A UML profile to support requirements engineering with KAOS

William Heaven and Anthony Finkelstein

Department of Computer Science, University College London, London WC1E 6BT
{w.heaven, a.finkelstein}@cs.ucl.ac.uk

Abstract

One of the most important approaches to requirements engineering of the last ten years is the KAOS model as presented in [1] and [2]. We introduce a profile that allows the KAOS model to be represented in the UML. The paper includes an informal presentation of the profile together with a full account of the new stereotypes and tags. We also outline an integration of requirements models with lower level design models in the UML, leading to a uniform and comprehensive specification document. A UML profile can increase the usefulness of KAOS. A method can be truly successful only if a large number of professionals are sufficiently convinced of its potential to use it in industrial cases. Use of the UML to support requirements engineering with KAOS may help achieve this end.

1 Introduction

1.1 Requirements engineering and the UML

Requirements engineering is helped enormously by methods that guide a practitioner in the task of identifying the requirements of the system-to-be. One of the most important approaches to requirements engineering of the last ten years is the KAOS model as presented in [1] and [2]. We introduce a profile that allows the KAOS model to be represented in the UML. The ability to model the KAOS approach in the UML offers the potential for extra tool support. A tool has been developed that supports KAOS [3], but it uses the non-standard graphical notation specifically developed for the approach. The notation and tool are not familiar to many. On the other hand, UML editors are ubiquitous in the software industry, and many can be updated to recognize new profiles. Furthermore, by modelling KAOS in the UML – an industry standard – we remove the need for a software engineer to master a new graphical notation in order to employ KAOS methods.

Additionally, UML documentation of the requirements engineering process will sit more comfortably with all other UML documentation for a software project. For purposes of traceability between models, integrated documentation of this sort is highly desirable. The UML currently provides no explicit support for the requirements phase of a project. Indeed, an exclusively object-oriented framework creates difficulties in mapping requirements to lower level entities in a design model, since requirements typically capture concerns that transgress class boundaries at the design level. We do not address this problem, but we note how our profile might be complemented by subject-oriented design techniques in the UML. A subject-oriented design approach that provides mechanisms for arranging UML classes by concern, together with a profile for representing requirements in the UML, would facilitate the mapping from one domain to the other. A requirements model and a design model would both be instances of a single UML meta-model.

The claim for our contribution is modest: a UML profile can increase the usefulness of KAOS. A method can be truly successful only if a large number of professionals are sufficiently convinced of its potential to use it in industrial cases. Use of the UML to support requirements engineering with KAOS may help achieve this end.

1.2 KAOS

The elucidation and manipulation of goals is a natural part of doing requirements engineering: requirements by their very nature present targets to be reached. Previous requirements engineering techniques have focused foremost on either entities or activities. Goal-oriented methodologies give primary place to the intentional – goals – in the development of software systems, with entities and activities now determined via the identification of goals. Consideration of goals leads to the exploration of alternative system designs – "One can state a goal without having to specify how it is to be achieved" [4] – thus helping the designer build a better system.

Perhaps the most successful goal-oriented method is KAOS ([1] and [2]). KAOS is *goal-driven*: having identified a few preliminary goals for a system-to-be, the KAOS framework facilitates the identification of further goals – and the requisites, objects, agents, and actions of the system. KAOS supplies a rich ontology for capturing and modelling requirements. Its meta-level is essentially a taxonomy of concepts that guide the identification of requirements and their relationships. **Goals** are taken to describe expected properties of the system, an example being "Achieve Ambulance Intervention". The KAOS framework asks an analyst to define this network of concepts for a given domain. Goals are analyzed and better understood by identifying higher-level goals – goals that will explain why this goal is desired – and they are more clearly defined by reducing them to subgoals. By identifying the relationships a goal has with other goals, a goal graph is developed. Goal graphs are semantic networks of AND/OR/XOR relationships between goals where a goal is either **reduced** by a conjunction of subgoals or an (exclusive) disjunction. **Conflict** relationships between goals may also be traced.

The leaves of goal graphs are **requisites** – goals that cannot be further reduced and can be assigned as the **responsibility** of individual **agents**. A requisite is either assigned to a software agent and taken as a **requirement**, or to an agent in the domain and taken as an **assumption**. Distinguishing requirements from assumptions identifies the boundary between software and environment for the system-to-be. Intuitively, requirements are *prescriptive* statements and assumptions *descriptive.* Requisites are **operationalized** by **actions** performed by the agents responsible for each requisite in such a way that the composite system satisfies the goals. Agents may also **monitor** and **control** certain **objects** that are identified when defining goals. These objects can be the **inputs** and **outputs** of actions. Goals **concern** objects, and objects **ensure** a requisite if they take part in an action that operationalizes that requisite. Finally, identifying possible obstacles to a goal allows one to reformulate the goal graph and strengthen the requirements specification by choosing alternative reduction paths or agent assignations, or by introducing new goals to mitigate the obstacle. The whole method roughly consists of

1. identifying and reducing goals until requisites are identified;
2. identifying objects from goal descriptions; and
3. assigning the requisites, objects, and actions to agents.

Many new goals may be identified by employing established patterns of goal-type. For example, a goal may be of the "Achieve such-and-such" or "Maintain such-and-such" form. The form of a goal can determine how it may be reduced according to patterns (see [1] and [2]). These patterns hide from the requirements engineer formal groundwork describing relationships between goals. A formal language with real-time temporal operators is provided for when formalization is necessary, but the methodology discourages it's overuse. There is a time and place for formalization: on many occasions we cannot be certain of the correctness of a model without it, but it can become a hindrance where its use is redundant. Instances of the KAOS concepts goal, requisite, and action – that is, goals, requisites, and actions identified for a particular domain – can be given a formal definition. Tactics for selecting between alternative goal reductions and assignments are also provided.

1.3 KAOS and the UML

The KAOS model lends itself well to representation in the UML. Its three-tiered structure involves a meta-level, a domain level, and an instance level that respectively mirror the meta-model layer, model layer, and user model (instance) layer of the UML. Entities in a lower level are instances of entities in the level above. Moreover, a KAOS model is a semantic network in which the nodes are concepts and the connections between nodes are associations between the concepts. Again this parallels the UML. There are two parts to a KAOS model: a graphical representation of the semantic network together with a supplementary textual definition, and the optional formal representation expressed in a temporal logic. The profile introduces some new stereotypes and tags. The extension allows us to model the KAOS semantic network graphically in the UML using stereotyped classes and associations, and to represent the informal and formal descriptions using tags of these stereotyped classes and associations. Formal expressions need to be rewritten using only ASCII characters since most UML editors do not support the symbols of the KAOS temporal logic. In some cases, it may be useful to capture a formal expression in the OCL — where the expression directly constrains a diagram of a model, for instance — and the OCL might be extended with temporal keywords to this end. However, in most cases, formalization is for the purpose of reasoning about goal conflicts or obstacles, and for this the OCL is probably not the best medium for formalization.

The UML is extended by introducing new stereotypes to the language. These stereotypes are applied to existing UML entities, such as classes and associations, and augment the semantics of these standard elements with newly defined meanings, allowing one to model previously unsupported concepts. A particular extension of the UML is packaged as a profile. For heuristic purposes, we can think of a profile as an extension of the UML meta-model layer – the layer at which the UML's concepts of class, object,

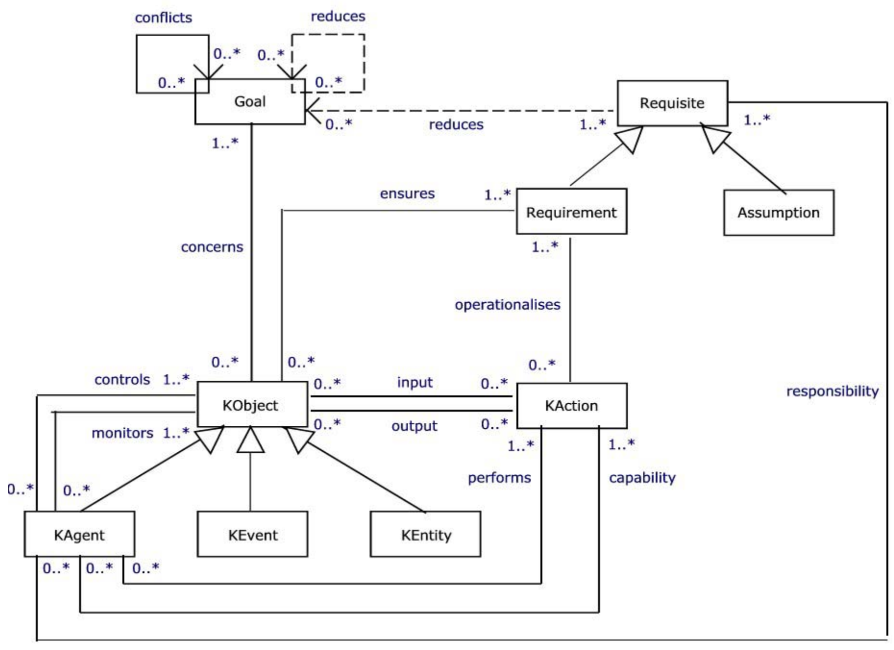


Fig. 1. Profile metamodel

association, etc are defined: "The stereotype concept provides a way of branding model elements so that they behave in some respects as if they were instances of new virtual meta-model constructs" ([5]). At the UML's model level, we can think of a class stereotyped with <<goal>> as both an instance of Class by classification and an instance of Goal (a virtual extension of the Class construct) by stereotyping. In this way we provide extra categorization for elements in our model. A stereotype may have attributes that can be modelled as the tags of a stereotyped class. New tags may also be introduced as independent elements. Figure 1 shows the meta-model of the proposed profile with minimal syntax, omitting the attributes of the classes for readability. The stereotype <<reduces>> is a specialization of the existing UML abstraction stereotype <<refines>> (an abstraction is a specialization of a dependency).

The rest of the paper is set out as follows. Section 3 is an informal presentation of our profile using examples from a case study of Letier's ([2]) that is based on the well-known software fiasco of the London Ambulance Service, and Section 4 outlines the potential for integrating a goal-driven requirements model with lower level design models. A fuller case study is presented in Section 5; some related work is covered in Section 6; and Section 7 concludes the paper. An appendix provides a full account of the stereotypes and tags of the profile.

In the course of presenting the profile, we are going to refer to concepts from the KAOS method, on one hand, and the UML on the other. Entities from KAOS are modelled by the stereotyped classes of a UML model. When talking about KAOS concepts we use normal font, for example, "a goal can be reduced by another goal". When talking about UML stereotypes, we use a sans serif font, and stereotype names will be enclosed in guillemets, for example, "a <<goal>> class can be linked with another <<goal>> class by a <<reduces>> association".

2 The stereotypes of the KAOS profile

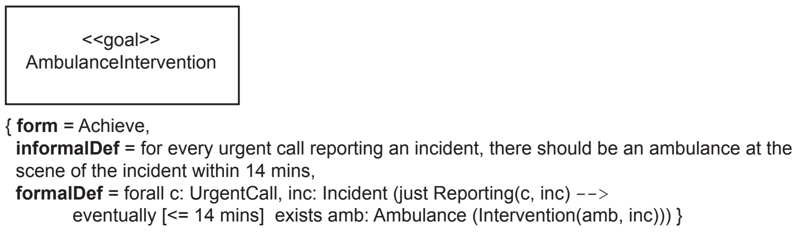
Our UML KAOS profile will be presented by means of examples taken from a case study found in [2]. The case study concerns a part of the London Ambulance Service's failed ambulance dispatching system that covers the handling of urgent calls.

