Formal Refinement Patterns for Goal-Driven Requirements Elaboration

Robert Darimont and Axel van Lamsweerde
Université catholique de Louvain, Département d'Ingénierie Informatique
B-1348 Louvain-la-Neuve (Belgium)
{rd, avl}@info.ucl.ac.be

Abstract. Requirements engineering is concerned with the identification of high-level goals to be achieved by the system envisioned, the refinement of such goals, the operationalization of goals into services and constraints, and the assignment of responsibilities for the resulting requirements to agents such as humans, devices and programs. Goal refinement and operationalization is a complex process which is not well supported by current requirements engineering technology. Ideally some form of formal support should be provided, but formal methods are difficult and costly to apply at this stage.

This paper presents an approach to goal refinement and operationalization which is aimed at providing constructive formal support while hiding the underlying mathematics. The principle is to reuse generic refinement patterns from a library structured according to strengthening/weakening relationships among patterns. The patterns are once for all proved correct and complete. They can be used for guiding the refinement process or for pointing out missing elements in a refinement. The cost inherent to the use of a formal method is thus reduced significantly. Tactics are proposed to the requirements engineer for grounding pattern selection on semantic criteria.

The approach is discussed in the context of the multi-paradigm language used in the KAOS method; this language has an external semantic net layer for capturing goals, constraints, agents, objects and actions together with their links, and an inner formal assertion layer that includes a real-time temporal logic for the specification of goals and constraints. Some frequent refinement patterns are highlighted and illustrated through a variety of examples.

The general principle is somewhat similar in spirit to the increasingly popular idea of design patterns, although it is grounded on a formal framework here .

Keywords: Goal-driven requirements engineering, refinement, reuse of specifications and proofs, design patterns, formal methods.

1. Introduction

Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families. This general definition, borrowed from [Zav95], stresses the leading part played by goals during requirements elaboration.

There are multiple reasons why goals must be made explicit in the requirements engineering process. Goals drive the identification of requirements to support them; they provide an explanation to clients of the rationale underpinning requirements; they represent the roots for detecting conflicts among requirements and for resolving them eventually [Rob89]; they provide a completeness criterion for the requirements specification --the specification is complete if all stated goals are met by the specification; they generally represent the most stable information in the requirements product. To sum up, requirements "implement" goals much the same way as programs implement design specifications.

Surprisingly, the explicit modelling of goals and the integration of goal specifications in requirements specifications has been considered only rather recently in the literature. Goals are generally modelled by intrinsic features such as their type and attributes, and by their links to other goals and to requirements. Various goal taxonomies have been proposed to *type* goals [Myl92], [Dar93], [Nix93], [Ant94]. For instance, information goals are functional goals concerned with keeping agents informed about object states; accuracy goals are non-functional goals concerned with maintaining the consistency between objects in the environment and their software image. Useful *attributes* for a goal may include its priority [Dar93] and utility [Rob89]. Information about goal types and attributes can be used to define heuristics for goal acquisition, goal refinement, and requirements elaboration. *Links* between goals are aimed at capturing situations where goals positively or negatively support other goals. Directly borrowed from problem reduction methods in Artificial Intelligence [Nil71], AND/OR graphs may be used to capture goal refinements [Dar91], [Dar93]. *AND-reduction* links relate a goal to a set of subgoals (called reduction); this means that satisfying all subgoals in the reduction is a sufficient condition for satisfying the goal. *OR-reduction* links relate a goal to an alter-

Proceedings 4th ACM Symposium on the Foundations of Software Engineering (FSE4), San Francisco, Oct. 1996, 179-190.

native set of reductions; this means that satisfying one of the reductions is a sufficient condition for satisfying the goal. In this framework, a *conflict* link between two goals is introduced when the satisfaction of one of them may preclude the satisfaction of the other. *Operationalization* links can also be introduced in goal models to relate goals to requirements. Weaker versions of such link types have been proposed to relate very high-level goals [Rob89], [Myl92]; the idea is that such goals can rarely be said to be satisfied in a clear-cut sense. Instead of goal satisfaction, goal satisficing is introduced to express that lower-level goals or requirements are expected to achieve the goal within acceptable limits, rather than absolutely. A subgoal is then said to *contribute* partially to the goal, regardless of other subgoals; it may contribute *positively* or *negatively*.

The purpose of goal modelling is to support some form of reasoning about goals during requirements elaboration. Two kinds of reasoning can be distinguished, namely, qualitative reasoning and formal reasoning. *Qualitative reasoning* is most appropriate for very high-level, non-formalizable goals. The labelling procedure described in [Myl92] is a typical example of what can be done; this procedure determines the degree to which a goal is satisficed/denied by lower-level requirements, by propagating such information along positive/negative support links in the goal graph. Goal models can also be at the root of requirements engineering processes; for example, [Boe95] proposes an iterative process model for goal-based negotiation of requirements among multiple stakeholders. When goal formulations can be formalized, *formal reasoning* techniques are expected to do much more than qualitative reasoning. For example, one can use planning techniques to generate admissible system behaviors showing that some desired goal is not achieved, and propose recovery actions [And89], [Fic92]. Beside specification exploration and debugging, one might expect that formal goal models would ultimately enable specifiers to formally verify that goals are achieved or, more constructively, to formally derive requirements specifications that satisfy the goals specified. Preliminary attempts in this direction are encouraging [Dar93], [Fea94], [Lam95].

Formal methods are being increasingly popular for the later stages of the specification process where the boundary between the software and its environment has already been established and the set of software objects/operations has already been made precise. Many languages have been proposed to capture such lower-level specifications; they differ mainly by the particular specification paradigm used. For example, languages such as Z [Pot91], VDM [Jon90] or their object-oriented variants [Lan95] support *state-based* specifications, where the system is described by logical predicates that restrict the set of admissible states; languages such as ERAE [Dub91] or TRIO [Mor92] support *history-based* specifications, where the system is described by predicates that restrict the set of admissible histories; the formalisms in STATECHARTS [Har87] or SCR [Sch93] support *transition-based* specifications in the finite state machine tradition; languages like LARCH [Gut93], ASL [Ast86] and PLUSS [Gau92] support *algebraic* specifications, where the system's data types are specified as universal algebras; languages such as PAISLEY [Zav82] and GIST [Bal83] support *operational* specifications, which may be seen as very high-level programs. The benefit of full formalization is that the specifications can be manipulated formally --e.g., logical consequences, inconsistencies, or refinements can be derived using theorem proving techniques [Jon90], [Pot91], [Geo95]; prototypes can be produced [Bal82], [Hek88], [Doug94]; test data can be generated [BGM91]; counterexamples to claims can be generated through model checking [Jac96]; partial specifications can be matched against reusable specifications from domain libraries [Reu91], [Zar95]; and so forth.

Most formal specification techniques to date are limited in several respects.

- *Limited scope:* while addressing WHAT questions, they do not address the WHY, WHO or WHEN questions that are found in real requirements documents; the latter are captured by goals, agents, and causing events, respectively.
- *Poor separation of concerns:* the languages do not allow specifiers to introduce a strict separation between descriptions of the domain objects/operations and the actual requirements; the need for such separation has been convincingly argued [Jac93].
- *Poor guidance:* while the techniques provide well-defined notations and support for a posteriori analysis, they are not equipped with constructive techniques for guiding the modelling/specification process.
- *Difficulty and cost of use:* applying such methods requires high expertise in mathematical logic and formal systems; due to the scarcity of such expertise the use of formal methods in industrial projects is nowadays limited to core parts of critical systems.

The purpose of this paper is to explore a formal technique for requirements elaboration that can be used (i) upstream in the specification process, as it supports formal reasoning about goals, (ii) constructively, as it suggests ways of refining goals and operationalizing them, and (iii) at reasonable cost, as it hides proofs and their underlying mathematics.

