Goal-directed requirements acquisition

Anne Dardenne

Institut d'Informatique, Facultés Universitaires de Namur, B-5000 Namur, Belgium

Axel van Lamsweerde

Unité d'Informatique, Université Catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium

Stephen Fickas

Department of Computer Science, University of Oregon, Eugene, OR 97403, USA

Abstract

Dardenne, A., A van Lamsweerde and S. Fickas, Goal-directed requirements acquisition, Science of Computer Programming 20 (1993) 3-50.

Requirements analysis includes a preliminary acquisition step where a global model for the specification of the system and its environment is elaborated. This model, called requirements model, involves concepts that are currently not supported by existing formal specification languages, such as goals to be achieved, agents to be assigned, alternatives to be negotiated, etc. The paper presents an approach to requirements acquisition which is driven by such higher-level concepts. Requirements models are acquired as instances of a conceptual meta-model. The latter can be represented as a graph where each node captures an abstraction such as, e.g., goal, action, agent, entity, or event, and where the edges capture semantic links between such abstractions. Well-formedness properties on nodes and links constrain their instances—that is, elements of requirements models. Requirements acquisition processes then correspond to particular ways of traversing the meta-model graph to acquire appropriate instances of the various nodes and links according to such constraints. Acquisition processes are governed by strategies telling which way to follow systematically in that graph; at each node specific tactics can be used to acquire the corresponding instances. The paper describes a significant portion of the meta-model related to system goals, and one particular acquisition strategy where the meta-model is traversed backwards from such goals. The meta-model and the strategy are illustrated by excerpts of a university library system.

Keywords: Requirements engineering; specification acquisition; nonfunctional requirements; conceptual modeling; domain analysis; meta-level inference; specification reuse.

Correspondence to: A. van Lamsweerde, Unité d'Informatique, Université Catholique de Louvain, Place Sainte Barbe 2, B-1348 Louvain-la-Neuve, Belgium. Telephone: (+32-10) 472529. E-mail: avl@info.ucl.ac.be.

1. Introduction

Requirements analysis is a highly critical step in the software lifecycle. A great variety of problems can arise during this step, e.g., inadequacies, incompleteness, contradictions, ambiguities, noises, forward references, or overspecifications [34, 41, 49]. Such errors and deficiencies can have disastrous effects on the subsequent development steps and on the quality of the resulting software product. Therefore, it is essential that requirements engineering be done with great care and precision. Formal methods, supported by automated tools, enable engineers to capture and specify the software requirements carefully and precisely.

Recently, researchers have devoted considerable effort to the design of formal specification languages. The use of such languages allows the requirements specification to be manipulated formally. The specification can be checked against a set of desired properties, can be used to generate a prototype implementation, and so forth. Specification languages differ mainly by the particular specification paradigm used. For example, Z (Spivey [42]) and VDM (Jones [30]) support *state-based* specifications; INFOLOG (Fiadeiro and Sernadas [16]) and ERAE (Dubois et al. [12]) support *history-based* specifications; STATECHARTS (Harel [28]) supports *transition-based* specifications; languages like LARCH (Guttag and Horning [25]), ASL (Astesiano and Wirsing [2]), and PLUSS (Gaudel [21]) support *algebraic* specifications; PAISLEY (Zave [50]) and GIST (Balzer et al. [3]) support *operational* specifications. In using such languages to formalize the requirements for complex systems, requirements engineers face two difficulties—the limited scope of the languages and the preliminary acquisition of relevant requirements.

The scope problem

Most existing specification languages focus on *functional requirements*—that is, requirements about what the software system is expected to do. Nonfunctional requirements are most often left outside of any kind of formal treatment [35]. Such requirements form an important part of real requirements documents [32]; they refer to operational costs, responsibilities, interaction with the external environment, reliability, integrity, flexibility, and so forth. The limited scope of current formal specification languages results from the restricted set of built-in abstractions in terms of which the requirements must be captured. For example, in state-based languages requirements must be expressed in terms of typed entities and operations on them; in algebraic languages they must be expressed in terms of abstract data types. (Such languages thus appear to be more appropriate in the design phase that follows requirements analysis.) To overcome these limitations language designers have proposed richer constructs, in particular for expressing temporal requirements (e.g., [12, 22, 24, 27]) and for capturing requirements about agents and their behavior in the composite system under consideration [14, 20]. (In the sequel, the term *composite system* will be used to refer to the automated system together with the relevant part of its environment [9, 14].)

The acquisition problem

To formalize the requirements, one must first know what these requirements are. Requirements acquisition and elicitation is not an easy task. Often the clients are unable to formulate all relevant requirements explicitly and precisely, the analysts have limited knowledge about the environment in which the system will be used, different automation alternatives arise so that a most suitable one must be selected, and so forth. Surprisingly, very little attention has been paid so far to the requirements acquisition process. The role of knowledge about the application domain and about similar systems has been recognized [17]. For example, requirements clichés available in domain-specific libraries can be instantiated and/or specialized [38]. Preliminary models for acquisition dialogues to support multiple viewpoints, negotiation, and cooperative elaboration of requirements have also been proposed [19, 40].

In this context, we view requirements analysis as being made of two coordinated tasks, requirements acquisition and formal specification.

- In *requirements acquisition* a preliminary model for the specification of the entire composite system is elaborated and expressed in a "rich" language. This language needs a variety of built-in concepts to structure requirements about the composite system in terms of the kind of abstractions usually found in requirements documents, such as objectives and constraints to be met by the composite system, entities, relationships, events, and actions taking place in it, agents controlling the actions, responsibilities assigned, possible scenarios of system behavior, and so forth. The language should also provide facilities for capturing multiple automation alternatives in a form amenable to evaluation and negotiation between the analysts and the clients. (In the sequel, the term *requirements model* will be used to refer to the preliminary model elaborated during acquisition; the language used to express this model will be called *acquisition language*.) The acquisition language should be formal enough to provide some formal basis for elicitation of requirements; on the other hand its use should not require too much hard coding by the analysts, and the preliminary model being sketched must be made visible to the clients.
- In *formal specification* a specific automation alternative that emerged during acquisition is considered, and the part to be automated in the corresponding composite system is retained; the preliminary specification obtained for the data and operations of that subsystem is refined and made more precise using a formalism suitable for detailed formal proofs and prototype generation.

Requirements acquisition and formal specification are not necessarily sequential tasks; from a process programming perspective, one could see them as coroutines. We justify this decomposition into two tasks by making the following observations.

- Formal specification needs some input to start with.
- The acquisition of knowledge about the composite system involves concepts such as objectives, agents, and responsibilities; such concepts are not found in the final formal specification given to the designers.

- Requirements acquisition relies more on knowledge about the application domain (e.g., library management or aircraft control) whereas formal specification relies more on knowledge about the sophisticated formalism being used (e.g., modularization or import mechanisms).
- The basic processes involved in requirements acquisition and in formal specification are rather different. Requirements acquisition involves learning [44] and negotiation [40], whereas formal specification involves data/operation refinement and structuring, assertion strengthening and weakening, and so forth [11, 15].

The focus of this paper is on the requirements acquisition task. Our aim is to present elements of a general approach to requirements acquisition we have developed in the context of the KAOS project. (KAOS stands for Knowledge Acquisition in autOmated Specification [45].) The driving forces of this approach are the reuse of domain knowledge and the application of machine learning technology [44]. Two learning strategies have been adapted to the context of requirements acquisition: *learning-by-instruction*, where the learner conducts the acquisition process by using meta-knowledge about the kind of knowledge to be acquired [4, 8, 43], and *learning-by-analogy*, where the learner retrieves knowledge about some "similar" system to map it to the system being learned [26].

The overall approach taken in KAOS has three components.

(i) a conceptual model for acquiring and structuring requirements models, with an associated acquisition language,
(ii) a set of strategies for elaborating requirements models in this framework, and
(iii) an automated assistant to provide guidance in the acquisition process according to such strategies.

To introduce the context of this paper, we first outline these three components briefly.

The conceptual model

The conceptual model provides a number of abstractions in terms of which requirements models have to be acquired; it is thus a *meta-model.* It is aimed at being sufficiently rich to allow *both* functional and nonfunctional requirements for *any* kind of composite system to be captured in a precise and natural way. Work on knowledge representation [5] has already been shown to be highly relevant in this context. For example, RML proposes abstractions such as the "entity", "activity", and "assertion" concepts together with the "subclass specialization" link type [24]. It was felt, however, that a richer set of abstractions is needed if one wants to also capture objectives of the system under consideration, constraints that make such objectives operational, agents like human beings or programs that control the system's behavior according to such constraints, events that cause the application of actions on entities, and so forth. Also, other structuring link types are needed beside subclass specialization, like (alternative) refinement links between objectives or between constraints, (alternative) assignment links between agents and constraints, and so forth. The meta-model for requirements acquisition can be represented as a conceptual graph where nodes represent abstractions and edges represent structuring links. (Figure 2 illustrates a portion of this graph.)

Acquisition strategies

An acquisition strategy in this framework defines a well-justified composition of steps for acquiring components of the requirements model as instances of meta-model components. In other words, a strategy corresponds to a specific way of traversing the meta-model graph to acquire instances of its various nodes and links. For example, the meta-model can be traversed backwards from the objectives to be fulfilled by the composite system, or backwards from the agents available in the system and their respective views, or backwards from client-supplied scenarios for combining actions. Each step in a strategy is itself composed from finer steps like, for example, question-answering, input validation against known properties of meta-model components, application of tactics to select among alternatives, deductive inferencing based on property inheritance through specialization links, analogical inferencing based on knowledge about similar systems, or conflict resolution between multiple views of human agents involved.

The acquisition assistant

The acquisition assistant is aimed at providing automated support in following one acquisition strategy or another. It is built around two repositories: a requirements database and a requirements knowledge base. Both are structured according to the meta-model components. The *requirements database* maintains the requirements model built gradually during acquisition; the latter can be analyzed using query facilities similar to those provided by project database systems [47]. The *requirements knowledge base* contains two kinds of knowledge. *Domain-level* knowledge concerns concepts and requirements typically found in the application domain considered. As in [4, 38], this knowledge is organized into specialization hierarchies; requirements fragments for a particular class of systems known to the assistant (e.g., library management, airline reservation, telephone network) are thereby inherited from more general applications (e.g., resource management, transportation, communication) and from more general tasks (e.g., transaction processing, history tracking, device control). Besides, *meta-level* knowledge concerns properties of the abstractions found in the meta-model (e.g., "a constraint that can be temporarily violated needs to be restored by some appropriate action") and ways of conducting specific acquisition strategies. The latter aspect includes tactics that can be used within strategies (e.g., "prefer those alternative refinements of objectives which split responsibility among fewer agents").

In this overall framework, the objective of this paper is to present a significant portion of the KAOS meta-model together with one specific acquisition strategy associated with it. The part of the meta-model considered relates to the refinement of system objectives, their operationalization through constraints, the specification of objects and actions to satisfy such constraints, and the assignment of agents like human beings, devices, or programs to constraints and actions. The acquisition strategy considered in the paper is a learning-by-instruction one. The meta-model graph is traversed backwards from the *Goal* node through adjacent links. Instances of the *Goal* node that are acquired represent objectives to be achieved by the composite system (like, e.g., satisfy as many book requests as possible, provide bibliographical knowledge in relevant domains, or maintain privacy about user interests). At each node on the path prescribed by the strategy the corresponding meta-level knowledge is used for guiding instance acquisition.

A distinguished feature of the approach presented in the paper is the importance given to high-level goals as opposed to their operationalization into constraints to be ensured by agents through appropriate actions. Instead of starting directly from lower-level process- or action-oriented descriptions as is usually done in current requirements engineering methods, the approach starts from system-level and organizational objectives from which such lower-level descriptions are progressively derived.

Goals are important in several respects. They lead to the incorporation of requirements components which should support them. They justify and explain the presence of requirements components which are not necessarily comprehensible to clients. They may be used to assign the respective responsibilities of agents in the system; more precisely, they may provide the basis for defining which agents should best perform which actions to fit prescribed constraints (according to their capabilities, reliability, cost, load, motivation, and so forth). Finally, they provide basic information for detecting and resolving conflicts that arise from multiple viewpoints among human agents [39].

The remainder of the paper is organized as follows. Section 2 describes the various abstractions involved in the portion of the meta-model relevant to the goal-directed acquisition strategy. Requirements fragments from a library system are also provided there to illustrate the use of the acquisition language. The strategy itself and its various associated tactics are then discussed in Section 3. Section 4 then concludes by discussing achievements and open issues.

2. A conceptual meta-model for requirements acquisition

The three levels involved in our approach to requirements acquisition are first clarified in Section 2.1. The central role played by the model at the meta level is then discussed in greater detail in Section 2.2. Section 2.3 introduces some background material used in the sequel to define the meta-model components. The remainder of Section 2 is devoted to a tour through the portion of the meta-model relevant to this paper (Sections 2.4–2.9).

2.1. *The meta, domain, and instance levels*

As shown in Fig. 1, our approach to requirements acquisition involves three levels of modeling. (In the sequel, the "meta" prefix will be used wherever felt necessary to avoid confusions between levels.)

