A KNOWLEDGE-BASED SUPPORT SYSTEM FOR THE REUSE OF STRUCTURED SPECIFICATIONS AND DESIGNS OF EMBEDDED COMPUTER SYSTEMS

Tuomas Ihme

Technical Research Centre of Finland (VTT)
Computer Technology Laboratory
P.O. Box 201
SF-90571 Oulu
FINLAND

ABSTRACT

Reuse promises to be one of the key factors in enhancing quality and productivity in software development. However, CASE tools for real-time systems are usually focused on the development of new software. In this paper we describe a domain-based support system for the reuse of structured software specifications and designs of embedded software[1].

Keywords: Reuse, Structured Analysis,
Software Engineering, Knowledge-Based Systems,

[1] This research was carried out as a part of the FINPRIT research programme, funded mainly by the Technology Development Centre of Finland (TEKES). Financial support was also provided by the Technical Research Centre of Finland (VTT), Kone, Nokia-Mobira, and Edacom Corporations.

1. Introduction

The productivity of a software development organization depends on many factors, technical and non-technical. *Software reusability* has been identified as one of the factors that has a direct impact on productivity. In routine software construction, program patterns recur frequently. One of the goals of software reusability is to capture these patterns as reusable modules. Repeated reuse allows to amortize the cost of developing the reusable components, thus allowing the development of higher-quality software.

In practice, despite of some success stories, the reusability in software construction is much less widespread than it could be (Meyers 1987). The attention of practitioners has focused on CASE for *developing new software* (CASE Outlook 1988). This is, in part, because there is still a gap between the results of research on reusability and their implementation for production use, and because of the lack of integration between those results. We will comment briefly on these two aspects.

There does not seem to be any absolute characterization of what makes a piece of information reusable. In principle, *application-independent* reuse systems can be flexible and generic, but they present two kinds of problems: it is difficult to determine which information should be captured as reusable components and how to locate and select those components. A lesson learned from experimental reuse systems is that practical approaches to reusability tend to be application or *domain specific*,(e.g. (Mittermeir & Oppitz 1987), (Prieto-Diaz 1987) and (Arango 1988)), specific pieces of information are reusable *with respect to* the solution of specific classes of applications, or *domains*. Domain-specific approaches have two kinds of practical advantages, 1) it is easier to support the identification and acquisition of reusable components (a process similar to knowledge engineering for expert systems, called *domain engineering* in (Arango 1988)); and 2) actual reuse can be supported (and in many cases, mechanized) using simpler mechanisms for locating, retrieving and adapting components.

In summary, to make reusability in software construction practical two preconditions must be met:

Infrastructural condition:

there exist systematic methods to identify, acquire, represent and evolve reusable information.

Operational condition:

there exist tools to aid in locating, adapting and configuring reusable components.

The problems of *how to develop reusable components*, and *how to reuse them* are mutually dependent. The former presupposes knowledge of how reusable resources are going to be reused, and the latter, the existence of appropriate reusable resources.

Most research on reusability has focused on satisfying the second condition. A variety of technologies have been proposed or developed to support the reuse of components: component data-bases, application generators, configuration systems, transformation systems, and so on. Satisfying the first condition has become a priority only recently. Still, little has been done in integrating methods and tools in such a way to satisfy both conditions.

1.1 The Infrastructural and the Operational Problems

The generation of reusable components in a *systematic* and *economic* way faces the same problems of the so-called "knowledge acquisition bottleneck" in expert systems (Arango 1988). These tasks share much with knowledge acquisition for knowledge-based systems:

1-identifying the boundaries of the application domain. That is, defining which classes of applications need to be implemented and what sources of information need to be consulted;

2-analysing the domain to identify which information is relevant in the specification and implementation of systems in the domain; and

3-representing the relevant, reusable information in particular formats to support actual reuse.

From a reuse perspective there are two kinds of reusable knowledge that must be captured about a domain: specification and implementation knowledge.

Specification Knowledge

From the point of view of **specification**, an application domain is characterized (using conceptual modelling techniques) by a collection of reusable objects, operations and relationships, which in practice play the role of a problem-oriented specification language. Information about an application domain can be classified using the ontological analysis approach to knowledge engineering (Alexander et al. 1986). The levels of the analysis are:

1. *Static ontology* that defines the physical objects and identifies with their inherent properties and relationships,

2. *Dynamic ontology* that defines the operations applied on them, and

3. *Epistemic ontology* that defines the knowledge structures which specify which relations between objects and operations are possible.

These (finite, and often small) collections of definitions simplify the process of reuse by facilitating the location and adaptation of *appropriate* reusable components. "Appropriateness" is the key word: domain specificity makes possible to determine which information is relevant to the implementation of a particular class of applications.

Implementation Knowledge

We must also capture sufficient information for carrying out the implementation of the specifications. General methods have been developed (e.g., reverse engineering and design recovery) but specific procedures need to be customized to each software development environment. The ontological analysis framework can be also applied here, but to the implementation process instead of to the problem domain. For instance, if an organization applies Yourdon's RTSA methodology to the development of embedded systems, then the static ontology includes the description of transformations, data-flows, state machines, and so on. The dynamic ontology includes the description of such operations as model configuration and refinement; and the epistemic ontology, the description of constraint over the configuration and refinement of SA models.

2. Objectives, Constraints and Strategy

The objective of our reuse system is to *integrate existing approaches to support practical reusability* in the construction of *embedded software systems* and in the using of the RTSA domain-independent system design method.

The part of the software process to be supported is specification and preliminary design (Davis 1988). We will outline a application-specific infrastructure that yet relies on a quite wide spectrum of the RTSA and object-oriented application-independent design methods. The intended product development environment is an extensively interactive system, where the target reuser is a human being, and therefore the design selection problem is automated only partially.

