GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering

P. Bertrand, R. Darimont, E. Delor, P. Massonet
CEDITI-UCL
B-6041 Gosselies, Belgium
{pb,rd,ed,phm}@cediti.be

A. van Lamsweerde
Université Catholique de Louvain
B-1348 Louvain-la-Neuve, Belgium
avl@info.ucl.ac.be

ABSTRACT

The KAOS methodology provides a language and method for goal-driven requirements elaboration. GRAIL is an environment under development to support the KAOS methodology. The GRAIL kernel combines a graphical view, a textual view, an abstract syntax view, and an object base view of specifications. GRAIL has been used to elicit and specify the requirements of several real, industrial projects.

1 THE KAOS METHODOLOGY

The KAOS methodology provides a specification language for capturing *why*, *who*, and *when* aspects in addition to the usual *what* requirements; a goal-driven elaboration method; and meta-level knowledge used for local guidance during method enactment [1, 2]. The language provides a rich ontology for capturing requirements in terms of goals, constraints, objects, actions, agents, etc. Links between requirements are represented as well to capture refinements, conflicts, operationalizations, responsibility assignments, etc.

Goals, for example, describe invariant properties expected of the composite system, i.e. the system and its environment. Fig. 1 shows examples of goals for a train system: "Achieve rapid passenger transportation", "Maintain the train doors closed between stations" and "Avoid trains on same block". Goals are organized into AND/OR graphs. A goal can be and-refined into sub-goals, such that the conjunction of the subgoals is a sufficient condition to achieve the parent goal. OR-refinements capture alternative way of achieving goals. Fig. 1 shows part of the goal refinement for a train system.

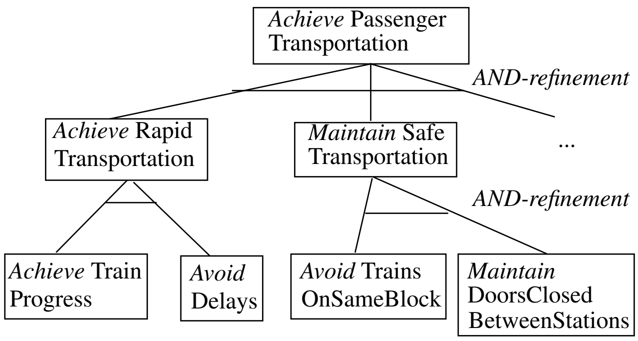

Fig. 1 Train System Goal Refinement

The KAOS language is a multi-paradigm specification language with a two-level structure: an outer semantic net layer for declaring concepts, their attributes and links to other concepts; an inner formal assertion layer for formally defining the concept. The latter combines a real-time temporal logic for the specification of goals, constraints, and objects, and standard pre-/postconditions for the specification of actions and their strengthening to ensure the constraints.

Goal Maintain [DoorsClosedBetweenStations]
InstanceOf SafetyGoal
Informal Definition The train doors must remain closed while the train is moving between two stations.
FormalDef ($\forall$ tr:Train, s: Station)
$\bullet$ At (tr, s) $\wedge \neg$ At (tr, s)
$\Rightarrow$ tr.doorState="closed" W At(tr,Next(s))

Goals are operationalized by actions/objects in such a way that the composite system, i.e. automated and physical parts, satisfies the goals. Agent behavior is restricted by strengthening the pre- and post-conditions of the actions they perform, and by strengthening the object invariants.

Action OpenDoors
Informal Definition Opens the train doors.
Input Train {**Arg**: tr}, Station {**Arg**:s}
Output Train {**Res**: tr'}
DomPre (The train doors are closed)
tr.doorState="closed"
DomPost (The train doors are open)
tr'.doorState="open"
RequiredPre for DoorsClosedBetweenStations
(The train is stopped at a station)
At(tr, s)
RequiredPost for DoorsClosedBetweenStations
(The train is stopped at a station)
At(tr', s)
PerformedBy TrainController

The language makes a clear distinction between the domain pre- and post-conditions and the strengthenings that are needed to satisfy the goals of the system.