The general principle is to reuse generic refinement patterns. The latter are organized in a rich library according to various strengthening/weakening relationships among abstract goal formulations. The patterns are once for all proved correct and complete; reusing a pattern entails reusing its proof. Tactics are proposed to the requirements engineer for grounding pattern selection on semantic criteria. The general principle is somewhat similar in spirit to the increasingly popular idea of design patterns [Gam95], although it is grounded here on a solid, formal framework.

The reuse of generic refinement patterns is detailed in the paper in the context of the KAOS requirements elaboration methodology [Dar93], [Lam95]. KAOS provides a multi-paradigm specification language and a goal-directed elaboration method. The language combines semantic nets

[Bra85] for the conceptual modelling of goals, consraints, agents, objects and operations in the system; temporal logic [Man92], [Koy92] for the specification of goals, constraints and objects; and state-based specifications [Pot91] for the specification of operations. It supports the separation of requirements from domain descriptions. The method roughly consists of (i) identifying and refining goals progressively until constraints that are assignable to individual agents are obtained, (ii) identifying objects and operations progressively from goals, (iii) deriving requirements on the objects and operations to meet the constraints, and (iv) assigning the constraints, objects and operations to the agents.

The paper is organized as follows. Section 2 provides some background material on the KAOS methodology. Section 3 introduces refinement patterns in this framework and discusses patterns for refining goals into subgoals. Section 4 applies the same principle for operationalizing goals into requirements according to stimulus/response heuristics.

2. Goal-Driven Requirements Elaboration in KAOS

The KAOS methodology is aimed at supporting the whole process of requirements elaboration --from the high-level goals that should be achieved by the composite system to the operations, objects and constraints to be implemented by the software part of it. (The term “composite system” is used here to denote the intended software together with its environment [Fea87].) 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 [Dar93] for more details (various simplifications have been made here based on some rather extensive experience on industrial projects).

2.1 The KAOS language

The specification language provides constructs for capturing various kinds of concepts that appear during requirements elaboration, namely, goals, constraints, agents, entities, relationships, events, actions, views, and scenarios. There is one construct for each type of concept. We define some of these types first; the constructs for specifying their instances are introduced next.

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 domain whose instances may evolve from state to state. It is in general specified in a more specialized way -as an *entity, relationship,* or *event* according as the object is an autonomous, subordinate, or instantaneous object, respectively. Objects are described by invariant assertions.
- *Action:* an action is an input-output relation over objects; action applications define state transitions. Actions may be *caused/stopped* by events. They are characterized by pre-, post- and trigger conditions.
- *Agent:* an agent is another kind of object which acts as processor for some actions. An agent *performs* an action if it is effectively allocated to it; the agent *knows* an object if the states of the object are made observable to 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 often *conflict* with others. Goals *concern* the objects they refer to.
- *Constraint:* a constraint is an implementable goal, that is, a goal that can be formulated in terms of states controllable by some individual agent. Goals must be eventually AND/OR *refined* into constraints. Constraints in turn are AND/OR *operationalized* by actions and objects through strengthenings of their pre-, post-, trigger conditions and invariants, respectively. Alternative ways of assigning responsible agents to a constraint are captured through AND/OR *responsibility* links; the actual assignment of agents to the actions that operationalize the constraint is captured in the corresponding *performance* links.

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; an inner formal assertion layer for *formally defining* the concept. The generic structure is instantiated to specific types of links and assertion languages according to the specific type the concept is an instance of.

For example, a goal that appears at some stage of the elaboration of a meeting scheduler system is to get all participants’ constraints known to the scheduler. The concept ParticipantsConstraintsKnown is of type “Goal”; it could be partially declared and formally defined as follows:

Goal *Achieve* [ParticipantsConstraintsKnown]
InstanceOf InformationGoal
Concerns Meeting, Participant, Scheduler, ...
RefinedTo ConstraintRequested, ConstraintProvided
InformalDef *A meeting scheduler should know the constraints of the various participants invited to the meeting within C days after appointment*
FormalDef $\forall$ m: Meeting, p: Participant, s: Scheduler
Invited (p, m) $\wedge$ Scheduling (s, m)
$\Rightarrow \Diamond_{\leq Cd}$ Knows (s, p.Constraints)

The declaration part of the specification above states that the goal ParticipantsConstraintsKnown (i) is concerned with keeping agents informed about object states, (ii) refers to objects such as Participant or Scheduler, and (iii) is refined into two subgoals. (The latter turn to be constraints assignable to single agents; the fact that the ConstraintRequested constraint is assignable to the Scheduler agent would be stated in the declaration of that constraint.)

The formal assertion defining this goal is written in a real-time temporal logic borrowed from [Koy92]. In this paper we will use some classical operators for temporal referencing: **o** (in the next state), • (in the previous state), ◊ (eventually), ♦ (some time in the past), ❑ (always in the future), ■ (always in the past), $\mathcal{U}$ (always in the future until), $\mathcal{W}$ (always in the future unless). Real-time restrictions are indicated by subscripts (e.g., $_{\leq Cd}$ where d denotes the day time unit). Such restrictions will not be considered in the refinement patterns discussed in this paper; we will stick to "classical" temporal logic [Man92].

In the formal assertion above, the predicate Invited (p, m) means that, in the current state, an instance of the Invited relationship links variables p and m of sort Participant and Meeting, respectively. The Invited relationship, Participant agent and Meeting entity are declared in other sections of the specification, e.g.,

Agent Participant
  CapableOf CommunicateConstraints, ...
  Has Constraints: **Tuple** [ExcludedDates: **SeqOf** [*TimeInterval*],
                                   PreferredDates: **SeqOf** [*TimeInterval*]]
  ...
Relationship Invited
  Links Participants {card: 0:N}, Meeting {card: 1:N}
  DomInvar ∀p: Participant, m: Meeting
        Invited (p, m) ⇔ p ∈ Requesting[-,m].ParticipantsList

In the declarations above, Constraints is declared as an attribute of Participant (this attribute was used in the formal definition of ParticipantsConstraintsKnown). Also note that the invariant defining Invited is not a requirement, but a domain description. Object inheritance is of course supported, e.g., an agent type ImportantParticipant can be introduced as a specialization of Participant with some extra attributes and invariants.

As mentioned earlier, operations are specified formally by pre- and postconditions, for example,

Action DetermineSchedule
  Input Requesting, Meeting {**Arg**: m};
  Output Meeting {**Res**: m}
  DomPre Requesting (-,m) ∧ ¬ Scheduled (m)
  DomPost Feasible (m) ⇒ Scheduled (m)
        ∧ ¬ Feasible (m) ⇒ DeadEnd (m)

In a KAOS specification, the outer declaration layer is much useful for requirements traceability (through semantic net navigation) and specification reuse (through queries). The inner assertion level is introduced for formal reasoning, and will be used later in the paper. Note that the specifier does not necessarily need to use the full power of this "two-button" language; in fact, industrial users tend to use the semantic net level (for which a concrete graphical syntax is available) without "pressing the formal button".

2.2 The elaboration method

The following steps may be followed to systematically elaborate KAOS specifications from high-level goals.

- Elaborate the goal AND/OR structure by defining goals and their refinement/conflict links until implementable constraints are reached; offspring goals are identified by asking HOW questions whereas parent goals are identified by asking WHY questions.
- Identify the objects concerned by goals and describe their domain properties.
- Identify object state transitions that are meaningful to the goals, specify them as domain pre- and postconditions of actions, and identify agents that could have those actions among their capabilities.
- Derive additional pre- and postconditions of actions and invariants of objects so as to ensure that all constraints are met (requirements that operationalize the goals are obtained thereby).
- Identify alternative responsibilities for constraints; make decisions among refinement, operationalization, and responsibility alternatives (with process-level objectives such as resolving conflicts, reducing costs, increasing reliability, avoiding overloading agents, etc.); assign the actions to agents that can commit to guaranteeing the requirements in the alternatives selected.

The steps above are ordered by data dependencies; they may be running concurrently, with possible backtracking at every step. The refinement patterns discussed below are to be used in the first step above.