A reuse-centred production environment should be able to store and reuse not only product components, but also production process, test, simulation and operation environment components. A variety of reusable software artifacts can be assembled to support a wide-spectrum reuse methodology (Lubars 1987). The others than the reusable product components are, however, out of the scope of this study. We will leave most of the process issues either implicit or manual.

The constraint on first-order reuse (Arango 1988) and on a particular class of problems (embedded systems and the RTSA method) has allowed us to focus on methods and representations for the acquisition of specification and implementation information based on generic notions of horizontal and vertical composition (Goguen 1986) that are applicable in any first-order reuse schema. Vertical relations are those which characterize implementation design decisions. Thus, our library structure includes components consisting of reusable specification patterns and reusable design patterns. We emphasize the construction, organization and reuse of libraries of RTSA-based specification and design components.

3. The Reuse Infrastructure

A first-order base (or, infrastructure (Arango 1988)) for specification and preliminary design reuse is presented. The first-order information consists of software specifications and designs in explicit forms as in parts of RTSA models or in sets of RTSA primitives.

The first-order infrastructure supports the representation of the semantics of problem-specific abstractions because of the practicable modeling power of the RTSA method (Yourdon 1988). It is also a persistent repository of domain-specific knowledge relevant to a reuse system (Arango 1988). It has the potential of being used as a tool for communication and training because of visibility of the analysis and design effort of the RTSA method.

Reuse-centred software construction requires a component-based viewpoint to the software process and to its products. The RTSA methods supported by the existing commercially available CASE tools are based on visual techniques, but in a reuse-centred approach the visual representations are only one dimension of the component base.

3.1 Data Flow Diagram Items

Data flow, entity relationship and state transition diagrams are used for the data and function modeling in the RTSA method. We have remodelled the data flow diagram items to support first-order reuse. The following data flow diagram primitives are modelled: data and control transformations, terminator, data-store and data and control flows. Some extra components are required to represent informal specifications, interconnections between components, and product constructions.

Terminators

The terminators show the external entities from and to which the data and control flows; that is, the entities with which the system is required to interface. Terminators are not as passive as described by the Yourdon-RTSA method, but they play active roles in a connection model, too.

Transformations

Transformations represent actions and events in the problem domain. The following property categories are used for describing activities: name, type, comment, input, output, control, invariants, precond, postcond, stopcond, default_parent, parent, default_neighbours, specialization-procedure, specialization-guide, specialization-default, instantiation, constraints, goals, plan, minispecification, assertions and coordinates (Figure 2).

Data and Control Flows

The information about the relationships between the entities in a data flow diagram is maintained in data and control flows. The advantages of keeping the relationship information only in the relationship objects are:

- easy to access needed information from anywhere from the system, and
- easy to maintain consistence, because the trace information is stored in one place only.

Data Stores

A store represents a time-delayed relationship between processes. Stores model passive entities in the domain. The basic notation and concepts of the entity-relationship diagram are used in partitioning of complex stored data. The resultant partitioning can be transferred to the transformation schema. The data dictionary provides an exact definition of every store.

3.2 Motivation Information

The model building process of the RTSA method proceeds by deriving features of the model from the previous features of the model or from previous models and documents. A snapshot situation in a reuse based software design process is shown in Figure 1.

Traces between the logical and physical models of RTSA (Ward & Mellor 1985) are described by trace objects. A trace object describes the relationships between one component in a logical model and one or more components in a corresponding physical model, or between several components in the logical model and one component in the physical model.

The pieces of requirements in the narrative requirements document are saved in requirements objects (e.g. Requirements for Gong Device) that are used to construct traces between the narrative requirements and the models. A Design-Decision object (e.g. Control Lanterns and Gong) includes the recorded design decisions about the component (e.g. how a component definition come about or why is it reasonable to have an abstraction in a model, Figure 7). Recorded design decisions with rationales and reusability supports, describe the background, characteristics and attributes of the reusable components and its dependencies on different issues (Taramaa et al. 1988). Reuse-Log objects are used to gather feedback information about reusable components from the actual reuse process.

Assertions are used to define the semantics of objects by defining constraints on structure and behaviour. Also, they state facts about the problem domain or software design that justify the concept descriptions in the model (Arango 1988). Assertions do not always need to be separate objects, but they can appear as a value of a component attribute. Note objects gather comments and temporary motivations that cannot be assigned to any above information type when they are created. All these motivation objects can be traced to the requirements objects.

The Motivation Network

Specifications most often evolve from small seeds to larger, more complete specifications in an extensively iterative manner. Our reuse system is interactive, and therefore the human-computer interaction must be effectively solved in using and evolving the reuse infrastructure.

The network of motivation objects is a kind of semantic network which mixes informal textual material with the more formal and mechanized material of the RTSA method. A motivation network allows the information and knowledge elements to be structured into multiple hierarchies and graphs, thus allowing the world to be "sliced up" into several orthogonal decompositions (Conklin 1987). The motivation network supports both hierarchical trees via organizational links and several nonhierarchical graphs via the typed links and links between the typed motivation objects.

The Use of the Motivation Network

Motivation objects in the motivation network (e.g. the trace-based documents) of the design process are used to capture and describe useful background knowledge of potential reusable components. The motivation objects provide a basis for revision during the evolution of the component library. The use and operational advantages of the motivation network are (Conklin 1987):

- ease of tracing references: machine support for link tracing means that all references are equally easy to follow forward to their referent, or backward to their reference. If a user wants to modify a transformation, he or she can easily check the related pieces of the product requirements, the related design decisions or other related information. The user can follow a selected link further and can find that the transformation depends logically on another transformation, which was not depicted by the RTSA diagrams. The user can also trace references from certain product requirements to the related RTSA diagrams.