- The *meta level* refers to domain-independent abstractions. This level is made of *meta-concepts* (e.g., "Agent", "Action", "Relationship", etc.), *meta-relationships* linking meta-concepts (e.g., "Performs", "Input", "Link", "IsA" specialization, etc.), *meta-attributes* of meta-concepts or meta-relationships (e.g., "Load" of "Agent", "PostCondition" of "Action", "Cardinality" of "Link", etc.), and *meta-constraints* on meta-concepts and meta-relationships (e.g., "a constraint that can be temporarily violated must have a restoration action meta-linked to it").
- The *domain level* refers to concepts specific to the application domain (e.g., resource management, telephone network, etc.) and to the type of task considered (e.g., transaction processing, history tracking, etc.). This level is made of *concepts* that are instances of meta-level abstractions (e.g., for the library management subdomain, the "Borrower" concept which is an instance of the "Agent" meta-concept, the "CheckOut" concept which is an instance of the

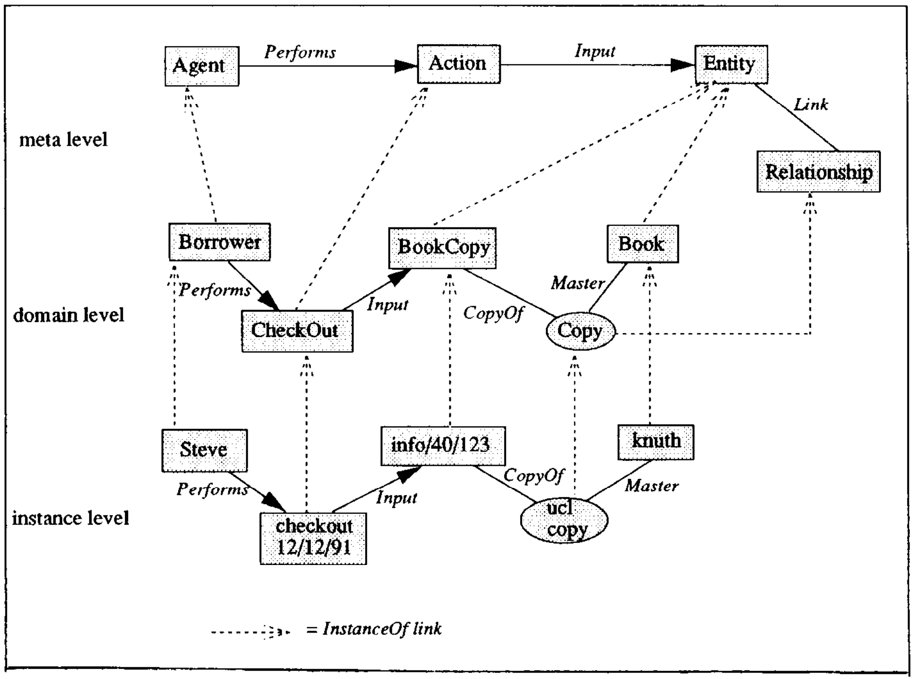

Fig. 1. The meta, domain, and instance levels.

"Action" meta-concept, the "Copy" concept which is an instance of the "Relationship" meta-concept, etc.). Domain-level concepts are linked through *instances* of the meta-relationships linking the corresponding meta-level concepts they are an instance of (e.g., "Borrower" *Performs* "CheckOut", "Copy" *Links* "Book" and "BookCopy"). Domain-level concepts must also satisfy instantiations of the meta-constraints on the corresponding meta-level concepts they are an instance of (e.g., the constraint of "limited borrow time" can be violated and must thus have a restoration action associated with it—such as sending a reminder). The requirements model to be acquired is thus structured from such domain-level concepts according to instances of the corresponding meta-relationships inherited from the meta level. The domain knowledge which can be reused during acquisition is structured in a similar way.
- The *instance level* refers to specific instances of domain-level concepts (see Fig. 1).

A similar distinction between object and meta levels has been used before for requirements modeling, see [23]. The meta, domain, and instance levels are thus made from meta-types, types, and type instances, respectively. The KAOS meta-model is a conceptual model for the meta level, thus consisting of meta-level concepts, relationships, attributes, and constraints.

2.2. Role of the conceptual meta-model

In the context of requirements acquisition, the KAOS meta-model fills several roles.

(i) As seen before, the components of a requirements model are acquired as domain-specific *instances* of meta-concepts, linked by instances of meta-relationships, characterized by instances of meta-attributes, and constrained by instances of meta-constraints.

(ii) As a consequence of (i), the meta-model determines the structure of the acquisition language.

(iii) As another consequence of (i), the components of a requirements model inherit all the features defined once for all for the corresponding meta-level abstractions they are an instance of.

(iv) The meta-model drives the acquisition process as in learning-by-instruction systems. For example, with the goal-directed strategy presented in Section 3, the *Goal* meta-concept is the first to be considered; instances of it are acquired through *Reduction* and *IsA* specialization links (see Fig. 2); the objects *Concerned* by the goals acquired are also preliminarily defined. Then the *Agent* and *Capability* meta-level abstractions are considered to identify relevant instances of them. Next the *Constraint* meta-concept is considered; instances of it are acquired from goals through *Operationalization* links, and so forth. It is important to recognize that *the more domain-independent knowledge is attached to these meta-concepts and meta-relationships under the* *form of meta-attributes and meta-constraints, the more knowledge-based guidance can be provided in the acquisition process.*

(v) The various components of the meta-model yield criteria for measuring conceptual similarity when a learning-by-analogy strategy is followed [10]. Analog requirements fragments are retrieved in the domain knowledge base and mapped to the target requirements from similarities that are evaluated between goals, constraints, actions, agents, events, and the like.

(vi) The meta-model determines the structure of the requirements database where the requirements model is gradually elaborated; similarly, it determines the structure of the domain knowledge base where model fragments can be retrieved during acquisition and reused. As shown in [47], meta-models provide a basis for defining generic environment architectures where tools know nothing about domain-level concepts; they know just about meta-level concepts.

2.3. Characterizing model components

Models at the meta level and at the domain level were already seen to consist of *concepts*, *relationships* linking concepts, and *attributes* attached to concepts or relationships (see Section 2.1). This style of model definition is close to the one used in semantic data models [6, 29] or structured object representations [5]. What is meant by attribute and relationship is now made more precise; next we will see how meta-level and domain-level concepts, relationships, and attributes are characterized.

2.3.1. Terminology

Attributes. An attribute *Att* of a concept or relationship T is defined as a function

$$Att : T \rightarrow D$$

where D is called the *domain of values* for the attribute.

Relationships. An n-ary relationship R over concepts C_1 through C_n is defined by

$$R = \text{TUPLE}\,(C_1, \ldots, C_n)$$

where TUPLE denotes the tuple type constructor (that is, any instance of R is a tuple of corresponding instances of C_k). Sometimes the *role* played by C_k in R is given an explicit name. The *cardinality* of R is defined by a sequence of pairs $\{\min_k : \max_k\}_{1 \leq k \leq n}$, where $\min_k$ and $\max_k$ denote the minimum and maximum number of instances of R, respectively, in which every instance of C_k must participate. Cardinalities allow various kinds of constraints on relationships to be expressed. For example, the cardinality of the *Copy* relationship that links the *BookCopy* and *Book* concepts in Fig. 1 is $\{1{:}1, 1{:}N\}$. This expresses that a book copy must be a copy of one and exactly one book whereas a book may have a number of copies ranging from one to an arbitrary number N. Cardinality constraints for a number of relationships in the KAOS meta-model are depicted in Fig. 2.

AndOr relationships. AND/OR graph structures [37] need to be introduced if one wants to support the tracking of *alternative* requirements options at the domain level. For example, a goal can be refined into several alternative combinations of subgoals, a goal can be made operational through several alternative combinations of constraints, a constraint can be under responsibility of several alternative combinations of agents, and so forth. *AndOr* relationships are introduced at the meta level for that purpose. (Instances of such relationships will be declared in the acquisition language using the corresponding logical connectives.) An *AndOr* relationship R over concepts C_1 and C_2 is a compound binary relationship defined as follows:

$$R = AndR \circ OrR \quad \text{(“}\circ\text{” denotes relation composition)}$$

with

$$OrR = \text{TUPLE}\,(C_1, AltR), \qquad AndR = \text{TUPLE}\,(AltR, C_2),$$

that is, any instance of R is a pair of concept instances (c_1, c_2) such that there exists an alternative alt for which (c_1, alt) and (alt, c_2) are instances of OrR and $AndR$, respectively. ($AltR$ thus represents the set of possible alternatives to link C_1 and C_2—in AND/OR graph terminology, instances of $AltR$ correspond to AND-nodes whereas instances of C_1 and C_2 correspond to OR-nodes.)

AndOr relationships have a *Selected* attribute with "yes" and "no" as possible values to record which alternative is eventually selected during acquisition. These values must be updated in case of backtracking to explore another alternative.

IsA relationship. Subclass specialization is captured through the binary *IsA* relationship over concepts. This relationship is defined by

$$\text{IsA}\,(C_1, C_2) \quad \text{iff every instance of } C_1 \text{ is also an instance of } C_2.$$

As a result, features of C_2 are inherited by C_1 according to the inheritance mode specified; a feature is uninheritable, instance-inheritable, type-inheritable, or fully inheritable (for more details, see [46]). A concept may be linked to several others through *IsA* relationship instances; thus multiple inheritance is supported. As first shown by Greenspan [24], specialization hierarchies are of great benefit in the development of conceptual requirements models.

2.3.2. Defining concepts, relationships, and attributes

At the meta and domain levels, model components are characterized as follows.

A *concept C* is defined by a set of features; a *concept feature* is either an attribute of C or a relationship involving C. For example, the "Agent" meta-level concept in Fig. 1 could have a "Load" attribute and is involved in the "Performs" relationship; the "Book" domain-level concept could have a"Title" attribute and is involved in the "Copy" relationship.

A *relationship R* is also defined by a set of features; a *relationship feature* is either an attribute of R or the ordered list of concepts linked by R together with their respective role and cardinality. For example, the "Performs" meta-level relationship In Fig. 1 could have a "Reliability" attribute and links the "Agent" and "Action" meta-level concepts; the "Borrowing" domain-level relationship could have a "DateOfCheckOut" attribute and links the "Borrower" and "BookCopy" domain-level concepts.

An *attribute Att* is defined by a set of characteristics like its name, informal definition, domain of values, and unit of values. A domain is either elementary or compound. An *elementary* domain is a set of atomic values; this set can be simple, linearly ordered, or *IsA*-structured (e.g., the "Keyword" domain is *IsA*-structured). A *compound* domain is built from other domains through the *Union*, *Tuple*, *SetOf*, or *SequenceOf* domain constructors, or through abstract syntax domain constructors [31]. The latter are used for those attributes attached to concepts or relationships which have *formal assertions* as values.

Formal assertions as attribute values. At the domain level, formal assertions can thus be attached to domain-specific concepts; they are values for attributes inherited from the meta level, like the *Invariant* attribute that can be attached to objects, the *PreCondition*, *PostCondition*, *TriggerCondition*, and *StopCondition* attributes that can be attached to actions, or the *FormalDef* attribute attached to goals or constraints (see Sections 2.4, 2.5, and 2.8). The assertion sublanguage used for writing such attribute values is a typed temporal first-order logic equipped with real-time temporal constructs [33]; it is inspired by ERAE [12]. The primary notations used in the examples below are summarized as follows. For a well-formed formula P,

P means "property P holds in the *current* state",
$\bigcirc P$ means "property P holds in the *next* state",
$\Diamond P$ means "property P holds in *current or some future* state",
$\Box P$ means "property P holds in *current and all future* states",
$\bullet P$ means "property P holds in the *previous* state",
$\blacklozenge P$ means "property P holds in *current or some previous* state",
$\blacksquare P$ means "property P holds in *current and all previous* states".

Inheritance mode. An additional characteristic that may be attached to a relationship or an attribute is its inheritance mode through *IsA* specialization hierarchies (with "uninheritable", "instance-inheritable", "type-inheritable", or "fully inheritable" as possible values, see above). All features of meta-concepts and meta-relationships are fully inheritable by each corresponding instance at the domain level. Similarly, constraints on model components at the meta level are correspondingly instantiated at the domain level. For example, consider the following meta-constraint.

A constraint that may be temporarily violated must have a restoration action meta-linked to it.

A possible instantiation of it at the domain level might be

The RectifyLibraryDatabase action must be introduced with appropriate conditions to restore the constraint of consistency between the library database and the library shelves.

Figure 2 summarizes the portion of the meta-model of relevance to this paper; a detailed description of the complete meta-model can be found in [46]. We proceed now from the more-or-less classical concepts, which already appear in some form in existing specification languages, to the new concepts which have been introduced in KAOS. (The order of presentation thus does not correspond to the order in which instances must be acquired at the domain level; the latter issue will be addressed in Section 3.) In the sequel, where no ambiguity arises we will say "a *C*" instead of "an instance of meta-concept *C*".

2.4. Objects, entities, relationships, and events

An *object* is a thing of interest which can be referenced in requirements. Instances of objects may evolve from state to state (because of applications of actions, see Section 2.5). The *state* of an object instance *Ob* at some time is defined as a mapping from *Ob* to the set of values at that time of all features of *Ob*.

In addition to *Name* and *InformalDefinition*, the primary meta-attributes of the OBJECT meta-concept include:

- *Exists*, with values **true** at the instance level if the corresponding object instance exists in the current state, and **false** otherwise;

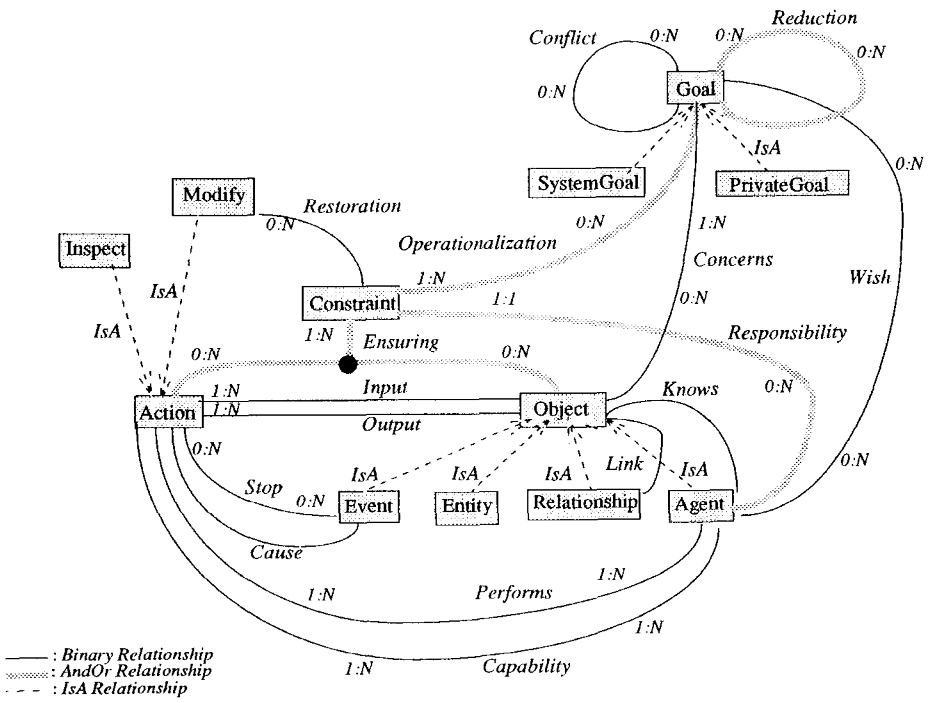

Fig. 2. A portion of the KAOS conceptual meta-model.

- *Invariant*, whose values at the domain level are assertions that restrict the class of possible states for the corresponding object. An invariant is implicitly universally quantified over the object states.