The method roughly consists of (i) identifying and refining goals progressively until constraints that are assignable to individual agents are obtained, (ii) identifying objects and actions progressively from goals, (iii) deriving requirements on the objects and actions to meet the constraints, and (iv) assigning the constraints, objects and actions to the agents composing the system. Fig. 2 illustrates step (iv) where

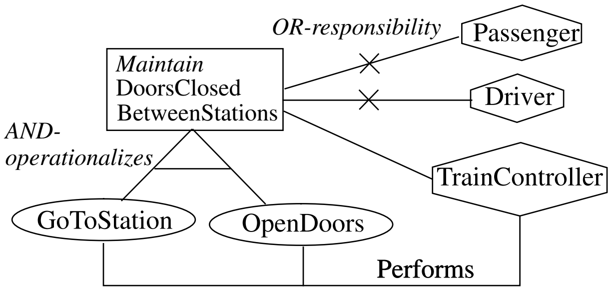

Fig. 2 Operationalization and Responsibility Graph

alternative responsibilities have been considered and one assigned to the "TrainController" agent. The "TrainController" is thus responsible for maintaining the doors closed between stations and the actions he performs, i.e. "GoToStation" and "OpenDoors", have been restricted accordingly by the operationalization link.

Meta-level knowledge is used to guide the elaboration process; it takes the form of conceptual taxonomies, well-formedness rules and tactics to select among alternatives. For example, goals may be classified by the pattern of temporal behavior they require:

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

Maintain: $P \Rightarrow \Box Q$ or *Avoid*: $P \Rightarrow \Box \neg Q$

Goals can also classified according to a category taxonomy: satisfaction, robustness, efficiency, safety, privacy, information, consistency, etc.

2 THE GRAIL ENVIRONMENT

GRAIL is an environment under development to support the KAOS methodology. Its architecture has been influenced by the ALMA lifecycle support system [3]. The GRAIL kernel combines a graphical view, a textual view, an abstract syntax view, and an object base view of specifications. The current version in use integrates a graphical editor, a syntax-directed editor, a hypertext navigator, a LaTex report generator, and an object base.

The KAOS declaration level is manipulated either graphically or textually; the assertion level is manipulated textually only. See Fig. 3 and Fig. 5. The **textual editor** supports template-driven requirements acquisition, and incremental checking of syntax and static semantics at the declaration level, assertion level and between them. Fig. 4 shows an example of a check between the declaration and the assertion levels. In this example the "At" relationship between "Train" and "Station" has not been declared yet, but is already used in several assertions. A list of "Not

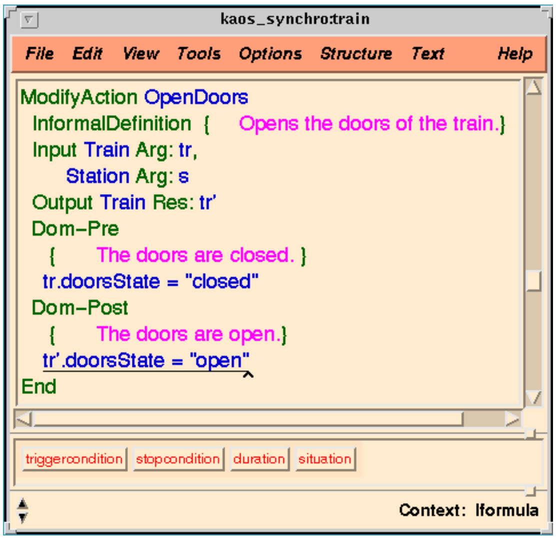

Fig. 3 Textual Editor

Defined" errors are given in the lower "ErrorView" window of Fig. 4, each line indicating an undeclared use of "At". The underlined error in the "error view" of Fig. 4 corresponds to the use of "At" above the cursor in the assertion of "DoorsClosedBetweenStations". Verifying that