- ease of creating new references: users can grow their own networks, or simply annotate someone else's RTSA diagrams with a note.

- global and local views: browsers provide table-of-contents style global and local views, that can be mixed effectively. Different kinds of views can easily be implemented using the KEE expert system development shell.

- customized documents: objects can be threaded together in many ways, allowing to generate documents for example for baselines and working documents.

- modularity of information: since the same motivation object can be referenced from several RTSA components, motivations can be expressed with less overlap and duplication.

- several views: the user is supported in having several paths of inquiry active and displayed on the screen windows at the same time (e.g. Assertion, Requirement, Design-Decisions, Reuse-Log and Note paths), such that any given path can be unwound to the original task, for example to modify a certain transformation.

3.3 The Component Library

At the highest level in the library there is a component class hierarchy for an application (Figure 1). The hierarchy includes background information needed to consult the user to use lower level reusable components to build product specifications in the application domain. It holds also information for product- and component-specific analyses.

The component class hierarchy specifies domains, whose purpose is to restrict the selection space of the reuser by defining a class of similar, reusable components. From a reuse perspective a domain is a catalogued set of reusable objects, operations and their relationships, with attributes and links to another objects and operations (Seppänen 1989). We can have several hierarchical domains as application domains, product domains and component domains.

The basic components in the library are typed RTSA items, whose types are the characteristics of a product (Figure 1). A composite component is created from the RTSA basic items and from other composite components. Subsystems are composite components modelled up to the terminator level of the RTSA method (e.g. Signalling in Figure 1). Product types are completely modelled RTSA systems and consist of components from the lower levels.

The RTSA logical and physical models form together a total system specification (Hatley & Pirbhai 1987). Not only a system component, but each reusable component may have a logical model and a physical model (e.g. Signalling). Many real-life products are too large to model as one RTSA system, but the subsystems of a large product are often modelled as separate RTSA systems. A model of a large product is then a composite model of several RTSA system that are represented as terminators in the composite model. As a matter of fact, each reusable component needs to have some kind of an external or environment model and an internal model.

3.3.1 The Configuration Knowledge

Configuration knowledge can be categorized in one of the four ways (Frayman & Mittal 1987):

1. Knowledge about the different components that are actually available and their properties.
2. Knowledge about the relations between components.
3. Constraints on the properties of or relations between components.
4. Preferences on how the system is configured.

Configuration knowledge consists of organization and composition principles that deal with the architectures for creating and organizing components and with the rules for composing components. It is used to inform the user of the relevant design choices on each design stage and support the use of

similar structures of components. The integration and coherence of the large variety of configuration knowledge are achieved by relying on a object-oriented programming paradigm (Wu et al. 1986).

Configuration knowledge consists of a hierarchy of knowledge involved in configuration. A hierarchy is a mean to decompose the knowledge of configuration into more manageable units.

The Component Model

Information about components and system types are described by the component model which defines the attributes and options of the components and systems (Wu et al. 1986), for example the signalling component in Figure 2. The component model captures information about individual systems and components. The component model constitutes a taxonomic hierarchy of a class and sub-classes and a specialization hierarchy to depict the relationship between parts and sub-parts. Properties may be inherited along the descendant path within a class hierarchy. The following properties are included in the specialization hierarchy: specialization-guides, specialization-defaults and specialization-procedures. The parent property is an inverse relationship to the specialization-procedures property. The parent and specialization-procedure relations are always lists to indicate an one-to-many relationship.

3.3.2 Hierarchical Design Components

Hierarchical design components provide a means for abstracting software designs into broadly reusable components that can be assembled and refined into new software designs (e.g. the signalling component in Figure 3). Generalization is an appropriate abstraction principle to exploit when the difficulty of modeling is caused by a large number of details and components that need to be captured (Borgida et al. 1985). The common aspects of the gong, lanterns and position indicator devices are presented as the description of the signalling superclass in Figure 3.

Features of the Product

The order of a product specifies the custom feature set of the product. Features depend upon market area (e.g. Scandinavia, USA), use area (e.g. goods lift, hospital lift), product type and level of abstraction. A combination of these factors defines a domain that we will designate as feature space (Shah 1988). Features in a feature space can be considered to be organisational elements used for structuring design components in a component library.

Feature spaces of the same dimension may be partially overlapping or be completely disjoint. In the overlapping (conjoint) regions one will find features with identical semantics. In regions that do not overlap, features are meaningful in only one domain. We use the term "conjugate spaces" to identify those sub-spaces which contain features that are composed of different variations of the same elements. Adjoint spaces are created by associating elements in one sub-space to certain elements in another sub-space. Simple features are the lowest-order canonical forms possible in a sub-space; compound features can always be decomposed into dislike simple features without loss of information (Shah 1988).

A feature can be available explicitly and be independent of other data in the model; or the feature can be determined from the explicit feature data of another feature; or the feature can depend on a relation between two or more features (Shah & Rogers 1988). The presence or absence of some kinds of features could trigger certain tests or actions. Since the information that needs to be extracted is not known in advance, one must have the ability to access the feature database as and when required by the search.

Constraints in Design Components

The use of constraints incorporates one important difference between design components, and the simple forms of code and design templates (Lubars 1986). The custom feature set of the product is used as constraints of hierarchical design components. The lanterns feature specifies a choice between the signalling.with.lanterns and signalling.without.lanterns refinements in the specialization procedure of the signalling component in Figure 4. The gong feature in Figure 5 triggers downward movement and newly applicable refinements (specialization) through the abstraction hierarchy of the signalling component. Refinements (e.g signalling.with.lanterns) may set additional constraints, thus continuing the cascading effect of constraint propagation and refinement.