As seen in Fig. 2, the OBJECT meta-concept is involved in a number of meta-relationships. These will be defined later when the other meta-concepts involved in them will be defined. In addition to those inherited from the meta level, new domain-specific attributes can be attached to objects at the domain level (e.g., a "Title" attribute for the "Book" concept).

The ENTITY, RELATIONSHIP, EVENT, and AGENT concepts are specializations of the OBJECT meta-concept and inherit all its features.

An *entity* is an autonomous object; its instances may exist independently from other object instances. Examples of entities are "Borrower", "Book", "BookCopy", "Library", and so forth. In the acquisition language, one might write

```
Entity Library
  Has available, checkedOut, lost: SetOf [BookCopy]
      coverageArea:  SetOf [Subject]
    % declaration of domain-specific attributes %
  Invariant (∀lib: Library)
                 (lib = lib.available ∪ lib.checkedOut ∪ lib.lost)  ∧
                 (lib.available ∩ lib.checkedOut ≠ ∅  ∧
                  lib.available ∩ lib.lost = ∅  ∧
                  lib.checkedOut ∩ lib.lost = ∅)

  . . .
end Library
```

The acquisition language can be seen to have a two-level structure: an outer level for *declaring* domain-level concepts in terms of meta-model components, and an inner level for expressing *assertions* as values for some meta-attributes. The outer declaration level has an entity-relationship structure which yields the structure of the requirements database; static semantics checking can thereby be performed through entity-relationship queries [47]. The inner assertion level corresponds to typed temporal first-order logic.

A *relationship* is a subordinate object; the existence of its instances depends upon the existence of the corresponding object instances linked by the relationship. If OB_1 through OB_n denote the linked objects, its structure is

$$\text{TUPLE}\,(OB_1, \ldots, OB_n).$$

The *Exists* meta-attribute inherited from the OBJECT meta-concept is renamed; in the acquisition language, the expression

$$R(ob_1, \ldots, ob_n)$$

is used rather than

$$[R(ob_1, \ldots, ob_n)].\text{Exists} = \textbf{true}.$$

Since a relationship links objects, the RELATIONSHIP meta-concept has a specific meta-relationship with the OBJECT meta-concept (see Figs. 1 and 2): the *Link* meta-relationship. In this meta-relationship, the meta-role "Links" is played by RELATIONSHIP with meta-cardinality $1{:}N$ (that is, relationships are n-ary with $n \geqslant 1$); the meta-role "LinkedBy" is played by OBJECT with meta-cardinality $0{:}N$ (that is, an object does not necessarily participate in relationships). The *Link* meta-relationship has two meta-attributes: *Role* and *Cardinality*. Values for these meta-attributes yield roles and cardinalities for relationships at the domain level. (The reader now may understand why the "meta" prefix is used: meta-roles and meta-cardinalities of a meta-relationship should not be confused with the *Role* and *Cardinality* meta-attributes of the *Link* meta-relationship.) For example, the *Borrowing* relationship might be partially described in the acquisition language as follows.

```
Relationship Borrowing
  Links Borrower {Role Borrows, Card 0:N}
        BookCopy {Role BorrowedBy, Card 0:1}
      % Borrowers may have no copy borrowed, and may borrow several copies at same
        time; copy may be not borrowed, and may be borrowed by at most one borrower %
  Invariant (∀lib: Library, bor: Borrower, bc: BookCopy)
            [Borrowing (bor, bc) ∧ bc∈lib ⇒
            bc∈lib.checkedOut ∧ ◆Requesting (bor, bc)]
            (∀lib: Library, bc: BookCopy)
              [bc∈lib.checkedOut ⇒
              (∃bor: Borrower) Borrowing (bor, bc)]
  ...
end Borrowing
```

An *event* is an instantaneous object; that is, its instances exist at the instance level in one state only. Again, the *Exists* meta-attribute inherited from the OBJECT meta-concept is renamed; in the acquisition language, the expression "Occurs (ev)" is used instead of "ev.Exists = **true**".

For example, the event of a reminder being sent for a borrower to return a book copy might be expressed as follows.

```
Event ReminderIssued
  Has ToWhom: Borrower, What: BookCopy, Message: TEXT
  Invariant (∀rs: ReminderIssued)
              (Occurs (rs) ⇔
              (∃p: Staff) Performs (p, IssueReminder))
  ...
end ReminderIssued
```

(We will come back to this example later when the *IssueReminder* action and *Performs* meta-relationship will be introduced.)

The EVENT meta-concept has specific meta-attributes in addition to those inherited from OBJECT. Among them, the *Frequency* meta-attribute can be used to express the time interval between repeating instances of the event. Specific meta-relationships are introduced to model the fact that events may cause the application of actions or are produced by such applications, as shown below.

2.5. Actions

An *action* is a mathematical relation over objects. (If the action is deterministic, this relation reduces to a function.) Action applications define state transitions.

In addition to *Name* and *InformalDefinition*, the primary meta-attributes of the ACTION meta-concept include

- *PreCondition*, whose values at the domain level are the weakest *necessary* conditions on initial states for application of the corresponding action;
- *TriggerCondition*, whose values at the domain level are the weakest *sufficient* conditions on initial states for application of the corresponding action;
- *PostCondition*, whose values at the domain level are the strongest conditions on final states that describe the net effect of applying the corresponding action.

The pair (*PreCondition*, *PostCondition*) captures the *state transition* produced by application of the action. This pair often has a pattern $(P \wedge \cdots, \neg P \wedge \cdots)$ or $(\neg P \wedge \cdots, P \wedge \cdots)$.

Note the difference between a precondition and a trigger condition. An action can only be applied if its precondition holds whereas it must be applied if its trigger condition becomes true. A *meta-constraint* here is that the action's precondition must be logically implied by the trigger condition taken in conjunction with the invariants on the objects referred to in the precondition.

Other attributes, such as *StopCondition* or *Duration*, can be attached to ACTIONs. See [46] for more detail.

The ACTION meta-concept is linked to OBJECT through the Input/Output meta-relationships and to EVENT through the Cause/Stop meta-relationships. They are defined as follows. Let *act*, *ob*, and *ev* denote instances of the ACTION, OBJECT, and EVENT meta-concepts, respectively.

Input (*act*, *ob*) iff *ob* is among the types making up the domain of *act*.
Ouput (*act*, *ob*)
iff *ob* is among the types making up the codomain of *act*.
Cause (*ev*, *act*)
iff *ev* instances are among those causing applications of *act*.
Stop (*ev*, *act*)
iff *ev* instances are among those causing abortions of *act*.

Two specializations of ACTION are distinguished in the meta-model, viz. INSPECT and MODIFY actions. For a modification action *act*, Output (*act*, *ob*) means that instances of object *ob* are created, are deleted, or have their features updated. The Input and Output meta-relationships have optional *Argument* and *Result* meta-attributes, respectively, to declare instance variables referenced in the assertions attached to the corresponding action.

The examples below suggest how these meta-relationships and meta-attributes are reflected in the acquisition language.

```
Action CheckOut
  Input BookCopy {Arg: bc},
        Library {Arg: lib}, Borrower {Arg: bor}
  Output Library {Res: lib}, Borrowing
  PreCondition $bc \in lib.available$
  PostCondition $\neg(bc \in lib.available) \wedge bc \in lib.checkedOut \wedge$
                Borrowing (bor, bc)
Action IssueReminder
  Input Borrower {Arg: bor}, BookCopy {Arg: bc}
  Output Reminder
  TriggerCondition
    $\blacksquare_{>2w}$ Borrowing (bor, bc) $\wedge$
    $\neg \blacklozenge_{\leq 1w} (\exists r$: ReminderIssued) [Occurs (r) $\wedge$ r = (bor, bc, -)]
PostCondition . . .
```

The trigger condition above states the condition and frequency under which reminders must be produced. (The *ReminderIssued* event was introduced before.) This example also shows the use of real-time temporal constructs [12, 33]; e.g., "$\blacksquare_{>2w}$" means "in every past state from the current one up to more than 2 weeks".

2.6. Agents

An *agent* is an object which is a processor for some actions; agents thus control state transitions. As opposed to the other kinds of objects (i.e., entities, relationships, and events), agents have choice on their behavior [14]. Examples of agents are human beings, physical devices, or programs that exist or are to be developed in the automated part of the composite system.

Agents have states like any other kind of object. They inherit all features that may characterize objects. In particular, they can be structured from other agents through the tuple type constructor; agent refinement into finer ones can thereby be supported.

In addition to the features inherited from the OBJECT meta-concept, the AGENT meta-concept has a *Load* meta-attribute, whose values denote occupation rates of the corresponding agents. The load of an agent will increase progressively during requirements acquisition as responsibility assignments are made. Initially it might be non-null if the agent already has assignments in another composite system. The process according to which loads are evaluated is outside the current scope of our approach. The *Load* attribute acts as a placeholder where values resulting from cost analysis can be integrated; such values are used in the tactics for responsibility assignment, see Section 3.

As shown in Fig. 2, the AGENT meta-concept has two meta-relationships with ACTION: *Capability* and *Performs*. They are defined as follows.

Capability (*ag*, *act*) iff agent *ag* is capable of performing action *act*.

Performs (*ag*, *act*) iff agent *ag* is a processor allocated to *act*.

An agent is thereby partly defined by two sets of actions: the set of actions it can perform and the set of actions it must perform after assignment decisions have been made (see Section 2.8 and Section 3). An obvious meta-constraint here is that

$$(\forall ag\colon \text{AGENT}, act\colon \text{ACTION})$$
$$(\text{Performs}\ (ag, act) \Rightarrow \text{Capability}\ (ag, act)).$$

The *Performs* relationship is more precisely captured through the following meta-level *Performance Axiom*:

$$(\forall ag\colon \text{AGENT}, act\colon \text{ACTION})$$
$$(\text{Performs}\ (ag, act) \equiv \{?\text{Pre}, !\text{Trig}\}\ [ag, act]\ \{\text{Post}\}).$$

The right-hand side of this equivalence has the following meaning:

if agent *ag* actually performs action *act*, it must guarantee to start *if* the trigger condition of *act* becomes true and *only if* the precondition of *act* is true, to yield a state satisfying the postcondition of *act*.

In terms of the FOREST notations [20], the Performance Axiom can be stated as

$$(\forall ag\colon \text{AGENT}, act\colon \text{ACTION})$$
$$(\text{Performs}\ (ag, act) \equiv$$
$$(\text{Pre} \rightarrow [ag, act]\ \text{Post})\ \&\ (\text{Trig}\ \&\ \text{Pre} \rightarrow \text{obl}\ (ag, act))).$$

A last meta-relationship which will be used below is the *Knows* meta-relationship between the AGENT and OBJECT meta-concepts. This meta-relationship is defined by

Knows (*ag*, *ob*)
iff the states of object *ob* are made observable to agent *ag*.

Note that Performs (*ag*, *act*) means that *ag* can actually control the state transitions associated with *act* whereas Knows (*ag*, *ob*) means that *ag* can actually observe *ob* states. Also note that *ob* can be an object of any kind. In particular, the state of an agent can thereby be made observable to other agents.

Often the interface through which an agent can observe object states should be made precise in the requirements [9]. An *Interface* meta-attribute attached to the *Knows* meta-relationship is introduced for that purpose; its values are references to other objects—and, in particular, to other agents.

To illustrate these various notions, one could get the following description at some stage during acquisition:

```
Agent Staff
  Has competenceArea,...
    % declaration of domain-specific attributes %
  Invariant (∀st: Staff) (InstanceOf (st, ResearchStaff) ∨
                         InstanceOf (st, SecretaryStaff))
Load...
CapableOf AddCopy, RemoveCopy, BiblioQuery,
          CheckOut, Return, IssueReminder,...
Performs AddCopy, RemoveCopy,...
Knows Borrowing {Interface: BorrowingSheet},...
```

2.7. *Goals*

A *goal* is a nonoperational objective to be achieved by the composite system. *Nonoperational* means that the objective is not formulated in terms of objects and actions available to some agent in the system; in other words, a goal as it is formulated cannot be established through appropriate state transitions under control of one of the agents.

For example, a standard objective for a library system would be to have any book request eventually satisfied; requests should be made by registered borrowers and refer to books relevant to the subject area covered by the library. This objective might be captured by the following requirements fragment:

```
SystemGoal Achieve [BookRequestSatisfied]
  InstanceOf SatisfactionGoal
    % declaration of goal category %
  Concerns Borrower, Book, Borrowing,...
  FormalDef
    (∀bor: Borrower, b: Book, lib: Library)
      Requesting (bor, b) ∧ b.subject ∈ lib.coverageArea ⇒
      ◇(∃bc: BookCopy) (Copy(bc, b) ∧ Borrowing (bor, bc))
```

This objective is nonoperational in that it cannot be achieved by application of actions available to some agent. For example, a *Borrower* agent cannot establish the objective by application of the *MakeBookRequest*, *CheckOut*, and *Return* actions it is capable of; the *CheckOut* action can make the predicate Borrowing (*bor, bc*) become true, but the precondition $bc \in$ lib.available for that action cannot be made true by application of actions available to that agent. (As it will be seen in Section 2.8, such a goal needs to be "implemented" by operational constraints; the latter can be established through state transitions under control of some agents.)

As seen in Fig. 2, the *Concerns* meta-relationship links the GOAL and OBJECT meta-concepts. Explicit links can thereby be established at the domain level between a goal and the objects the goal refers to; these links are used during acquisition to get object descriptions from goal descriptions. (See Section 3.) For example, the formulation of the *BookRequestSatisfied* goal above requires the introduction and later description of the *Borrower* agent and the *Borrowing* and *Requesting* relationships.

A *goal taxonomy* is defined at the meta level to support guidance during acquisition, reuse of goal descriptions, and formal checks. Goals are classified according to their pattern and their category.

The *pattern* of a goal is based on the pattern of its formal definition. Five patterns can be identified:

Achieve:	pattern $P \Rightarrow \Diamond Q$,
Cease:	pattern $P \Rightarrow \Diamond \neg Q$,
Maintain:	pattern $P \Rightarrow \Box Q$,
Avoid:	pattern $P \Rightarrow \Box \neg Q$,
Optimize:	pattern Maximize (objective function) or Minimize (objective function).

These patterns have an impact on the set of possible behaviors of the system; *Achieve* and *Cease* goals generate behaviors, *Maintain* and *Avoid* goals restrict behaviors, and *Optimize* goals compare behaviors [1]. Goal patterns are declared by making the pattern name precede the goal name, see the example above.