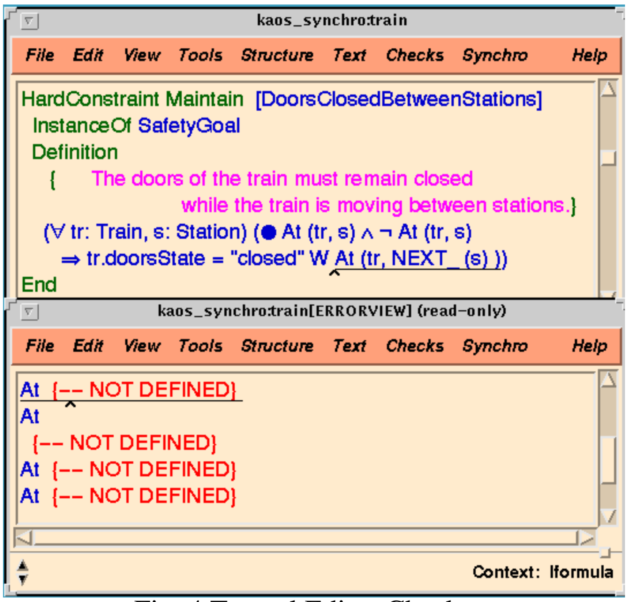

Fig. 4 Textual Editor Checks

a concept has been declared every time it is used, or verifying that a concept has not been already declared are examples of declaration level checks. Verifying that a variable used in a predicate has been declared is an example of an assertion level check. The checks are non-blocking and incremental. The former supports ease of use by allowing temporary errors to subsist and the latter supports change control by indicating the errors that a change could produce in the specification.

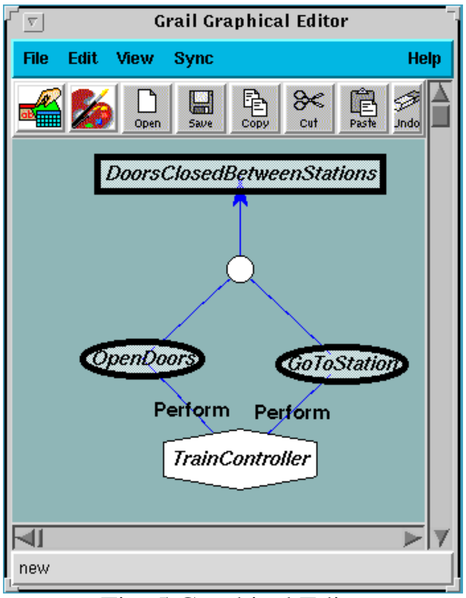

Fig. 5 Graphical Editor

The **graphical editor** complements the detailed textual view with a high-level graphical view of the specification. Automatic layout of the semantic net is provided to make the graphical view of the specification more quickly and easily understandable. The graphical and textual editors are built on top of Tcl/Tk and SG [4], respectively. Fig. 3 shows the definition of an action for a simple train system. Fig. 5 shows an excerpt of the semantic net, where the "TrainController" agent "Performs" the "OpenDoors" and "GoToStation" actions, that operationalize the "DoorsClosedBetweenStations" constraint.

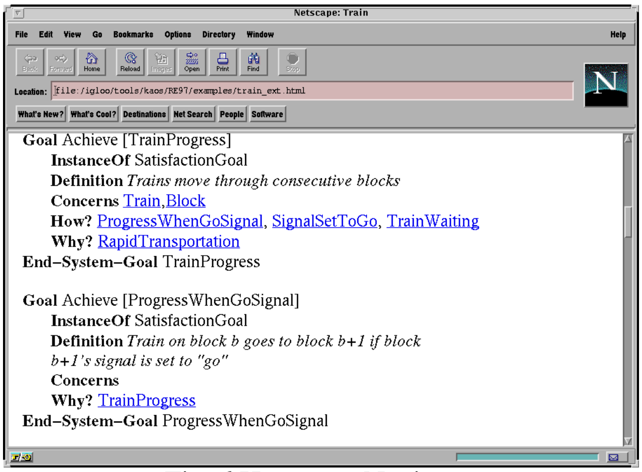

Fig. 6 Hypertext Navigator

The syntax-tree internal representation of SG provides report generation facilities. Currently HTML and Latex reports are supported. The navigator allows specifiers to browse the entire specification in hypertext mode (see Fig. 6). An **explanator tool** is under development. It will allow generating a report by assembling the informal definitions of concepts into a continuous text whose structure is compliant with the IEEE Std-830-1993 for requirements documents.