Overloaded Design Components

Design components are overloaded, if the particular form of the domain situation determines which design solution to select (Lubars 1986). A hierarchical design component is constructed for a design family, with several applicable specializations (e.g. signalling.with.lanterns and signalling.without.lanterns in Figure 3), each appropriate under different sets of constraints alias features.

4. Reuse Process

Reuse tasks can be divided roughly in the following subclasses:

1. Initial reuse-based design
   - Redesign
2. Configurative production
   - Reconfiguration

These two subclasses have different emphases in the production processes of different products, roughly so that the former has the main emphasis in small batch production, and the later in mass production. After the initial product design cycle, there are needs to hardwire most of the design choices for efficient enough production or for different products inside a product family. This task is very much a configuration problem. In it, an important part is to find the relevant constraints and the related design variables.

We point out that these two tasks are likely to be somewhat interwoven, but that we want to conceptually separate them from each other: the rough difference is that for solving a configuration problem one actually does not invent anything new. This is one approach to reuse. However, in practice one is usually forced to make modifications to the reusable pieces, and then there is a flavour of redesign in the configuration task. Redesign is naturally interconnected to the initial design task, since in the case of a design failure some backtracking is needed with corrective actions. The hope is that these actions may preserve as much as possible from the previous trial to avoid waste of efforts.

The design of a complicated embedded product is typically a complex, domain oriented and product specific process. Some design decisions and selections of product options must be made early, because of timing and control dependencies between components and subsystems of the product. A reuse-based product design process starts usually with the configuration task, but continues in redesign.

4.1 Configuration Process

The task of configuring a system from its components is defined as follows (Frayman & Mittal 1987): "*Given a set of components, an architecture that defines certain fixed ways of connecting these components, and user requirements, either produce a configuration that satisfies all requirements or detect inconsistencies in the requirements.*" Requirements can be classified into two categories: requirements for individual components and requirements for the overall configuration.

In many cases a configuration task is not merely a constraint satisfaction, it is really an optimization task, where one can judge and compare configuration alternatives and ultimately select the best or optimal configuration. It is difficult for a system to be able to determine which configuration is optimal. Thus, it is desirable for the system to be able to provide multiple configurations and possibly leave the task of selecting the optimal configuration to the user (Frayman & Mittal 1987).

Design Specialization and Refinement

Design specialization is a constraint driven process that enables the selection of design sub-families from larger design families (Lubars 1986). Each of the specialized and overloaded components can have associated specialization procedures as shown in Figure 6. Collectively they represent the possible refinements of the design family, which the specialization process (the instantiation procedure) chooses the most appropriate design component from.

Configuration knowledge is used to instantiate reusable components from the library into the designer's working desk (Figure 6). An instantiation procedure controls and performs instantiations of components. From the point of view of the instantiation procedure the only links which connect components in the library to each other are specialization procedures as shown in Figure 6. So the library system is flexible, and the specialization procedures make components hierarchical.

At the beginning of the component instantiation the user specifies which features he would like to have in the model, in the form of a features list, and the starting point of the procedure (i.e. which design component to start with and onto which model refinement level it should be copied) as shown in

Figure 6. Models of complete systems or individual components without hierarchy can supplied as the first component. Then the component instantiation procedure evaluates the specialization procedure of the specified starting component against the features list to determine which sub-components must be copied into the model as the component's refinements. The sub-components are copied into the model and the procedure recurs with each sub-component in turn. The instantiation procedure can ask the designer to specify additional features, if needed by a lower level component in the hierarchy. The result is the instantiated components in the model and a list of traces indicating which components are part of the model.

4.2 Design Process

The system design process can be seen as the representation and manipulation of system models by synthesis and decomposition tools. The synthesis tools are essential for the representation of complex systems, since they enable the engineer to put models of parts together to form models of complex entities. The key to successful synthesis is composition. The composition mechanisms must provide well defined connectives for effective inspection and analysis. The decomposition tools are used to decompose a complex design problem or structure into simpler and manageable steps and guides the user through these simple steps toward a complex goal.

The specification and design begins with creating the logical model of the RTSA method from the conceptual model of the systems analyst. The RTSA representation provides a uniform model throughout the development process. Initial specifications describe a new system using high-level abstractions of domain-oriented concepts. The user can select appropriate components for top-level system description. The specification and design process can also begin in any later specification or design phase, if the earlier designs are same in an old design. The reuse-based design process is described more detailed in (Hakkarainen et al. 1989).

Design Component Selection

Different component selection situations must cope with different combinations of incomplete, incorrect, and partial design specifications (Lubars 1986). The component selection strategy is based on the domain-oriented nature of design components and their correspondence with domain operations. The user has product and subsystem specific views and he can access the library catalogue from all four levels of the component hierarchy shown in Figure 1. These views are also specific to the current design phase. One selection is enough to add a composite component or subsystem into the current design. If the selection has an influence on later design phases, the implementation of these parts is delayed until these design phases are in progress.

The user is provided with a menu-based library catalogue of the available components. To understand the component and its characteristics, the user has an access to its graphical representation from the library catalogue as shown in Figure 7. In addition, the user can get important characteristics of door components in the library when designing door control.

