SPADE - TOWARDS CASE TOOLS THAT CAN GUIDE DESIGN

Veikko Seppänen, Marko Heikkinen and Raino Lintulampi
Technical Research Centre of Finland
Computer Technology Laboratory
P.O. Box 201, SF-90571 Oulu
FINLAND

Abstract

A CASE tool to assist a designer in the task of guiding the software design process is presented. The architecture of the tool consists of design editors, databases of reusable software components and their descriptors, a dictionary of design control knowledge, a component proposer, a component applier, an agenda, an explainer, a design history recorder and a simulation-based design analyzer.

The implementation of these functional building blocks in an experimental CASE tool is discussed. Experiences gained from the use of the tool in the design of a large real-life telecommunication software system are outlined.

1. INTRODUCTION

This paper reports on a prototype of a CASE tool that helps software engineers make design decisions and test the results immediately by graphical simulation.

We call the task of guiding a software design process navigation [Seppänen 1990]. The context of our work is a reuse-driven design process of real-time embedded software, derived from a popular structured design method [Ward 1986]. We believe, however, that some of the results are applicable in a wider context as well. We demonstrate the results using the design of a considerable large commercial telecommunication system as a real-life example. This system, called Layer 1, represents well the kinds of software products developed by the industry at present.

The emphasis of our demonstration is on the allocation of specified data transmission functions to concurrent processing units, such as real-time tasks and interrupt handlers. Some aspects related to interconnecting these units and making them interoperate using for example mailbox-based task communication and synchronization principles, are also addressed.

This choice of the focus is justified by the fact that such automated design tools as code generators [Anon. 1987] will mechanize the subsequent module design and software implementation activities in the longer run. The drawback of the emphasis on this part of the software design process is the lack of a direct connection with the software embedded in the target hardware. This is compensated in part by the use of advanced simulation-based analysis features.

Since the tool that we have developed is interactive rather than automatic and makes explicit design knowledge, we view it as a knowledge-based design assistant. The schemas of knowledge-based software assistants, as presented originally in [Balzer *et al.* 1983] and later by others [Puncello *et al.* 1988], are too vague for addressing our specific topic of interest. Compared with the architectures of these systems, our focus is a mechanism that guides the software construction process interactively and automates some

aspects of design reuse. The tool aids a designer in the identification and selection of reusable software components used to implement a given system specification.

To evaluate the core of the tool, an experimental design assistant called Spade has been implemented using the Smalltalk-80 language and its programming environment [Heikkinen 1990]. The architecture of Spade is shown in Figure 1. Its major building blocks are called the component proposer and the component applier. The former is a mechanism for identifying and choosing reusable software components, whereas the latter is needed to integrate the chosen components as parts of the artifact being designed.

