8.

7 Related Work

The informal use of scenarios for requirements elicitation, validation and documentation is widely recognized among practitioners [69], [72] and is promoted by many methodologies [68], [67], [34], [58]. Proposals for representing scenarios have flourished [63]; they generally consist in extensions or adaptations of existing notations for capturing dynamic behaviors of specific agents. For type-level scenarios, these include, e.g., regular expressions [11], statecharts [25], decision trees [32], operational scripts [67] enriched with contextual information [59], or narrative use cases [34] possibly annotated with interaction traces that contain state information [61]. Extensions to support loops, modularization and hierarchical decomposition have also been suggested [61]. For instance-level scenarios, notations based on message sequence charts [33] to capture interaction traces have been the most popular to date [68], [23]. There have also been proposals for explicitly linking scenarios to the goals underlying them [22], [11], [59], [29], [65]. Our choice of a specific notation in this paper was dictated by the simplicity, popularity, and closeness of the notation to the informal narratives provided by stakeholders. (The type-level scenario notation supported by KAOS [11] was felt too rich and complex in this context.) Our experience also suggests that the goals underpinning scenarios are often left implicit in practice (see Section 3.1).

Most research work on using scenarios in requirements engineering has been on the validation side. Prototyping [31], [70], animation [18], simulation [30] and symbolic execution [17] tools generate execution traces which can be seen as scenarios to be submitted to stakeholders for adequacy checking. The ARIES Simulation Component [3], [4] allows a specification to be executed, even if ambiguities, inconsistencies and incompletenesses are present, by supporting various levels of abstraction. Scenarios there formalize validation questions that are asked to check whether some desirable or undesirable system behavior is covered. CRITTER [22] is a scenario-based requirements debugging system. Goals are formalized in some restricted temporal logic, and type-level scenarios are expressed in a Petri net-like language. The general approach consists in:

1) detecting inconsistencies between scenarios and goals, and
2) applying operators that modify the specification to remove the inconsistencies.

Step 1) is carried out by a planner that searches for scenarios leading to some goal violation. The operators offered to the analyst in step 2) encode heuristics for specification debugging—e.g., introduce an agent whose responsibility is to avoid the last state transition before the goal is broken. Nitpick [37] is another example of scenario-based debugging of specifications. Given a specification and a claim formulated in a Z-like relational language, the system tries to generate a counterexample to that claim which can be seen as a scenario that may help correcting the specification. In the same spirit, model checking tools can be viewed as scenario-based validation tools [10], [51]; given a declarative specification in some temporal logic and an operational specification of a finite state machine, these tools try to show that the temporal logic specification is satisfied by the finite state machine through enumeration of all execution traces; in case of failure they exhibit a counter-example scenario that violates the specification.

The work on scenario-based elicitation has not yet reached such a level of technical sophistication and maturity. At the very extreme of informality, guidelines are provided for building finite state machine models from event trace diagrams [68]. Process models that integrate scenarios as important artifacts for requirements elicitation have been suggested [58]; the role of scenarios to identify obstacles obstructing goals has been demonstrated [59]. [71] suggests using scenarios as "keys" for retrieving generic requirements satisfied by them and reusable for the problem at hand. In [65], a semiformal approach is proposed for eliciting goals from scenarios. Goals are represented there by parameterized action verbs; alternative goals correspond to alternative parameter values. Assuming that a bidirectional coupling between a type-level scenario and such a goal can be made explicit, heuristic rules are suggested for finding out alternative goals to establish the scenario, missing companion goals, or subgoals of the goal considered.

The WATSON and ISAT systems are examples of work closely related to ours. WATSON is an automated elicitation environment that tries to induce a plausible formal specification from scenarios expressed in natural language [39]. The system relies on domain knowledge to correct routine ommissions and errors in scenarios, constrain the space of possible scenario generalizations, and plan queries to the user about variant scenarios. ISAT is a scenario-based elicitation/validation assistant that helps requirements engineers acquire and maintain a specification consistent with scenarios provided [27]. The system uses explanation-based learning techniques to generalize the scenarios in order to state and prove validation lemmas. Recent developments have been focussed on the validation side. [28] describes a formal method for generating scenarios that establish some target goal specification. The principle is to retrieve from a knowledge base fragmentary scenarios that establish some of the conjuncts of the goal predicate, generalize them, apply some constrained coinstantiation to make them refer to common objects, and compute a merge of the resulting scenarios that establishes the whole set of conjuncts. A scenario integration step is thus involved there. Another technique for scenario integration is proposed in [16]. Scenarios are represented there by mathematical relations; scenario integration involves the application of complex relational "meet" operators. The result is a relational description of a big scenario—unlike the declarative goals, requirements and assumptions obtained here.