5. Conclusions

Domain-independent reuse requires complex techniques for acquiring the reusable components, and strongly domain-specific approaches need sophisticated methods for capturing the domain knowledge. Our reuse system is a knowledge-based software design reuse system that combines the benefits of the two approaches. We have outlined methods and structures for building and using libraries of reusable RTSA components that can be applied to practical software construction in the industrial production of embedded computer systems. We have adapted the RTSA method to the construction and reuse of RTSA components.

We have developed a reuse prototype system as an extension of an intelligent RTSA model editor (Hakkarainen et al. 1989). It provides a graphics-based user interface and an effective frame-based knowledge representation scheme. A menu-based library component catalogue and graphical visualization facilities are also available. We have used the prototype system for the evaluation and demonstration of the promises of the presented reuse ideas. This software design expert system is implemented in KEE expert system development environment running on a Symbolics workstation.

Once the reusable components are developed and saved in the library, it is possible to reuse the same component in different product families and in different versions of one product. Product knowledge can also be used in semi-automating some design steps and giving intelligent advice and product consultations. Reuse promises to move a considerable amount of the analysis and design effort of embedded real-time systems into defining the product itself.

6. References

Alexander et al. 1986. Knowledge Level Engineering: Ontological Analysis, in proceedings of the Fifth National Conference on Artificial Intelligence (AAAI'86), pp. 963-968.

Arango, G. 1988. Domain Engineering for Software Reuse. ICS-RTP-88-27, Department of Information and Computer Science, University of California, Irvine.

Borgida, A. et al. 1985. Knowledge Representation as the Basis for Requirements Specifications. IEEE Computer, April 1985, pp. 82 - 91.

CASE Outlook 1988. CASE Tools for Reverse Engineering. CASE Outlook 2, 2, p. 1.

Conklin, J. 1987. Hypertext: An Introduction and Survey. IEEE Computer, Vol. 20, 9, pp.17 - 41.

Davis, A. 1988. A Taxonomy for the Early Stages of the Software Development Life Cycle. The Journal of Systems and Software 8, pp. 297 - 311.

Frayman, F. & Mittal, S. 1987. Cossack: A Constraint-based Expert System for Configuration Tasks. Proceedings of 2nd International Conference on Applications of AI to Engineering, Boston.

Goguen, J. 1986. Reusing and Interconnecting Software Components, IEEE Computer, February 1986, pp. 16-28.

Hakkarainen, K., Ihme, T. & Oivo, M. 1989. A Knowledge-based system for embedded computer analysis and design, The first Nordic Conference on Advanced Systems Engineering, CASE89, May 9 - 11 1989, Stockholm, Sweden.

Hatley, D. & Pirbhai, I. 1987. Strategies for Real-Time System Specification. New York, Dorset House Publishing.

Lubars, M. 1986. A Knowledge-based Design Aid for the Construction of Software Systems. Urbana, University of Illinois at Urbana-Champaign, Report no. UIUCDCS-R-86-1304. 201p.

Lubars, M. 1987. Wide-Spectrum Support for Software Reusability. Austin TX, Microelectronics and Computer Technology Corp., MCC Technical Report Number STP-276-87. 8p.

Meyers, B. 1987. Eiffel: Programming for Reusability and Extendibility. Goleta, CA, Interactive Software Engineering Inc. 12 p.

Mittermeir, R. & Oppitz, M. 1987. Software Bases for the Flexible Composition of Application Systems, IEEE Transactions on Software Engineering, April 1987, pp. 461-471.

Prieto-Diaz, R. 1987. Domain Analysis for Reusability. Proceedings COMPSAC'87, Tokyo, October, pp. 23-29.

Shah, J & Rogers, M. 1988. Functional requirements and conceptual design of the Feature-Based Modelling System, Computer-Aided Engineering Journal 5, 1, pp. 9 - 15.

Shah, J. 1988. Feature transformations between application-specific feature spaces. Computer-Aided Engineering Journal, pp. 247 - 255.

Seppänen, V. 1989. Navigation Dimensions in Knowledge-based Software Reuse. SCAI89, Tampere, June 1989.

Taramaa, J. et al. 1988. Decision-based Approach in Software Engineering. STeP-88, Suomen Tekoälytutkimuksen Päivät, 15.-18.8.1988. Helsingin yliopisto.

Ward, P. T. & Mellor, S. J. 1985. Structured development for real-time systems, 1-3. New York, Yourdon Press.

Wu, H. et al. 1986. ISCS - A Tool Kit for Constructing Knowledge-Based System Configurators. Proceedings of the 5th National Conference on Artificial Intelligence, Philadelphia. pp. 1015-1021.

Yourdon, E. 1988. Managing the System Life Cycle. New Jersey, Yourdon Press.