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, constraints, objects and actions 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 ⇒ ◊ Q or *Cease:* P ⇒ ◊ ¬ Q
    Maintain: P ⇒ ❑ Q or *Avoid:* P ⇒ ❑ ¬ Q
  – Goals are also classified by type of requirements they will drive with respect to the agents concerned (e.g., SatisfactionGoal, InformationGoal, ConsistencyGoal, SafetyGoal, PrivacyGoal, etc.)
  – Constraints are in the HardConstraint category if they may never be violated, or in the SoftConstraint category if they are likely to be temporarily violated.
  – Actions are Modify or Inspect actions according as they modify some object states or not.

  Such taxonomies are constrained by rules, e.g.,
  – SafetyGoals are AvoidGoals to be refined in HardConstraints;
  – PrivacyGoals are AvoidGoals on *Knows* predicates;
  – SoftConstraints must have associated ModifyActions to restore them.

  Meta-level rules can be used to ensure consistency and completeness of the requirements model being elaborated. For example, declaring the constraint ConstraintProvided that appeared in the refinement of the goal ParticipantsConstraintsKnown above as a *SoftConstraint* would prompt a question about what restoration action should be foreseen in case of violation; as a result, actions like emailing a reminder or giving a phone call to negligent participants would be introduced.

- Tactics capture heuristics to drive the elaboration or select 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.

2.4 Constructive formal support

Formal derivation rules can be used to derive requirements from goals. Consider, for example, the following inference rule.

$$\frac{\textit{Constraint: } \Box\,[\, C \wedge (\, P1 \wedge \circ\, P2 \;\Rightarrow\; Q1 \wedge \circ\, Q2 \,)\,]\,,\quad \textit{DomPre: } P1,\ \textit{DomPost: } P2}{\textit{RequiredPre: } Q1,\ \textit{RequiredPost: } Q2}$$

As a very simple example of using this inference rule, consider the following SafetyConstraint for a lift system:

HardConstraint Maintain [DoorsClosedWhileMoving]
  Refines SafeTransportation
  FormalDef ∀ l: Lift, d: Doors, f,f’: Floor
  PartOf (d, l) ⇒ ❑ [LiftAt (l,f) ∧ o LiftAt (l,f’) ∧ f’   f
  ⇒ d.State = 'closed' ∧ o (d.State = 'closed')]

The following required pre-/postconditions are then formally derived using the rule above:

Action GoToFloor
  Input Lift {**arg**: l}, Floor {**arg**: f, f’}, Passenger {**arg**: p};
  Output LiftAt
  DomPre LiftAt (l, f) ∧ Requesting (p, f’) ∧ f’   f
  DomPost LiftAt (l, f’)
  RequiredPre for DoorsClosedWhileMoving: *d.state = 'closed'*
  RequiredPost for DoorsClosedWhileMoving: *d.state = 'closed'*

3. Refining Goals into Subgoals

3.1 Basic idea

Experience with KAOS has revealed that correct goal refinements are often hard to find; goal decompositions made by hand are usually incomplete and sometimes inconsistent [Lam95]. Moreover, the goal graph usually has few OR-branches and contain implicit choices; interesting alternatives may be overlooked. The idea is therefore to provide formal support for building goal refinement graphs that are *complete*, proved *correct*, and integrate *alternatives*. This section introduces domain-independent refinement patterns as a way to achieve this objective. Patterns are proved correct and complete once for all by the pattern designer; reusing the pattern entails reusing the proof, and the mathematics can therefore be hidden to the user of patterns. The concept of *refinement* and *refinement patterns* are first carefully defined. Next an example of frequent refinement pattern is shown and proved. The benefits of using refinement patterns are then discussed. Finally, we detail the contents and structure of our library of patterns.

Definition. A set of goal assertions G1, G2,..., Gn is a *complete refinement* of a goal assertion G iff the following conditions hold:

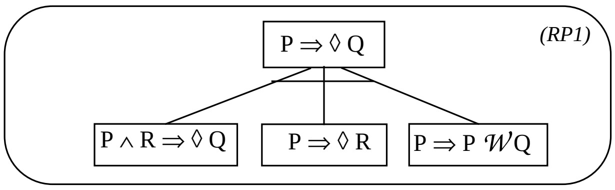
1. $G1 \wedge G2 \wedge \ldots \wedge Gn \vdash G$ (*entailment*)
2. $\forall i: \bigwedge_{j\ i} Gj \nvdash G$ (*minimality*)
3. $G1 \wedge G2 \wedge \ldots \wedge Gn \nvdash \text{false}$ *(consistency)*

To avoid trivial refinements consisting in rewriting G into logically equivalent forms, the following rule is added:

4. n > 1 or the refinement relies on domain knowledge.

Definition. A *refinement pattern* is a one-level AND-tree of abstract goal assertions such that the set of leaf assertions is a complete refinement of the root assertion.

As a first example, the following refinement pattern proposes a way to decompose *Achieve* goals into three subgoals.



This pattern is generic; it can be instantiated to completely different situations. A simple instantiation is presented first; other instantiations will be used later.

Consider the train control system studied in [Fea94]. One functional goal is to ensure that trains move through consecutive blocks:

Goal Achieve [TrainProgress]
  FormalDef (∀ t: Train, b: Block) [At (t, b) ⇒ ◊ At (t, b+1)]

A particular case that comes directly to mind is when block*b+1*’s signal is set to ‘go’. One may thus instantiate the meta-variable *R* in the first subgoal of the pattern above to the predicate Go [b+1] formalizing this situation; hence the refinement into the following three subgoals:

Goal Achieve [ProgressWhenGoSignal]
  FormalDef ∀ t: Train, b: Block
  At (t, b) ∧ Go[b+1] ⇒ ◊ At (t, b+1)

Goal Achieve [SignalSetToGo]
  FormalDef ∀ t: Train, b: Block
  At (t, b) ⇒ ◊ Go[b+1]

Goal Maintain [TrainWaiting]
  FormalDef ∀ t: Train, b: Block
  At (t, b) ⇒ At (t, b)𝒲At (t, b+1)

The last subgoal obtained from this pattern states that a train *t* must stay in block *b* unless it is in block *b+1*; backward moves are thereby discarded.

Refinement patterns are useful for the following reasons.

- They allow formal reasoning to be hidden to the requirements engineer;
- They may help detecting incomplete refinements and reconnoitering requirements;
- They allow choices underlying refinements to be made explicit.

Each point is illustrated now in turn.

3.1.1 Hiding mathematics

Patterns are proved correct once for all and reused many times through instantiation. The pattern above, for instance, can be proved using the proof theory of temporal logic [Man92] as follows.

Proof

1.	$P \Rightarrow \Diamond R$	Hyp
2.	$P \wedge R \Rightarrow \Diamond Q$	Hyp
3.	$P \Rightarrow P \mathcal{W} Q$	Hyp
4.	$P \Rightarrow (P \mathcal{U} Q) \vee \Box P$	3, def of Unless
5.	$P \Rightarrow \Diamond Q \vee \Box P$	4, def of Until
6.	$P \Rightarrow \Diamond R \wedge (\Diamond Q \vee \Box P)$	1, 5, strengthen consequent
7.	$P \Rightarrow (\Diamond R \wedge \Diamond Q) \vee (\Diamond R \wedge \Box P)$	6, distribution
8.	$P \Rightarrow (\Diamond R \wedge \Diamond Q) \vee \Diamond (R \wedge P)$	7, trivial lemma
9.	$P \Rightarrow (\Diamond R \wedge \Diamond Q) \vee \Diamond \Diamond Q$	8, 2, strengthen consequent
10.	$P \Rightarrow (\Diamond R \wedge \Diamond Q) \vee \Diamond Q$	9, $\Diamond$-idempotence
11.	$P \Rightarrow \Diamond Q$	10, absorption

Temporal logic proofs tend to be rather complex even for simple refinements. Such proofs are built for every pattern before inclusion into the library. A pattern may be reused by requirements engineers without need to prove that its instantiations produce correct refinements; the proofs are instantiated accordingly.