The formal method presented in this paper is closely related to inductive learning from examples. Admissible de-

scriptions are obtained there by bidirectional search through a space of descriptions. A typical algorithm iterates over the instances provided. If the current instance is positive, the candidate concept descriptions obtained so far are transformed minimally by use of *generalization rules* so as to cover this new positive instance without covering the negative instances already considered; if the current instance is negative, the candidate concept descriptions obtained so far are transformed minimally by use of *specialization rules* so as to exclude this new negative instance without excluding the positive instances already considered. Various inductive learning algorithms have been proposed which vary according to the search strategy followed, the specific generalization/specialization rules applied, the characteristics of the set of admissible descriptions found, and the use or not of domain knowledge. Mitchell's candidate elimination algorithm is probably the best known among them [53]. The interested reader may refer to [41] for a comparative analysis of these algorithms.

A notable difference between inductive learning algorithms and our goal inference procedure is that the learning of a new concept from examples requires starting from the bottom of the lattice of concept descriptions and climbing up by generalizations to cover positive examples. In our case, we start from the top of the lattice of all possible behaviors (where everything is permited), and go down by removing behaviors each time a new scenario has to be covered or excluded.

Learning techniques have already been used in a variety of software engineering applications, including the synthesis of list processing programs from execution traces [6], the inference of process models from process traces [24], and the generation of test cases [5].

8 Conclusions

While scenarios are a proven effective means for eliciting requirements, they cannot replace such requirements because they are most often partial, instance-level, procedural, and often leave the underlying goals, requirements and assumptions implicit. The approach advocated in this paper is to use the narrative, concrete, and informal style of description provided by scenarios to elicit more abstract, type-level, and declarative specifications; such specifications cover more behaviors and assert goal-based properties explicitly and formally. Thanks to the inference of goal specifications from scenarios, formal analysis can be applied to the specifications to generate further specifications by goal abstraction and refinement; to generate obstructing obstacles that prompt for new goals to handle exceptional situations; to detect conflicts among goals/requirements and resolve them; to find out new scenarios that operationalize their intended purpose in a more effective way.

The inference method presented in the paper allows declarative specifications of goals, requirements and assumptions to be generated from scenarios in a systematic, formal way. The various kinds of formal analysis at the goal level illustrate the benefits of inferring high-level, temporal logic specifications rather than more operational ones such as finite state machines or state-based specifications, on which such analysis could not have been applied.

Event trace diagrams were chosen as a simple, semiformal notation for representing scenarios because they are widely used in practice and because they provide an intuitive visualization of the logical models the inferred temporal logic formulas need to satisfy. Our method exploits the temporal goal patterns provided by KAOS to inductively infer candidate goal assertions satisfied by the event trace diagrams provided. These assertions are progressively integrated as new scenarios are provided so that in the end they cover all positive scenarios and exclude all negative ones. The generated assertions need to be validated, and may be abstracted/refined formally using the techniques discussed in Section 6.

Long, complex scenarios in which unrelated interaction events are involved will correspondingly make the inference process more complex. A scenario filtering mechanism was therefore proposed, based on views associated with KAOS goal categories, to simplify the scenarios and make the inference process more focussed. Other clean up mechanisms beside those presented here need to be investigated in order to further simplify scenarios when necessary. Among them, scenario chunking to represent macro-interactions seems a promising approach which proved effective in AI hierarchical planning.

Another direction we want to investigate in the future is the inference, in parallel, of a finite state machine whose execution paths cover the positive scenarios provided; our validation step would then be supported by model checking tools applied to the temporal logic specifications and the finite state machine inferred in parallel.