Goals found in requirements documents can be of different *categories*. At the meta level, such categories are organized into a specialization hierarchy. (Only the top level of this taxonomy is shown in Fig. 2.) A first distinction is made between *SystemGoals* and *PrivateGoals*. *SystemGoals* are application-specific goals that *must* be achieved by the composite system; e.g., the *BookRequestSatisfied* goal above is declared as a *SystemGoal*. *PrivateGoals* are agent-specific goals that *might* be achieved by the composite system; e.g., the goal of keeping borrowed copies as long as needed would be private to the *Borrower* agent. (We will come back to private goals below.) System goals are specialized into several categories. The categories considered so far include:

- *SatisfactionGoals* concerned with satisfying agent requests;
- *InformationGoals* concerned with getting agents informed about object states;
- *RobustnessGoals* concerned with recovering from foibles of human agents or from breakdowns of automated agents;
- *ConsistencyGoals* concerned with maintaining the consistency between the automated and physical parts of the composite system;
- *SafetyGoals* and *PrivacyGoals* concerned with maintaining agents in states which are safe and observable under restricted conditions, respectively.

The specific category of a domain-level goal is declared by an *InstanceOf* clause which makes the goal inherit all features of the corresponding goal category.

As another example, the objective of safe transportation in a lift system might be captured by the following requirements fragment:

> **SystemGoal** Maintain [SafeTransportation]
> **InstanceOf** SafetyGoal
> **Concerns** Passenger
> **InformalDef** . . .

Note again that this objective is nonoperational because as formulated it cannot be established by the *MakeRequest*, *GetIn*, and *GetOut* actions available to the *Passenger* agent nor by the *OpenDoors*, *CloseDoors*, *Stop*, and *GoToFloor* actions available to the *LiftController* agents. One difference with the previous example is that the *SafeTransportation* goal is even more abstract than the *BookRequestSatisfied* goal; it cannot be directly formalized. Goals can thus be described formally or informally. Abstract/informal goals need to be refined into concrete/formal ones; the latter may also need to be further refined in order to obtain subgoals that can be more easily made operational through constraints. (The notions of formality and operationality should not be confused.)

The *Reduction* meta-relationship is thus introduced for goal refinement. Since a goal can be reduced into several alternative combinations of subgoals, *Reduction* is an AndOr meta-relationship; it corresponds to the classical problem reduction operator in problem solving [37]. The precise definition is as follows.

> Reduction (G, g)
> iff achieving goal g possibly with other subgoals is
> among the alternative ways of achieving goal G.

At the domain level, goals are thus structured as AND/OR graphs (a goal node can have several parent nodes as it can occur in several reductions, see the corresponding cardinality in Fig. 2).

Let us come back to the *BookRequestSatisfied* goal above. The predicate Borrowing (bor, bc) will eventually become true from states where Requesting (bor, b) holds provided the precondition $bc \in$ lib.available of the *CheckOut* action becomes true sooner or later. This could be achieved *either* by (i) having a reasonable amount of relevant book copies in the library, (ii) guaranteeing the regular availability of such book copies, and (iii) notifying borrowers in case of requested book copies being returned, *or* by guaranteeing that a copy of any book is available for any borrower at any time. (The latter alternative would probably be later rejected due to the cost of ensuring the constraint operationalizing it, see below.) These possible refinements would be captured during acquisition as follows.

> **SystemGoal** Achieve [BookRequestSatisfied]
> **InstanceOf** . . .
> **Concerns** . . .
> **FormalDef** . . .
> **ReducedTo** EnoughCopies, RegularAvailability, AvailabilityNotified
> **or ReducedTo** AsManyCopiesAsNeeded

Systemgoal Maintain [RegularAvailability]
Concerns Library
FormalDef
$(\forall \text{lib: Library, bc: BookCopy})$
$\text{bc} \in \text{lib} \Rightarrow$
$\Box[\neg(\text{bc} \in \text{lib.available}) \Rightarrow \Diamond_{\leq Nw}(\text{bc} \in \text{lib.available})]$
% *N* is a system parameter %

. . .

Systemgoal Achieve [AvailabilityNotified]
InstanceOf InformationGoal
Concerns Borrower, Library, . . .
FormalDef
$(\forall \text{lib: Library, bor: Borrower, b: Book, bc: BookCopy})$
$\text{Requesting (bor, b)} \wedge$
$\bullet\neg(\exists \text{bc: BookCopy}) (\text{Copy (bc, b)} \wedge \text{bc} \in \text{lib.available}) \wedge$
$(\exists \text{bc: BookCopy}) (\text{Copy (bc, b)} \wedge \text{bc} \in \text{lib.available}) \Rightarrow$
$\Diamond \text{Knows (bor, lib.available)}$
% The *Knows* predicate was introduced in Sec. 2.6 %

. . .

The *Reduction* meta-relationship allows one to capture goals that contribute positively to other goals. Goals can also contribute *negatively* to other ones; this is captured in the *Conflict* relationship. The latter is defined as follows.

Conflict (*g1, g2*) iff goals *g1, g2* cannot be achieved together.

Suppose, for example, that the following private goal has been acquired from borrowers:

PrivateGoal Maintain [LongBorrowingPeriod]
InstanceOf SatisfactionGoal
Concerns Borrower, Borrowing
FormalDef
$(\forall \text{bor: Borrower, b: Book, bc: BookCopy})$
$\Box[\text{Borrowing (bor, bc)} \wedge \text{Copy (bc, b)} \wedge \bigcirc\text{Need (bor, b)} \Rightarrow$
$\bigcirc\text{Borrowing (bor, bc)}]$

This goal can clearly be seen to conflict with the *RegularAvailability* goal above (see the formal definitions of these goals and the invariants attached to the *Library* entity and the *Borrowing* relationship). The description of *LongBorrowingPeriod* is therefore complemented with

ConflictsWith RegularAvailability

Thus, conflicts between goals can be made explicit. Recording such conflicts is required to support subsequent conflict resolution through evaluation and negotiation. In that prospect, a *Priority* meta-attribute is also attached to the GOAL meta-concept. Its values can be used for conflict resolution and for agent responsibility assignment. (Priorities takes values in the range 0 to 1; the latter value being the highest priority.) A meta-constraint here is that *SafetyGoals* always are of highest priority.

Finally, a *Wish* meta-relationship is introduced between the HumanAGENT specialization of the AGENT meta-concept (not shown in Fig. 2) and the GOAL meta-concept; it is defined by

Wish (*ag*, *G*) iff human agent *ag* wants goal *G* to be achieved.

For example, the *LongBorrowingPeriod* goal above is clearly *Wished* by *Borrower* agents; the system goal *Avoid* [*LostCopies*] is *Wished* by *Staff* agents who do not want copies to disappear improperly.

The introduction of private goals and *Wish* links provides useful information at the domain level for making decisions among alternative responsibility assignments and conflict resolutions. (See Section 3.) For example, private goals have low *Priority* values and therefore will often be dropped in case of conflict; if an agent wishes some system goal, the constraint operationalizing that goal will be assigned preferably to that agent—e.g., staff agents will be in charge of the constraint operationalizing *Avoid* [*LostCopies*]; an agent is not assigned a constraint operationalizing a goal that conflicts with its private goals—e.g., the *Borrower* agent would not be in charge of the *LimitedBorrowingAmount* constraint that operationalizes the *EnoughCopies* goal above.

2.8. Constraints

A *constraint* is an *operational* objective to be achieved by the composite system. As opposed to goals, a constraint is formulated in terms of objects and actions available to some agent in the system; that is, it can be established through appropriate state transitions under control of one of the agents.

For example, the *LimitedBorrowingPeriod* constraint might be captured by the following requirements fragment.

SoftConstraint Maintain [LimitedBorrowingPeriod]
FormalDef
(∀bor: Borrower, bc: BookCopy)
$\Box$[Borrowing (bor, bc) $\wedge$ $\bullet\neg$Borrowing (bor, bc) $\Rightarrow$
$\Diamond_{\leq Nw}\neg$Borrowing (bor, bc)]
...

This constraint is operational in that it can be achieved by application of actions available to some agent in the system. The *Borrower* agent has the *Return* action in its capabilities; the latter has Borrowing (*bor*, *bc*) in its precondition and ¬Borrowing (*bor*, *bc*) in its postcondition. The constraint can thus be established through appropriate state transitions under control of the *Borrower* agent.

Goals are made operational through constraints. The link between goals and constraints is captured in the *Operationalization* meta-relationship defined as follows:

> Operationalization (C, G)
> iff meeting constraint C is among the
> operational ways to achieve goal G.

A constraint operationalizing a goal thus amounts to some abstract "implementation" of this goal. In general a goal can be operationalized through several alternative combinations of constraints; like *Reduction*, *Operationalization* is an *AndOr* relationship. A meta-constraint here is that a goal operationalized into constraints may not be reduced further.

Coming back to the *RegularAvailability* goal, one can verify from its formal definition and from the definitions of the *Library* entity and *Borrowing* relationship that the *LimitedBorrowingPeriod* constraint above is among the operational ways to achieve it. The *NoLostCopies* constraint is another way to make the goal operational. The description above would thus be complemented as follows:

> **Systemgoal** Maintain [RegularAvailability]
> **Concerns** . . .
> **FormalDef** . . .
> **OperationalizedBy** LimitedBorrowingPeriod, NoLostCopies

The *Operationalization* meta-relationship propagates all features of the GOAL meta-concept to the CONSTRAINT meta-concept (see Fig. 2). Thus, constraints can be AND/OR-reduced in the same way that goals are; they can be conflicting, may have assigned priorities and may be wished; they concern objects and are classified by pattern and by category. Moreover, conflicts and categories are propagated through *Operationalization* links at the domain level as well; that is, constraints that operationalize conflicting goals are conflicting as well, a *SatisfactionGoal* is operationalized into a *SatisfactionConstraint*, and so forth.

A goal can be achieved provided the constraints operationalizing it can be met. To meet these constraints, appropriate restrictions may be required in turn on the actions and objects already identified; new actions and objects might also be required. Also, the possible agents that are able to enforce the constraints through these restricted and/or new actions need to be identified before most appropriate ones can be selected. The *Ensuring* and *Responsibility* meta-relationships are introduced for that purpose. (See Fig. 2.)

To introduce the definition of *Ensuring*, it is important to recognize that constraint satisfaction may require:

- the *strengthening* of *PreConditions*, *TriggerConditions*, and *PostConditions* of several actions, and of *Invariants* of several objects,
- the acquisition of new specific actions and objects,
- the acquisition of new specific features for actions and objects already acquired.

To make that point more precise, consider the *SafeTransportation* goal introduced in Section 2.7. For a lift system, this goal might be operationalized by the *DoorsClosedWhileMoving* constraint formally defined by

$$(\forall l\text{: Lift, d: Doors, f, f': Floor})$$
$$\text{PartOf}(d, l) \Rightarrow \Box[\text{LiftAt}(l, f) \wedge \circ\text{LiftAt}(l, f') \wedge f' \neq f \Rightarrow d.\text{State} = \text{'closed'} \wedge \circ(d.\text{State} = \text{'closed'})]$$

Assume the specifications of the *GoToFloor* and *OpenDoor* actions have already been acquired under the following form.

Action GoToFloor
  Input Lift {**arg**: l}, Floor {**arg**: f, f'}, Passenger {**arg**: p};
  Output LiftAt
  PreCondition LiftAt (l, f) $\wedge$ Requesting (p, f') $\wedge$ f' $\neq$ f
  PostCondition LiftAt (l, f')

Action OpenDoors
  Input Lift {**arg**: l}, Doors {**arg**: d};
  Output Lift {**res**: l}, Doors {**res**: d}
  PreCondition PartOf (d, l) $\wedge$ d.State = 'closed'
  PostCondition d.State = 'open'

To meet the *DoorsClosedWhileMoving* constraint above, one can make the following derivations.

(i) From the formal expression of the constraint and the pre- and postconditions of the *GoToFloor* action, one derives that the predicate d.State = 'closed' must hold both in the initial and final states where *GoToFloor* is applied; a strengthening of the precondition and postcondition of *GoToFloor* with the assertion

  d.State = 'closed'

is thus required.

(ii) From the formal expression of the constraint and the pre- and postconditions of the *OpenDoors* action, one derives that the antecedent in the constraint must be false in the initial states of *OpenDoors* (because the consequent is then false), that is, the predicate LiftAt (l, f) must hold *both* in the initial and final states where *OpenDoors* is applied; a strengthening of the precondition and postcondition of *OpenDoors* with the assertion

  LiftAt (l, f)

is thus required.

(iii) From the formal expression of the constraint, one derives that another action is required to yield transitions from states such that $\neg$(d.State = 'closed') to states such that d.State = 'closed'—the latter conditions define a precondition and a postcondition for a new action that might be named *CloseDoor*.

The *Ensuring* meta-relationship thus has the following meta-attributes attached to it: *StrengthenedPre*, *StrengthenedTrig*, *StrengthenedPost*, and *StrengthenedInv*. Their values at the domain level represent constraint-oriented preconditions, trigger conditions, postconditions, and invariants, respectively. Since a constraint can in general be met by several alternative combinations of strengthenings, *Ensuring* is an *AndOr* relationship. It is defined between the ACTION or OBJECT meta-concepts on one hand and the CONSTRAINT meta-concept on the other. The precise definition is as follows.

> Ensuring (*act*, *C*)
> iff the application of action *act* under strengthened conditions *Pre* ∧ *StrengthenedPred*, *Trig* ∧ *StrengthenedTrig*, *Post* ∧ *StrengthenedPost* guarantees that constraint *C* holds in the initial and final state of *act*.

> Ensuring (*obj*, *C*)
> iff the restriction of *ob* states to the strengthened condition *Inv* ∧ *StrengthenedInv* guarantees that constraint *C* holds in the initial and final states of any action on *ob*.

The combination of *Operationalization* and *Ensuring* links at the domain level provides explanations about the rationale of requirements on actions and objects with regard to system-level or organizational goals; it can be seen as a refinement of the notion of operationalization used in explanation-based learning [13, 36].

Constraints operationalizing goals must be assigned to agents that will be in charge of the strengthened actions. This is captured by the *Responsibility* meta-relationship. Since a constraint can in general be assigned to several alternative agents, *Responsibility* is an *AndOr* relationship. It is defined by

> Responsibility (*ag*, *C*)
> iff agent *ag* is among the candidates to enforce constraint *C* through some restricted behavior prescribed by *Ensuring* links.