3.1.2 Checking refinements for completeness

Refinement patterns can be used to check whether given decompositions are complete refinements. This way of using patterns is very important; it is our experience that intuitive refinements made "by hand" tend to be incomplete [Lam95]. Incomplete goal refinements result in incomplete requirements.

Consider the specification of a general resource management system. A root in the goal refinement graph states that any relevant request for a resource should be eventually satisfied by providing a resource unit from the repository to the requestor.

Goal Achieve [ResourceRequestSatisfied]
FormalDef $\forall$ u: User, res: Resource, rep: Repository
Requesting (u, res, rep) $\wedge$ Registered (res, rep)
$\Rightarrow \Diamond$ ($\exists$ ru: ResourceUnit) (Unit (ru, res) $\wedge$ Has (u, ru))

The above goal can be trivially satisfied in the particular case where some resource unit is available. This suggests the following decomposition:

Goal Achieve [AvailableResourceProvided]
FormalDef $\forall$ u: User, res: Resource, rep: Repository
Requesting (u, res, rep) $\wedge$
($\exists$ ru: ResourceUnit) (Unit (ru, res) $\wedge$ Available (ru, rep))
$\Rightarrow \Diamond$ ($\exists$ ru: ResourceUnit) (Unit (ru, res) $\wedge$ Has (u, ru))

Goal Achieve [ResourceUnitAvailable]
FormalDef $\forall$ u: User, res: Resource, rep: Repository
Requesting (u, res, rep) $\wedge$ Registered (res, rep)
$\Rightarrow \Diamond$ *($\exists$ ru: ResourceUnit) (Unit (ru, res) $\wedge$ Available (ru, rep))*

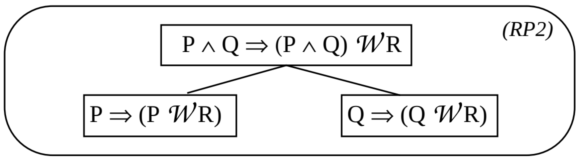
In the first subgoal above, the conjunct Registered (res, rep) has been kept implicit in the antecedent due to the following domain invariant attached to the Available relationship:

($\exists$ ru: ResourceUnit) (Unit (ru, res) $\wedge$ Available (ru, rep))
$\Rightarrow$ Registered (res, rep)

The pattern (RP1) introduced above tells the specifier that this refinement is incomplete. There is a missing Maintain subgoal that can be derived systematically, namely,

Goal Maintain [??]
FormalDef $\forall$ u: User, res: Resource, rep: Repository
Requesting (u, res, rep) $\wedge$ Registered (res, rep)
$\Rightarrow$ (Requesting (u, res, rep) $\wedge$ Registered (res, rep))
$\mathcal{W}$ ($\exists$ ru: ResourceUnit) (Unit (ru, res) $\wedge$ Has (u, ru))

This missing subgoal can in turn be refined by application of another frequently used pattern, called the *"Maximize Cohesion"* pattern:



The resulting refinement for the missing subgoal is:

Goal Maintain [PendingRequest]
FormalDef $\forall$ u: User, res: Resource, rep: Repository
Requesting (u, res, rep)
$\Rightarrow$ (Requesting (u, res, rep)
$\mathcal{W}$ ($\exists$ ru: ResourceUnit) (Unit (ru, res) $\wedge$ Has (u, ru))

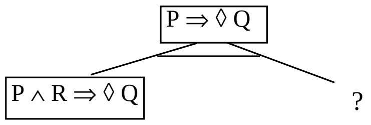
Goal Maintain [ResourceRegistered]
FormalDef $\forall$ res: Resource, rep: Repository
Registered (res, rep) $\Rightarrow \Box$ Registered (res, rep) ,

where a stronger version for the second subgoal has eventually been selected. Asking what is meant by "requests should be maintained" or "resource registration should be maintained" (or "why a resource should *not* be kept registered") triggers the discovery of new, more concrete requirements --e.g., reservation instalments or fines for the first subgoal, policies to avoid lost or stolen resource units for the second subgoal.

Another example of refinement debugging through such patterns, for a quite different (and real) system, can be found in [Lam95].

3.1.3 Constructive refinement and reconnoitering of alternatives

Refinement patterns can be used to help completing partial refinements. The requirements engineer might for instance be interested in retrieving all complete patterns that match the following partial pattern:



As suggested before, this partial pattern corresponds to a frequent situation where some easy or limit case *R* comes directly to mind to define a "trivial" subgoal. A formal query on the pattern library allows one to retrieve candidate patterns to complete the above refinement, e.g.,

	$P \Rightarrow \Diamond R$ and $P \mathcal{W} Q$	as other subgoals,
or	$P \wedge \neg R \Rightarrow \Diamond R$ and $P \mathcal{W} Q$	as other subgoals,
or	$\neg R \Rightarrow \Diamond R$ and $P \mathcal{W} Q$	as other subgoals.

To illustrate the point, consider the train control system again. The first candidate pattern has already been instantiated above. The instantiation of the second candidate pattern would produce the following alternative subgoal for SignalSetToGo:

Goal Achieve [SignalSetToGo]
FormalDef ∀ t: Train, b: Block
$\text{At}(t, b) \land \neg \text{Go}[b+1] \Rightarrow \lozenge \text{Go}[b+1]$

whereas the third retrieved pattern would produce a third alternative:

Goal Achieve [SignalSetToGo]
FormalDef ∀ t: Train, b: Block
$\neg \text{Go}[b+1] \Rightarrow \lozenge \text{Go}[b+1]$

The three alternative subgoals generated result in different policies. In the first two alternatives, signal changes are triggered by train arrivals. In the third alternative, signal changes occur under every circumstance regardless of train arrivals. The requirements "implementing" the SignalSetToGo subgoal issued from the two first patterns need to introduce a new agent, namely, a train detection device near each signal.

3.1.4 Making choices explicit

Pattern-directed refinement may sometimes allow requirements engineers to make explicit some design choices that were hidden in informal elaborations or past designs. We experienced this in reengineering the requirements of a cable phone system for a major telecommunication company. The system had to be built on top of an existing technology that imposed pre-existing choices. As an example abstracted from this system, consider the following specification of a point-to-point communication between two agents:

Goal Achieve [CommunicationSatisfied]
FormalDef ∀ p1, p2: Point, m: Message
$\text{CallRequest}(p1, p2) \land (p1.\text{msgOut} = m) \Rightarrow \lozenge (p2.\text{msgIn} = m)$

The way taken to satisfy this goal was to require that (i) a channel be allocated for the communication, and (ii) the communication use the allocated channel. Consider subgoal (i). The weakest refinement pattern we used led to the following formalization:

Goal Achieve [ChannelAllocatedUponCall]
FormalDef ∀ p1, p2: Point
CallRequest (p1, p2)
$\Rightarrow \lozenge (\exists c: \text{Channel}) (\text{Allocation}(p1, p2, c))$

A stronger refinement pattern produced the requirement that channels be allocated not just when calls are issued, but as soon as pairs are plugged into the network:

Goal Achieve [ChannelAllocatedUponPlugIn]
FormalDef ∀ p1, p2: Point
Connection (p1, p2)
$\Rightarrow \square (\exists c: \text{Channel}) (\text{Allocation}(p1, p2, c))$

Under this refinement the following domain invariant was attached to the CallRequest relationship:

$(\forall p1, p2: \text{Point})\ \text{CallRequest}(p1, p2) \Rightarrow \text{Connection}(p1, p2)$

This version still allows the channel to vary over time. An even stronger refinement pattern produced the requirement that the same channel be used forever:

Goal Achieve [ChannelAllocatedStatically]
FormalDef ∀ p1, p2: Point
Connection (p1, p2)
$\Rightarrow (\exists c: \text{Channel}) (\square\ \text{Allocation}(p1, p2, c))$

(Note that the difference is just in the ❑ operator jumping over the existential quantifier.) The use of patterns explicitly structured in a weakening/strengthening hierarchy may thus help selecting refinements that make design choices explicit (here, dynamic vs. static allocation of channels) and better fit actual needs (in view of other parent goals in the graph that need to be achieved as well).