2.1 Goals

An initial goal that an ambulance is to arrive at the scene of an incident within 14 minutes of the incident being reported can be identified at an early stage:

Goal
  Achieve[AmbulanceIntervention]
InformalDef
  For every urgent call reporting an incident, there should be an ambulance at the scene of the incident within 14 mins
FormalDef
  $\forall\, c : \text{UrgentCall}, \, inc : \text{Incident} \; (@ \; \text{Reporting}(c, inc) \Rightarrow$
    $\Diamond_{\leq 14 \text{ mins}} \; \exists\, amb : \text{Ambulance} \; (\text{Intervention}(amb, inc)))$

In the UML we can represent this as



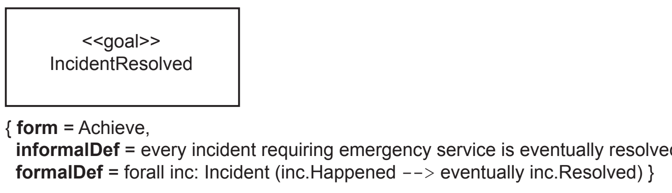
{ **form** = Achieve,
informalDef = for every urgent call reporting an incident, there should be an ambulance at the scene of the incident within 14 mins,
formalDef = forall c: UrgentCall, inc: Incident (just Reporting(c, inc) -->
    eventually [<= 14 mins] exists amb: Ambulance (Intervention(amb, inc))) }

– that is, as a class stereotyped by <<goal>> together with some tagged values – one specifying an informal definition, another a formal definition, and one indicating what type of goal this is. In this case we have an "Achieve" goal. We use three tags of the <<goal>> stereotype: form, informalDef, and formalDef.

The tag form can take one of the values from the set {Achieve, Maintain, Avoid, Cease, Minimize, Maximize}, while informalDef and formalDef take a string. The formalDef string is an expression in KAOS temporal logic, with the temporal operators written using ASCII text. The <<goal>> stereotype also has a priority attribute to help with conflict resolution, an instanceOf attribute, and a boolean soft attribute indicating whether or not the goal can be formalized (false) or not (true). Values of priority are floats between 0 and 1. A soft goal is usually an "Optimize" goal – either a "Maximize" or a "Minimize" goal. Values of instanceOf are taken from the set {satisfactiongoal, safetygoal, securitygoal, informationgoal, accuracygoal}. A goal with an instanceOf value of safetygoal must have a priority of 1.

The KAOS meta-level contains the high-level concepts of the KAOS method such as Goal and Action. Instances of these concepts are modeled at the domain level by particular goals and actions, such as Achieve[IncidentResolved] and AllocateAmbulance. At the instance level of the KAOS modelling hierarchy we can then model specific cases involving the goals and actions (and so on) of the domain. We keep this three-tiered hierarchy in our profile. KAOS concepts are introduced as stereotypes in the UML meta-model, instances of these concepts are modeled at the model layer as stereotyped classes, and instances of these classes can be modeled at the user model layer as objects of these classes. For example, given an Ambulance class of <<kentity>> stereotype, an EmergencyCall class of <<kevent>> stereotype, and a Dispatcher class of <<kagent>> stereotype (modeled at the domain level), we can model at the instance level a specific ambulance allocation involving an instance of the action AllocateAmbulance and instances of the Ambulance, EmergencyCall, and Dispatcher classes: AllocateAmbulance(ambulance1,event1,dispatcher1). However, we should note that it makes little sense for there to be many instances of goal or requisite classes at the user model layer. While we may model several specific ambulances participating in several specific ambulance allocations, we do not want several specific instances of the Achieve[IncidentResolved] goal at that level: all of the particular cases of the AllocateAmbulance action we might model will be motivated by the same goal. We should therefore think of each class of <<goal>> (and <<requisite>>) stereotype as a singleton, that is, as only being instantiated once per model.

Asking WHY questions about goals can often lead to the identification of higher level goals. By asking a WHY question about Achieve[AmbulanceIntervention], we might be led to the higher level goal Achieve[IncidentResolved]:



{ **form** = Achieve,
informalDef = every incident requiring emergency service is eventually resolved,
formalDef = forall inc: Incident (inc.Happened --> eventually inc.Resolved) }

The identification of this goal in turn drives the identification of the assumptions Achieve[IncidentReported] and Achieve[IncidentResolvedByIntervention]:

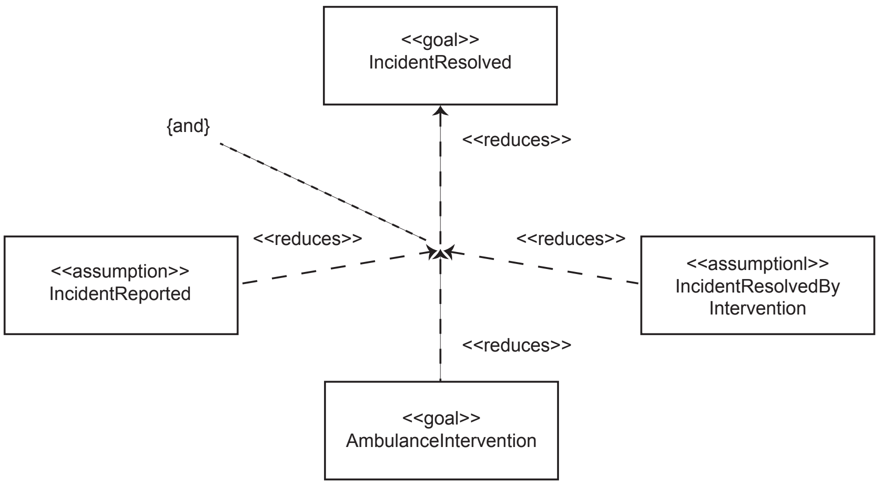
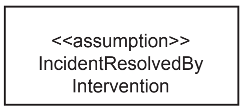


Fig. 2. One way to represent goal reduction

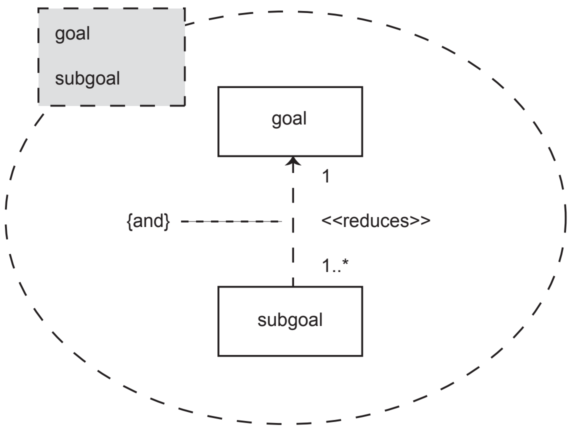
{ **form** = Achieve,
informalDef = every incident requiring emergency service is eventually reported to the LAS,
formalDef = forall inc: Incident (inc.Happened -->
eventually exists c: UrgentCall (Reporting(c, inc)) }



{ **form** = Achieve,
informalDef = an incident is resolved by the intervention of a single ambulance,
formalDef = forall amb: Ambulance, inc: Incident (Intervention(amb, inc) -->
eventually inc.Resolved) }

We might show the relationships between these goals and assumptions in the UML as in Figure 2. Using abstraction associations (an abstraction is a specialization of a dependency) stereotyped by <<reduces>> we show that IncidentResolved is reduced by the conjunction of IncidentReported, AmbulanceIntervention, and IncidentResolved-ByIntervention. An and tag appropriately constrains the branching of the abstraction links, indicating that the reduction of IncidentResolved is achieved collectively. Alternative tags of or and xor can be used to constrain such branching to indicate disjunction and exclusive disjunction respectively. The xor tag is already defined in the UML. We introduce the companion tags and and or.

However, where many classes are involved in a reduction relationship, this notation can soon make a diagram too cluttered to be of much use. An alternative is to use the



AND-Reduction Pattern Constraints:

– goal and subgoal roles are played by instances of <<goal>>, <<requirement>>, or <<assumption>>, or by another AND/OR/XOR-Reduction collaboration
– a goal g is reduced by a set S of subgoals if the satisfaction of S is sufficient for satisfying g
– a goal g is minimally reduced by a set S of subgoals if g is reduced by S but not by S / $\{sg\}$ for any subgoal sg in S

Fig. 3. AND-Reduction collaboration template

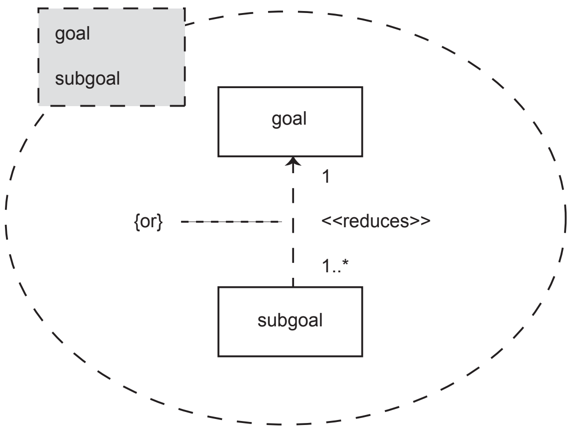
UML's collaboration notation. A collaboration template, or *pattern*, is defined for each of AND-reduction, OR-reduction, and XOR-reduction as Figures 3–5 illustrate.

Graphically, our collaboration templates are defined within a dotted ellipse, and the roles involved in the collaboration have been declared in a dotted box intersecting the ellipse. Supplementary text is provided. In the model, it is then simply a matter of specifying the classes that are to fulfill the defined roles in each instance (the classes must of course be of the correct type for the role). Figure 6 demonstrates how the reduction of the goal IncidentResolved can be shown using a collaboration template. Note that the links between classes and collaboration are labelled with the roles defined for that collaboration and not with a stereotype. AND/OR/XOR relationships for assignment of responsibility to agents and for operationalization can be modelled in a similar way.

Goals may also conflict with other goals. We represent this in the UML using a <<conflicts>> association between the conflicting goals.

2.2 Requisites

A goal that can be reduced no further and is assignable to an individual agent either in the domain or in the software-to-be is a requisite. A requisite assigned to an agent in the software-to-be is a requirement, while a requisite effectively assigned to an agent



OR-Reduction Pattern Constraints:

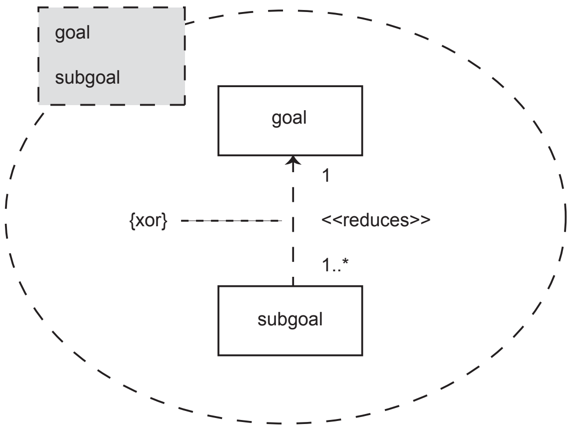
- goal and subgoal roles are played by instances of <<goal>>, <<requirement>>, or <<assumption>>, or by another AND/OR/XOR-Reduction collaboration
- a goal g is reduced by a set S of subgoals if the satisfaction of $g_1, .., g_n$ (for $1 \leq n \leq |S|$ and where $\{g_1, .., g_n\}$ is a subset of S) is sufficient for satisfying g

Fig. 4. OR-Reduction collaboration template

in the domain is an assumption. The assumption IncidentReported can be more fully modelled in the UML again using tagged values:

{ **form** = Achieve,
informalDef = an incident is resolved by the intervention of a single ambulance,
formalDef = forall amb: Ambulance, inc: Incident (Intervention(amb, inc) -->
eventually inc.Resolved,
category = SatisfactionGoal }

In many ways requisites are like goals, but we are unable to model them at the meta-level as specializations of the <<goal>> stereotype, since we do not want to inherit the <<reduces>> association. The <<requirement>> and <<assumption>> stereotypes have the attributes form, informalDef, formalDef, priority, and category. In a model, the value of category will be the value of the instanceOf tag of the goal that this requisite reduces. The <<requirement>> stereotype adds the tag specialization, which takes a value from the set {hard, soft}. A hard requirement may never be violated, while a soft one may be violated temporarily. A requirement with a category of safetygoal cannot be soft. Again, like classes of <<goal>> stereotype, <<requisite>> classes are singletons.