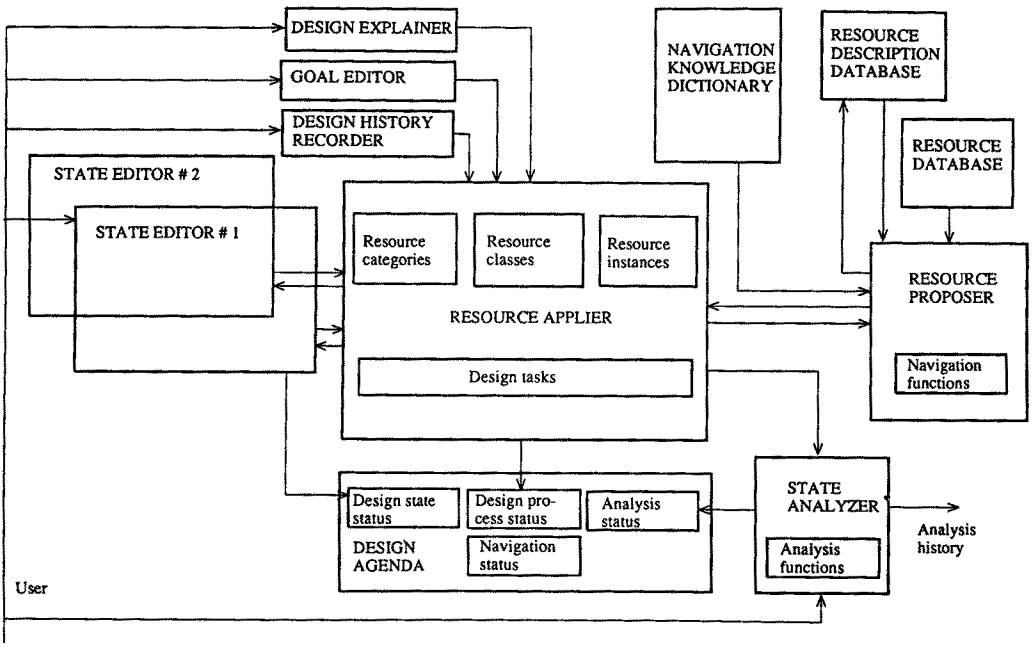

Figure 1. The architecture of Spade

Some clerical design support is provided by the Espex environment [Lintulampi 1990], with which Spade has been integrated. More importantly, Espex is used as a simulation-based interactive analysis tool to test the resulting software designs by graphical animation. The combination of Spade and Espex therefore forms the basis of a CASE tool for rapid prototyping.

2. THE ARCHITECTURE OF SPADE

The functional architecture of Spade consists of the following parts: design editors, databases of reusable software components and their descriptors, a dictionary of design guidance information alias navigation knowledge, a software component proposer and a component applier, a design agenda, an explanation facility, a design history recording mechanism and a simulation-based design analyzer.

We now discuss the purpose and implementation of each of these functional building blocks. We demonstrate them using the design of Layer 1 as a running example throughout the paper. Therefore, we start by a brief overview of the example.

2.1 AN OVERVIEW OF LAYER 1

Layer 1 is a part of a commercial software-intensive telecommunication system where several portable user devices are connected with central stations. The main task of Layer 1 is to take care of physical data transmission between a user device and one of the central stations that are available. The software implementation of Layer 1 is executed by the MCU (Master Control Unit) processor of the user device. The processor is connected with a number of other local processors via a shared bus.

MCU is a standard microprocessor, supplied with an in-house operating system that is based on a pre-emptive priority-driven scheduling scheme. Within the physical context of MCU, Layer 1 is interconnected with three upper data transmission layers. In addition, it is connected with the interprocessor bus. We focus below on the design of the part of Layer 1 that serves these interconnections.

The demonstration of Spade using Layer 1 as an extensive example covers several system development activities:

- Respecification of the system as an executable structured specification [Ward 1986], using the Espex tool.

The purpose was to specify a large real-life system using a prototype of a tool that supports operational specifications [Agresti 1986]. We do not address this activity at length, since it is outside the design-centered context of the paper.

- Allocation of the specification to concurrent processing units, using Spade as a navigation assistant.

The use of Spade to aid a designer in allocation was studied. One of the allocations that we evaluated is similar to the corresponding design constructed in the actual systems development project, independently of our study. Another allocations are variations generated by applying different design criteria.

- Testing of the properties of the resulting designs.

The properties of the alternative designs were evaluated using the simulation facilities of the Espex tool, available as an integrated part of Spade.

2.2 THE DESIGN EDITORS OF SPADE

A specification to implement and the designs derived from it are represented in Spade externally using graphical notations, as in most commercial CASE tools [Anon. 1988]. A graphical user interface has become a *de facto* standard in modern software engineering facilities. In state-of-the-practice approaches, however, visual system representations are often defined informally and not treated rigourously.

One way to compensate for this is to supply graphical notations with textual specifications that can be manipulated mechanically. Another approach is to describe graphical notations in terms of other more rigourous representational formalisms. Object Petri nets applied in Spade are an example of the latter approach [Pulli 1988].

A design editor is used to perform manual design activities. Different types of graphical and textual editors for constructing design artifacts are perhaps the most common form of software design tools. Design editors for describing other than functional design requirements are, however, still rare. The means used in some CASE tools to define attributes that complement functional designs can be used to represent non-functional requirements (or, design goals).