The method presented in this paper is grounded on several years of experience with scenario-based goal elicitation in industrial projects. We hope to have convinced the reader through the variety of examples given in the paper, some of them being fairly complex, that the method is systematic and effective in nontrivial elicitation situations. Our goal now is to experiment them on the largest, complex scenarios found among those we have been working on at the CEDITI tech transfer institute. In parallel, our formal method needs to be complemented with a tool to be integrated in the existing KAOS/GRAIL environment for goal-based requirements engineering [15].

Acknowledgments

The work reported herein was partially supported by the "Communauté Française de Belgique" (FRISCO project, Actions de Recherche Concertées No. 95/00-187—Direction générale de la Recherche). Partial support was also provided while the first author was on sabbatical at SRI International by the Advanced Research Projects Agency under Air Force Research Laboratory Contract No. F30602-97-C-0040.

We are deeply indebted to Robert Darimont, Emmanuelle Delor, Philippe Jamart, Philippe Massonet, and Christophe Ponsard for sharing their experience and insights in the use of scenarios in industrial projects at CEDITI. Thanks go also to Jose Fiadeiro for clarifying discussions on scenario integration in a temporal logic framework. The comments and feedback provided by the reviewers were greatly appreciated.

REFERENCES

[1] J.S. Anderson and S. Fickas, "A Proposed Perspective Shift: Viewing Specification Design as a Planning Problem," *Proc. IWSSD-5, Fifth Int'l Workshop Software Specification and Design*, pp. 177–184, IEEE, 1989.

[2] A.I. Anton, W.M. McCracken, and C. Potts, "Goal Decomposition and Scenario Analysis in Business Process Reengineering," *Proc. CAISE'94, Sixth Conf. Advanced Information Systems Eng.*, pp. 94–104, Lecture Notes in Computer Science 811, Springer-Verlag, 1994.

[3] K.M. Benner, "The ARIES Simulation Component," *Proc. KBSE'93*.

[4] K.M. Benner, M.S. Feather, W.L. Johnson, and L.A. Zorman, "Utilizing Scenarios in the Software Development Process," *Information System Development Process*, Elsevier Science, B.V. North-Holland, pp. 117–134, 1993.

[5] F. Bergadano and D. Gunetti, "Testing by Means of Inductive Program Learning," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 2, Apr. 1996.

[6] A.W. Biermann and R. Krishnaswamy, "Constructing Programs from Example Computations," *IEEE Trans. Software Eng.*, vol. 2, no. 9, pp. 141-153, 1976.

[7] A. Borgida, J. Mylopoulos, and R. Reiter, "On the Frame Problem in Procedure Specifications," *IEEE Trans. Software Eng.*, vol. 21, no. 10, pp. 785–798, Oct. 1995.

[8] R.J. Brachman and H.J. Levesque, eds., *Readings in Knowledge Representation*. Morgan Kaufmann, 1985.

[9] J.M. Carroll and M.B. Rosson, "Narrowing the Specification Implementation Gap in Scenario-Based Design," *Scenario-Based Design: Envisioning Work and Technology in System Development*, J.M. Carroll, ed., pp. 247–278, John Wiley & Sons, 1995.

[10] E.M. Clarke and E.A. Emerson, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. Program. Language Systems* ,vol. 8, no. 2, pp. 244–263, 1986.

[11] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-Directed Requirements Acquisition," *Science of Computer Programming*, vol. 20, pp. 3–50, 1993.

[12] A. Dardenne, "On the Use of Scenarios in Requirements Acquisition," Technical Report CIS-TR-93-17, Dept. of Computer and Information Science, Univ. of Oregon, Aug. 1993.

[13] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration," *Proc. FSE'4—Fourth ACM SIGSOFT Symp. Foundations of Software Eng.*, pp. 179–190, San Francisco, Oct. 1996.

[14] R. Darimont and E. Delor, "Goal-Driven Requirements Specification of a Copyright Tracking System," CEDITI Deliverable, Sept. 1996. In French

[15] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde, "GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering," *Proc. ICSE'97—19th Int'l Conf. Software Eng.*, Boston, Apr. 1997.