As this example suggests, our pattern library is structured according to a weakening/strengthening hierarchy. Section 3.4 investigates this structure more precisely. Before that, Sections 3.2 and 3.3 provide typical examples of propositional and first-order refinement patterns, respectively.

3.2 Propositional refinement patterns

Propositional patterns are worth being considered for global refinements in which quantifiers do not interfere with the refinement.

A sample of frequently used propositional patterns for *Achieve* goals is shown in Table 1. Each row in the table gives a possible refinement of the parent goal $P \Rightarrow \lozenge Q$. Pattern RP3 defines a *milestone-driven* refinement where an intermediate state satisfying *R* must first be reached, from which a final state satisfying *Q* must be reached. Pattern RP4 advocates for a decomposition by cases. Pattern RP5 identifies a condition *R* which must be set in order to reach *Q*. Pattern RP6 *strengthens* RP5 by requiring that condition *R* hold independently of the truth value of *P*. RP7 strengthens RP3 by requiring *R* to continuously hold until *Q* holds.

	Subgoal	**Subgoal**	**Subgoal**
RP3	$P \Rightarrow \lozenge R$	$R \Rightarrow \lozenge Q$	
RP4	$P \land P1 \Rightarrow \lozenge Q1$	$P \land P2 \Rightarrow \lozenge Q2$	$\square (P1 \lor P2)$ $Q1 \lor Q2 \Rightarrow Q$
RP5	$P \land \neg R \Rightarrow \lozenge R$	$P \land R \Rightarrow \lozenge Q$	$P \Rightarrow \square P$
RP6	$\neg R \Rightarrow \lozenge R$	$P \land R \Rightarrow \lozenge Q$	$P \Rightarrow \square P$
RP7	$P \Rightarrow \lozenge R$	$R \Rightarrow R\ \mathcal{U}\ Q$	

TABLE 1. Some propositional patterns for *Achieve* goals

As already suggested, the use of one pattern instead of another may lead to completely different systems. For example, RP5 and RP6 in a library system lead to completely different policies for book loan. Suppose *P* stands for "a member requests some book" and *Q* stands for "the member can borrow a copy of that book". Now suppose *R* stands for "a copy of that book is available". Patterns RP5 and RP6 both require that a request for a book which has a copy available is eventually satisfied. They disagree however on what happens when there is no copy available. RP5 requires that a book requested with no copy available has a copy eventually available; RP6 requires that every copy of a book, be it requested or not, must eventually become available. In the former case, borrowers are allowed to keep copies as long as there is no other member requesting a copy of the corresponding book; in the latter case, they have to return copies even if there is no request.

An extensive set of patterns for *Achieve* and *Maintain* goals can be found in [Dar95]; all proofs of correctness and completeness are given there.

3.3 First-order refinement patterns

First-order goal assertions are classified into four categories with respect to refinement:

1. the assertion consists of a composition, via logical/temporal connectives, of closed formulas that are not decomposed by the refinement;
2. the consequent of the assertion is a closed formula taking the form of a conjunction/disjunction of subformulas which have to be decomposed by the refinement and cannot be rewritten to match case 1;
3. the antecedent of the assertion is a closed formula taking the form of a conjunction/disjunction of subformulas which have to be decomposed by the refinement and cannot be rewritten to match case 1;
4. the antecedent and the consequent of the instantiated assertion are not closed formulas, because they are sharing variables.

Formulas in the first category can be treated as propositional formulas; the closed subformulas are abstracted as propositional symbols. The third category is just the dual of the second. We will therefore concentrate on cases 2 and 4 successively.

Notations. When no ambiguity arises,

- free variables in formulas will be assumed to be universally quantified; for instance,
  $P \land Q1 (x) \Rightarrow \Diamond (Q1 (x) \land Q2 (x))$ *stands for*
  $(\forall x) [P \land Q1 (x) \Rightarrow \Diamond (Q1 (x) \land Q2 (x))]$
- predicate arguments will be dropped; for instance,
  $\Diamond (\exists x) Q1 \lor \Diamond (\exists x) Q2 \Rightarrow \Diamond (\exists x) (Q1 \land Q2)$ *stands for*
  $\Diamond (\exists x) Q1 (x) \lor \Diamond (\exists x) Q2 (x) \Rightarrow \Diamond (\exists x) (Q1 (x) \land Q2 (x))$

3.3.1 Decomposing consequents

The most frequent generic assertion in this category is $P \Rightarrow \Diamond (\exists x) (Q1 \land Q2)$. Some possible refinements of it are shown in Table 2. Each row contains a refinement that has been proved correct and complete. More patterns can be found in [Dar95]. For instance, other patterns can be derived by dropping the term *P* in the second subgoal (as in propositional patterns) or by replacing the *eventually* operator by an *until* operator.

	Subgoal	Subgoal	Subgoal
FO1	$P \Rightarrow \Diamond (\exists x) Q1$	$P \land Q1 (x)$ $\Rightarrow \Diamond (\exists x) (Q1 \land Q2)$	$P \Rightarrow \Box P$
FO2	$P \Rightarrow \Diamond (\exists x) Q1$	$P \land Q1 (x)$ $\Rightarrow \Diamond (Q1 (x) \land Q2 (x))$	$P \Rightarrow \Box P$
FO3	$P \Rightarrow \Diamond (\exists x) Q1$	$P \land Q1 (x) \Rightarrow \Diamond Q2 (x)$ $Q1 (x) \Rightarrow \Box Q1 (x)$	$P \Rightarrow \Box P$

TABLE 2. Some first-order patterns for *Achieve* goals

To illustrate the use of these patterns, consider the meeting scheduler system again. A meeting can be planned (*Q*) if there is a date on which the participants agree and a venue for the meeting. The three refinements in Table 2 propose to sequentialize the problem of finding a date and a venue. First a date must be found (*Q1*). FO2 and FO3 state that a venue must be found for the date found (*Q2*). Pattern FO1 is weaker; provided a date can be found, it requires one to find a date (not necessarily the one initially foreseen) and a venue for the latter date.

3.3.2 Variables common to antecedents and consequents

Universally quantified variables common to the antecedent and consequent of the parent goal do not interfere with refinements; one may (i) use the intantiation rule from classical logic on the universally quantified variables, (ii) apply a refinement pattern to the resulting formula, and (iii) use the inverse generalization rule on each subgoal to reintroduce the quantifiers for the variables frozen in (i).

For an existentially quantified variable common to the antecedent and consequent of the parent goal, the refinements must guarantee that the consequent of the parent formula holds for the value satisfiying the antecedent. This explains why subgoals generally have to introduce universal quantifiers instead of existential ones. For instance, a possible refinement for the pattern $(\exists x) (P \Rightarrow \Diamond Q)$ is:

$$(\exists x) (P \Rightarrow \Diamond R) , R(x) \Rightarrow \Diamond Q(x)$$

Other patterns can be found in [Dar95].

3.4 Exploring the pattern space

To be effective in practice, the pattern library should have the following properties.

- *Relevance*: the library should provide patterns that are actually needed by requirements engineers;
- *Retrievability:* relevant patterns should be retrieved easily.

3.4.1 Relevance

A two-step process was followed to identify relevant refinement patterns.

- *Inductive step:* candidate patterns were identified by abstracting goal refinements from a wide variety of case studies --among others, resource management systems with multiple specializations (e.g., sharable resources, returnable resources, reservable resources, chargeable resources, convertible resources); transportation systems; communication systems; process monitoring; meeting scheduling; conference organization. The candidate refinement patterns that emerged from these case studies were proved correct using the proof theory of temporal logic [Man92]. The proofs were *bottom-up*, in that the conjunction of subgoals was shown to minimally entail the parent goal. The result of this step was a preliminary library of complete *AND*-refinements reusable through instantiation. This library had the following weaknesses:

patterns were unrelated to each other; they did not explore all alternatives; they contained implicit choices.