A frame-based or an object-oriented knowledge representation scheme allows a similar approach in a more rigourous manner. Property attributes are treated as explicit and integral parts of a comprehensive design notation, rather than as informal inscriptions of graphical symbols. An object-oriented scheme is used in Spade.

The design goal editor built for Spade includes two different views, one for implementation constraints and another for performance requirements. Goals are represented as attributes attached to a functional representation. They are needed during a design session to help in making design decisions.

Functional design descriptions are used in Spade as finishing conditions for the design process, for example to test if a specification is entirely allocated to concurrent processing units. They also force the sequencing of some steps in the design process, e.g. by requesting interconnection of some allocated processing units. Non-functional requirements act in practice as constraints that cannot be violated during the design process.

The original structured specification of Layer 1 provided by the production organization consists of twenty pages of dataflow and state transition diagrams, forty textual specifications for the lowest-level data transformations (written in structured English) and sixteen pages of data declarations organized into a data dictionary. Fourteen pages of graphical models, thirty transformation specifications and the corresponding data declarations were chosen as the context of the demonstration of Spade.

An executable version of this specification was first produced, by the aid of the design editors of Spade. It is expressed using object Petri nets and consists of 263 data transformations and states (shown as places of object Petri nets, called T-elements) and 216 data stores and data flows (shown as transitions of object Petri nets, called S-elements). 176 of the T-elements are at the lowest-level of the hierarchical structured specification. One part of the executable specification is shown in Figure 2.

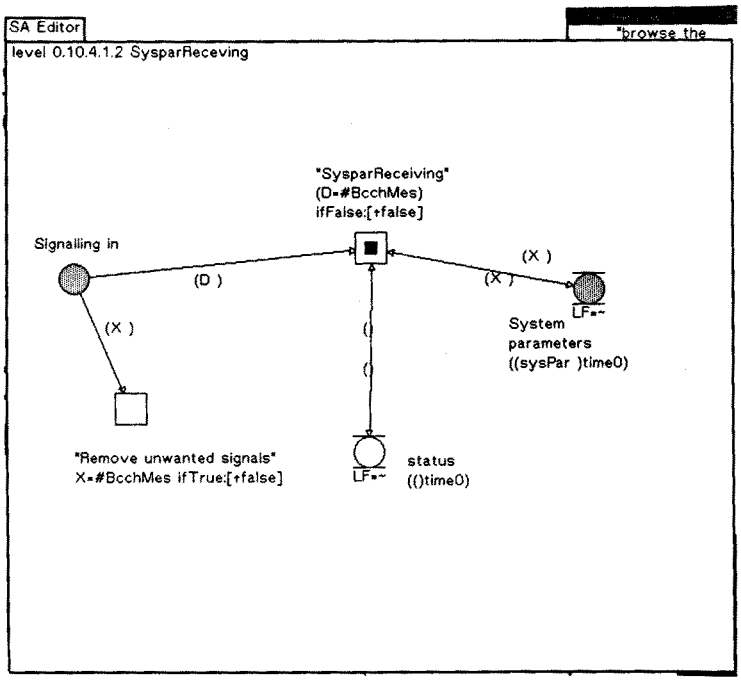

Figure 2. A snapshot of the specification of Layer 1

Some of the elements of the specification are used to describe the target system in which to embedded Layer 1, i.e. the behaviour of the terminators in the structured context diagram of the system [Ward 1986]. This is necessary for the execution of the specification. It can also be considered as an extension of an ordinary functional specification of a computer system with a separate *target system specification.*

The uppermost level of the functional specification of Layer 1 consists of four groups of functions. *System control* functions interact with the upper data communication layers and thereby activate the other functions of Layer 1. *Data sending* functions transmit data frames to the shared bus, to be sent further to the active central station. *Data receiving* functions operate in the other data transmission direction than the sending functions. *System parameter handling* functions, a snapshot of which is shown in Figure 2, read and interpret system parameters received from the central station and store them for the use of the other functions.

Layer 1 has a few distinguished states, where particular functions are active or operate in a particular way. These states are *initialization* in the power on situation (a connection with some central station is established), the *idle* mode (a connection has been established, but the user is not active), *start communication* (the user communicates or is being communicated with the central station), *stop communication* (an active communication session is closed) and *shut-down* (the power of the user device is switched off). In addition, the reselection of the central station with which the user is interconnected can take place on the fly when the power of the system is on.