[16] J. Desharnais, M. Frappier, R. Khedri, and A. Mili, "Integration of Sequential Scenarios," *Proc. ESEC'97—Sixth European Software Eng. Conf.*, pp. 310–326, Zurich, Lecture Notes in Computer Science 1301, Springer-Verlag, Sept. 1997.

[17] J. Douglas and R.A. Kemmerer, "Aslantest: A Symbolic Execution Tool for Testing ASLAN Formal Specifications," *Proc. ISTSTA '94—Int'l Symp. Software Testing and Analysis*, ACM Software Engineering Notes, pp. 15–27, 1994.

[18] E. Dubois, Ph. Du Bois, and M. Petit, "Object-Oriented Requirements Analysis: An Agent Perspective," *Proc. ECOOP'93—Seventh European Conf. Object-Oriented Programming*, pp. 458-481, Lecture Notes in Computer Science 707, Springer-Verlag, 1993.

[19] M. Feather, "Language Support for the Specification and Development of Composite Systems," *ACM Trans. Programming Languages and Systems*, vol. 9, no. 2, pp. 198-234, Apr. 1987.

[20] M. Feather, "Towards a Derivational Style of Distributed System Design," *Automated Software Eng.*, vol. 1, no. 1, pp. 31–60, 19**94**.

[21] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behaviour," *Proc. IWSSD'98—Ninth Int'l Workshop Software Specification and Design*, Isobe, IEEE CS Press, Apr. 1998.

[22] S. Fickas and R. Helm, "Knowledge Representation and Reasoning in the Design of Composite Systems," *IEEE Trans. Software Eng.*, pp. 470–482, June 1992.

[23] M. Fowler, *UML Distilled—Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[24] P.K. Garg and S. Bhansali, 'Process Programming by Hindsight," *Proc. ICSE14—14th Int'l Conf. Software Eng.*, Melbourne, pp. 280–293, 1992.

[25] M. Glinz, "An Integrated Formal Model of Scenarios Based on Statecharts," *Proc. ESEC'95—Fourth European Software Eng. Conf.*, Lecture Notes in Computer Science 989, Springer-Verlag, 1995.

[26] D. Gries, *The Science of Programming*. Springer-Verlag, 1981.

[27] R.J. Hall, "Systematic Incremental Validation of Reactive Systems via Sound Scenario Generalization," *Automated Software Eng.*, vol. 2, pp. 131–166, 1995.

[28] R.J. Hall, "Explanation-Based Scenario Generation for Reactive System Models," *Proc. ASE'98, Hawaii*, Oct. 1998.

[29] P. Haumer, K. Pohl, and K. Weidenhaupt, "Requirements Elicitation and Validation with Real World Scenes," CREWS Report 98–16, 1998.

[30] C. Heitmeyer, R. Jeffords, and B. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Trans. Software Eng. and Methodology* vol. 5, no. 3, pp. 231–261, July 1996.

[31] S. Hekmatpour and D. Ince, *Software Prototyping, Formal Methods, and VDM*. Addison-Wesley, 1988.

[32] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen, "Formal Approach to Scenario Analysis," *IEEE Software*, pp. 33–41, Mar. 1994.

[33] ITU-T, Message Sequence Charts. Recommendation Z.120, 1993.

[34] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, ACM Press, 1993.

[35] M. Jackson and P. Zave, "Domain Descriptions," *Proc. RE'93—First Int'l IEEE Symp. Requirements Eng.*, pp. 56–64, Jan. 1993.

[36] M. Jackson, *Software Requirements & Specifications—A Lexicon of Practice, Principles and Prejudices*. ACM Press, Addison-Wesley, 1995.

[37] D. Jackson, "Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector," *Proc. ACM ISSTA'96*, pp. 239–249, San Diego, 1996.

[38] S.E. Keller, L.G. Kahn, and R.B. Panara, "Specifying Software Quality Requirements with Metrics," *Tutorial: System and Software Requirements Engineering*, R.H. Thayer and M. Dorfman, eds., pp. 145–163, IEEE CS Press, 1990.