• *Deductive step:* the library of AND-refinements obtained in the inductive step was reorganized by deriving refinement patterns *top-down* from basic patterns. Refinement choices were thereby made explicit; this opened the way to a systematic investigation of other refinement alternatives. The result is an *AND/OR* library of refinement patterns, which can be seen as a a directed graph of patterns linked through weakening/strengthening relationships. Fig. 1 shows a piece of this graph. The higher a goal in the figure, the weaker it is.

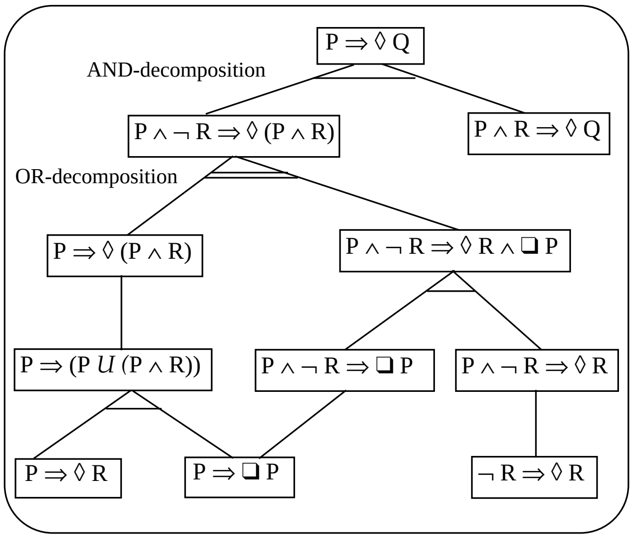


FIGURE 1 A portion of the refinement graph

3.4.2 Retrievability

The refinement graph structures the pattern space for the library designer. This structure should also help library users select appropriate refinements in some given situation. Patterns are selected by retrieving all *AND*-subtrees in this graph whose root matches the goal assertion to be refined; the leaves of the retrieved subtrees provide the subgoals to be instantiated according to the substitutions revealed by the match. As suggested before,there are frequent cases where the user is looking for ways to complete partial refinements already identified; the match is then constrained by the subgoals given. The retrieval of patterns can thus be supported by a query system implemented in terms of standard tree matching primitives on abstract syntax trees. Although this facility has not been implemented yet, we see no difficulty as the KAOS environment kernel we are currently building uses the CSG syntax-directed environment generator [Rep89].

If no refinement pattern is found to meet the needs exactly, the user or designer may define new patterns on top of existing ones. Frequently used patterns that are not yet in the library should clearly be incorporated into it.

The retrieval mechanism above is based on matching concrete and abstract patterns syntactically. Further assistance is provided to help selecting appropriate patterns according to more semantic criteria. Meta-level *refinement tactics* structure the pattern space according to the nature of the decomposition proposed. The patterns are indexed by tactics; they can be used in the context described by the tactics. Fig. 2 shows the domain-independent tactics identified so far.

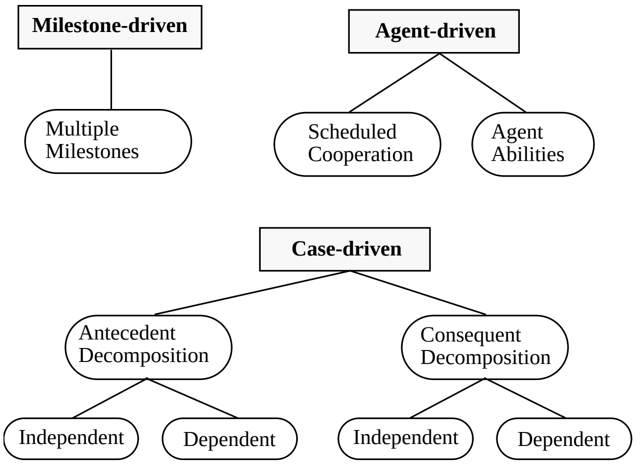


FIGURE 2 Domain-independent tactics for goal refinement

Milestone-driven tactics. This tactics suggests identifying milestone states, that is, states that must be reached sooner or later to achieve the target predicate *Q* appearing in the parent goal P ⇒ ◊ Q. This tactics corresponds to a well-known heuristics in planning by problem reduction [Nil71].

Case-driven tactics. These tactics suggest identifying different cases to satisfy the goal. The cases may concern the current state satisfying *P* from which the target predicate *Q* has to be reached (*antecedent decomposition*), or the target state satisfying *Q*. Cases may be completely independent from each other; they may also be dependent. An example of a dependent, antecedent decomposition, case-driven tactics is the tactics which suggests identifying particular cases from which the target predicate *Q* can be reached trivially, and general cases that reduce to such cases eventually. The distinction between normal and exceptional cases in the antecedent *P* or target predicate *Q* yields other case-driven tactics.

Agent-driven tactics. These tactics suggest identifying the group of agents involved in the achievement of the parent goal and splitting this group into smaller subgroups of agents that can achieve corresponding subgoals according to their abilities or to some known schedule.

The patterns RP3, RP7, and FO1 to FO3 above are indexed by the milestone-driven tactics. Patterns RP4, RP5 and RP6 are indexed by case-driven tactics.

4. Operationalizing goals

Goal refinement must eventually result in the identification and specification of requirements whose responsibility must be assigned to individual agents such as programs in the software to be developed, or devices, humans or existing programs in the environment. The boundary between the new software and its environment is obtained thereby. As suggested in the previous sections, the refinement process can be supported at a first stage by reuse of nonoperational patterns --that is, patterns which do not refer to operational notions such as agents, events, actions, etc. In a second stage, *operational patterns* must be considered.

We have studied one family of such patterns to date. *Stimulus-response* patterns apply when the cooperation between agents required to achieve some goal can be modelled by the exchange of stimuli and responses between the agents.

4.1 Stimulus-response patterns

A *stimulus* is an event perceived by some agent which requires some action to be performed by the agent. A *response* is an agent reaction to some stimulus.

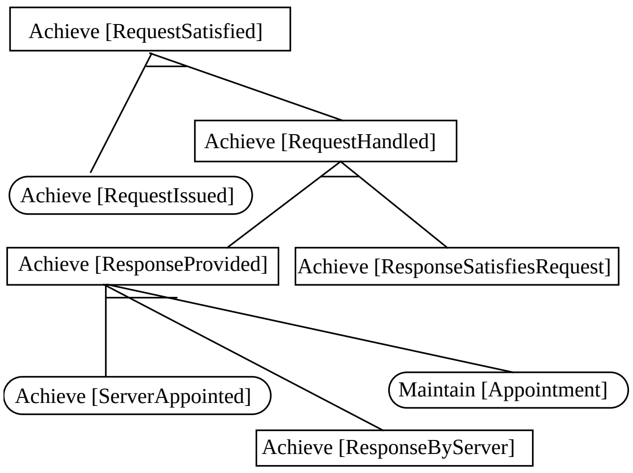
Examples. In a phone network system, a user picking up her phone defines a stimulus being sent to the exchange system; the latter reacts by sending a dial tone. In a patient monitoring system, stimuli can be events that occur whenever some monitored values become greater or lower than some accepted treshold. Responses can be provided by raising alarms which in turn can be stimuli for nurses on duty.