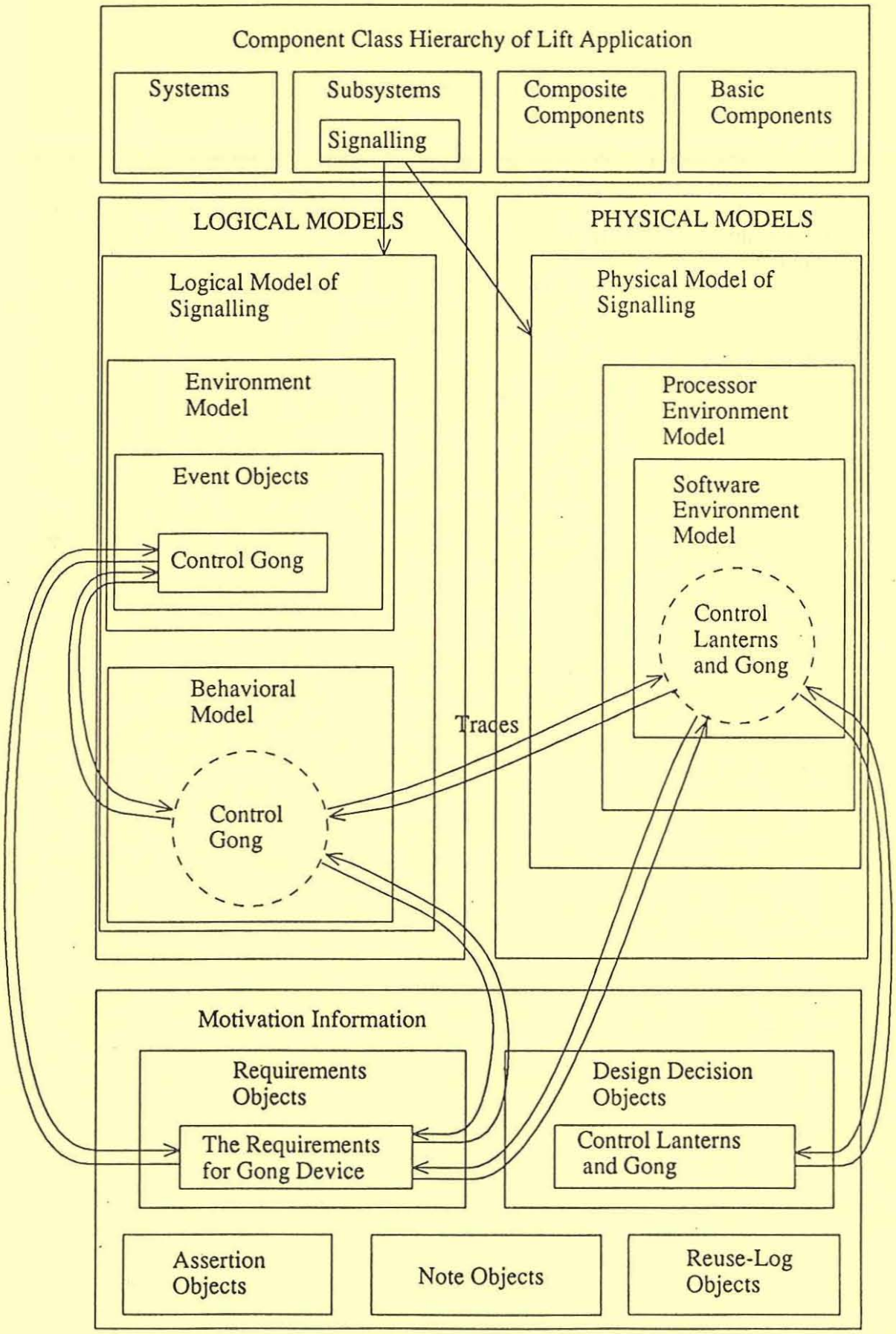

Figure 1. A snapshot situation in a reuse-based design process.

Name:
    Comment: the name of the transformation
    Value: Signalling
Type:
    Comment: the type of the transformation or reference to the component library
    Value: Signalling
Comment: Generalization of signalling devices
Input:
    Comment: input data/control flows
    Value: DECELERATING, POSITION_CHANGE, MOVING, LANTERNS_ON/OFF, POSITION, LANDING_CALL
Output:
    Comment: output data/control flows
    Value: SIGNALLING_COMMANDS
Control:
    Comment: external control of the transformation
    Value: E/D SIGNALLING
Invariants:
    Comment: conditions that must be satisfied all the time
    Value: all inputs and outputs are mandatory
Precond:
    Comment: conditions that must be satisfied whenever the transformation is activated
Postcond:
    Comment: conditions that are guaranteed to be true after the execution of the transformation
Stopcond:
    Comment: conditions, when the execution of the transformation is stopped
Default_parent:
    Comment: the default type of the parent transformation
    Value: CONTROL_LIFT
Parent:
    Comment: the actual parents of the transformation
Default_neighbours:
    Comment: the default types of the neighbour transformations
    Value: SIGNALLING_DEVICES, DRIVE, CAR, LANDING_CALL_DEVICES
Specialization-procedure:
    Comment: the selection procedure of specialization
    Value:
        (if (equal LANTERNS T)
            SIGNALLING.WITH.LANTERNS
            SIGNALLING.WITHOUT.LANTERNS)
Specialization-default:
    Comment: the default component of specialization
    Value: SIGNALLING.WITH.LANTERNS
Implementation:
    Comment: reference to implementation
    Value: (SIGNALLING PHYSICAL_MODEL)