The *Responsibility* relationship is more precisely characterized through the following *Responsibility Axiom* which is a strengthened version of the Performance Axiom in Section 2.6.

$$
\begin{array}{l}
(\forall ag\text{: AGENT, } c\text{: CONSTRAINT})\\
\quad \text{Responsibility } (ag, c) \Rightarrow\\
\quad (\forall act\text{: ACTION})\\
\qquad \text{Ensuring } (act, c) \wedge \text{Performs } (ag, act) \Rightarrow\\
\qquad \{?\text{Pre} \wedge \text{StrPre}, !\text{Trig} \wedge \text{StrTrig}\}\ [ag, act]\ \{\text{Post} \wedge \text{StrPost}\}
\end{array}
$$

Constraints may thus restrict the behavior of responsible agents, in a way similar to [14] (note, however, that no distinction is made in [14] between nonoperational goals and operational constraints).

Beside the fact that a constraint is assigned to exactly one agent in each alternative assignment considered (see the cardinality in Fig. 2), *Responsibility* is subject to two other meta-constraints:

$$(\forall ag\text{: AGENT, } c\text{: CONSTRAINT, } act\text{: ACTION})$$
$$\text{Performs } (ag, act) \wedge \text{Ensuring } (act, c) \Rightarrow \text{Responsibility } (ag, c)$$

$$(\forall ag\text{: AGENT, } c\text{: CONSTRAINT, } act\text{: ACTION})$$
$$\text{Responsibility } (ag, c) \wedge \text{Ensuring } (act, c) \Rightarrow \text{Capability } (ag, act)$$

The *Responsibility* meta-relationship has optional meta-attributes like the *Cost* for the agent to take responsibility over the constraint, the *Reliability* of the agent with respect to the constraint, and the *Motivation* of the agent to control the system behavior so as to meet the constraint. Values for such meta-attributes at the domain level are used in tactics for selecting among several alternative responsibility assignments.

For example, the *EnoughCopies* goal that appeared as a reduction of the *BookRequestSatisfied* goal in Section 2.7 might be operationalized through two constraints: *HighCoverage* and *LimitedBorrowingAmount*. The formal definition of the latter is

$$(\forall \text{ lib: Library, bor: Borrower, bc: BookCopy})$$
$$\Box\ [\#\ \{\text{bc} \mid \text{Borrowing (bor, bc)} \wedge \text{bc} \in \text{lib}\} \leq \text{Max (bor)}]$$

% Max (bor) defines an upper limit for the number of borrowed copies as a function on borrowers %

Following a same line of reasoning as above, one may derive a strengthened precondition to be attached to the *Ensuring* instance linking that constraint and the *CheckOut* action; after determination of responsibility links and acquisition of values for the *Cost*, *Reliability*, and *Motivation* attributes, one might get the following requirements fragment:

Constraint Maintain[LimitedBorrowingAmount]
  Operationalizes EnoughCopies
  FormalDef ... (see above)
  EnsuredBy
    CheckOut {**StrengthenedPre**:
        $\#\{\text{bc} \mid \text{Borrowing (bor, bc)}\} < \text{Max (bor)}$}
  UnderResponsibilityOf
    Borrower {**Reliability**: low, **Motivation**: low, **Cost**: low}
  or UnderResponsibilityOf
    Staff {**Reliability**: high, **Motivation**: high, **Cost**: low}

From this fragment one would most probably decide that the *Performs* link for the *CheckOut* action will be assigned to *Staff* rather than *Borrower*; the decision can be based on negotiation or use of tactics, see Step 7 in Section 3.

The difference between the *Responsibility* and *Performs* meta-relationships is important. Responsibility is defined between agents and constraints. It captures *alternative* assignments of constraints to agents. On the other hand, performance is defined between agents and actions. It captures the *decisions* of *actual* assignment of actions to agents; as a consequence, additional restrictions are imposed upon the agent's behavior, that is, the strengthenings of the conditions.

The CONSTRAINT meta-concept has two specializations (not represented in Fig. 2). *HardConstraints* may never be violated; *SoftConstraints* may be temporarily violated. For example, "no planes on same portion of air corridor" is a *Hard-Constraint.* Another example of a meta-constraint built into the meta-model is that "domain-level constraints in the *SafetyConstraint* category always are *Hard-Constraints*".

SoftConstraints need specific actions to restore them. This knowledge is captured in the meta-model by introducing the *Restoration* meta-relationship, defined by

Restoration (*act*, *C*)
  iff action *act* contributes to re-establishing soft constraint *C*.

Restoration has a meta-cardinality 1 : *N* for the *RestoredBy* role; every soft constraint must have at least one restoration action associated with it. This meta-constraint is a source of acquisition of new requirements fragments; e.g., the *IssueReminder* action introduced in Section 2.5 was acquired from the fact that the *Limited-BorrowingPeriod* constraint shown at the beginning of this section is declared as a *SoftConstraint.*

2.9. *Other features of the KAOS meta-model*

The KAOS meta-model incorporates other meta-concepts and meta-relationships that are not directly related to the goal-directed strategy discussed in Section 3. Here is a short list of them.

- The *Structuring* and *Composition* meta-relationships are used to structure complex objects and complex actions into components according to various structuring modes.
- The SCENARIO meta-concept is linked to the ACTION meta-concept through a *Combination* meta-relationship to support sequential, parallel, alternative, and repetitive compositions of scenarios; the latter can then be discussed with clients to validate requirements.
- The *View* ternary meta-relationship links agents, concepts (playing the role of *master* concept), and concepts (playing the role of *facet* concept). This allows domain-level concepts to be visible by agents under restricted facets only; conflicting views can also be thereby recorded for later resolution. For example, the *ProceedingsCopy* entity can be seen by the *Borrower* agent as a *Borrowable-Proceedings* entity structured as a pair (*title*, *set of papers*) whereas it could be

seen by the *staff* agent as a *LocatableProceedings* entity structured as a triple (*title, date, location in shelves*).

- The *Mapping* meta-relationship is a reflexive one on OBJECT or ACTION; it is used to specify requirements about the links between the objects and actions in the automated part of the composite system and the corresponding objects and actions in the "manual" part of it. For example, the constraint of consistency between the library database and the physical library is captured as follows.

```
Entity LibraryDatabase
  IsA Library
    % inherits features of Library, e.g., the available attribute, invariants, etc. %
  . . .
    % specific features %

SoftConstraint Maintain [SameLibraries]
  InstanceOf ConsistencyConstraint
  FormalDef
    (∀libdb: LibraryDatabase, lib: Library)
      Mapping(libdb, lib) ⇒
      □ [libdb.available = lib.available  ∧
         libdb.checkedOut = lib.checkedOut  ∧
         libdb.lost = lib.lost]
```

For more details, see [46].

3. A goal-directed acquisition strategy

In a learning-by-instruction framework [4, 8, 43], requirements about the composite system are acquired as domain-specific instances of elements of the conceptual meta-model. Such instances must satisfy the meta-constraints specified once and for all—like, e.g., the cardinality constraints on meta-relationships (see Fig. 2) or the various meta-constraints made explicit all along Section 2. The requirements gradually acquired are expressed in the acquisition language which closely reflects the structure of the meta-model, as suggested in the examples introduced in Section 2.

Acquisition processes are guided by strategies and domain models. *Strategies* define specific ways of traversing the meta-model graph to acquire instances of its various nodes and links. Each step in a strategy is itself composed from finer steps like question-answering, input validation against meta-constraints, application of tactics to select preferred alternatives for the various *AndOr* meta-relationship instances that arise during acquisition, deductive inferencing based on property inheritance through specialization links, or analogical reuse of domain models. *Domain models* are described in the same acquisition language as requirements are. They are organized as *IsA* inheritance hierarchies in the domain knowledge base; one finds various levels of specialization of goals, constraints, objects, actions, and agents involved in resource management systems, transportation systems, communication systems, and so forth. Ultimately the acquisition assistant's knowledge base should include a rich variety of domain models, strategies, and tactics. During the acquisition process, the critical decisions are made by the analyst based upon the knowledge and guidance provided by the assistant.

The strategies considered so far differ by the meta-concept(s) around which they are centered; goal-directed, view-directed, and scenario-directed strategies have been identified. The strategy of interest in this paper is the goal-directed one. It is made of the following steps. (Upper case letters are used to refer to meta-level concepts.)

(1) Acquisition of Goal structure and identification of Concerned Objects.
(2) Preliminary identification of potential Agents and their Capabilities.
(3) Operationalization of Goals to Constraints.
(4) Refinement of Objects and Actions.
(5) Derivation of strengthened Actions and Objects to Ensure Constraints.
(6) Identification of alternative Responsibilities.
(7) Assignment of Actions to responsible Agents.

In this strategy, the steps are ordered but some of them may overlap (notably, Steps 1 and 2). Moreover, backtracking is possible at every step. For example, information acquired during the responsibility identification step (Step 6) may induce changes to the results of the operationalization step (Step 3). The changes made to the latter step must then be propagated through the succeeding steps.

To understand the proposed strategy, we must address three questions for each step: *What* tasks are done during the step? *Why* are those tasks necessary? *How* are the tasks carried out? We must also identify which components of the meta-model are involved in the step. (In Fig. 2, we can trace the acquisition path from the GOAL node back through the meta-model graph.)

Step 1. Acquire goal structure and identify concerned objects

What

The system goals given by the client are incrementally refined into an overall goal/subgoal structure—an AND/OR graph. In other words, instances of the GOAL meta-concept and *Reduction* meta-relationship are acquired under the constraints specified at the meta-level. The leaf goals of this structure are primitive goals which can be made operational through constraints in Step 3. A portion of a possible goal structure related to borrower goals in a library system is visualized in Fig. 3.

The elaboration of the goal structure consists of three substeps:

(i) Identify *SystemGoals*, their *category*, and their *pattern*, and associate them with the parent goal(s) they *Reduce*; formalize refined subgoals according

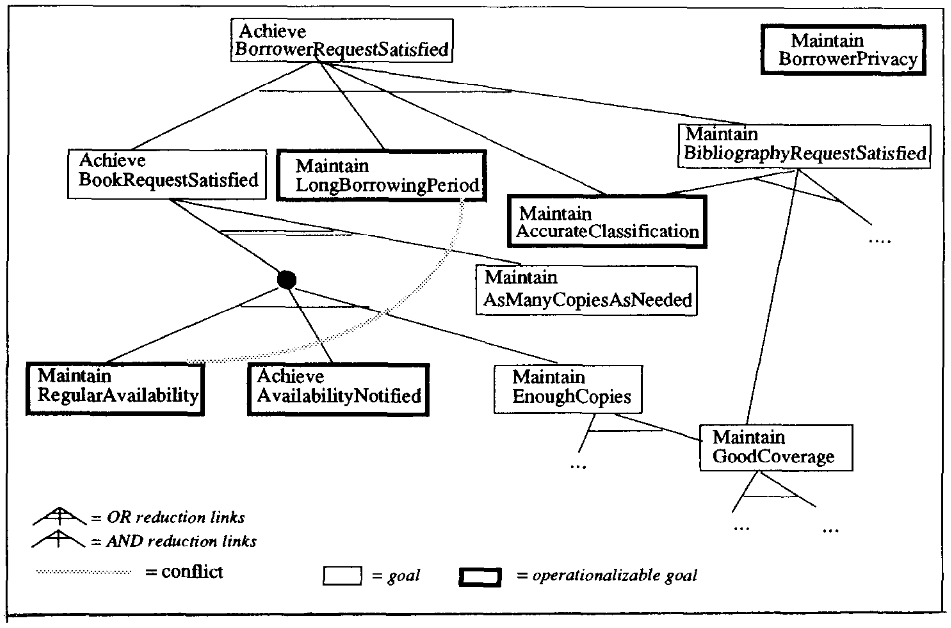
Achieve BorrowerRequestSatisfied
Maintain BorrowerPrivacy
Maintain BibliographyRequestSatisfied
Achieve BookRequestSatisfied
Maintain LongBorrowingPeriod
Maintain AccurateClassification
Maintain AsManyCopiesAsNeeded
Maintain RegularAvailability
Achieve AvailabilityNotified
Maintain EnoughCopies
Maintain GoodCoverage
= OR reduction links
= AND reduction links
= conflict
= goal
= operationalizable goal

Fig. 3. Portion of a Goal structure for borrower goals.

to their specified pattern as soon as they become concrete enough. (In general abstract goals near roots of trees cannot be formalized.)

(ii) As the reduction proceeds, identify the objects concerned by the goals and elaborate a preliminary definition of their features—e.g., basic domain-specific attributes that appear in goal descriptions, preliminary invariants to be attached to them.

(iii) Identify possible conflicts among system goals; that is, define instances of the *Conflict* meta-relationship. For each conflict being detected, assign priorities to the conflicting goals. (Those priorities define a partial order on goals.)

The three substeps above are not sequential; they are intertwined. When conflicts are identified, it may be necessary to find an alternative reduction that has fewer conflicts. As a result, new goals may be identified.

Why

The reduction of system goals into primitive goals is necessary because global goals usually cannot be directly translated into constraints; only simple, primitive goals can be operationalized. Moreover, the system-wide goal structure records the history of the acquisition process. This structure is important because:

- it ties specification components to their rationale (i.e., goal descriptions);
- it will be used in case negotiation is required to resolve goal conflicts [39];
- it can be used to replay some part of the acquisition process in other circumstances where similar portions of the goal structure are recognized.

The preliminary identification and characterization of objects from goals ensures that only those objects which are relevant to goals are under consideration.

How

The identification and reduction of goals is a nontrivial, but critical, task. Analysts and clients must interact a lot at this stage. The following tactics help the analyst to refine the goal structure.

1. Reuse relevant generic goals and reductions by specializing/instantiating their description

Generic goals are retrieved in the domain knowledge base; the indexing scheme for retrieval is based on goal *category*, goal *pattern*, and *IsA* links between the *Concerned* objects already identified and their generalizations in the domain models available. The retrieved goals and their reductions are then considered for specialization and adaptation to the specific composite system being modeled.