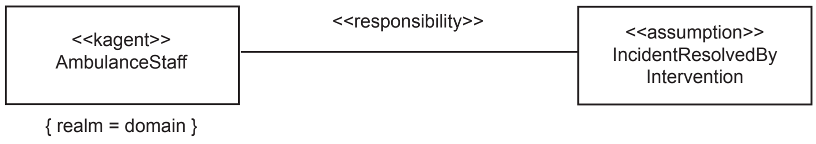
XOR-Reduction Pattern Constraints:

- goal and subgoal roles are played by instances of <<goal>>, <<requirement>>, or <<assumption>>, or by another AND/OR/XOR-Reduction collaboration
- a goal g is reduced by a single subgoal sg from the set S of alternative subgoals if the satisfaction of sg is sufficient for satisfying g (the <<reduces>> association here indicates potential alternative reductions of the goal, where only one alternative may be selected in the model)

Fig. 5. XOR-Reduction collaboration template

2.3 Agents, Entities, and Events

We can model the responsibility assignments of requisites to agents in the UML as follows:



As well as the tags of <<kobject>>, <<kagent>> has a realm tag that can be used to indicate whether the agent is a domain agent or an agent in the software-to-be. An alternative would of course be to introduce two stereotypes – <<softwareagent>> and <<domainagent>> – but for reasoning purposes it is arguably easier to consider agents (both software and domain) in a system-to-be as a whole.

An agent cannot be represented by an actor in the UML. An actor generally represents someone or something that is outside the system-to-be; an actor interacts with the system, but is not part of it. So far this rules out agents in the software. But we must also note that actors are *users* of a system's functionality, while agents perform the actions that *produce* this functionality. The one cannot be mapped to the other.

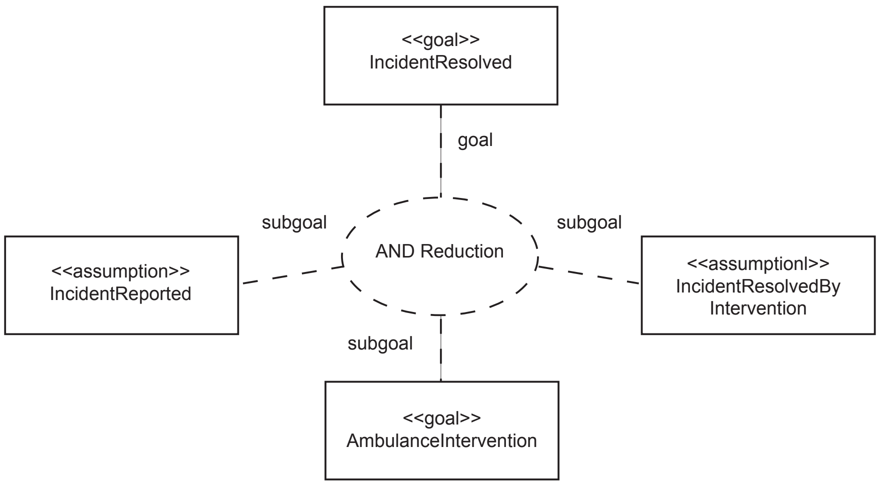


Fig. 6. Another way to represent goal reduction

The identification of objects is driven by the definition of goals. Here, we are of course talking of 'objects' in KAOS terminology, as in *things of interest to the system.* From the goals we have identified and defined above, we can draw a partial object model diagram in the UML as in Figure 7. If we had wanted to express the formal definition of any of the goals we've met so far in the OCL, we would have had to explicitly model these objects before now. An OCL expression refers to elements in a diagram; so, in order to refer to the objects Ambulance or Incident, we would have to depict them in a diagram. The OCL expression would then concern a model of the goal we are defining formally and it's relationships to the objects Ambulance and Incident etc.

Like the <<kagent>> stereotype, <<kentity>> and <<kevent>> are both specializations of <<kobject>>. Tags common to these stereotypes are informalDef, invariant, and strengthenedInv. The <<kevent>> stereotype has an additional frequency property. The invariant of an object is a domain property concerning the object and is assumed to hold for the purposes of designing a system. The strengthening of this invariant is used to ensure the satisfaction of whatever goal concerns the object. The value of a frequency tag in the model specifies a time interval between consecutive occurrences of an event. The classes in Figure 7 have no tags shown. The relationships of the KAOS model — here, Intervention, Mobilization, and Reporting — can of course straightforwardly be modelled by associations in the UML.

Objects can be the concern of goals. This relationship is modelled with a <<concerns>> association. All this really declares is that the object is referenced in the definition of the goal. If an object's strengthened invariant contributes to the satisfaction of a requirement it is said to ensure that requirement. This relationship is modelled with an <<ensures>> association.

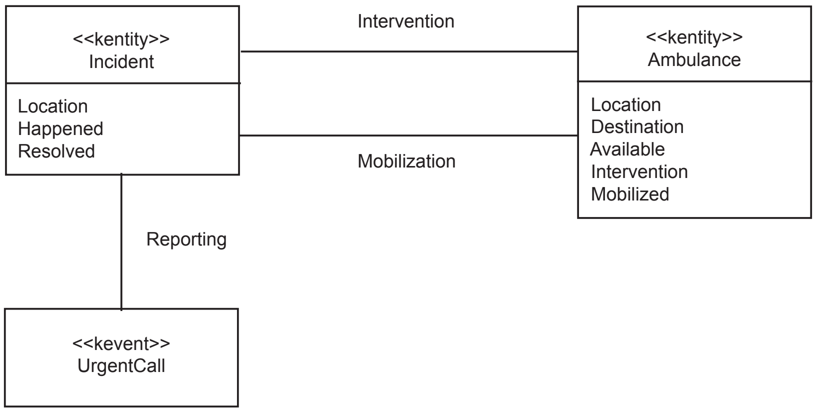
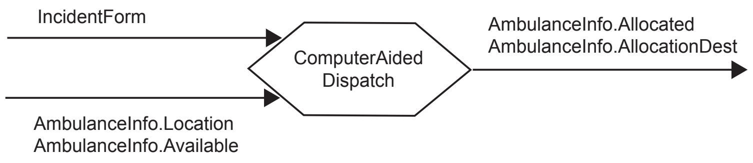


Fig. 7. Partial object model

2.4 Actions

If an agent is responsible for a requirement it must meet certain conditions. The agent must perform the action that operationalizes the requirement. To do this it must monitor any object that is input to the action and control any object that is output from the action. An action is shown in the graphical notation of KAOS like this:



ComputerAidedDispatch is an agent. An arrow hitting an agent indicates that the agent monitors the object (or attributes) labelling the arrow. An arrow leaving an agent indicates that the agent controls the object (or attributes) labelling the arrow. The diagram is supplemented by explanatory text:

Action
  AllocateAmbulance
PeformedBy
  ComputerAidedDispatch
Input
  IncidentForm {**arg** if}
  AmbulanceInfo {**arg** ai}/ Location, Available
Output
  AmbulanceInfo {**res** ai}/ Allocated, AllocationDest
DomPre
  $\neg(ai.\text{Allocated} \wedge ai.\text{AllocationDest} = if.\text{Location})$
DomPost

ai.Allocated $\wedge$ ai.AllocationDest = if.Location)

ReqTrigFor

AllocationBasedOnIncidentFormAndAmbulanceInfoWhenNearAmbulanceAvailable,
$\blacksquare_{\leq \text{ allocationdelay}}$ if.Encoded

ReqPreFor

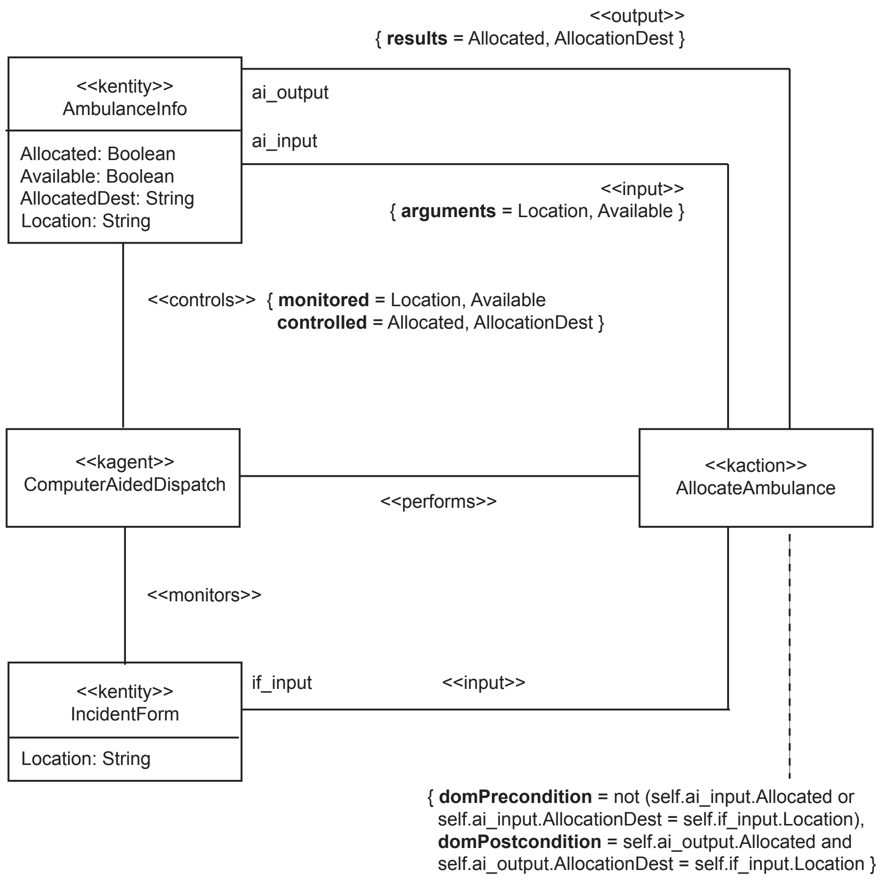
AllocationBasedOnIncidentFormAndAmbulanceInfoWhenNearAmbulanceAvailable,
(ai.Available $\wedge$ $\neg ai$.Allocated $\wedge$ TimeDist(ai.Location, if.Location) = 11mins) $\vee$
$\neg\ \exists\ x$: AmbulanceInfo
((x.Available $\wedge$ $\neg x$.Allocated $\wedge$ TimeDist(x.Location, if.Location) = 11mins))

The diagram and text together represent the action. An equivalent in the UML is shown in Figures 8 and 9. Figure 9 simply covers the aspects of the action's specification in relation to the goal it operationalizes – here, these are the required trigger condition and the required post condition. Diagrammatically, these should be related to the association between the action and the goal.