Figure 2. Attributes of the SIGNALLING component

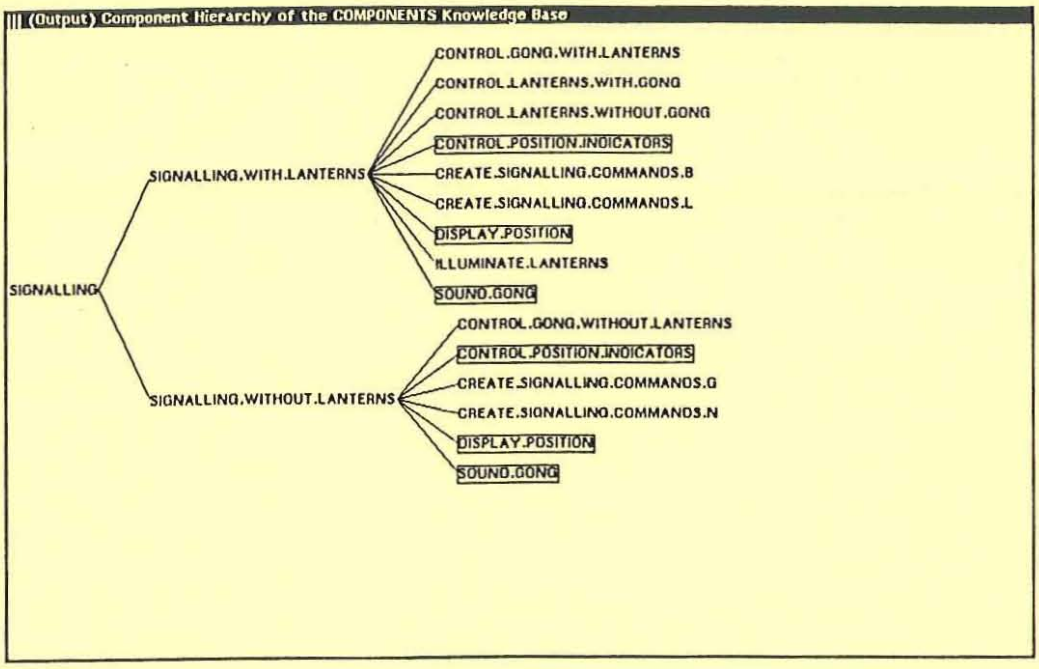

Figure 3. The hierarchical signalling component is constructed for a design family, with several applicable specializations.

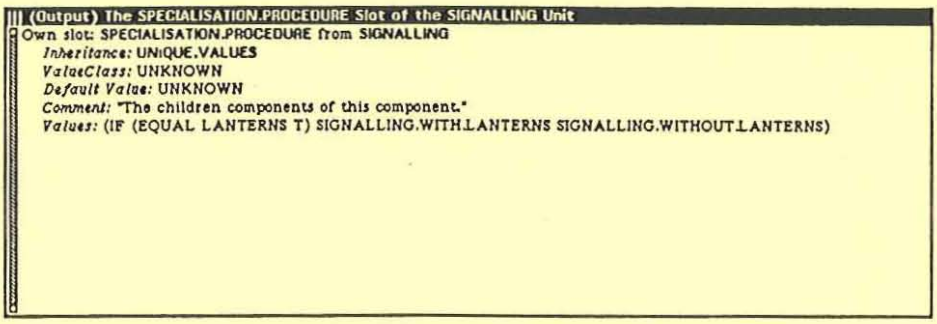

Figure 4. The lanterns feature specifies a choice between the signalling.with.lanterns and signalling.without.lanterns refinements in the specialization procedure of the signalling component.

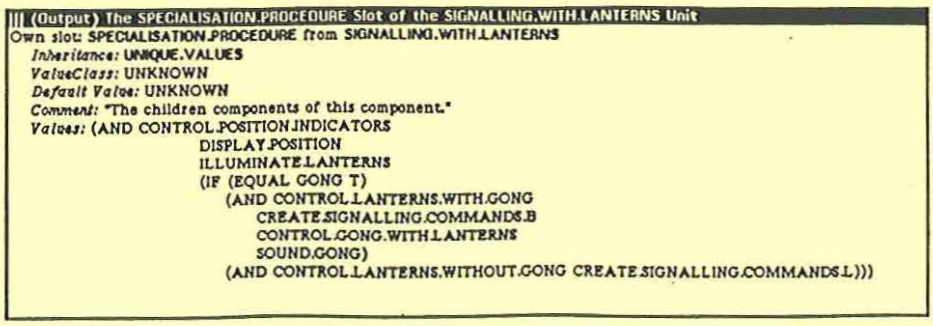

Figure 5. The gong feature triggers downward movement and newly applicable refinements in the specialization procedure of the signalling.with.lanterns component.