The **object base** provides persistency, querying and concurrent access to the specification. Queries are used to make checks on the requirements specification. For example the following query finds all unrefined goals that need to be refined:

set result = **Goal**
which not [(exist g:**Goal**) (**RefinedBy** g)]

In order to manage large specifications it is necessary to have **partial views** on the specification. Fig. 7 shows the entity-relationship view for a portion of the Train specification. Typical views include

- refinement view: shows how goals are refined into constraints, which alternatives have been selected and how goal conflicts are resolved.
- operationalisation view: shows which actions and objects operationalize the constraints.
- entity-relationship view: shows the object model in terms of entities and relationships.
- the agent view: shows the agents that perform the actions that operationalize the constraints for which they are responsible.

Other views may easily be created by selecting the appropriate combination of language concepts to view. The partial view mechanism was implemented using the tool synchronization facilities described below.

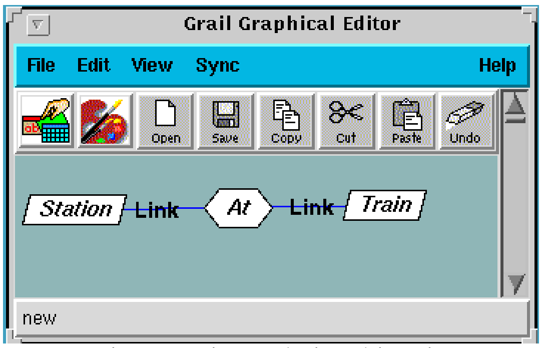

Fig. 7 Entity-Relationship View

3 THE GRAIL ARCHITECTURE

The architecture of the GRAIL kernel is shown in Fig. 8. Tool integration is achieved by combining event-based and repository-based architectural styles [5, 6]. An integration module provides synchronization and communication services; the former allows the specification to be maintained in a consistent state among tools, and the latter allows independent tools to communicate; tools thus do not make assumptions about each other. The synchronization/ communication capability is implemented on top of Orbix, a CORBA compliant facility. The local synchronization module provides local event-based integration, i.e. between tools running on the same machine. It is used to synchronize in real time the textual, graphic and object views of the specification. For example, when a new goal is created in the textual view, it automatically appears in the graphical and object views.

The global synchronization module provides global event-based integration, i.e. between tools running on different machines. This second level of synchronization provides the multi-user services that are needed for defining cooperative work policies. An experiment was carried out using the Orbix implementation of CORBA on a subset of the KAOS language illustrating the need for a model of cooperative work with its rules.