A principal difference of the UML representation is that the action itself is shown graphically as a class of <<kaction>> stereotype, whereas the action in KAOS notation is taken to be the sum of the interactions of the agent. A UML diagram clearly separates the monitors and controls relationships on one hand, and the input and output relationships on the other. Specific attributes of objects are declared in the monitored and controlled tags of the <<monitors>> and <<controls>> stereotypes respectively. Similarly, the arguments and results tags are used to declare the specific attributes of objects that are involved in <<input>> and <<output>> associations respectively. The domain pre- and post-conditions and the required pre-, post-, and trigger conditions are defined in tagged values of a <<kaction>> class. The attributes reqPrecondition, reqPostcondition, and reqTriggercondition specify constraints on the <<operationalizes>> association between action and goal (Figure 9). These diagrams provide greater detail than may be necessary for a model. In many cases, minimal tags are sufficient.

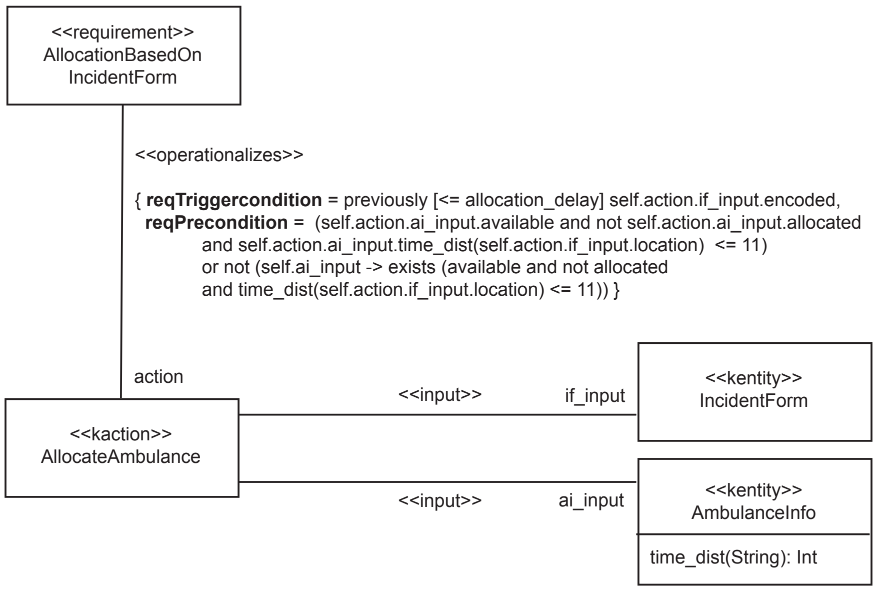
To perform an action, an agent must be capable of performing that action. It is unnecessary to show a <<capability>> association in Figure 8 since an agent performing an action implies its capability to do so. However, it may be useful to model capabilities before determining responsibility relationships to see what can do what. Similarly, if an agent controls an object, it also monitors that object. Essentially, controlling an object is the ability to write that object's attributes, while monitoring an object is the ability to read that object's attributes. Again, it is unnecessary to show a <<monitors>> association between ComputerAidedDespatch and AmbulanceInfo, since controls implies monitors.

In Figure 9, the value of the reqPrecondition tag of the <<operationalizes>> association, given here as an OCL expression, accesses the time_dist() operation of AmbulanceInfo. To express in the OCL the TimeDist() function from the KAOS specification of the action, we have introduced an extra operation into one of the classes modelled. The TimeDist() function in the KAOS specification gives the time it takes to travel between two locations. In our UML model we have invented an operation in the AmbulanceInfo class that takes a location value and returns the distance between AmbulanceInfo's location and the location of the argument. We have also introduced a keyword previously, which can take a temporal qualifier, as an example of how the real time constraints of KAOS may be captured. There is not room to cover an extension of the OCL and we will not elaborate further. However, this would allow us to express fully, using elements



<<output>>
{ **results** = Allocated, AllocationDest }
<<kentity>>
AmbulanceInfo
Allocated: Boolean
Available: Boolean
AllocatedDest: String
Location: String
ai_output
ai_input
<<input>>
{ **arguments** = Location, Available }
<<controls>> { **monitored** = Location, Available
controlled = Allocated, AllocationDest }
<<kagent>>
ComputerAidedDispatch
<<performs>>
<<kaction>>
AllocateAmbulance
<<monitors>>
<<kentity>>
IncidentForm
Location: String
if_input
<<input>>
{ **domPrecondition** = not (self.ai_input.Allocated or
self.ai_input.AllocationDest = self.if_input.Location),
domPostcondition = self.ai_output.Allocated and
self.ai_output.AllocationDest = self.if_input.Location }

Fig. 8. AllocateAmbulance action model



<<requirement>>
AllocationBasedOn
IncidentForm
<<operationalizes>>
{ reqTriggercondition = previously [<= allocation_delay] self.action.if_input.encoded,
reqPrecondition = (self.action.ai_input.available and not self.action.ai_input.allocated
and self.action.ai_input.time_dist(self.action.if_input.location) <= 11)
or not (self.ai_input -> exists (available and not allocated
and time_dist(self.action.if_input.location) <= 11)) }
action
<<kaction>>
AllocateAmbulance
<<input>>
if_input
<<kentity>>
IncidentForm
<<input>>
ai_input
<<kentity>>
AmbulanceInfo
time_dist(String): Int

Fig. 9. AllocateAmbulance action model

of the UML, what we find in a KAOS specification. But again, in many cases, use of ASCII text for formalization is more suitable.

3 Model Integration

The worth of a UML profile for KAOS should be measured by how it benefits the software engineering process. Given that KAOS is a powerful approach to an essential part of this process, the provision of the means to model it in the UML – a ubiquitous modelling notation – is surely beneficial. Potentially, KAOS could reach a wider user-base. However, there is a further bonus. Modelled in the UML, a KAOS model may be incorporated into the rest of a system's UML design documentation, seamlessly linking the documentation for the requirements elicitation part of a project to the whole.

3.1 Requirements Traceability

An obvious result of the interleaving of requirements elicitation and design documentation is the facilitation of requirements traceability. A KAOS model supports backwards requirements traceability through its reduces relationships. By navigating back from a goal along <<reduces>> associations, that goal's purpose can be ascertained by identifying the goal or goals it reduces. Forward requirements traceability is supported through <<operationalizes>>, <<responsibility>>, and <<ensures>> associations between

<<requisite>> classes and <<kaction>>, <<kagent>>, and <<kobject>> classes respectively. However, we might also want to trace forward from a requirement or goal to lower levels of design. We would like to be able to integrate our KAOS model for the system-to-be with other models of the system: use case models, sequence models, or (UML) object models, for example. Application of the UML's existing <<trace>> association would allow us to relate elements between requirements model and design model. The <<trace>> association is used for "tracking requirements and changes across models" ([5]). Signaling interrelations in this way produces a more manageable requirements document. Drawing the connections using UML associations is both simple and clear.

Use of the <<trace>> association is left to the discretion of the modeler. It is syntactically applicable to any model elements, though, of course, its use in some cases is far more meaningful or appropriate than in others. A relationship between a use case and goal, for example, is readily intelligible. Syntactically, it is legal to draw the association between goal and the whole use case model – actor, use cases, and links together – since this is itself a UML element, but we would need to take care, for instance, that our goal indeed concerned an actor.

3.2 Interleaving Models

A more interesting possibility resulting from the profile is the *interleaving* of KAOS models with standard UML models. In this way, the KAOS approach can be fully integrated with the rest of the design documentation for a software system. The KAOS method encourages the consideration of different design alternatives: key relationships such as those of reduction and responsibility assignment can be modelled as (possibly exclusive) disjunctive associations. Alternatives can be modelled and investigated before choosing the most attractive. An analysis of these alternatives using UML models can now more neatly be carried out using the very classes identified and modelled through KAOS. A KAOS object represented in the UML can become a model element in a UML sequence diagram, for instance. Because we can represent it in the UML, a KAOS model may now form part of a comprehensive whole rather than remaining an essentially separate document.

A good example of correlation is the relationship between a use case and the UML model of a KAOS action. We could consider the use case to be an abstracted view of the action and the action model a more detailed description of the use case. Another example is illustrated in the top half of Figure 10. StationPrinter is an agent and modelled as a <<kagent>> class in the KAOS model. We might then model an instance of this class at the design level and there would then be dependency between the two models. Also, in general, events, actions, and agents could all be represented in other UML models such as collaborations, sequences, and state machines. However, the correspondence between elements of a KAOS model and those of design models is not always straightforward. To begin with, the level of abstraction at which an element is represented in a KAOS model will often not coincide with the representation of the corresponding element in a design model. It is unlikely that a CommunicationsInfrastructure agent, for instance, though again, modelled as a <<kagent>> class in the KAOS model, would be represented as a single class at the design level. In this case, the communications architecture would be better modelled by a package of classes.

Differing levels of abstraction are not the only concern. Models of requirements also tend to be structured in a way that is not obviously compatible with the conventional

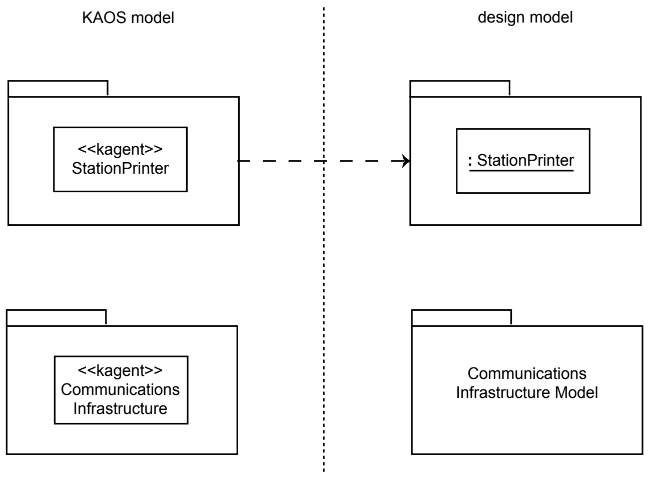


Fig. 10.

object-oriented structure of design models. Requirements specifications are generally concerned with the features and capabilities of a system. These features and capabilities may eventually affect many classes, or conversely, a single class may end up covering several features on its own. Our UML profile for KAOS does not avoid this. A KAOS model, though represented in the UML by means of classes, will still not decompose directly into the class structure of a design model. In modelling requirements as classes, we are still simply modelling these concerns and capabilities. A class representing a general system concern cannot straightforwardly be mapped to a class or classes at design level that represent actual components of a system. The UML offers no explicit support for decomposition of models from requirements.

However, work presented in [6] and [7], which extends object-oriented design in the UML by adding "additional decomposition capabilities that support the direct alignment of design models with requirements" ([7]), may help here. Though explicitly addressing class diagrams, the work seems generally applicable to UML design models. The suggestion is that classes in design models might be arranged by subject in a way that mitigates the UML's object-oriented bias. This *subject-oriented* approach allows the design of different requirements to be modelled in (possibly overlapping) UML design models. Relationships between models governing their composition or merger are specified by stereotyped associations. Augmenting the UML with our profile together with a profile such as the one introduced in [6] would support smooth development of a system in the UML from requirements to implementation. The mapping between requirements model and design model would be facilitated. Using single design models to model each requirement gives us a one-to-one mapping between design model and KAOS model. The agents, actions, and objects of our KAOS model would correspond closely with subsets of the elements in the design model that modelled the requirement in question. An implementation need not follow subject-oriented programming principles, though these may be appropriate in some cases. Subject-oriented design models can be composed so that object-oriented code may be developed from them. This sees

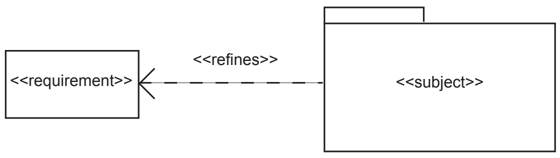


Fig. 11.

the gap between requirements and software bridged. Figure 11 shows a package we should imagine containing a design model of a requirement. The package carries the <<subject>> stereotype ([6]).