The stimulus-response family of patterns suggests ways of operationalizing goals according to their classification:

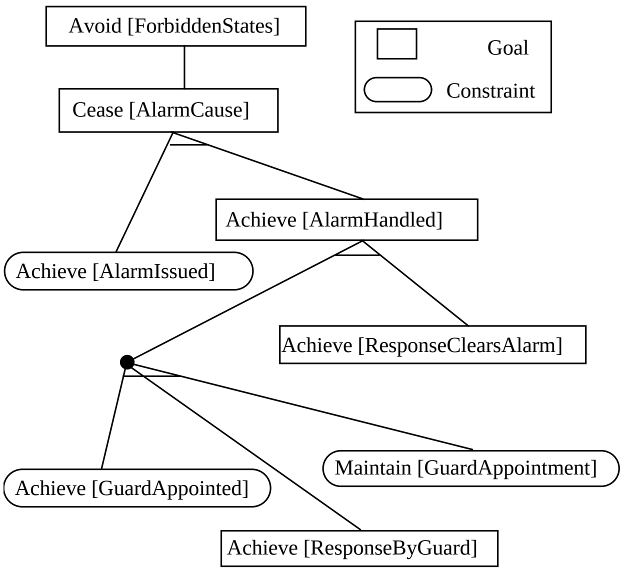
- in case of Satisfaction goals, stimuli represent requests for services; responses indicate that the requested services have been provided;
- in case of Safety, Robustness and Privacy goals, stimuli represent alarms that detect situations leading to goal violations; responses indicate that alarm causes have been resolved;
- in case of Information goals, stimuli represent object state changes of interest to agents; responses represent notifications to the agents of the state change.

Fig. 3 shows AND-trees of goals and constraints proposed by stimulus-response patterns for *Satisfaction* and *Safety* goals. In Fig. 3 (a), a stimulus is produced to satisfy the RequestIssued constraint. The latter is under the responsibility of the agent who requests the service. The response is modelled by the right part of the tree; the service request is satisfied as soon as there is a response to the request by some appointed server that addresses the request appropriately. In Fig. 3 (b), the violation of Safety, Robustness and Privacy goals is prevented whenever undesirable situations can be detected in advance to raise alarms (stimuli). Responses to such alarms must be provided by some appointed guard in a way that clears the alarm causes. Note that ResponseByGuard is not necessarily a constraint as it may need the cooperation of several agents to be achieved. For instance, a response can itself be a stimulus sent to other agents; stimuli can be chained, forwarded, broadcasted, acknowledged, and so on.

It is worth noticing that each goal and constraint appearing


Achieve [RequestSatisfied]
Achieve [RequestHandled]
Achieve [RequestIssued]
Achieve [ResponseProvided]
Achieve [ResponseSatisfiesRequest]
Achieve [ServerAppointed]
Maintain [Appointment]
Achieve [ResponseByServer]

(a) Satisfaction goals



(b) Safety, robustness, and privacy goals

FIGURE 3 Goal operationalization through stimuli-responses

in Fig. 3 has a formal definition; the refinement patterns were all proved correct once for all [Dar95]. We show a derivation leading to Fig. 3 (b).

4.2 Safety requirements

Safety, robustness, and privacy goals are formalized in terms of *Avoid* patterns:

Goal Avoid [ForbiddenStates]
InstanceOf {SafetyGoal, PrivacyGoal, RobustnessGoal}
FormalDef ❑ ¬ C

Alarms are introduced to warn the environment about some possible accidental violation of goals – e.g., when some intrusion is detected, or when monitored values exceed some threshold. The alarm is effective if the violation can be anticipated and if the response to the alarm allows the violation to be avoided. Therefore, we suppose that the following *effectiveness condition* holds:

$$S \wedge \Box_{\geq D} P \Leftrightarrow \Diamond_{\geq D} C$$

The effectiveness condition states that some contextual condition *S* and some persistent property *P* holding continuously over a period *D* altogether result in subsequent goal violation. For instance, in a house alarm system *C* is instantiated to robber intrusion, *S* is instantiated to the alarm system being on, and *P* is instantiated to intrusion detection. In a patient monitoring system, *C*, *S*, and *P* might be instantiated to patient death, monitoring working, and some monitored value exceeding its treshold, respectively.

According to the effectiveness condition, the above goal can be rewritten as

Goal Avoid [ForbiddenStates]
FormalDef $\Box \neg (S \wedge \Box_{\geq D} P)$

which in turn can be rewritten as:

Goal Cease [AlarmCause]
FormalDef $S \wedge P \Rightarrow \Diamond_{\leq D} \neg P$

We now introduce the Alarm event type:

Event Alarm
Has Source: <Agent>
Target: <Agent>
About: <Object>
DomInvar $(\forall$ a: Alarm) [Occurs (a) $\Leftrightarrow (\bullet \neg P \wedge P)$]

The goal Cease [AlarmCause] can then be partially refined into

HardConstraint Achieve [AlarmIssued]
FormalDef $S \wedge \bullet \neg P \wedge P \Rightarrow (\exists$ a: Alarm) (Occurs (a) $\wedge$ ...) ,

Goal Achieve[AlarmHandled]
FormalDef $(\forall$ a: Alarm) Occurs (a) $\Rightarrow \Diamond_{\leq D} \neg P$

Pattern-directed refinement of the latter goal introduces a guard appointed to react on alarms. The refinement results in the following subgoals/constraints:

HardConstraint Achieve [GuardAppointed]
FormalDef $\forall$ a: Alarm
Occurs (a) $\Rightarrow \Diamond (\exists$ g: Guard) Appointed (g, a)

Goal Achieve [ResponseByGuard]
Formal Def $\forall$ a: Alarm, g: Guard
Appointed (g, a) $\Rightarrow \Diamond$ Response (g, a)

HardConstraint Maintain [GuardAppointment]
Formal Def $\forall$ a: Alarm, g: Guard
Appointed (g, a) $\Rightarrow \Box$ Appointed (g, a)

Goal Achieve [ResponseClearsAlarm]
Formal Def $\forall$ g: Guard, a: Alarm
Response (g, a) $\Rightarrow \Diamond_{\leq D} \neg P$

Depending on the nature of the response provided, the above goals can be converted or not into constraints assignable to single agents.

5. Conclusion

This paper has presented a constructive approach to goal refinement and operationalization. The principle is to reuse generic refinement patterns from a library structured according to weakening/strengthening relationships among patterns. The retrieval of relevant patterns is based on matching partial goal formulations to AND-trees in the library; complementary tactics allow patterns to be selected on more semantic criteria. A distinction has also been made between nonoperational refinement patterns and operational patterns for the introduction of constraints, agents, events, and actions.

The patterns are once for all proved correct and complete. Their use allows (i) the tedious mathematics involved in proofs to be hidden, (ii) given refinements to be checked against completeness and consistency, (iii) partial refinements to be completed, (iv) design choices to be made explicit, and (v) alternative patterns for a same goal to be explored for requirements reconnoitering.

Our pattern library and set of tactics are by no means complete. As suggested in Sections 3.1.4 and 3.4.1, they both were constructed inductively from our experience over the past four years in using KAOS to elaborate requirements for a wide variety of systems, ranging from research case studies to a few industrial projects. The abstract examples in this paper were aimed to reflect this variety. More experience will certainly suggest more relevant patterns and tactics. Our current experience though has convinced us that the approach is effective for *real* systems. Many of the patterns illustrated here occurred recurrently. They were especially helpful in debugging specifications, notably by pointing out subgoals we had overlooked. Problems were sometimes found even without detailed/formal pattern matching.

Beside goal refinement patterns, we also recently started investigating patterns for operationalizing constraints into specifications of actions. Some experience with a real telecommunication project allowed us to exhibit a few such patterns.

The refinement patterns discussed in this paper are domain-independent. To complement them, domain-dependent patterns and frameworks should also be considered. Some initial work on this is reported in [Mas96].

Our approach should be supported by a tool for browsing the library, updating the AND/OR refinement graph, and retrieving appropriate patterns through syntactic matching and tactics. We gave some hints in the paper on how such a tool would work given the way our current KAOS environment is architectured. Our plan is to integrate such a tool on top of the abstract syntax tree engine in a near future.

Further work is also needed to "refine" our patterns to account for the real-time temporal constructs we are using [Koy92]. On the other hand, we would also like to provide a "light" version of our library for industrial partners who want to take benefit of the approach without having to become acquainted with temporal logic. A light version would provide textual but still structured patterns in which all temporal formulas would be hidden.