With regard to non-functional specifications we focused in the case study on one of the basic performance requirements of Layer 1, described using the goal editor of Spade. The requirement is derived from the rate of data frames transmitted to and received from the central station by Layer 1. The duration of one frame is 2.5 milliseconds. This is the required operational speed of many functions in Layer 1. In particular, data transmission functions have to be synchronized with the frame frequency. The specification of this requirement as the required execution times of data transformations of Layer 1 was done on the basis of estimates given by one of the designers of the system. Minimum and maximum stimulus-to-response requirements and the availability of responses were specified.

The implementation constraints that were used in the demonstration of Spade to guide the selection of reusable software components (e.g., memory constraints) are less accurate than the above performance requirement. They were not validated by the designers of Layer 1, but estimated from the number and size of elements in the specification. The reason is that the choice of the target hardware and the design of the system software integrated with Layer 1 were not yet finished at the time of the demonstration. We do not therefore discuss the role of modelling and using implementation constraints in design guidance in this paper.

2.3 THE SOFTWARE COMPONENT DATABASE

Reusable software components used to implement a given specification need to be stored in a repository. Component descriptors are stored separately from the corresponding components in Spade, in order to provide more flexibility in reuse. Such design tools that do not make this distinction are inflexible with respect to the following aspects:

- Optimization of the physical storage of reusable components.

Only differences between the variants of a component need to be stored, as opposed to distinct copies of almost similar components. Each variant can be described separately and the component applier may build the selected variant when it is retrieved for application. A similar technology has become common in code-level configuration management systems but is still missing from most existing CASE tools.

- Modification of component classification and selection schemes.

A snapshot of a simple classification scheme of reusable software components to implement Layer 1 is shown in Figure 3. There is, however, no single taxonomy of reusable components that fits all needs. The same components must therefore be classified in several different ways.

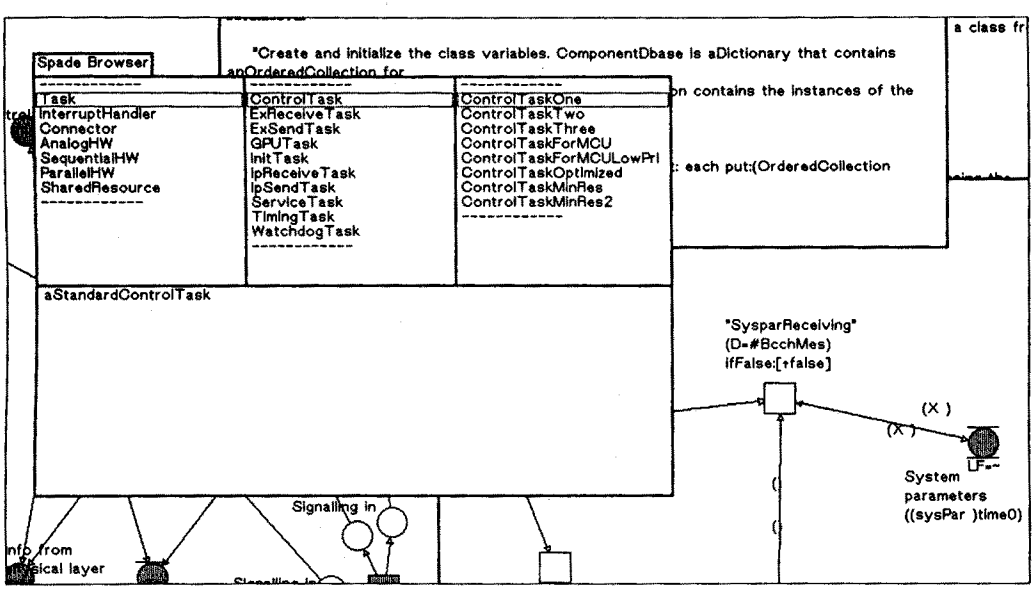

Figure 3. A classification of reusable software components

It is easier to reorganize a component description database than to change the organization of components themselves. Several alternative classifications of the same components may coexist. Classification schemes can also highlight different properties of the same components, such as reliability, extendability and efficiency in addition to functionality, to help the designer.