[39] V. Kelly and U. Nonnenmann, "Reducing the Complexity of Formal Specification Acquisition," *Automated Software Design*, M. Lowry and R. McCartne, eds., pp. 41–64, AAAI Press, 1991.

[40] R. Koymans, *Specifying Message Passing and Time-Critical Systems with Temporal Logic*, Lecture Notes in Computer Science 651, Springer-Verlag, 1992.

[41] A. van Lamsweerde, "Learning Machine Learning," *Introducing a Logic Based Approach to Artificial Intelligence*, A. Thayse, ed., vol. 3, pp. 263–356, John Wiley & Sons, 1991.

[42] A. van Lamsweerde, R. Darimont, and P. Massonet, "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learned," *Proc. RE'95—Second Int'l Symp. on Requirements Eng.*, York, IEEE, 1995.

[43] A. van Lamsweerde, "Divergent Views in Goal-Driven Requirements Engineering," *Proc. Viewpoints'96—ACM SIGSOFT Workshop Viewpoints in Software Development*, Oct. 1996.

[44] A. van Lamsweerde and E. Letier, "Integrating Obstacles in Goal-Driven Requirements Engineering," *Proc. ICSE-98: 20th Int'l Conf. Software Eng.*, Kyoto, Apr. 1998.

[45] A. van Lamsweerde, R. Darimont, and E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering," *IEEE Trans. Software. Eng.*, special issue on *Inconsistency Management in Software Development*, Nov. 1998.

[46] J.C. Leite and P.A. Freeman, "Requirements Validation Through Viewpoint Resolution," *IEEE Trans. Software Eng.*, pp. 1,253–1,269, Dec. 1991.

[47] J.C. Leite, G. Rossi, F. Balaguer, V. Maiorana, G. Kaplan, G. Hadad, and A. Oliveros, "Enhancing a Requirements Baseline with Scenarios," *Proc. RE'97—Third Int'l Symp. Requirements Eng.*, pp. 44–53, Anapolis, IEEE, 1997.

[48] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[49] Z. Manna and the STep Group, "STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems," *Proc. CAV'96—Eighth Int'l Conf. Computer-Aided Verification*, pp. 415–418, Lecture Notes in Computer Science 1102. Springer-Verlag, July 1996.

[50] D. Marca and M. Harandi, "Problem Set for the Fourth Int'l Workshop on Software Specification and Design," *Proc. Fourth*