4 Case Study

This case study is concerned with the development of a portion of the Advanced Automatic Train Control (AATC) system used in San Francisco's Bay Area Rapid Transit (BART) rail service. The purpose behind this system is to serve more passengers by running trains more closely spaced. The case study is based on that presented in [2] which is in turn taken from an informal description of the AATC specifications in [8]. The aim is to show, by following a path from initial goals to agent assignment, that all necessary documentation of the KAOS process can be captured in the UML using our profile.

The case study is specifically concerned with those aspects of the system needed to control the acceleration and speed of the trains. The problem addressed is the development of an acceleration and speed control system responsible for getting trains running as swiftly and as smoothly as possible within these safety constraints:

1. a train should not enter a closed gate (in the BART system, a gate is not a physical object, but a signal, received by the acceleration and speed control system, that establishes whether a train is allowed to enter a segment of the track)
2. a train should never get so close to a train in front so that if the train in front stopped suddenly — perhaps due to derailment — the following train would hit it
3. a train should stay below the maximum speed the segment of track it is on can handle

4.1 Identification and formalization of primary goals

By searching the problem statement for keywords such as 'purpose', 'objective', 'in order to', 'intent' etc, some initial goals can be identified at an early stage. Figure 12 shows the UML goal graph obtained after a first reading of this document.

The two main goals identified are ServeMorePassengers and Minimize[Costs]. ServeMorePassengers is quickly found to reduce to either NewTracksAdded or TrainsMoreCloselySpaced, while Minimize[Costs] reduces to the conjunction of Minimize[DevelopmentCosts] and Minimize[OperationalCosts]. It is also clear that achieving TrainsMoreCloselySpaced would minimize development costs, and so TrainsMoreCloselySpaced is shown to reduce Minimize[DevelopmentCosts] as well. Moreover, building

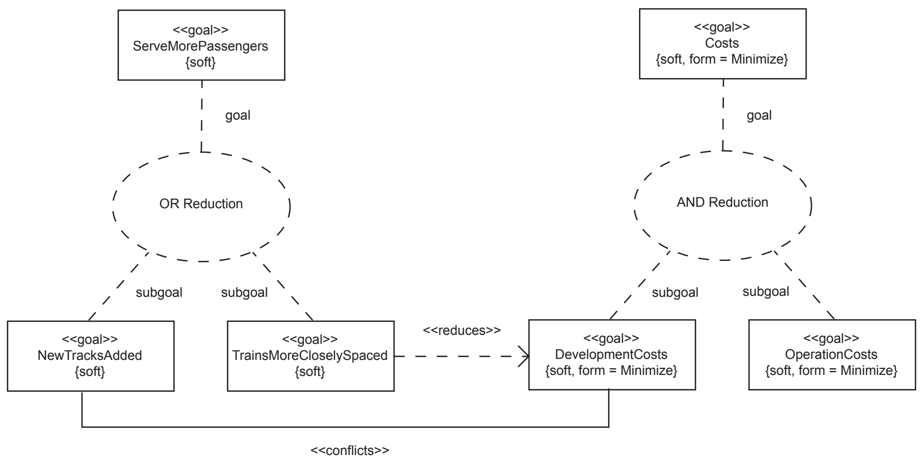
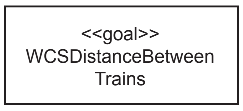


Fig. 12. Preliminary goal graph for the BART system

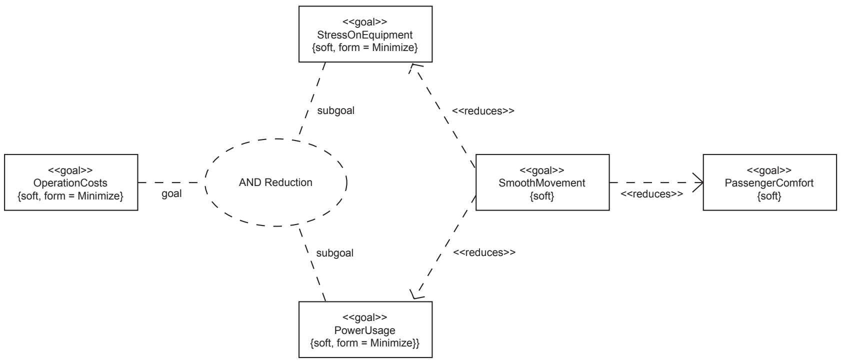
new tracks would be very expensive: NewTracksAdded conflicts with Minimize[DevelopmentCosts]. Minimize[OperationalCosts] is reduced as in Figure 13 resulting in the goal SmoothMovement that reduces the goals that reduce Minimize[OperationalCosts] — Minimize[StressOnEquipment] and Minimize[PowerUsage]. SmoothMovement also reduces a further identified goal, PassengerComfort.

All of the goals in Figures 12 and 13 are soft goals, that is, they are goals that are as yet too vague to be formalized. Since they are not sufficiently defined, it is also not yet fully clear what it would take to satisfy one of these goals. Ordinarily, a reduces relationship holds between a goal and a subgoal if satisfaction of the subgoal is sufficient for satisfaction of the goal it reduces. However, where a reduces relationship exists between goals in which any of the reduced goals are soft, we talk in terms of satisficing rather than satisfying. The concept of satisficing is weaker than satisfying: a lower-level goal is supposed to achieve its parent goal within acceptable limits rather than absolutely [2]. The goals identified so far must be further reduced. An important aspect of the AATC system is safety, so formalization in this case is important.

From the given safety conditions, we can identify three safety goals, Maintain-[WCSDistanceBetweenTrains], Avoid[TrainEnteringClosedGate], and Maintain[Track-SegmentSpeedLimit]. These goals can be formally defined:

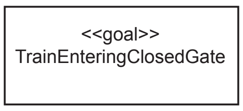


{ **form** = Maintain,
informalDef = a train should never get so close to a train in front so that if the train in front stopped suddenly the following train would hit it: it must maintain Worst Case Stopping distance,
formalDef = forall tr1, tr2: Train (Following(tr1, tr2) --> tr2.Loc - tr1.Loc >= tr1.WCSDist) }

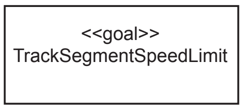


<<goal>> StressOnEquipment {soft, form = Minimize}
subgoal
<<reduces>>
<<goal>> OperationCosts {soft, form = Minimize}
goal
AND Reduction
<<goal>> SmoothMovement {soft}
<<reduces>>
<<goal>> PassengerComfort {soft}
subgoal
<<reduces>>
<<goal>> PowerUsage {soft, form = Minimize}}

Fig. 13. Reduction of Minimize[OperationCosts]



{ **form** = Avoid,
informalDef = a train should not enter a closed gate,
formalDef = forall g: Gate, s: TrackSegment, tr: Train (g.Status = 'closed' and HasGate(s, g)
--> ~ just (~ On(tr, s) }



{ **form** = Maintain,
informalDef = a train should stay below the maximum speed the track segment can handle,
formalDef = forall tr: Train, s: TrackSegment (On(tr, s) --> tr.Speed <= s.SpeedLimit) }

4.2 Identification of Objects

The definition of goals drives the identification of objects. In the definitions of the above three goals we have identified four entities — Train, Track, Gate, and TrackSegment — and three relationships between them — Following, HasGate, and On. The predicate Following is defined formally by

$$\begin{aligned}&\forall\, tr_1, tr_2 : \text{Train } (\text{Following}(tr_1, tr_2) \equiv \\ &\quad \exists\, trk : \text{Track } (\text{OnTrack}(tr_1, trk) \wedge \text{OnTrack}(tr_2, trk) \wedge tr_1.\text{Loc} \leq tr_2.\text{Loc} \\ &\quad \wedge \neg\exists\, tr_3 : \text{Train } (\text{OnTrack}(tr_3, trk) \wedge tr_1.\text{Loc} \leq tr_3.\text{Loc} \wedge tr_3.\text{Loc} \leq tr_2.\text{Loc})))\end{aligned}$$

This definition introduces the OnTrack relationship. A composition relationship between Track and TrackSegment is also introduced. Figure 14 shows a UML object diagram that captures the portion of the KAOS object model identified.

The definitions of goals identified at an early stage often need revising. From these revisions more objects may be uncovered. An example is the goal Avoid[TrainEntering-ClosedGate]. A train cannot stop instantaneously; if a gate closes when a train is too

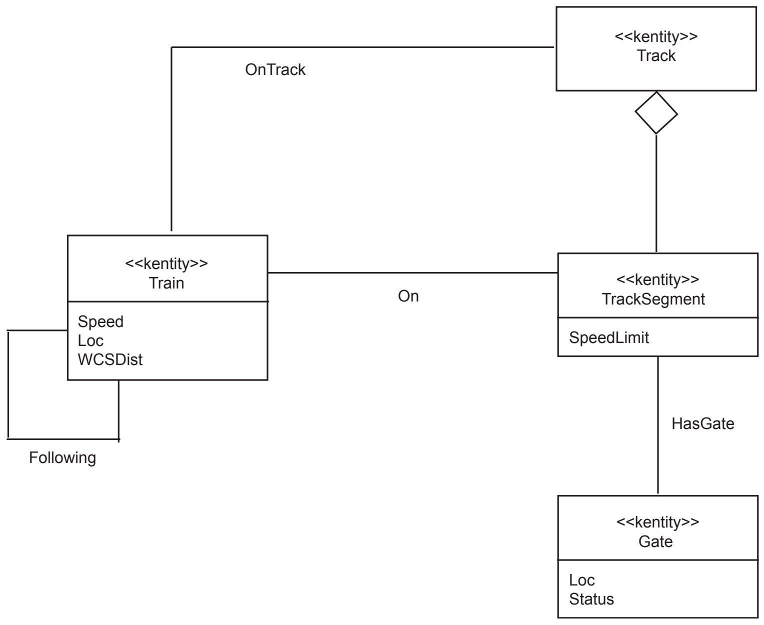
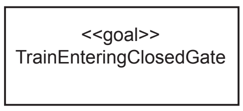


Fig. 14.

close to it to stop then the train must be allowed to enter the gate even though it is closed. We must weaken Avoid[TrainEnteringClosedGate]. This is also an example of the use of the KAOS tactic **weaken goal with unsatisfiable condition**. A weakened Avoid[TrainEnteringClosedGate] is depicted here:



{ **form** = Maintain,
informalDef = a train should not enter a closed gate provided that the gate has been closed when the distance between the train and the gate was more than the worst case stopping distance of the train; if the gate is open when the distance between the train and the gate is less than the worst case stopping distance of the train, the train may ignore the gate, even if it becomes closed later,
formalDef = forall g: Gate, s: TrackSegment, tr: Train

```
        (g.Status = 'closed' backto (g.Loc - tr.Loc)  >=  tr.WCSDist) and HasGate(s, g)  -->
        ~ just (~ On(tr, s)) }
```

This weakening allows trains to enter a closed gate if the gate becomes closed when it is impossible for the train to stop in time. Allowing a train to enter a closed gate need not be unsafe, but this really becomes clear only once the rationale for the original Avoid[TrainEnteringClosedGate] is understood. We need to identify the higher-level goal that this goal reduces by asking WHY Avoid[TrainEnteringClosedGate]. The resulting extension to the goal model is shown in Figure 15.

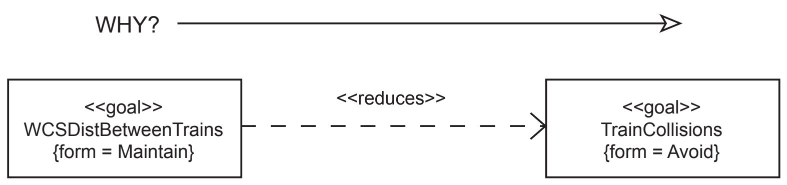
The definition of Maintain[GateClosedInTimeWhenSwitchInWrongPosition] is elicited formally by matching a chain-driven refinement pattern to the formalization of the parent goal Avoid[TrainOnSwitchInWrongPosition] and of the initial goal Avoid[TrainEnteringClosedGate].