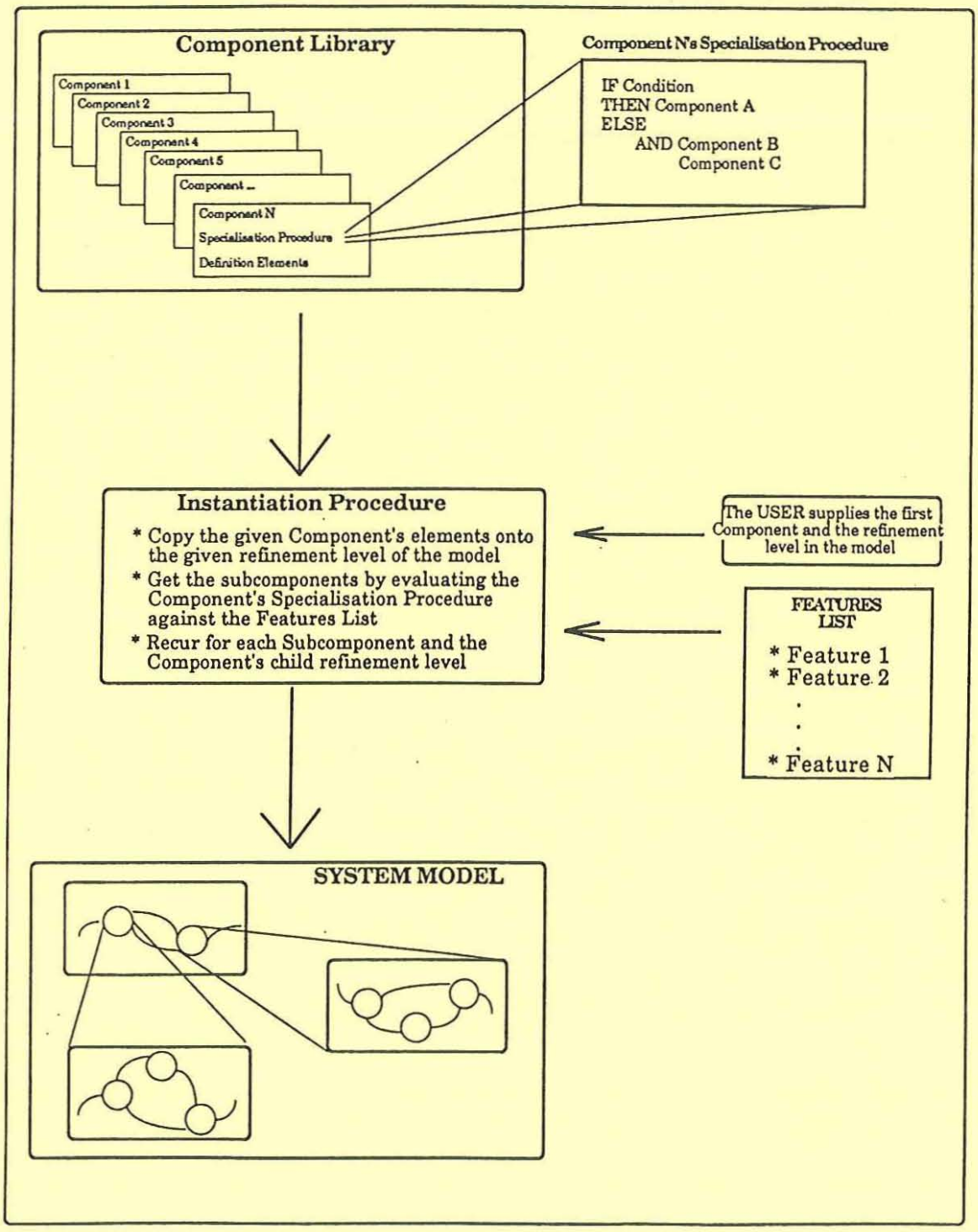

Figure 6. The user specifies which features he would like to have in a model, and which design component to start with. The component instantiation procedure copies the starting component, and evaluates recursively the specialization procedures of the components in the component hierarchy against the feature list to determine which sub-components must be copied into model as refinements.

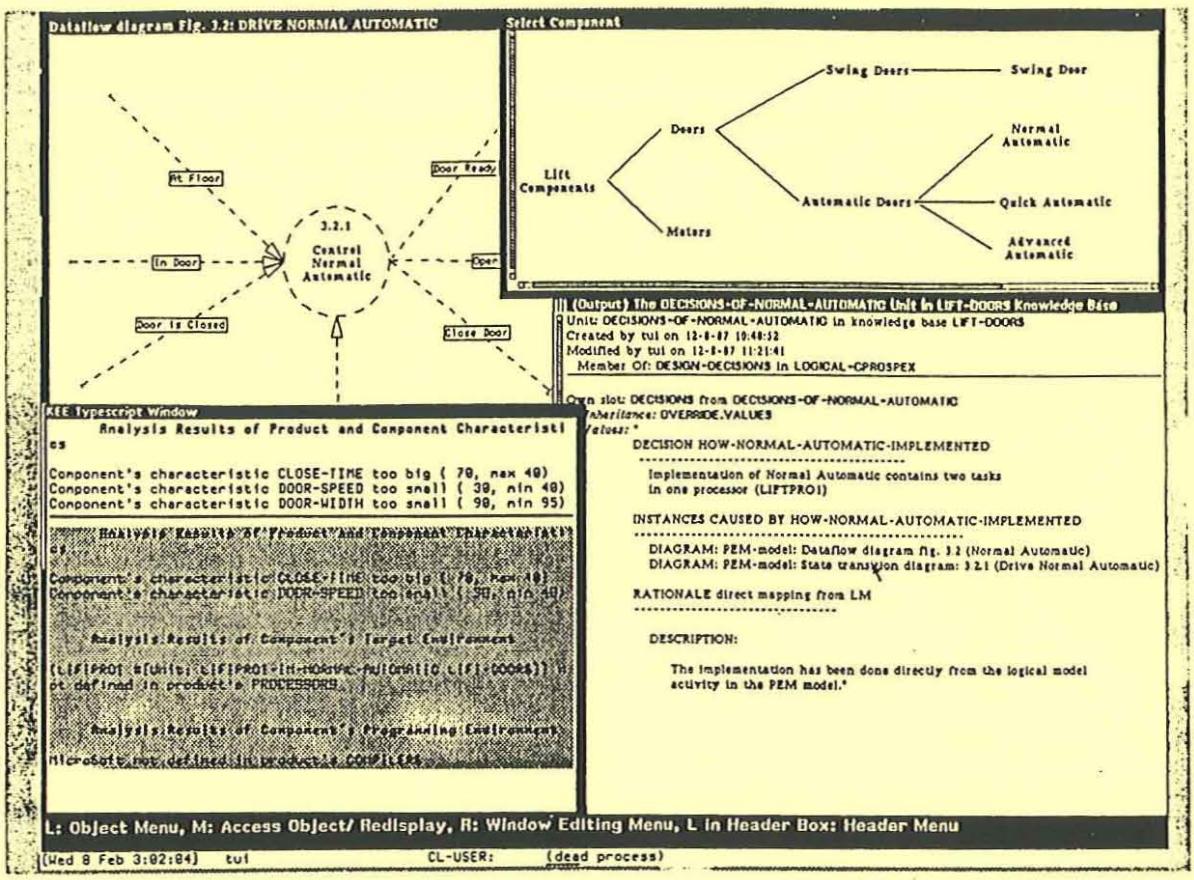

Figure 7. The user has an access to the graphical representation of the library components, and he or she can get important information about the characteristics of the components.