Well-established database technologies illustrate the importance of keeping physical data models, logical data models and varying user views separate. In mechanical design systems the basic separation of concerns between part descriptors and true physical parts is obvious. In software-intensive design tools for constructing software systems the three aspects may become unnecessarily blurred.

Different users may also need different ways (or, domains) of viewing component descriptors. For example, it is possible in Spade to identify all data transformations that are available (on the basis of one classification of design concepts, called the embedded systems modelling domain) or just those data transformations used in some particular types of systems (classified in an application domain).

Component descriptors specify the properties of a reusable piece of software, as illustrated in Figure 4 for the "L1SioReceiveTask" component used to implement Layer1. The component proposer mechanism of Spade uses descriptors to identify and select appropriate components, given a specification and design requirements to achieve.

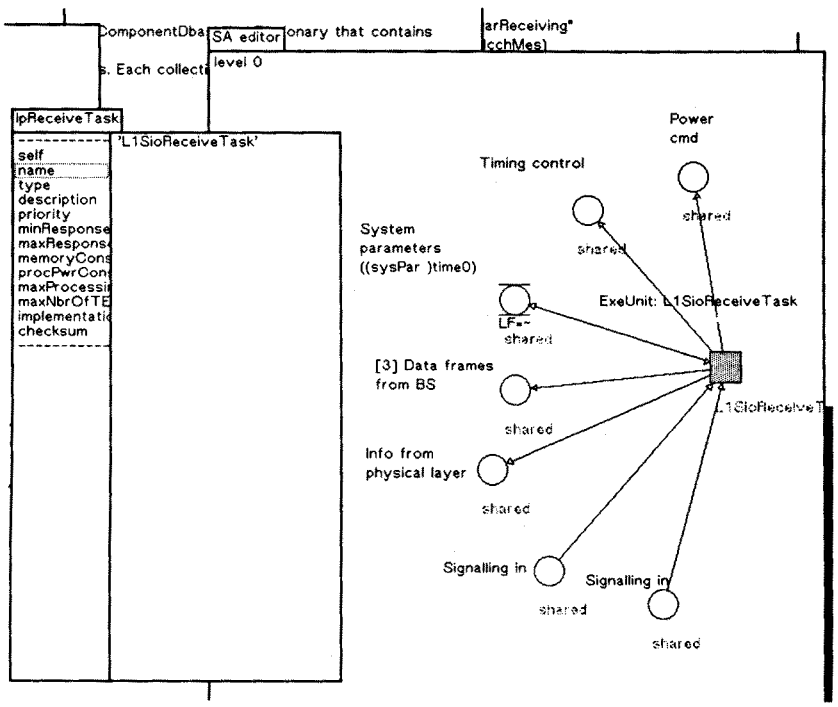

Figure 4. Software component descriptors

A simple copy-and-edit approach has been taken to the capture of reusable components in Spade. A chosen part of a design representation can be stored as a software component. Alternatively, generic templates (or, schemas) for certain types of designs can be defined and reused. This corresponds to the kind of an approach to reuse taken in some commercial CASE tools.

We have left the capture and representation of other reusable components than instances of designs out of the context of Spade. A justification for the simple reusability approach is our focus on the kinds of properties that a reusable component may have and on their use in guiding design, as opposed to the problem of how to acquire software components that do have such properties.

2.4 THE NAVIGATION KNOWLEDGE DICTIONARY

We classify design guidance knowledge into generic, design task-specific and domain-specific categories [Seppänen 1990]. An object-oriented scheme is used in Spade to represent software design knowledge. Object-oriented representational formalisms are common in knowledge-based software and hardware design tools.

A frame-based knowledge representation scheme could have offered some additional features (e.g., truth maintenance and integrated production rule representations) not contained in the Smalltalk-80 language used in Spade. The benefits of the integration with the Espex tool implemented in Smalltalk-80 were, however, considered greater than the lack of such features.