Two domain properties can also be identified:

– *every track segment leading to a switch ends with a gate*
  $\forall\ sw$: Switch, s : TrackSegment, trk : Track
  (NextSegmentOnTrack$(trk, s, sw) \Rightarrow \exists\ g$: Gate (HasGate(s, g)))
– *a train enters a switch iff it leaves a track segment preceding the switch*
  $\forall\ sw$: Switch, tr : Train
  (@ On$(tr, sw) \equiv \exists\ trk$: Track, s : TrackSegment (NextSegmentOnTrack(trk, s, sw)
  $\wedge$ @ $\neg$On(tr, s)))

These new goals and domain properties in turn augment the object model as shown in Figure 16. The definition of the domain properties should also be attached to the object model since the properties constrain the objects' behaviour. Binary relationships between objects are modelled using UML associations. NextSegmentOnTrack and ApproachingSwitchOnTrack are ternary relationships and modelled using collaborations with three roles — the Segment role of the NextSegmentOnTrack collaboration has a multiplicity of 2.

We can ask WHY questions about Maintain[WCSDistBetweenTrains] yielding:



As before, we gain greater perspective on a goal by finding out what higher-level aim it is intended to achieve.

4.3 Identification of requirements and agent assignments

Obviously, all achievable goals must be reduced until they are irreducible and assignable to an agent, and all unachievable goals must be weakened and then reduced in their new form. When a goal can be assigned to an agent it is called a requisite. A requisite that is assigned to an agent in the software-to-be is a requirement. However, to better illustrate this process we will here concern ourselves only with the reduction of Maintain[WCSDistBetweenTrains] in order to identify some initial requirements for the system. Complete reduction of this goal — let alone that for all goals of the system — would result in a goal graph too large for our purposes. We will restrict the example to the identification of a few requirements, allocating them each to an agent.

Reducing Maintain[WCSDistBetweenTrains], three more goals are identified: Avoid-[TrainEnteringTrackInFrontOfCloseTrain], Avoid [TrainEnteringTrackBehindClose-Train], and Maintain[WCSDistBetweenTrainsOnSameTrack]:

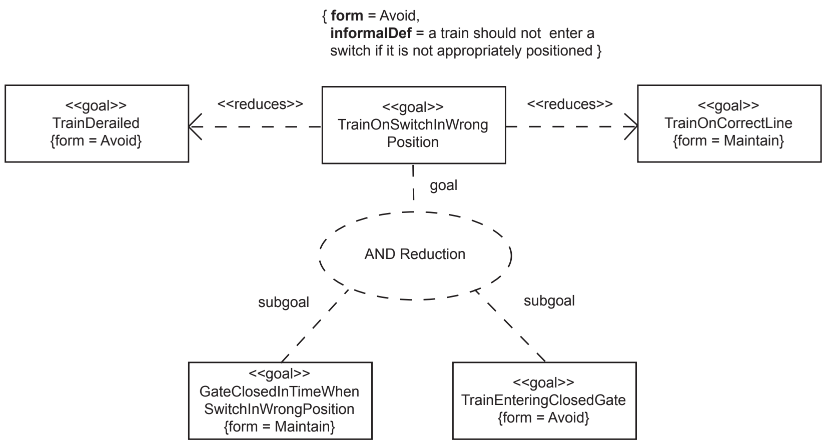

{ form = Avoid,
informalDef = a train should not enter a switch if it is not appropriately positioned }
<<goal>> TrainDerailed {form = Avoid}
<<reduces>>
<<goal>> TrainOnSwitchInWrong Position
<<reduces>>
<<goal>> TrainOnCorrectLine {form = Maintain}
goal
AND Reduction
subgoal
subgoal
<<goal>> GateClosedInTimeWhen SwitchInWrongPosition {form = Maintain}
<<goal>> TrainEnteringClosedGate {form = Avoid}

Fig. 15. Goals identified by asking WHY questions for Avoid[TrainEnteringCloseGate].

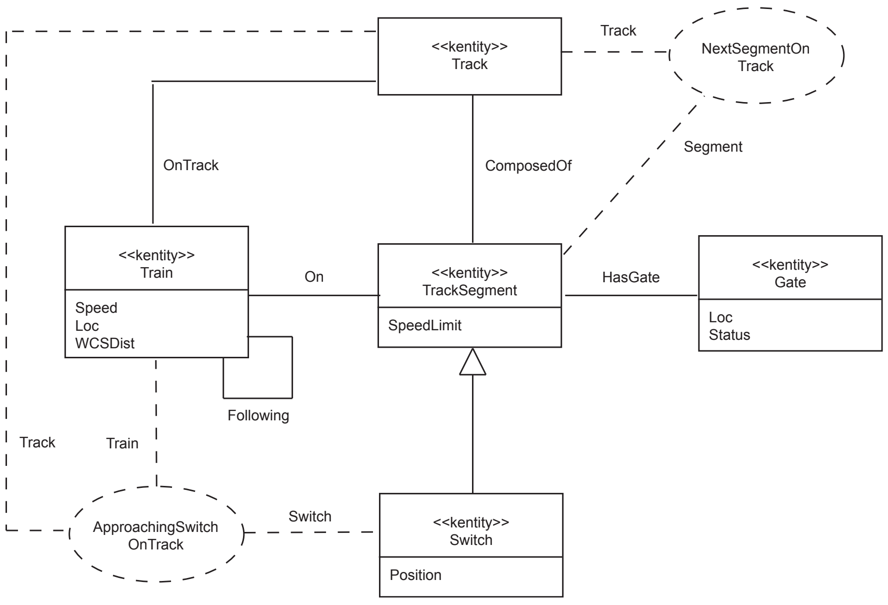


Fig. 16. Extended object model.

<<goal>>
WCSDistBetweenTrainsOn
SameTrack

```
{ form = Maintain,
  informalDef = if a train is following another so that the distance between the two trains is safe,
then the distance between the two trains must remain safe in the next state,
  formalDef = forall tr1, tr2: Train, trk: Track
          ((previous (Following(tr1, tr2) and tr2.Loc - tr1.Loc >= tr1.WCSDist) and
          Following(tr1, tr2)  -->  tr2.Loc - tr1.Loc >= tr1.WCSDist) }
```

<<goal>>
TrainEnteringTrackBehind
CloseTrain

```
{ form = Avoid,
  informalDef = a train should not enter a track behind another train if it violates the worst
  case stopping distance between trains,
  formalDef = forall tr1: Train, trk: Track (just OnTrack(tr1, trk) -->
          ~ exists tr2: Train (tr2 <> tr1 and OnTrack(tr2, trk) and tr2.Loc >= tr1.Loc and
          tr2.Loc - tr1.Loc >= tr1.WCSDist)) }
```

<<goal>>
TrainEnteringTrackInFront
OfCloseTrain

```
{ form = Avoid,
  informalDef = a train should not enter a track in front of another train if it violates the worst
  case stopping distance between trains,
  formalDef = forall tr1: Train, trk: Track (just OnTrack(tr1, trk) -->
          ~ exists tr2: Train (tr2 <> tr1 and OnTrack(tr2, trk) and tr1.Loc >= tr2.Loc and
          tr1.Loc - tr2.Loc >= tr2.WCSDist)) }
```

Have we yet found a goal that can be assigned to an agent? Not yet. Maintain[WCSDistBetweenTrainsOnSameTrack] cannot be operationalized by the centralized train control system agent — an obvious candidate already in the system — because this agent does not control the speed and location of the following train, nor does it monitor the location of both trains. We need to further reduce the goal.

Figures 17 and 18 together show a derivation graph with Maintain[WCSDistBetweenTrainsOnSameTrack] at the head. The graphs are pruned and do not show the entire model for simplicity. The part of the model shown in Figure 17 includes a requirement that is one of two goals reducing the goal Maintain[SafeAccCmdOfFollowingTrain]. This requirement — AccurateSpeed/LocationEstimates — has been assigned to the agent TrainTrackingSystem. Figure 18 identifies four requirements — ReceivedCmdMessageExercised, CmdMessageSentInTime, SafeAcc/SpeedCmdInCmdMessage, and SentCmdMessageDeliveredInTime — and the agents responsible for them. The Figures also show comments attached to the reduction collaborations indicating the KAOS methods used in identifying the new goals.

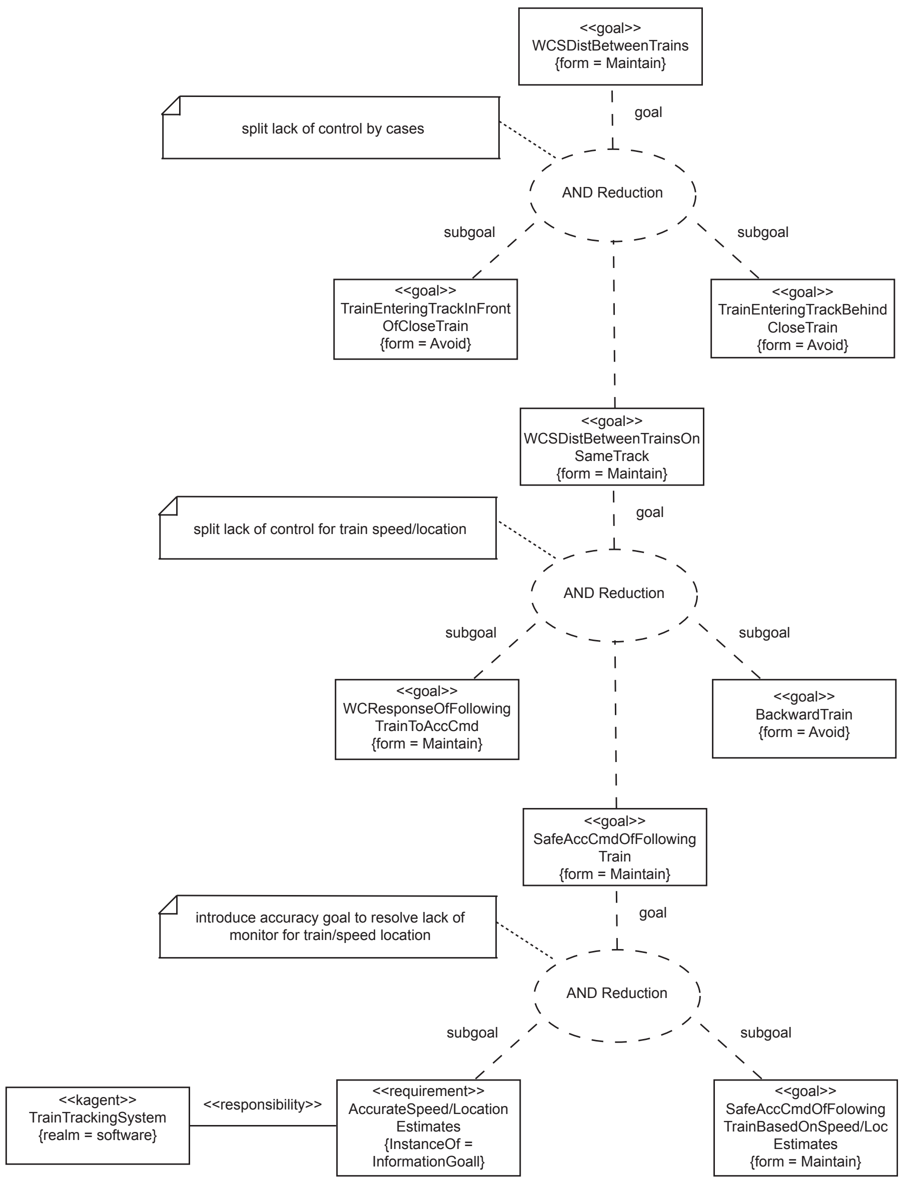


Fig. 17. Goal derivation graph for Maintain[WCSDistBetweenTrains].