For example, the *BookRequestSatisfied* introduced in Section 2.7 was handled in that way. This goal was classified in the *SatisfactionGoal* category and was declared to have an *Achieve* pattern; the Concerned *Borrower* and *Book* objects were declared to be *IsA* specializations of the generic *User* and *Resource* objects in the resource management domain model, respectively. The following requirements fragment is then retrieved in the domain knowledge base on that basis:

SystemGoal Achieve [ResourceRequestSatisfied]
  InstanceOf SatisfactionGoal
  Concerns User, Resource, Using, . . .
  FormalDef
    $(\forall u\text{: User, res: Resource, rep: Repository})$
      $\text{Requesting }(u, res) \wedge \text{InScope }(res, rep) \Rightarrow$
      $\Diamond(\exists ru\text{: ResourceUnit })\ (\text{Unit }(ru, res) \wedge \text{Using }(u, ru))$
  ReducedTo EnoughUnits, UnitsAvailable, AvailabilityNotified

The generic concept names are then instantiated to their library-specific counterpart, and the generic *InScope* predicate has to be specialized in an appropriate fashion to the library-specific context. Note that a *reduction* into three generic subgoals is also proposed. The *UnitsAvailable* subgoal and its formal expression could even be more specific if the *Book* concept is declared as an *IsA* specialization of a more specific concept than the *Resource* concept, namely, the *Returnable-Resource* concept; the *UnitsAvailable* subgoal becomes *RegularAvailability* in this case, and the formal expression of that goal in Section 2.7 is obtained as a straightforward instantiation of it. (The reader should be convinced that the same process can be replayed for the *BedRequestSatisfied* goal in a hospital management system.)

The reuse process is under control of the analyst, of course. At any time it is possible to adapt proposed goals, to reject them or to provide new specific ones. The more abstract and general the reused concept description is, the more important the required adaptation might be. For example, the *AvailabilityNotified* goal in Section 2.7 could be seen as an adapted specialization of the following very abstract goal.

> **Systemgoal** Achieve [UserInformedOfStateChange]
> **InstanceOf** InformationGoal
> **Concerns** User, SystemComponent
> **FormalDef**
> $(\forall u\text{: User, comp: SystemComponent})$
> $[P(\text{comp}) \wedge \bullet\neg P(\text{comp}) \Rightarrow \Diamond \text{Knows}(u, \text{comp})]$

Other tactics can be used during acquisition of the goal structure such as the following ones.

2. Stop reducing a goal when it can be operationalized

The sooner a goal description can be translated into an operational form, the better; formal reasoning can then take place to ensure constraints through appropriate actions and to assign responsibilities to agents. (Remember that operational constraints can themselves be reduced.) For example, the *BookRequestSatisfied* goal was seen in Section 2.7 to require further reduction as it could not be operationalized; its *RegularAvailability* subgoal requires no further reduction because it can be operationalized into the *LimitedBorrowingPeriod* constraint defined in Section 2.8.

3. Reduce goals into subgoals so that the latter require cooperation of fewer potential agents to achieve them

This tactics is the basic one to ensure that the reduction process makes progress towards a stage where all goals are operationalizable. Its use requires that some progress has already been made in Step 2 since information about potential agents and their capabilities need to be available. Thus, Step 1 and Step 2 are working like coroutines.

4. Choose an alternative reduction that minimizes costs

Costs are taken in a broad sense here (e.g., cost of achieving a goal by means of a human agent versus cost of achieving it by means of a program to be developed or to be acquired for that purpose, cost of purchasing the resources concerned by the goal, etc.). When it can be anticipated that a goal will be too costly to achieve, it might be necessary to find cheaper alternative reductions. The problem with this heuristic is that cost evaluation is a very complex task; moreover it is normally handled later when *ResponsibilityCosts* are evaluated, prior to assigning *Performs* links to agents. Nevertheless, we don't need complex evaluation functions to detect goals which appear from the beginning to be very costly.

For example, the *AsManyCopiesAsNeeded* subgoal introduced in Section 2.7 as one alternative reduction of the *BookRequestSatisfied* goal would be rejected using this tactics—at least in the case of large libraries with many potential borrowers.

5. Choose an alternative reduction with as few conflicts as possible

6. Resolve conflicts according to the relative priority among goals

The conflicts with the highest priority goals should be resolved first. Higher priority goals should obviously be favored. For example, the *LongBorrowingPeriod* goal described in Section 2.7 was seen to conflict with the *RegularAvailability* goal; the latter is of much higher priority so that it is most likely the one to be retained. (A fair value for the number N of weeks before return might emerge as a compromise during negotiation [39].)

As mentioned before, the various objects *Concerned* by goal descriptions are given partial characterizations. For example, the *RegularAvailability* goal concerns the *Library* object (see Section 2.7); moreover an *available* attribute of this object appears in the expression of the goal. At this stage one might recognize the *Library* object as being an instance of the ENTITY meta-concept, introduce an additional *checkedOut* attribute and write a partial invariant "$\text{lib} = \text{lib.available} \cup \text{lib.checkedOut}$"; the *lost* attribute might be introduced later in Step 3 when the constraint *Avoid* [*LostCopies*] is acquired as one of the ways to operationalize the *RegularAvailability* goal. Similarly, the *Borrowing* object is concerned by the *BookRequestSatisfied* goal; it meets the criterion for being a RELATIONSHIP instance (see Section 2.4), and the objects it links appear from the expression of the *BookRequestSatisfied* goal—namely, *Borrower* and *BookCopy*. A partial invariant recognized at this stage might be

$$\text{Borrowing (bor, bc)} \Rightarrow \text{bc} \in \text{lib.checkedOut.}$$

In case generic goals have been reused according to the first tactics above, the features of the generic objects *Concerned* by these goals are specialized/instantiated correspondingly. For example, the generic *Repository* and *Using* objects are instantiated to *Library* and *Borrowing*, respectively; the partial invariants of the *Library* and *Borrowing* objects suggested above can then be obtained as instantiations of the following generic invariants:

$$\text{rep} = \text{rep.available} \cup \text{rep.used}$$
$$\text{Using (u, ru)} \Rightarrow \text{ru} \in \text{rep.used}$$

(In these assertions, rep.available and rep.used are generic attributes of the *Repository* entity; their domain is known to be "**SetOf** [ResourceUnit]".)

Step 2. Identify potential agents and their capabilities

What

A preliminary identification is made of the agents that could be available in the composite system, together with their category (human agent, physical device, program) and the actions they are capable of performing on the objects involved in goal descriptions. In other words, goal-directed instances of the AGENT meta-concept, *Capability* meta-relationship, and ACTION meta-concept are acquired under the constraints specified at the meta-level. For each action appearing in the capability list of an agent, a pair of basic precondition and postcondition is specified; this pair must capture the elementary state transitions produced by application of the action on the objects identified in Step 1.

Why

Such preliminary information about agents potentially available is needed to determine when the reduction process can terminate (Step 1) and to guide the operationalization process (Step 3).

How

The various objects appearing in the goal descriptions elaborated in Step 1 are reviewed to determine those which can control state transitions of the others. For each agent obtained thereby, the actions corresponding to these state transitions are identified and the elementary pre- and postconditions describing these transitions are written down. For example, the *Borrower* agent is identified from the description of the *BookRequestSatisfied* goal introduced in Section 2.7. This object is an agent because it can control state transitions of the *Borrowing* object appearing in the formal expression of that goal; the two possible transitions are "Borrowing→ ¬Borrowing" and "¬Borrowing→Borrowing", from which *Return* and *CheckOut* actions are identified together with their elementary pre- and postcondition (e.g., Borrowing (*bor, bc*) and ¬Borrowing (*bor, bc*) for *Return*). Similarly, the *Passenger* object referenced in the *SafeTransportation* goal is identified as an agent because it can control transitions such as being in and then out of the lift or getting the doors open and then closed.

Additional agents and capabilities are acquired by interaction between the clients and the analysts. All the agents eventually required in the composite system are not necessarily identified at this stage, however. For example, a *Counter* device agent could be required to ensure that no book copy is improperly removed without being checked out; the need for such an agent might arise later in Step 6 when *Responsibility* links are identified for the *Avoid* [*LostCopies*] constraint operationalizing the *RegularAvailability* goal.

Also remember that agents can be organized into specialization hierarchies like any other domain-level concepts. A specialized agent then inherits all capabilities of the more general agents it specializes; in addition it may have specific capabilities (e.g., the *ResearchStaff* agent has all capabilities of the *Staff* agent plus specific ones such as ordering new book copies).

The following tactics may be helpful in identifying agents and capabilities.

1. Reuse relevant generic agents and capabilities by specializing/instantiating their description

Generic agents are retrieved in the domain knowledge base; the agents considered are those objects *Concerned* by the generic goals retrieved in Step 1 which are of the AGENT meta-type. The retrieved agents and the generic actions they are capable of are then considered for specialization and adaptation to the specific composite system being modeled.

For example, in case the generic *ResourceRequestSatisfied* goal has been reused in the more specific context of the *ReturnableResource* concept, the *User* agent is retrieved with the *GetResource* and *ReturnResource* actions from its capability list. The following partial description of, e.g., the *GetResource* action is then proposed for instantiation and specialization:

> **Action** GetResource
> **Input** ResourceUnit {**Arg**: ru},
> Repository {**Arg**: rep}, User {**Arg**: u};
> **Output** Repository {**Res**: rep}, Using
> **PreCondition** ru $\in$ rep.available
> **PostCondition**
> $\neg$(ru $\in$ rep.available) $\wedge$ ru $\in$ rep.used $\wedge$ Using (u, ru)

The process of instantiating, specializing, and adapting generic descriptions follows the same line as suggested for Step 1. The outcome in this simple case is the description of the *CheckOut* action given in Section 2.5.

2. *For each action in the capability list of a human agent, consider the relevance of an automated agent with a corresponding action in its capability list*

This tactics is the basic one for introducing new devices and programs as candidates in the space of alternative agent assignments (see Step 6). If the introduction of an automated action appears to be relevant, this action and the objects involved in it can be defined as *IsA* specializations of the action and objects already defined; in particular, pre- and postconditions on the corresponding image objects are thereby inherited as features of the automated action. Instances of the *Mapping* meta-relationship must then be introduced to link concepts and their automated counterpart. (See Section 2.9.) Most often new goals have then to be introduced; such goals are in the *ConsistencyGoal* category. For example, a *CheckOutTransaction* action might be identified as a possible capability of an automated *LibraryDatabase-Manager* agent; it is defined as an *IsA* specialization of the *CheckOut* action. Corresponding image objects are then identified and defined in a similar way (e.g., the *LibraryDatabase* entity introduced in Section 2.9). An object and its corresponding image must both be instances of the OBJECT meta-type; however, they need not necessarily be instances of the same specialized meta-type (e.g., the automated counterpart of the *Borrower agent* will be a *BorrowerRecord entity* which will record relevant information about borrowers). One of the *ConsistencyGoals* required in this example should concern the *Library* and *LibraryDatabase* entities; this goal is operationalized in the *Maintain*[*SameLibraries*] soft constraint given in Section 2.9. The use of this tactics allows one to avoid confusions between the physical and automated parts of the composite system; such confusions are frequently made in specifications [49].

Step 3. Operationalize goals into constraints

What

The leaf goals in the goal structure elaborated in Step 1 are transformed into system objectives formulated in terms of objects and actions available to some agent identified in Step 2. In other words, instances of the CONSTRAINT meta-concept and *Operationalization* meta-relationship are elaborated under the constraints specified at the meta level. A constraint definition can lead to the identification of new objects and actions involved in the constraint.

Why

The system objectives need to be made operational in order to (i) derive new or strengthened actions and objects which will support them (Steps 4 and 5) and (ii) assign responsibilities (Steps 6 and 7).

How

Like the elaboration of goals, the transformation of goals into constraints is a nontrivial task. Several alternative operationalizations can implement the same goal, just like several alternative programs can implement the same specification. In such situations, a best operationalization should be retained. The following tactics may be used to guide the analyst in carrying out the transformation.

1. Reuse relevant generic operationalizations by specializing/instantiating their description

Generic constraints are retrieved in the domain knowledge base; the constraints considered are those which operationalize the generic goals retrieved in Step 1. The retrieved constraints and their reduction (if any) are then considered for specialization and adaptation to the specific composite system being modeled. The process is similar to the one suggested in Steps 1 and 2.

For example, the *Maintain* [*LimitedBorrowingPeriod*] constraint was seen in Section 2.8 to operationalize the *RegularAvailability* goal. This constraint could have been obtained by instantiation of the following constraint found in the domain knowledge base to operationalize the *UnitsAvailable* generic goal—at the specialization level where the *ReturnableResource* concept is defined.

SoftConstraint Maintain [LimitedPeriodOfUse]
FormalDef
(∀u: User, ru: ResourceUnit)
$\Box[\text{Using}(u, ru) \wedge \bullet\neg\text{Using}(u, ru) \Rightarrow$
$\Diamond_{\leq Nw}\neg\text{Using}(u, ru)]$

As another example, consider the *AvailabilityNotified* goal introduced in Section 2.7. This goal is an instantiation of the following generic goal:

Systemgoal Achieve [AvailabilityNotified]
  InstanceOf InformationGoal
  Concerns User, Repository
  FormalDef
    ($\forall$rep: Repository, u: User,
      res: ReturnableResource, ru: ResourceUnit)
      Requesting (u, res) $\wedge$
      $\bullet\neg(\exists$ru: ResourceUnit) (Unit (ru, res) $\wedge$ ru $\in$ rep.available) $\wedge$
      ($\exists$ru: ResourceUnit) (Unit (ru, res) $\wedge$ ru $\in$ rep.available) $\Rightarrow$
      $\Diamond$Knows (u, rep.available)

This goal is known in the domain theory to be operationalizable into the following constraint:

SoftConstraint Achieve [UserNotified]
  InstanceOf InformationGoal
  Concerns User, Repository
  FormalDef
    ($\forall$rep: Repository, u: User,
      res: ReturnableResource, ru: ResourceUnit)
      Requesting (u, res) $\wedge$
      $\bullet\neg(\exists$ru: ResourceUnit) (Unit (ru, res) $\wedge$ ru $\in$ rep.available) $\wedge$
      ($\exists$ru: ResourceUnit) (Unit (ru, res) $\wedge$ ru $\in$ rep.available) $\Leftrightarrow$
      $\circ(\exists$ntu: NoticeSentToUser)
        [Occurs (ntu) $\wedge$ ntu = (res, u, 'message')]

In this constraint, *NoticeSentToUser* is an event with appropriate attributes that captures the required user notification. (This constraint exhibits a standard pattern of operationalizing *Knows* predicates.) The instantiation of the generic *User, Repository, ReturnableResource,* and *ResourceUnit* concepts to *Borrower, Library, Book,* and *BookCopy,* respectively, yields an instantiated constraint proposed to the analyst for possible adaptation.

A more general form of reuse could be supported at the process level. The knowledge base might contain a set of domain-specific operationalization rules that could be applied for a variety of similar goals [36]. The operationalization process would then be replayed as done in some derivational analogy systems [44]. This promising approach has not been explored yet.