Acknowledgement. The work reported herein was partially supported by ESPRIT Project Nr. 2537 (ICARUS) and the "Communauté Française de Belgique" (FRISCO project, Actions de Recherche Concertées Nr. 95/00-187 - Direction générale de la

Recherche). Thanks to Martin Feather and Steve Fickas for encouragements and critical remarks, and to the FSE reviewers for helpful comments and suggestions.

References

[And89] J.S. Anderson and S. Fickas, "A Proposed Perspective Shift: Viewing Specification Design as a Planning Problem", *Proc. IWSSD-5 - 5th Intl. Workshop on Software Specification and Design*, IEEE, 1989, 177-184.

[Ant94] Anton, A. I., McCracken, W. M., Potts, C., "Goal Decomposition and Scenario analysis in Business Process Engineering", *CAiSE'94*, LNCS 811, Springer-Verlag, pp. 94-104.

[Ast86] Astesiano, E., Wirsing, M., "An introduction to ASL", *Proc. IFIP WG2.1 Conf. on Program Specifications and Transformations*, North-Holland, 1986.

[Bal82] R.M. Balzer, N.M. Goldman, and D.S. Wile, "Operational Specification as the Basis for Rapid Prototyping", *ACM SIGSOFT Softw. Eng. Notes* Vol. 7 No. 5, Dec. 1982, 3-16.

[BGM91] G. Bernot, M.C. Gaudel, ad B. Marre, "Software Testing Based on Formal Specifications: A Theory and a Tool", *Software Engineering Journal*, 1991.

[Boe95] Boehm, B., Bose, P., Horowitz, E., Ming June Lee, "Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach", *Proc. ICSE-17 - 17th Intl. Conf. on Software Engineering*, Seattle, 1995, pp. 243-253.

[Bra85] R.J. Brachman and H.J. Levesque (eds.), *Readings in Knowledge Representation*, Morgan Kaufmann, 1985.

[Dar91] Dardenne, A., Fickas, S., van Lamsweerde, A., "Goal-Directed Concept Acquisition in Requirements Elicitation", *Proc. IWSSD-6 - 6th Intl. Workshop on Software Specification and Design*, Como, 1991, 14-21.

[Dar93] Dardenne, A., van Lamsweerde, A., Fickas, S., "Goal-Directed Requirements Acquisition", *Science of Computer Programming*, Vol. 20, 1993, 3-50.

[Dar95] Darimont, R., *"Process Support for Requirements Elaboration"*, PhD Thesis, Université catholique de Louvain, Dépt. Ingénierie Informatique, Louvain-la-Neuve, Belgium, 1995.

[Doug94] J. Douglas and R.A. Kemmerer, "Aslantest: A Symbolic Execution Tool for Testing ASLAN Formal Specifications", *Proc. ISTSTA '94 - Intl. Symp. on Software Testing and Analysis*, ACM Softw. Eng. Notes, 1994, 15-27.

[Dub91] Dubois, E., Hagelstein, J., Rifaut, A., "A Formal Language for the Requirements Engineering of Computer Systems", in *Introducing a Logic Based Approach to Artificial Intelligence*, A. Thayse (Ed.), Vol. 3, Wiley, 1991, 357-433.

[Fea87] M. Feather, "Language Support for the Specification and Development of Composite Systems", *ACM Trans. on Programming Languages and Systems* 9(2), Apr. 87, 198-234.

[Fea94] M. Feather, "Towards a Derivational Style of Distributed System Design", *Automated Software Engineering* 1(1), 31-60.

[Fic92] S. Fickas and R. Helm, "Knowledge Representation and Reasoning in the Design of Composite Systems", *IEEE Trans. on Software Engineering*, June 1992, 470-482.

[Gam95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns — Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Gau92] Gaudel, M.-C., "Structuring and Modularizing Algebraic Specifications: the PLUSS specification language, evolutions and perspectives", *Proc. STAS'92*, LNCS 557, 1992, 3-18.

[Geo95] C. George, A.E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J.S. Pedersen, *The RAISE Development Method*. Prentice Hall, 1995.

[Gut93] J.V. Guttag and J.J. Horning, *LARCH: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[Har87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, vol. 8, 1987, 231-274.

[Hek88] S. Hekmatpour and D. Ince, *Software Prototyping, Formal Methods, and VDM*. Addison-Wesley, 1988.

[Jac93] M. Jackson and P. Zave, "Domain Descriptions", *Proc. RE'93 - 1st Intl. IEEE Symp. on Requirements Engineering*, Jan. 1993, 56-64.

[Jac96] D. Jackson and C.A. Damon, "Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector", *Proc. ISTA '96 - Intl. Symp. on Software Testing and Analysis*, ACM Softw. Eng. Notes Vol. 21 No. 3, 1996, 239-249.

[Jon90] Jones, C.B., *Systematic Software using VDM*, 2nd ed., Prentice Hall, 1990.

[Koy92] Koymans, R., *Specifying message passing and time-critical systems with temporal logic,* LNCS 651, Springer-Verlag, 1992.

[Lam95] van Lamsweerde, A., Darimont, R., Massonet, P., "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learned", *Proc. RE'95 - 2nd Int. Symp. on Requirements Engineering*, York, IEEE, 1995.

[Lan95] Lano, K., *Formal Object-Oriented Development*, Springer-Verlag, 1995.

[Man92] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems,* Springer-Verlag, 1992.

[Mas96] P. Massonet and A. van Lamsweerde, "Analogical Reuse of Requirements Frameworks", to appear in Proc. RE-97 - *3rd Int. Symp. on Requirements Engineering*, 1997.

[Mor92] A. Morzenti, D. Mandrioli, and C. Ghezzi, "A Model Parametric Real-Time Logic", *ACM Transactions on Programming Languages and Systems*, Vol. 14 No. 4, October 1992, 521-573.

[Myl92] Mylopoulos, J., Chung, L., Nixon, B., "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach", *IEEE Trans. on Sofware. Engineering*, Vol. 18 No. 6, June 1992, pp. 483-497.

[Nil71] N.J. Nilsson, *Problem Solving Methods in Artificial Intelligence*. McGraw Hill, 1971.

[Nix93] B. A. Nixon, "Dealing with Performance Requirements During the Development of Information Systems", *Proc. RE'93 - 1st Intl. IEEE Symp. on Requirements Engineering*, Jan. 1993, 42-49.

[Pot91] B. Potter, J. Sinclair and D. Till, *An Introduction to Formal Specification and Z.* Prentice Hall, 1991.

[Rep89] Reps, T. and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, 1989.

[Reu91] H.B. Reubenstein and R.C. Waters, "The Requirements Apprentice: Automated Assistance for Requirements Acquisition", *IEEE Transactions on Software Engineering*, Vol. 17 No. 3, March 1991, 226-240.

[Rob89] Robinson, W.N., "Integrating Multiple Specifications Using Domain Goals", *Proc. IWSSD-5 - 5th Intl. Workshop on Software Specification and Design*, IEEE, 1989, 219-225.

[Sch93] A.J. van Schouwen, D.L. Parnas, and J. Madey, "Documentation of Requirements for Computer Systems", *Proc. RE'93 - 1st Intl Symp. on Requirements Engineering*, IEEE, 1993, 198-207.

[Zar95] A.M. Zaremski and J. Wing, "Signature Matching: A Tool for Using Software Libraries", *ACM Trans. on Software Engineering and Methodology* 4(2), April 1995, 146-170.

[Zav82] P. Zave, "An Operational Approach to Requirements Specification for Embedded Systems", *IEEE Transactions on Software Engineering*, vol. 8 no. 3, May 1982, 250-269.

[Zave 94] Zave, P., "Classification of Research Efforts in Requirements Engineering", *Proc. RE'95 - 2nd IEEE Int. Symposium on Requirements Engineering*, March 1995, 214-216.