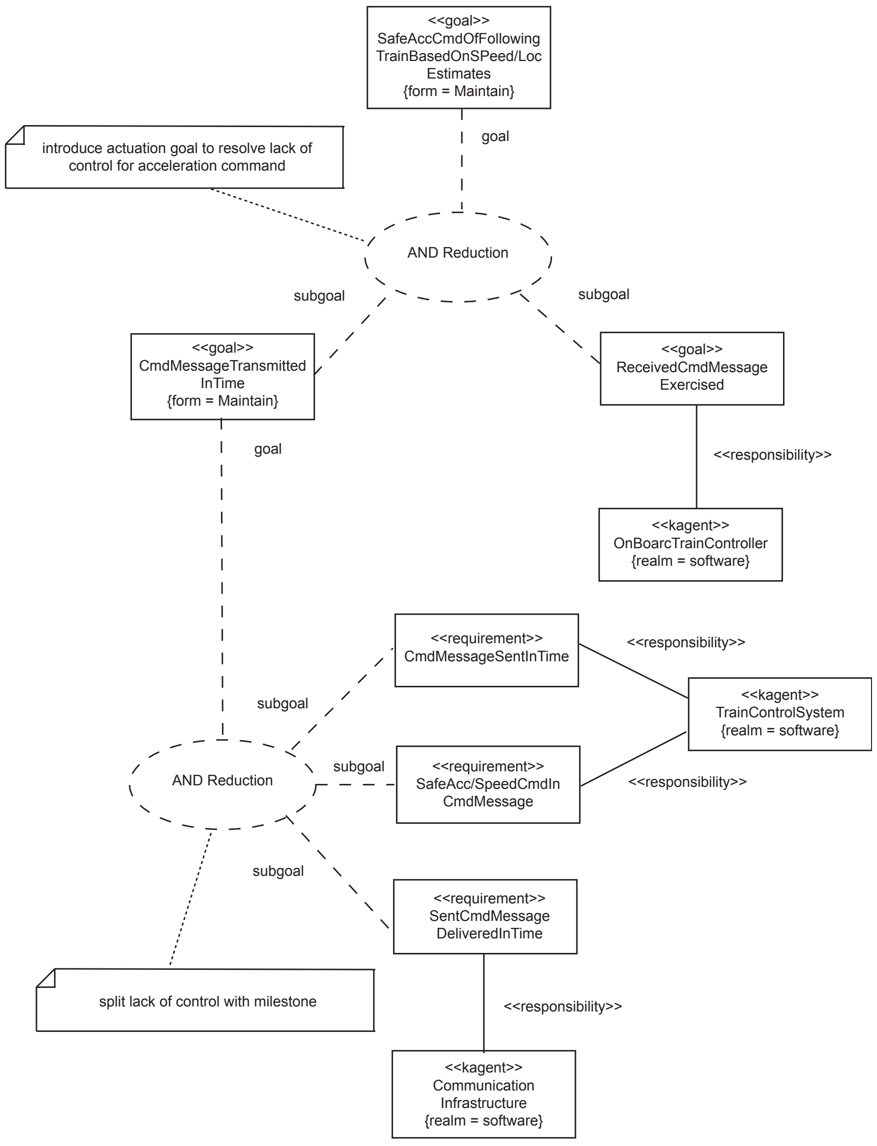


Fig. 18. Agent assignments for subgoals.

4.4 Goal operationalization

Agent interfaces and operational requirements can now be derived from the requirements identified and the agents assigned to them. For instance, the requirement SafeAcc/SpeedCmdInCmdMessage is assigned as the responsibility of the TrainControlSystem agent. A portion of the agent interface model derived from that responsibility assignment is given in Figure 19 (in the diagram the constraints on the action are given in OCL). We should also model the constraints on the <<operationalizes>> associations

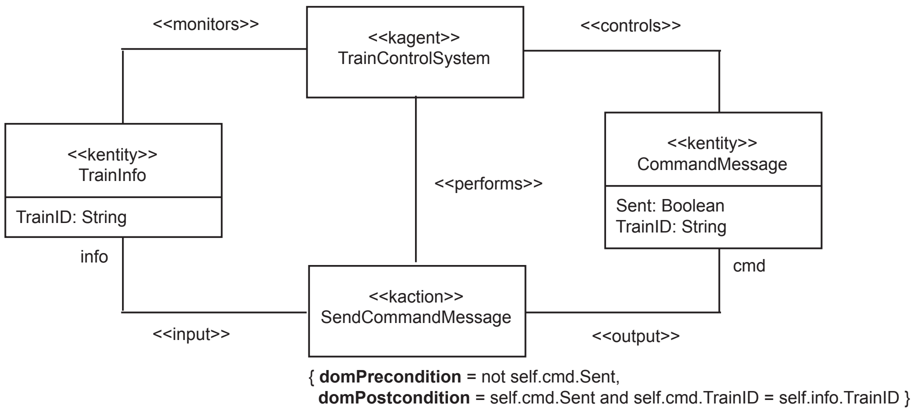


Fig. 19. Agent interface model

of the action as shown in Figure 20. In our UML representation we can include the action definition in the agent interface model. Further actions and agent interfaces are derived from the other responsibility assignments as shown minimally in Figure 21.

5 Related Work

KAOS is by no means the only goal-oriented methodology around (see [4] and [9] for brief surveys of the field). For example, [10] – though its focus is not so much on the acquisition of requirements – uses goals as the main guiding concept in developing requirements specifications. KAOS is, however, the most influential and widely cited approach. It is also unique in its conceptual ontology: lower level descriptions of a system-to-be are progressively derived from system-level and organizational objectives using a framework that is essentially a taxonomy of concepts, which are instantiated

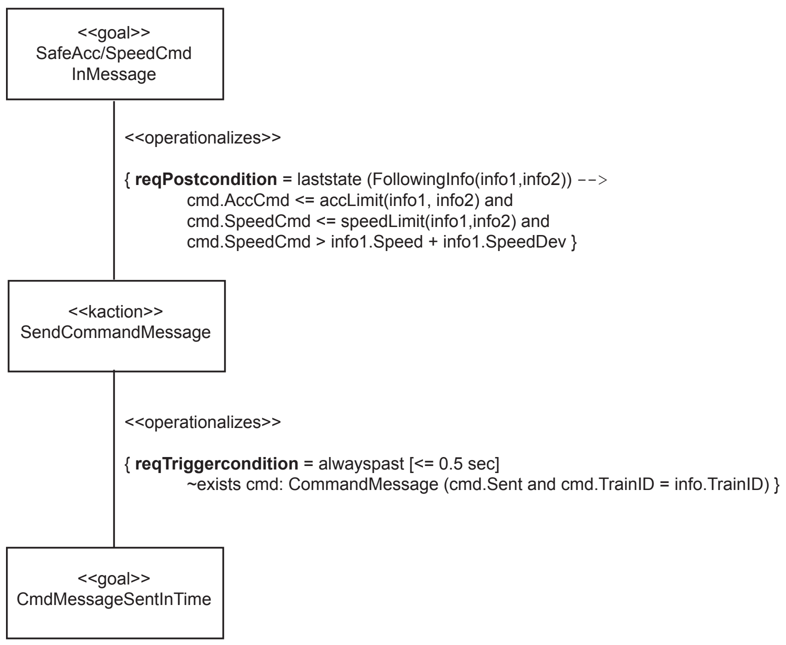

<<goal>>
SafeAcc/SpeedCmd
InMessage
<<operationalizes>>
{ **reqPostcondition** = laststate (FollowingInfo(info1,info2)) -->
cmd.AccCmd <= accLimit(info1, info2) and
cmd.SpeedCmd <= speedLimit(info1,info2) and
cmd.SpeedCmd > info1.Speed + info1.SpeedDev }
<<kaction>>
SendCommandMessage
<<operationalizes>>
{ **reqTriggercondition** = alwayspast [<= 0.5 sec]
~exists cmd: CommandMessage (cmd.Sent and cmd.TrainID = info.TrainID) }
<<goal>>
CmdMessageSentInTime

Fig. 20. Agent interface model with extra constraints

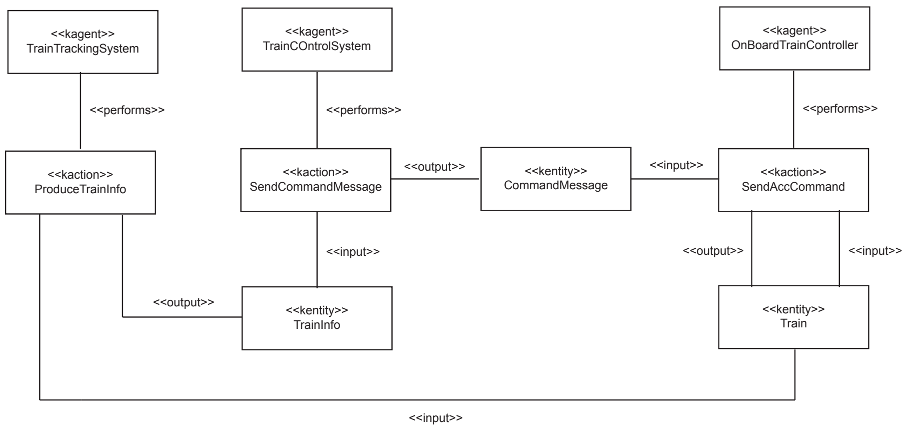


Fig. 21. A portion of the agent and action interface model

for particular domains. The structure KAOS gives its models also makes their representation in the UML straightforward.

KAOS has its own graphical notation and, indeed, in GRAIL, tool support that uses it ([3]). However, the notation is only informally defined – mainly through examples in the literature – and is non-standard. Furthermore, its expressive power is limited. As things stand, a KAOS diagram merely supplements the textual definition of a concept. On the other hand, a UML representation of a KAOS model can combine the ease and immediacy of visual comprehension with the full semantics of the KAOS textual definition by making use of a richer notational semantics together with tagged values. The GRAIL tool needs both a textual editor and a graphical editor, which are displayed in separate windows in the tool's GUI. Representing KAOS in the UML could also lead to a more widespread adoption of the approach.

The GRL – Goal-oriented Requirements Language ([11]) – is a graphical notation based in part on the NFR framework ([12] and [13]) and a candidate for standardisation through the International Telecommunication Union. There is also the potential for integration with the UML. However, again, the GRL does not have as well defined a semantics as the UML, and this is not a trivial matter if we want requirements models to be used as the basis for formal reasoning. Nor, and perhaps most importantly, does it yet allow for the integration of requirements and design models. While the NFR framework is mostly concerned with the evaluation and selection of alternatives with respect to qualitative non-functional goals such as usability, performance, accuracy, and security, KAOS is better suited to the generation of alternative system designs from high-level goals. We readily acknowledge this bias.

We also acknowledge that a UML profile sacrifices the simplicity of the above notations to a certain extent, but feel the advantages offered by a UML representation are mitigating. Our profile is not intended to supersede alternative notations but specifically to provide extra support for KAOS by extending the UML. We might have extended the UML merely by bolting on the existing KAOS diagrammatic notation, but a purely notational extension would carry no semantic content.