2. Use goal reduction tactics transposed to constraints

As seen in Section 2.8, constraints are operational system objectives; the *Operationalization* meta-relationship propagates all features of the GOAL meta-concept to the CONSTRAINT meta-concept—i.e., *Reduction*, *Conflict*, *Category*, etc. Some commonsense tactics used in Step 1 can thus be used for goal operationalization and constraint reduction, e.g., *choose an alternative operationalization or reduction that minimizes Ensuring costs*, *choose an alternative operationalization or reduction with as few conflicts as possible*, and so forth.

3. Choose an alternative operationalization that minimizes the need for restoration actions

If the soft constraints chosen to operationalize a given leaf goal need complex restoration actions, then an alternative way of operationalizing the leaf goal should be considered—an alternative where the constraints are violated in fewer situations or where less or simpler restoration actions are required.

Step 4. Refine objects and actions

What

The constraints obtained in Step 3 can involve new objects and new actions; entities, relationships, events, agents, and state transitions not identified in Steps 1 and 2 can emerge from the operational formulations. Also, new features of concepts already identified can be referred to (e.g., new domain-specific attributes of objects). In this step the analyst defines the objects and actions newly identified and completes the description of objects and actions already identified; new domain-specific attributes and new elements of invariants and pre- and postconditions are thereby introduced.

Why

The refined descriptions of objects and actions form the basis for the subsequent acquisition steps. Invariants and pre- and postconditions will be strengthened to ensure the constraints. They are also needed to identify the *Responsibility* links.

How

The process of acquiring additional requirements fragments about objects and actions from constraints is similar to the process of acquiring initial ones from goals, see Steps 1 and 2.

For example, the *Achieve* [*UserNotified*] constraint above once instantiated to *Achieve* [*BorrowerNotified*] yields an implication whose consequent is

$$(\exists \text{ntb: NoticeSentToBorrower})\\ [\text{Occurs}(\text{ntb}) \wedge \text{ntb} = (\text{b, bor, 'message'})]$$

A new object of EVENT meta-type is thus involved. A preliminary description of this event might include an invariant capturing its condition of occurrence. According to a meta-constraint on the EVENT meta-type, a new action must be acquired therefrom—that is, the *SendAvailabilityNotice* action having *NoticeSentToBorrower* as output; a pair of elementary pre- and postconditions for that action should thus be acquired. The acquisition process can be based on reuse of generic descriptions, as shown before.

Step 5. Derive strengthened actions and objects to ensure constraints

What

The descriptions of actions and objects completed in Step 4 do not necessarily guarantee that the constraints obtained in Step 3 will be met. In this step, strengthened actions and objects are derived to ensure that all required constraints are satisfied. The strengthenings are put on preconditions, postconditions, and invariants; trigger conditions are introduced for some actions; new actions can still be introduced to yield state transitions involved in the formulation of constraints; restoration actions are defined for soft constraints. In other words, instances of the *Ensuring* meta-relationship are elaborated under the meta-constraints specified in the meta-model; they link actions and objects to the constraints they *Ensure.* Note that *Ensuring* is an *AndOr* relationship (see Fig. 2 and Section 2.8); as a frequent case, a constraint can be *Ensured* by a combination of actions and objects in the physical subsystem or alternatively by some counterpart of this combination in the automated subsystem.

Why

Ensuring links are necessary to identify which actions and objects are going to contribute to the satisfaction of which constraint and to show how those actions and objects are contributing to constraint satisfaction. Action strengthening has an impact on the possible behavior of the agent allocated to the action; the information acquired in this step is therefore taken into account in the next steps when responsibilities are identified and assigned.

How

Trigger conditions and strengthenings on pre-, postconditions, and invariants are derived from the formal expression of constraints. Each action is matched against each constraint to check whether the state transitions defined by the action meet the constraint. The match may reveal subsidiary conditions for the action to meet the constraint; if this is the case, these conditions are taken as strengthenings on the action. The principle is similar for objects and their invariants. To make this process more precise, we illustrate it by giving some inference rules for three general patterns of constraints. (A full calculus for deriving strengthenings is out of the scope of this paper.)

Let *Cons* denote the pattern of the constraint, *Pre* and *Post* the patterns of the action's pre- and postcondition, *StrPre* and *StrPost* the required strengthenings on

Pre and *Post*, respectively, and *Trig* the required trigger condition (if any). The rules of inference are the following.

$$\frac{\mathit{Cons}\colon \Box[C \wedge (P_1 \wedge \bigcirc P_2 \Rightarrow Q_1 \wedge \bigcirc Q_2)],\ \mathit{Pre}\colon P_1,\ \mathit{Post}\colon P_2}{\mathit{StrPre}\colon Q_1,\ \mathit{StrPost}\colon Q_2}$$

$$\frac{\begin{array}{l}\mathit{Cons}\colon \Box[C \wedge (P_1 \wedge \bigcirc P_2 \Rightarrow Q_1 \wedge \bigcirc Q_2)],\\ \mathit{Pre}\colon P \text{ with } P \Rightarrow \neg Q_1 \textbf{ or } \mathit{Post}\colon Q \text{ with } Q \Rightarrow \neg Q_2\end{array}}{\mathit{StrPre}\colon \neg P_1 \textbf{ or } \mathit{StrPost}\colon \neg P_2}$$

$$\frac{\mathit{Cons}\colon \bigcirc P,\ \mathit{Pre}\colon P_1,\ \mathit{Post}\colon P_2, \neg[P \wedge P_1 \Rightarrow \bigcirc(P \wedge P_2)]}{\mathit{StrPre}\colon Q_1,\ \mathit{StrPost}\colon Q_2 \textbf{ such that } P \wedge P_1 \wedge Q_1 \Rightarrow \bigcirc(P \wedge P_2 \wedge Q_2)}$$

$$\frac{\mathit{Cons}\colon \bullet P_1 \wedge P_2 \Leftrightarrow \bigcirc Q,\ \mathit{Post}\colon Q}{\mathit{Trig}\colon \bullet P_1 \wedge P_2}$$

The first rule was applied to derive the strengthenings on the *GoToFloor* action from the *DoorsClosedWhileMoving* constraint in Section 2.8. The second rule was applied to derive the strengthenings on the *OpenDoors* action from that constraint. The third rule was applied to derive the strengthening

$$\#\{\text{bc} \mid \text{Borrowing (bor, bc)}\} < \text{Max(bor)}$$

on the precondition of the *CheckOut* action to ensure the *LimitedBorrowingAmount* constraint also formalized in Section 2.8. The fourth rule of inference is used to derive the trigger condition

$$\begin{array}{l}\bullet\neg(\exists \text{bc: BookCopy) (Copy (bc, b)} \wedge \text{bc} \in \text{lib.available)} \wedge \\ (\exists \text{bc: BookCopy)(Copy (bc, b)} \wedge \text{bc} \in \text{lib.available)} \wedge \\ \text{Requesting (bor, b)}\end{array}$$

to be attached to the *SendAvailabilityNotice* action revealed in Step 4 above; this trigger condition is derived from the *Achieve* [*BorrowerNotified*] instantiation of the *Achieve* [*UserNotified*] constraint introduced in Step 3 above.

New actions to restore soft constraints can be derived using the same general principle. (E.g., the *IssueReminder* action introduced in Section 2.5 is acquired to restore the *LimitedBorrowingPeriod* constraint; the *RectifyLibraryDatabase* action mentioned in Section 2.3.2 is acquired to restore the *Maintain* [*SameLibraries*] constraint introduced in Section 2.9.)

Tactics can also be used to help in the derivation of *Ensuring* links.

1. *Reuse relevant generic actions and strengthenings by specializing/instantiating their description*

Generic *Ensuring* links are retrieved in the domain knowledge base; the links considered are those which ensure the generic constraints retrieved in Step 3. The retrieved links and their associated strengthenings are then considered for specialization and adaptation to the specific composite system being modeled. The process is similar to the one suggested in Steps 1-3.

2. *Choose an alternative Ensuring link that minimizes the restrictions on the Ensuring actions*

Strengthened conditions defined on actions restrict the agents' behavior (see the *Responsibility* axiom in Section 2.8); therefore, it is often preferable to define an alternative *Ensuring* structure which imposes as few restrictions as possible.

Step 6. Identify alternative responsibilities

What

In this step the analyst acquires the *AndOr Responsibility* structure linking the agents to the constraints. For each constraint obtained in Step 3, the various possible *Responsibility* links are identified; the identification is based on the capabilities of the agents determined in Step 2. The acquisition of *Responsibility* links includes the determination of values for the *Cost, Motivation,* and *Reliability* attributes attached to the *Responsibility* meta-relationship. The various automation alternatives being considered are thus made explicit at this stage. *PrivateGoals* and *Wish* links (if any) are also acquired for the human agents identified in Step 2.

Why

The information acquired in this step is needed in the next step to make the right decisions about which processor (human agent or program) to assign to which action—so that all constraints operationalizing the system goals are guaranteed to be met.

How

The acquisition is guided by meta-constraints on the *Responsibility* meta-relationship. (See Section 2.8.) A constraint is assigned to one agent in each alternative assignment considered. An agent is a possible candidate provided (i) the actions *Ensuring* the constraint are in the capability list of the agent, and (ii) the agent can behave according to the requirements put on the actions—precondition, postcondition, trigger condition, and their respective strengthenings attached to *Ensuring* links. (In other words, the *Responsibility axiom* must be satisfied.)

PrivateGoals and *Wish* links are acquired by interaction between the analysts and the clients. Values for the *Cost, Motivation,* and *Reliability* meta-attributes attached to *Responsibility* links are also estimated through such interactions; cost estimation models can be integrated here. In general, costs will depend upon both the agent and the actions being involved to meet the constraint. *Motivation* can be partially estimated from the potential source of conflict or mutual support between the private goals of the agent and the leaf goal(s) operationalized by the constraint. The motivation of the agent for controlling actions to meet the constraint is expected to be low in case of conflict and high in case of mutual support.

The following tactic can also be used to help in the identification of alternative *Responsibility* links.

Reuse relevant Responsibility links between generic constraints and agents

Generic *Responsibility* links are retrieved in the domain knowledge base; the links considered are those which link the generic agents retrieved in Step 2 to the generic constraints retrieved in Step 3. The retrieved links are then considered for specialization and adaptation to the specific composite system being modeled. The process is similar to the one suggested in Steps 1–3.

Let us suggest a few examples for some of the constraints introduced before in the paper. The *LimitedBorrowingPeriod* constraint is assignable to the *Borrower* agent or to the *Staff* agent; both agents have the *Return* and *IssueReminder* actions in their capability list, and they both could enforce the corresponding strengthenings on these actions to ensure the constraint. (The strengthening amounts to a strengthened postcondition for *Return* and to a trigger condition for *IssuedReminder*.) The *Borrower* agent has a *LongBorrowingPeriod* private goal which conflicts with the *RegularAvailability* goal and *LimitedBorrowingPeriod* constraint operationalizing this goal. (See Fig. 3 and the formal expressions given in Sections 2.7 and 2.8.) Therefore, the values for *Motivation* and *Reliability* in the corresponding *Responsibility* links are very low for *Borrower* while being high for *Staff*. Automated counterparts for the *Return* and *IssueReminder* actions were identified in Step 2 and defined as specializations of these actions; the inherited *LimitedBorrowingPeriod* constraint referring to the automated representations of the corresponding objects is assignable to the *LibraryDatabaseManager* agent since the latter can enforce the constraint through corresponding strengthenings.

Similarly, the *LimitedBorrowingAmount* constraint is assignable to the *Staff* agent or to the *Borrower* agent; both can enforce the strengthening on the *CheckOut* action derived in Step 5 to ensure that constraint. For the alternative *Responsibility* link involving *Borrower*, the values of *Motivation* and *Reliability* are very low because there is a conflict with the *AsManyBooksAsNeeded* private goal. The automated counterpart of *LimitedBorrowingAmount* is assignable to the *LibraryDatabaseManager* agent which can enforce the corresponding strengthened precondition on the *CheckOutTransaction* action.

As a last example, consider the *AccurateClassification* leaf goal appearing in Fig. 3. This goal is operationalized through two constraints, namely, *AccurateShelfMark* and *AccurateKeywordsAssigned*. The former can be assigned to the *Staff* agent or to a shelf-mark allocation program; the latter can be assigned to the *SecretaryStaff* agent with low *Reliability* or to the *ResearchStaff* agent with high *Reliability*.

Step 7. Assign actions to responsible agents

What

Performs links are effectively assigned to agents for the various actions elaborated in Steps 2 and 4 on the basis of the alternative *Responsibility* links established in

Step 6. The allocation of actions to processors implies, in particular, that the *Performing* agents selected are contractually committed to satisfy the *Responsibility* axiom (see Section 2.8).

The agent *Load* values are gradually updated as the assignment of *Performs* links to the agent proceeds. (An agent can initially have a non-null load if it has assignments in other composite systems.) Backtracking on assignment decisions may take place when an agent becomes overloaded; some *Performs* links are then undone.

Why

The eventual assignment of actions to agents under commitment to the *Responsibility* axiom guarantees that the constraints operationalizing system goals will be met through appropriate behavior of the agents.

How

An action is assigned to an agent only if the agent has been determined to be among the alternative candidates for taking responsibility over the constraints the action ensures. (See Step 6.)

The following tactics can be used to help in deciding between alternative candidates.

1. Do not make effective assignments that would prevent other constraints from being met

For example, deciding that the *CheckOut* action is allocated to the *Borrower* agent can prevent the *SameLibraries* consistency constraint from being met; the *Borrower* agent has indeed no responsibility link with the latter constraint. (The formal expression of this constraint was given in Section 2.9. In fact, the *RectifyDatabase* restoration action and its specializations are not in the capability list of the *Borrower* agent.) The rational decision is thus to allocate the *CheckOut* action to the *Staff* agent. In that case, the *Staff* agent actually has the *CheckOutTransaction* action under supervision as well because he/she is also responsible for the *SameLibraries* constraint.

2. Reuse relevant Performs links between generic agents and actions

Generic *Performs* links are retrieved in the domain knowledge base; the links considered are those which link the generic agents retrieved in Step 2 to the generic actions retrieved in Step 5. The retrieved links are then considered for specialization and adaptation to the specific composite system being modeled.

3. First assign Performs links for actions ensuring constraints that operationalize the highest priority goals

Using this tactic, one would assign the *CheckOut* or *PutKeywords* actions (with *Load* values being increased correspondingly) before the action of issuing a list of recent book acquisitions.

4. Do not make effective assignments that would be conflicting with PrivateGoals

If a human agent is assigned an action ensuring constraints that operationalize goals in conflict with his/her private goals, the agent will not be very motivated to guarantee satisfaction of that constraint. The assignment of the *ReturnTransition* action to the *Borrower* agent would be rejected on that ground.