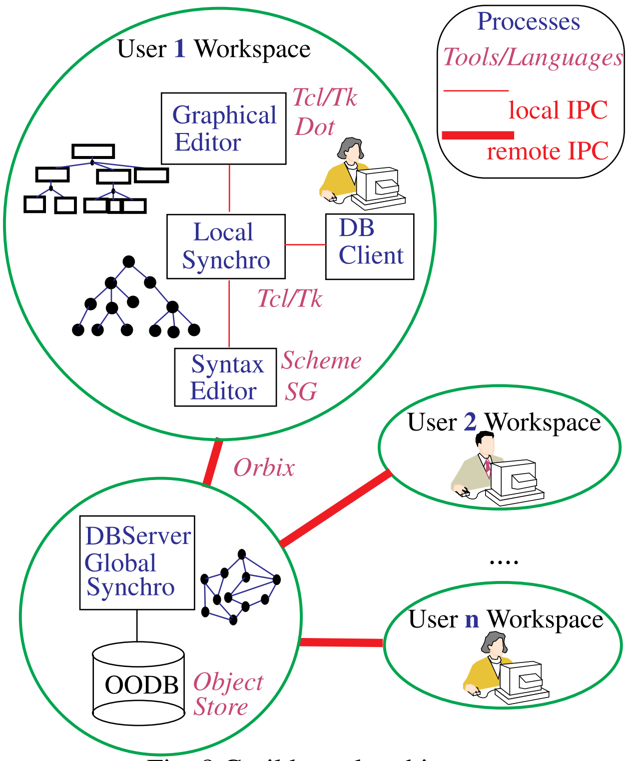

Fig. 8 Grail kernel architecture

4 A TOOL INTEGRATION EXAMPLE

The GRAIL/KAOS environment is a requirements acquisition tool; to be useful, it needs to be integrated with other case-tools. One experiment involved using the tool integration facilities to integrate it with the DOORS requirements and traceability tool, and the Rational Rose CASE tool. DOORS is used to create trace links between artifacts of different steps of the software life cycle. In this case it was used to create trace links between concepts of a KAOS specification and their counterparts in an OMT specification. The integration was achieved with the local synchronizer. This guarantees that every KAOS concept in GRAIL has a corresponding concept in DOORS. The DOORS to Rational Rose integration provided by DOORS was used to create traceability links to OMT concepts in Rational Rose. Fig. 7 shows the "Train" and "Station" entities in GRAIL. The upper part of Fig. 9 shows the "Train" and "Station" entities in DOORS along with their trace links to their OMT counterparts. The lower part of Fig. 9 shows Rational Rose and the corresponding "Train" and "Station" OMT concepts.

Ongoing developments concern the development of collaborative specification capabilities. Future plans include specialized tools on top of the GRAIL kernel for: coupling the environment with downstream CASE tools such as STP/ OMT, RTM and STATEMATE; generating natural language explanations; formal analysis and pattern-driven construction; specification reuse; and method enactment.

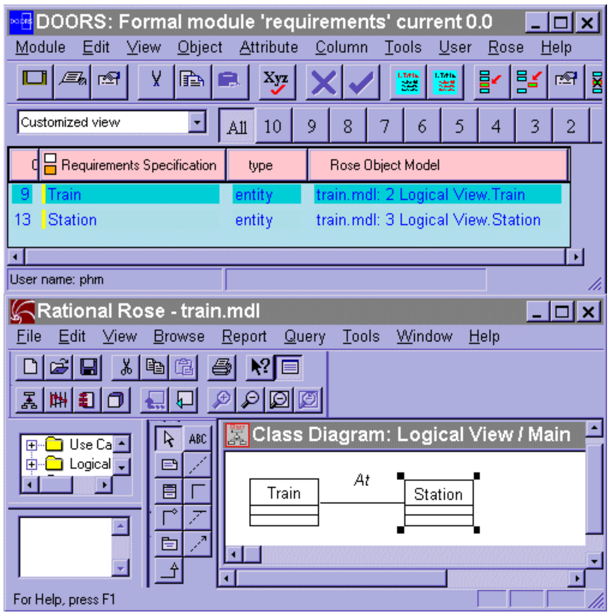

Fig. 9 DOORS and Rational Rose integration

5 INDUSTRIAL EXPERIENCE AND REQUIREMENTS PROCESS

The development of GRAIL is carried out in parallel with the use of KAOS on several industrial projects. The version being demonstrated has in particular been used for the requirements reengineering of a large telecommunication system. It has also been used for the requirements engineering of a complex copyright tracking system and a information system for a big pharmaceutical distribution company. The integration of new facilities above the kernel is driven by needs arising from such experience.

The important deliverable of these requirement analysis projects is a requirements document which is handed over to the customer upon completion. The "ideal" requirements process that is followed to ensure that the requirement document meets quality criteria such as correctness and traceability is illustrated in Fig. 10. The important feature of this process is that the Kaos model must be elaborated before the natural language requirements document is derived from it.