A UML profile for modelling goal-oriented requirements has previously been developed as part of the UWA project ([14]) and presented in [15]. The concern of the UWA project is the development of a design framework for facilitating the implementation of ubiquitous web applications. The requirements engineering model advocated is therefore lightweight and – though influenced by KAOS – tailored to the typically rapid development cycles of web services. The UML profile in this paper builds on the original idea from the UWA project but tackles the problem anew by taking the full KAOS model as its target rather than a considerably pared down variant. The UML profile of this paper is intended to be used to model large, critical software systems.

We have shown how our profile can be used to integrate a requirements acquisition model with other UML design models. [6] and [7] give another example of how to integrate a requirements model with lower-level design models. Decomposition of object-oriented systems by class is usually deemed necessary for good software engineering, but it is not sufficient. Requirements models are often not readily compatible with an object-oriented structure: concerns can be spread across many classes and a class may embody several concerns at once. A late change in requirements can thus cause havoc in the lower levels of design. A subject-oriented approach to design is offered as a remedy. UML models are organized by subject where a group of classes deals exclusively with a single requirement. In cases where several requirements are embodied in a single class we model this class several times, once per subject (requirement). This redundancy is accounted for by the semantics introduced with a UML profile governing how the subject-units of a design can be composed. While our profile allows for straightforward integration of requirements and design models, this integration is still flawed by the differences in structure between these types of models. Augmenting a subject-oriented approach with our profile, perhaps by combining our profile with that presented in [6], for example, so that a requirements acquisition model can be represented in UML, and then integrated with a subject-oriented UML design model, would provide a very powerful UML-based software modelling environment.

6 Conclusion

We have introduced a UML profile with which a KAOS model can be represented. Providing for the representation of KAOS in the UML opens up this powerful approach to requirements engineering to the support of the many UML tools available. It also makes KAOS more attractive to newcomers by rendering it in a familiar context. But perhaps more importantly, documentation for KAOS requirements engineering activities can be unified with other UML design documentation, making the UML specification comprehensive and more manageable.

A case study is presented in Section 6. The aim of the case study was to illustrate how all parts of the KAOS process could be modelled using our UML profile. The presentation of the profile in section 4 of this paper draws on the first of the case studies in [2] for its examples, while the case study of Section 6 uses the second case study from [2] – the BART train control system specification – reproducing the KAOS model in UML more comprehensively. As far as KAOS has been presented in the literature, the UML profile adequately supports KAOS.

Future work would include the straightforward task of incorporating support for the profile into a UML editor. A UML KAOS model could also easily be represented in XML, enabling automated consistency checking.

Acknowledgements

The authors wish to thank their colleagues in the UWA (Ubiquitous Web Applications) project for their comments and the European Union for the funding (IST-2000-25131) that has supported this work. We also acknowledge the work of Axel van Lamsweerde and Emmanuel Letier on which we have built. The anonymous referees have assisted in improving this paper.

References

1. Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas: "Goal-Dircted Requirements Acquisition" in *Science of Computer Programming*, 20 (1993)
2. Emmanuel Letier: *Reasoning About Agents in Goal-Oriented Requirements Engineering*, PhD Thesis, ftp://ftp.info.ucl.ac.be/pub/thesis/letier.pdf (2001)

3. P Bertrand, R Darimont, E Delor, P Massonet, and A van Lamsweerde: "GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering" in *Proceedings of ICSE '98 — 20th International Conference on Software Engineering* (1998)
4. Eric Yu and John Mylopoulous: "Why Goal-Oriented Requirements Engineering?" http://www.cs.toronto.edu/pub/eric/REFSQ98.html (1998)
5. Object Management Group: *OMG Unified Modelling Language Specification* version 1.4, http://cgi.omg.org/docs/formal/01-09-67.pdf (2001)
6. S Clarke, W Harrison, H Ossher, and P Tarr: "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code" in *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA) (1999)
7. Siobhan Clarke: "Extending the UML Metamodel for Design Composition" in *Science of Computer Programming*, 44 (2002)
8. V Winter, R Berg, and J. Ringland: "Bay Area Rapid Transit District, Advance Automated Train Control System: Case Study Description", Technical Report, Sandia National Labs, www.sandia.gov/ast/papers/ BART_case_study.pdf (1999)
9. Gil Regev: "Goal-Driven Requirements Engineering Overview" http://lamswww.eplf.ch/Reference/Goal/Default.htm (2001)
10. Annie I Anton: "Goal-Based Requirements Analysis" in *Proceedings of the Second International Conference on Requirements Engineering*, ICRE '96 (1996)
11. The Knowledge Management Lab: "GRL – Goal-oriented Requirements Language", http://www.cs.toronto.edu/km/GRL
12. L Chung, B Nixon, E Yu, and J Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, Boston, (2000)
13. J Mylopoulos, L Chung, and B Nixon, "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach", in *IEEE Trans on Software Engineering*, vol 18 no 6, June (1992)
14. A Finkelstein, A Savigni, G Kappel, W Retschitzegger, E Kimmerstorfer, W Schwinger, Th Hofer, B Pröll, and Ch Feichtner: "Ubiquitous Web Application Development — A Framework for Understanding" in *The 6th World Multiconference on Systematics, Cybernetics, and Informatics* (2002)
15. A Finkelstein and Andrea Savigni: "A Framework for Requirements Engineering for Context-Aware Services" in *Proceedings of STRAW 01 the First International Workshop From Software Requirements to Architectures* (2001)

Appendix: the stereotypes and tags of the profile

Two tables are included. The first gives an overview of the stereotypes, the second details the tags. The constraints given generally cover the constraints of the KAOS meta-model.

TABLE 1: stereotypes

Stereotype	Base Class	Parent	Description	Constraints
<<kobject>>	Class	-	Class defines a thing of interest that can be referenced in goal defintions	-
<<kentity>>	Class	<<kobject>>	Class defines an autonomous thing of interest: instances of <<kentity>> classes may exist independently of instances of other <<kobject>> classes	-
<<kevent>>	Class	<<kobject>>	Class defines an instantaneous thing of interest: instances of <<kevent>> classes only ever have the one state	-
<<kaction>>	Class	<<kobject>>	Class defines an input-output relation over <<kobject>> classes; defines state transitions	An action can only be applied if its domPrecondition holds
<<input>>	Association	-	Association denotes a relationship between a <<kobject>> class and a <<kaction>> class	Values of the association's argument tag must be attributes of the <<kobject>> class
<<output>>	Association	-	Association denotes a relationship between a <<kobject>> class and a <<kaction>> class	The value of the association's result tag must be attributes of the <<kobject>> class

<<kagent>>	Class	<<kobject>>	Class defines a processor for some actions; controls state transitions	*
				performs(ag,ac) --> capability(ag,ac)
				controls(ag,o) --> monitors(ag,o)
				capability(ag,ac) and input(o,ac)
				--> monitors(ag,o)**
				capability(ag,ac) and output(o,ac)
				--> controls(ag,o)**
				responsibility(ag,r) and
				operationalizes(ac,r)
				--> performs(ag,ac)

* agent : ag, action : ac, object : o, requisite : r
** the agent need actually only monitor / control those attributes of the object that specifically provide the input / output

Stereotype	Base Class	Parent	Description	Constraints
<<capability>	Association	-	Association denotes a relationship between a <<kagent>> class and a <<kaction>> class	See <<kagent>> constraints
<<performs>>	Association	-	Association denotes a relationship between a <<kagent>> class and a <<kaction>> class	See <<kagent>> constraints
<<monitors>>	Association	-	Association denotes a relationship between a <<kagent>> class and a <<kobject>> class	See <<kagent>> constraints; values of the monitored tag must be attributes of the <<kobject>> class
<<controls>>	Association	-	Association denotes a relationship between a <<kagent>> class and a <<kobject>> class	See <<kagent>> constraints; values of the controlled tag must be attributes of the <<kobject>> class and may not be values of the controlled tag of any other <<kagent>> class
<<goal>>	Class	-	Class defines an objective to be achieved by the system	A <<goal>> class is a singleton.
<<concerns>>	Association	-	Association denotes a relationship between a <<goal>> class and a <<kobject>> class	-
<<reduces>>	Abstraction	<<refines>>	Abstraction denotes a relationship between two <<goal>> classes, or between a <<goal>> class and a <<requisite>> class	-
<<conflicts>>	Association	-	Association denotes a relationship between two <<goal>> classes	-
<<requisite>>	Class	-	Class defines a goal that has been sufficiently refined to be assignable to a single agent	A <<requisite>> class is a singleton.

Stereotype	Base Class	Parent	Description	Constraints
<<assumption>>	Association	<<requisite>>	Class defines a requisite responsibility for which can effectively be assigned to an agent in the domain	-
<<requirement>>	Class	<<requisite>>	Class defines a requisite responsibility for which can be assigned to an agent in the domain	A requirement must be defined only in terms of things monitored and controlled by the software
<<operationalizes>	Association	-	Association denotes a relationship between a <<requisite>> class and a <<kaction>> class	-
<<ensures>>	Association	-	Association denotes a relationship between a <<kobject>> class and a <<requisite>> class	-
<<responsibility>>	Association	-	Association denotes a relationship between a <<requisite>> class and a <<kaction>> class	Only one agent is assigned the responsibility for each requisite; if an agent is responsible for an assumption it must be a domain agent; and if an agent is responsible for a requirement it must be a software agent

TABLE 2: tags

Tag	Stereotype	Type	Description	Constraints
informalDef	all KAOS stereotypes	string	An informal definition of a class or association	-
formalDef	<<goal>> <<requisite>> <<kaction>>	string	A formal definition of a class expressed in KAOS temporal logic (using ASCII characters)	-
form	<<goal>> <<requisite>>	{Achieve, Maintain, Avoid, Cease, Minimize, Maximise}	Specifies the type of goal or requisite; Achieve and Cease goals generate behaviours, Maintain and Avoid goals restrict behaviours, Minimize and Maximise goals are used in design comparisons	-
soft	<<goal>>	boolean	Indicates whether a goal is soft: soft goals are goals that are not clearly defined (and cannot be formalized)	A soft <<goal>> class cannot have a formalDef tag
instanceOf	<<goal>>	{Satisfaction, Safety, Security, Information, Accuracy}	Specifies the application-specific type of a goal	A <<goal>> class with an instanceOf value of Safety must have a priority value of 1
priority	<<goal>>	float (0..1)	Priority used to resolve conflicts: 1 indicates highest priority	See instanceOf constraint
category	<<requisite>>	{Satisfaction, Safety, Security, Information, Accuracy}	Specifes the application-specific type of the goal reduced by the requisite	-
invariant	<<kobject>>	string	Defines a domain property with respect to object: expressed in KAOS temporal logic (using ASCII characters)	-

Tag	Stereotype	Type	Description	Constraints
strengthenInv	<<kobject>>	string	Defines a strengthening of the invariant in order to satisfy a requirement: expressed in KAOS temporal logic (using ASCII characters)	-
frequency	<<kevent>>	string	An informal indication of the interval between consecutive occurrences of the event	-
domPrecondition	<<kaction>>	string	A general domain precondition for the action	-
domPostcondition	<<kaction>>	string	A general domain postcondition for the action	-
reqPrecondition	<<operationalize>>	string	A required precondition for the operationalization of a requirement by an action: expressed in KAOS temporal logic (using ASCII characters)	-
reqPostcondition	<<operationalize>>	string	A required postcondition for the operationalization of a requirement by an action: expressed in KAOS temporal logic (using ASCII characters)	-
reqTriggercondition	<<operationalize>>	string	A required postcondition for the operationalization of a requirement by an action; an action must be applied if trigger condition becomes true (and the precondition of the action holds): expressed in KAOS temporal logic (using ASCII characters)	reqTriggercondition must imply reqPrecondition
realm	<<kagent>>	{domain, software}	Indicates whether agent is domain agent or software agent	-