5. Maximize reliability

If there is a choice among several agents, select the agent with the highest reliability. Using this tactics, one would retain the *ResearchStaff* agent for the *PutKeywords* action to ensure the *AccurateKeywordsAssigned* constraint above.

6. Avoid overloading agent

An excessive load of actions to ensure constraints can seriously degrade the overall system performance.

7. Minimize cost of performance

If there is a choice among several agents, select the agent with the lowest performance cost.

Note that these tactics refer to one single meta-attribute/relationship. This kind of hill climbing search for local optima may not reach a global optimum; ideally all criteria should be considered together. Multicriteria analysis techniques might be of great help in this context [51]. (The same remark holds for the other tactics in the previous steps.) For example, the eventual decision of choosing the alternative where the *LibraryDatabaseManager* agent is allocated a number of transactions that automate their manual counterpart will be governed by a combination of tactics integrating goal achievement, reliability, cost, and load reduction.

4. Conclusion

This paper has proposed a meta-model for capturing initial requirements and a strategy for conducting the requirements acquisition process. The requirements considered here refer to the entire composite system—that is, the part to be automated, its physical environment, and the way both parts have to cooperate. A salient feature of the approach is the importance given to system-level goals and their operationalization through constraints. This contrasts with some traditional technology for formal or semi-formal specification, where all requirements are supposed to be captured in terms of "data" and "operation" abstractions.

Some experience with real requirements documents has convinced the authors that higher-level abstractions such as "goal", "operationalization", "ensuring action", "agent", "responsility", or "alternative assignment" are found informally and explicitly in the requirements of non-toy systems. The formal framework proposed here can be seen as a preliminary attempt to reason more formally in terms of such higher-level concepts. Meta-level constraints and rules of inference based on temporal logic allow formal checking of requirements and formal derivation of goal-directed strengthenings of them. It is encouraging to see that others have independently recognized the need for reasoning about system goals, their category and their reduction or interference links (Mylopoulos et al. [35]).

The strategy discussed in this paper amounts to a goal-directed traversal of the meta-model graph; specific tactics are applied at each node to acquire the corresponding requirements fragments. Another salient feature here is the reuse of both meta-level and domain-level knowledge. The principle of a rich meta-model to guide the acquisition process was inspired from work on machine learning; in some learning strategies, the acquisition process is guided by abstract knowledge about what should be acquired [4, 8]. The KAOS meta-model may appear to be rather complex; this is the price to pay for meta-level guidance during acquisition. The more domain-independent knowledge the meta-model privides, the more guidance the acquisition strategy can provide. On another hand, our experience in acquiring requirements for a variety of resource management systems (library systems, airline reservation, warehouse processing, hospital management) and transportations systems (lifts, trains, metros) has given us much confidence in the power of reusing generic descriptions. Such descriptions are retrieved in a domain knowledge base and then instantiated, specialized, and adapted to the system considered. Beside the usual benefits of reuse, the matching of such descriptions against the requirements already acquired often results in detecting problems which otherwise can be very hard to detect—notably, inadequacies, incompletenesses, and contradictions.

The requirements fragments given in the paper come from a rational rederivation of the requirements for a university library system currently in use. It may be worth to compare these fragments with the simplistic requirements of the classical library problem [49] to see how some of the informal requirements stated there are derived in our approach.

We have argued that requirements acquisition languages need much richer abstractions than those supported by traditional specification formalisms such as, e.g., state-based or algebraic ones. The latter are, however, needed but at a later stage where more sophisticated formal checking is undertaken on the specification of the automated subsystem. Acquisition languages and (design) specification formalisms may thus play complementary roles. Based on this, we have developed a set of rules for transforming KAOS objects and actions into Z data and operation schemas [42].

Other components of the KAOS meta-model, not discussed in the paper, provide the basis for defining other strategies—like *agent-directed* strategies where the meta-model is traversed from the views agents have about the composite system, or *scenario-directed* strategies where typical usage scenarios are elaborated first. Our current belief, however, is that a goal-directed strategy is the best one to establish that the system objectives will be achieved by proper cooperation of responsible agents.

There are some weak facets in our approach, on which we plan to work in a near future. The declaration part of the acquisition language should clearly have a graphical concrete syntax that would reflect the various concepts and links supported by the meta-model. The assertion sublanguage should incorporate deontic logic extensions to support a deeper level of formal reasoning about agent capabilities and responsibility assignment. Cooperation and communication among agents should also be supported more explicitly. As alluded to before, the tactics should also be refined to handle multiple criteria and extended to form a rich body of rules; in particular, goal conflict resolution strategies need to be carefully investigated. (The problem of interfering goals is well recognized as being a difficult one to tackle, see, e.g., [48].) Ultimately, tactics will have to be formalized for use by the acquisition assistant we are designing. In parallel with the reuse tactics suggested in this paper, we are also working on analogical acquisition techniques where requirements about similar systems are retrieved and transposed [10].

Acknowledgments

Thanks are due to Bruno Delcourt and Françoise Dubisy for their contribution to the first approximation of the KAOS meta-model. We are also grateful to John Anderson, Robert Darimont, Brian Durney, Martin Feather, Anthony Finkelstein, Jacques Hagelstein, Rob Helm, Jeff Kramer, Philippe Massonet, and Bill Robinson for useful questions and suggestions on this work. Constructive comments by the reviewers on the presentation of this paper are also gratefully acknowledged. The KAOS project was supported by the Belgian Ministry of Scientific Affairs in the context of the Belgian Incentive Program for Fundamental Research in Artificial Intelligence under grant no. RFO/AI/13. The third author has been supported by NSF grant no. CCR-8804085.

References

[1] J. Anderson, Private communication.

[2] E. Astesiano and M. Wirsing, An introduction to ASL, in: *Proceedings IFIP WG 2.1 Working Conference on Program Specifications and Transformation* (North-Holland, Amsterdam, 1986).

[3] R.M. Balzer, D. Cohen, M.S. Feather, N.M. Goldman, W. Swartout and D.S. Wile, Operational specification as the basis for specification validation, in: D. Ferrari, M. Bolognani and J. Goguen, eds., *Theory and Practice of Software Technology* (North-Holland, Amsterdam, 1983) 21-50.

[4] J. Benett, A knowledge-based system for acquiring the conceptual structure of a diagnostic expert system, *J. Automated Reasoning* **1** (1985) 49-74.

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

[6] P. Chen, The entity relationship model—towards a unified view of data, *ACM Trans. Database Systems* **1** (1) (1976) 9-36.

[7] A. Dardenne, S. Fickas and A. van Lamsweerde, Goal-directed concept acquisition in requirements elicitation, in: *Proceedings 6th International Workshop on Software Specification and Design*, Como, Italy (1991) 14-21.

[8] R. Davis, Teiresias: applications of meta-level knowledge, in: R. Davis and D. Lenat, eds., *Knowledge-Based Systems in Artificial Intelligence* (McGraw-Hill, New York 1982) 227-490.

[9] E. Doerry, S. Fickas, R. Helm and M.S. Feather, A model for composite system design, in: *Proceedings 6th International Workshop on Software Specification and Design*, Como, Italy (1991) 216-219.

[10] F. Dubisy and A. van Lamsweerde, Requirements acquisition by analogy, Report No. 13, KAOS Project, Institut d'Informatique, Facultés Universitaires de Namur, Belgium (1992).

[11] E. Dubois and A. van Lamsweerde, Making specification processes explicit, in: *Proceedings 4th International Workshop on Software Specification and Design*, Monterey, CA (1987) 169-177.

[12] E. Dubois, J. Hagelstein and A. Rifaut, A formal language for the requirements engineering of computer systems, in: A. Thayse, ed., *Introducing a Logic Based Approach to Artificial Intelligence*, Vol. 3 (Wiley, New York, 1991) 357-433.

[13] T. Ellman, Explanation-based learning: a survey of programs and perspectives, *ACM Comput. Surv.* **21** (2) (1989) 163-222.

[14] M.S. Feather, Language support for the specification and development of composite systems, *ACM Trans. Programming Languages Systems* **9** (2) (1987) 198-234.

[15] M.S. Feather, Constructing specifications by combining parallel elaborations, *IEEE Trans. Softw. Engrg.* **15** (2) (1989) 198-208.

[16] J. Fiadeiro and A. Sernadas, The INFOLOG linear tense propositional logic of events and transactions, *Inform. Systems* **11** (1986) 61-85.

[17] S. Fickas and P. Nagarajan, Critiquing software specifications, *IEEE Software* (November 1988) 37-46.

[18] J.P. Finance, J. Souquieres, A. van Lamsweerde, P. Du Bois and J. Hagelstein, First version of a model for the requirements development process, Intermediate Deliverable, ESPRIT Project 2537 (1991).

[19] A. Finkelstein and H. Fuks, Multi-party specification, in: *Proceedings 5th International Workshop on Software Specification and Design*, Pittsburgh, PA (1989) 185-195.

[20] A. Finkelstein and C. Potts. Building formal specifications using structured common sense, in: *Proceedings 4th International Workshop on Software Specification and Design*, Monterey, CA (1987) 108-113.

[21] M.C. Gaudel, Towards structured algebraic specifications, in: *ESPRIT'85 Status Report* (North-Holland, Amsterdam, 1986) 493-510.

[22] C. Ghezzi, D. Mandrioli and A. Morzenti, TRIO: a logic language for executable specifications of real-time systems, *J. Systems Softw.* (1990).

[23] S.J. Greenspan, Requirements modelling: a knowledge representation approach to software requirements definition, Ph.D. Thesis, Report CSRG-155, University of Toronto, Toronto, Ont. (1984).

[24] S.J. Greenspan, A. Borgida and J. Mylopoulos, A requirements modeling language and its logic, in: M.L. Brodie and J. Mylopoulos, eds., *On Knowledge-Based Management Systems* (Springer, Berlin, 1986) 471-502.

[25] J.V. Guttag and J.J. Horning, Report on the LARCH shared language, *Sci. Comput. Programming* **6** (1986) 24-36.

[26] R.P. Hall, Computational approaches to analogical reasoning: a comparative analysis, *Artificial Intelligence* **39** (1989) 39-120.

[27] K.M. Hansen, A.P. Ravn and H. Rischel, Specifying and verifying requirements of real-time systems, in: *Proceedings ACM SIGSOFT'91 Conference on Software for Critical Systems, ACM Softw. Engrg. Notes* **16** (5) (1991) 44-54.

[28] D. Harel, Statecharts: a visual formalism for complex systems, *Sci. Comput. Programming* **8** (1987) 231-274.

[29] R. Hull and R. King, Semantic database modeling: survey, applications and research issues, *ACM Comput. Surv.* **19** (3) (1987) 201-260.

[30] C.B. Jones, *Systematic Software Development Using VDM* (Prentice-Hall, Englewood Cliffs, NJ, 2nd ed., 1990).

[31] G. Kahn, B. Lang, B. Mélèse and E. Morcos, Metal: a formalism to specify formalism, *Sci. Comput. Programming* **3** (1983) 151-188.

[32] S.E. Keller, L.G. Kahn and R.B. Panara, Specifying software quality requirements with metrics, in: R.H. Thayer and M. Dorfman, eds., *Tutorial: System and Software Requirements Engineering* (IEEE Computer Society Press, Silver Spring, MD, 1990) 145-163.

[33] R. Koymans, J. Vytopil and W.P. de Roever, Real-time programming and asynchronous message passing, in: *Proceedings 2nd ACM Conference on Principles of Distributed Computing*, Montreal, Que. (1983).
[34] B. Meyer, On formalism in specifications, *IEEE Software* **2** (1) (1985) 6-26.
[35] J. Mylopoulos, L. Chung and B. Nixon, Representing and using nonfunctional requirements: a process-oriented approach, *IEEE Trans. Softw. Engrg* **18** (6) (1992) 483-497.
[36] J. Mostow, A problem solver for making advice operational, in: *Proceedings AAAI-83*, Washington, DC (1983) 279-283.
[37] N.J. Nilsson, *Problem-Solving Methods in AI* (McGraw-Hill, New York, 1971).
[38] H.B. Reubenstein and R.C. Waters, The requirements apprentice: automated assistance for requirements acquisition, *IEEE Trans. Softw. Engrg.* **17** (3) (1991) 226-240.
[39] W.N. Robinson, Integrating multiple specifications using domain goals, in: *Proceedings 5th International Workshop on Software Specification and Design*, Pittsburgh, PA (1989) 219-226.
[40] W.N. Robinson, Negotiation behavior during requirement specification, in: *Proceedings 12th International Conference on Software Engineering* (1990) 268-276.
[41] G.-C. Roman, A taxonomy of current issues in requirements engineering, *IEEE Comput.* **2** (April 1985) 14-22.
[42] J.M. Spivey, *The Z Notation* (Prentice-Hall, Englewood Cliffs, NJ, 1989).
[43] W. Swartout, XPLAIN: a system for creating and explaining expert consulting programs, *Artificial Intelligence* **21** (1983) 285-325.
[44] A. van Lamsweerde, Learning machine learning, in: A. Thayse, ed., *Introducing a Logic Based Approach to Artificial Intelligence*, Vol. 3 (Wiley, New York, 1991) 263-356.
[45] A. van Lamsweerde, A. Dardenne, B. Delcourt and F. Dubisy, The KAOS project: knowledge acquisition in automated specification of software, in: *Proceedings AAAI Spring Symposium Series, Design of Composite Systems*, Stanford, CA (1991) 59-62.
[46] A. van Lamsweerde, A. Dardenne and F. Dubisy, KAOS knowledge representations as initial support for formal specification processes, Report RR-91-8, Unité d'Informatique, University of Louvain, Louvain-la-Neuve, Belgium (1991).
[47] A. van Lamsweerde, B. Delcourt, E. Delor, M.C. Schayes and R. Champagne, Generic lifecycle support in the ALMA environment, *IEEE Trans. Softw. Engrg.* **14** (6) (1988) 720-741.
[48] R. Waldinger, Achieving several goals simultaneously, in: E. Elcock and D. Michie, eds., *Machine Intelligence* **8** (Ellis Horwood, Chichester, England, 1977).
[49] J.M. Wing, A study of 12 specifications of the library problem, *IEEE Software* (July 1988) 66-76.
[50] P. Zave, An operational approach to requirements specification for embedded systems, *IEEE Trans. Softw. Engrg.* **8** (3) (1982) 250-269.
[51] M. Zeleny, *Multiple Criteria Decision Making* (McGraw-Hill, New York, 1982).