Int'l Workshop Software Specification and Design, IEEE CS Press, 1987.
[51] K.L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer, 1993.
[52] B. Meyer, "On Formalism in Specifications," *IEEE Software*, vol. 2, no. 1, pp. 6–26, Jan. 1985.
[53] T. Mitchell, "Generalization as Search," *Artificial Intelligence*, vol. 18, pp. 203–226, 1982.
[54] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach," *IEEE Trans. Sofware. Eng.*, vol. 18, no. 6, pp. 483-497, June 1992.
[55] N.J. Nilsson, *Principles of Artificial Intelligence*. Springer-Verlag, 1982.
[56] B. Nuseibeh, J. Kramer, and A. Finkelstein, "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications," *IEEE Trans. Software Eng.*, vol. 2, no. 10, pp. 760–773, Oct. 1994.
[57] D.L. Parnas and J. Madey, "Functional Documents for Computer Systems," *Science of Computer Programming*, vol. 25, pp. 41–61, 1995.
[58] C. Potts, K. Takahashi, and A.I. Anton, "Inquiry-Based Requirements Analysis," *IEEE Software*, pp. 21–32, Mar. 1994.
[59] C. Potts, "Using Schematic Scenarios to Understand User Needs," *Proc. DIS'95—ACM Symp. Designing Interactive Systems: Processes, Practices and Techniques*, Univ. of Michigan, Aug. 1995.
[60] B. Potter, J. Sinclair, and D. Till, *An Introduction to Formal Specification and Z*, second edition. Prentice Hall, 1996.
[61] B. Regnell, K. Kimbler, and A. Wesslen, "Improving the Use Case Driven Approach to Requirements Engineering," *Proc. RE'95—Second Int'l Symp. Requirements Eng.*, pp. 40–47, York, IEEE, 1995.
[62] W.N. Robinson, "Integrating Multiple Specifications Using Domain Goals," *Proc. IWSSD-5—Fifth Int'l Workshop Software Specification and Design*, pp. 219–225, IEEE, 1989.
[63] C. Rolland, C. Ben Achour, C. Cauvet, J. Ralyte, A. Sutcliffe, N. Maiden, M. Jarke, P. Haumer, K. Pohl, E. Dubois, and P. Heymas, "A Proposal for Scenario Classification Framework," *Requirements Eng. J.*, vol. 3, no. 1, 1998.
[64] C. Rolland and C. Ben Achour, "Guiding the Construction of Textual Use Case Specifications," *Data and Knowledge Eng. J.*, vol. 25, nos. 1-2, pp. 125–160, Mar. 1998.
[65] C. Rolland, C. Souveyet, and C. Ben Achour, "Guiding Goal Modelling Using Scenarios," CREWS Report Series no. 98–27.
[66] D. Rosca, S. Greenspan, M. Feblovitz, and C. Wild, "A Decision Making Methodology in Support of the Business Rules Lifecycle," *Proc. RE'97—Third Int'l Symp. Requirements Eng.*, pp. 236–246, Anapolis, IEEE, 1997.
[67] K.S. Rubin and J. Goldberg, "Object Behaviour Analysis," *Comm. ACM*, vol. 35, no. 9, pp. 48–62, Sept. 1992.
[68] J. Rumbaugh, M. Blaha, W. Prmerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modelling and Design*. Prentice Hall, 1991.
[69] I. Sommerville and P. Sawyer, *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, 1997.
[70] A. Sutcliffe, "A Technique Combination Approach to Requirements Engineering," *Proc. RE'97—Third Intl. Symp. Requirements Eng.*, pp. 65–74, Anapolis, IEEE, 1997.
[71] A. Sutcliffe, N. Maiden, S. Minocha, and D. Manuel, "Supporting Scenario-Based Requirements Engineering," CREWS Report 98–08, 1998.
[72] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer, "Scenario Usage in System Development: A Report on Current Practice," *IEEE Software*, Mar. 1998.
[73] K. Yue, "What Does It Mean to Say that a Specification is Complete?" *Proc. IWSSD-4, Fourth Int'l Workshop Software Specification and Design*, Monterey, 1987.
[74] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering," *ACM Trans. Software Eng. and Methodology*, pp. 1–30, 1997.

Axel van Lamsweerde is full professor of computing science at the Université Catholique de Louvain, Belgium. He received the MS degree in mathematics from that university, and the PhD degree in computing science from the University of Brussels. From 1970-1980, he was research associate with the Philips Research Laboratory in Brussels, where he worked on proof methods for parallel programs and knowledge-based approaches to automatic programming. He was then professor of software engineering at the Universities of Namur and Brussels until he joined UCL in 1990. He is co-founder of the CEDITI technology transfer institute partially funded by the European Union. He has also been a visitor at the University of Oregon and the Computer Science Laboratory of SRI International, Menlo Park, California. Dr. van Lamsweerde's professional interests are in lightweight formal methods and tools for assisting software engineers in knowledge-intensive tasks. His current focus is on constructive, technical approaches to requirements engineering and, more generally, to formal reasoning about software engineering products and processes. van Lamsweerde was program-chair of the Third European Software Engineering Conference (ESEC'91); program co-chair of the Seventh IEEE Workshop on Software Specification and Design (IWSSD-7); and program co-chair of the ACM-IEEE 16th International Conference on Software Engineering (ICSE-16). He is a member of the Editorial Boards of the *Automated Software Engineering Journal* and the *Requirements Engineering Journal*. Since 1995, he is editor-in-chief of the *ACM Transactions on Software Engineering and Methodology (TOSEM)*. He is a member of the IEEE, ACM, and AAAI. His recent papers can be found at http://www.info.ucl.ac.be/people/avl.html.

Laurent Willemet received the degree of engineer in computer science in 1995 from the Université Catholique de Louvain, Belgium. He is now a PhD candidate in the Département d'Ingénierie Informatique of this university. His research concerns the use of scenarios in the requirements engineering process. He also has a strong interest in formal specification techniques.