The Kaos analysis takes inputs from a variety of data sources:

- relevant general domain knowledge that analysts already have about a domain.
- different types of stakeholders are interviewed. Transcripts of the interviews are available to create trace links, with a requirements and traceability tool like DOORS, to the Kaos model.
- documentation that may be useful for requirements analysis is gathered; it can be imported into a requirements and traceability tool like DOORS, and traced to the Kaos model. Typical documentation includes procedure manuals, standards, product specifications, information about the existing system, etc.

The Kaos model is elaborated by following the Kaos goal-

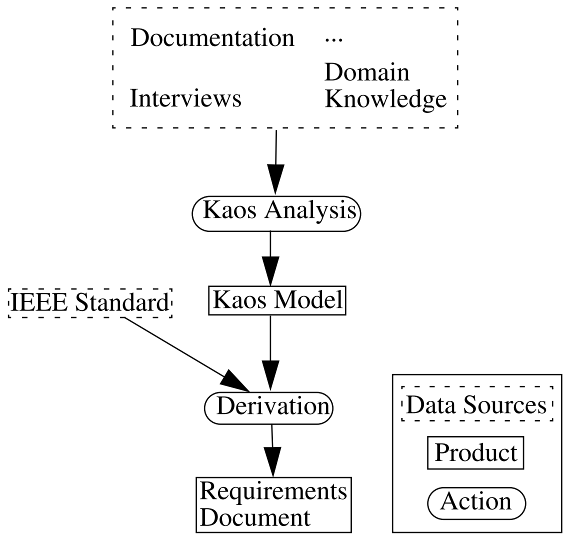

Fig. 10 Requirements Document Building Process

driven elaboration method described earlier. Once the Kaos model is finished, a natural language requirements document, compliant with the IEEE Std-830-1993, is derived from the Kaos model. The derivation is based on a mapping between the Kaos model structure and the requirements document structure. The derivation is currently done manually, but the explanator tool that is under development, will help make this step semi-automatic.

6 CONCLUSION

Conclusions which may be drawn from these experiences are that the Kaos goal-driven elaboration method helps to ask the right questions, favors precise requirements, makes the writing of a requirements document easier and produces specifications that are very well perceived by industrial partners. Elaboration of large/complex requirements such as those encountered on industrial projects is intractable without tool support. The GRAIL environment provides adequate support for goal-driven elaboration; it is already operational for analysts well acquainted to KAOS and the technologies underlying the tool.

ACKNOWLEDGEMENTS

Development of the GRAIL/KAOS prototype was partially supported by the European Union and the “Ministère de la Région Wallone” (convention Objectif 1, CEDITI-UCL 2905).

REFERENCES

1. A. Dardenne, A. van Lamsweerde, and S. Fickas, Goal-Directed Requirements Acquisition. Science of Computer Programming Vol. 20, North Holland, 1993, 3-50.
2. A. van Lamsweerde, R. Darimont and Ph. Massonet, Goal-Directed Elaboration of Requirements for a Meting Scheduler: Problems and Lessons Learnt. Proc. RE'95 - 2 nd. IEEE Symp. on Requirements Engineering (York, UK), March 1995, 194-203.
3. A. van Lamsweerde, B. Delcourt, E. Delor, M.C. Schayes, and R. Champagne. Generic Lifecycle Support in the ALMA Environment. IEEE Transactions on Software Engineering, Vol. 14, No. 6, June 1988, 720-741.
4. T. Reps, and T. Teitelbaum, The Synthesizer Generator, Springer-Verlag, 1989.
5. M. Shaw and D. Garlan, Software Architecture, Prentice Hall, 1996.
6. D.J. Barrett, L.A. Clarke, P.L. Tarr, A.E. Wise, A Framework for event-based software integration, ACM Transactions on Software Engineering and Methodology, vol. 5, nr. 4, Oct 1996, 378-421.