Design guidance knowledge in Spade is mostly coded in Smalltalk-80 methods. These methods evaluate alternative software components, represented as classes and instances of objects, and apply the selected components. Some navigation knowledge is therefore incorporated both in the design editors and in the component applier mechanism. It would be convenient for a designer to have a centralized design knowledge base available. This aim is, however, in conflict with the need to decentralize navigation knowledge by locating it close to the mechanisms of the design tool whose operation is being controlled.

The conflict between distribution and centralization be resolved by a knowledge dictionary. It makes explicit the organization of navigation knowledge and links together different pieces of knowledge distributed over the design tool. A dictionary is flexible in the sense that it does not depend on the physical location or particular form of the knowledge. This is contrary to the centralized rule base approach used in some proposed architectures of design expert systems. The dictionary is like the component description database discussed above, because it merely organizes various pieces of design knowledge.

A designer has a single entry point to navigation knowledge. Moreover, it is possible to record the use of the knowledge in order to explain design decisions and to record design histories. We discuss the importance of these features below. It is also very necessary to know how different pieces of knowledge are reused in order to evolve and maintain a navigation knowledge base. A distributed approach without any central access to the knowledge would make this difficult.

It is relatively straightforward to implement a navigation knowledge dictionary in an object-oriented environment. A dictionary is an object that is linked with the set of other objects and methods used to represent navigation knowledge. The dictionary object is informed of the use of any piece of navigation knowledge. A dictionary has been specified for Spade using this kind of an approach, but is not yet implemented as this is being written.

The details of how different types of design knowledge are represented in Spade are left out of the context of this paper. The goal, from our viewpoint, was to demonstrate that the representational formalisms of object Petri nets, as a rigourous foundation for a popular strutured design method, and the Smalltalk-80 language make it possible to codify some essential types of navigation knowledge.

Implementation-specific navigation knowledge was the easiest type of design knowledge to codify in Spade. The reason is that we focused on algorithmic design selection criteria calculated from various metrics attached to reusable components. The implementation of such algorithms in an object-oriented programming language is not difficult. A criterion may, for example, rank reusable software components according to their memory and power consumption characteristics and their processing times.

Certain design criteria derived from the generic guidelines of the structured design method that we apply were also codified in Spade, to demonstrate navigation knowledge specific to a particular design method. As an example, one piece of knowledge performs a consistency check for hierarchical design descriptions by analyzing their internal network consistency.

2.5 THE COMPONENT PROPOSER AND APPLIER

The component proposer and applier mechanisms are integrated in Spade and implemented as a browsing facility called *Spade Browser* [Heikkinen 1990]. The browser is based on the system browser of the Smalltalk-80 programming environment. When used manually, it corresponds to a non-automated component applier.

However, a collection of Smalltalk-80 methods can also be activated that automatically scan the component description database. This provides for a prototype of a component proposer mechanism. Reusable software components identified as plausible for implementing the given specification are evaluated and chosen on the basis of the kinds of non-functional requirements and design selection criteria discussed briefly above.

A designer may accept or reject a design choice made by the component proposer or first ask for an explanation to justify it. If a component is accepted, it is applied. The application is straightforward in Spade and results, for example, in the marking of an allocated specification element with the name of the processing unit to which it is allocated.

In addition to the allocation decision, the execution orders and times of the lowest-level data transformations in each real-time processing unit need to be chosen using Spade Browser. The priorities of the units must also be set. The result is then reconfigured so that at the uppermost level of the design description only the units and their interconnections are shown. The design representation constructed before the reconfiguration is saved automatically, to support a one-step automatic design backtracking procedure.

Spade Browser, when initialized, shows the classes of reusable software components available in the database. The given non-functional requirements are used to guide the search for appropriate components, to which to allocate the given specification. If the designer accepts the choice of a component, the parts of the specification marked as allocatable are allocated to the selected component.

Figure 5 shows Spade Browser in operation. The "SysparReceiving" data transformation of the specification of Layer 1 is being allocated. The component proposer recommends the "L1SioReceiveTask" instance of the "IpReceiveTask" class of interprocessor communication units for the allocation, which was shown earlier as an example of a reusable software component. The designer can accept or reject the choice, ask for an explanation and record the decision that is made.

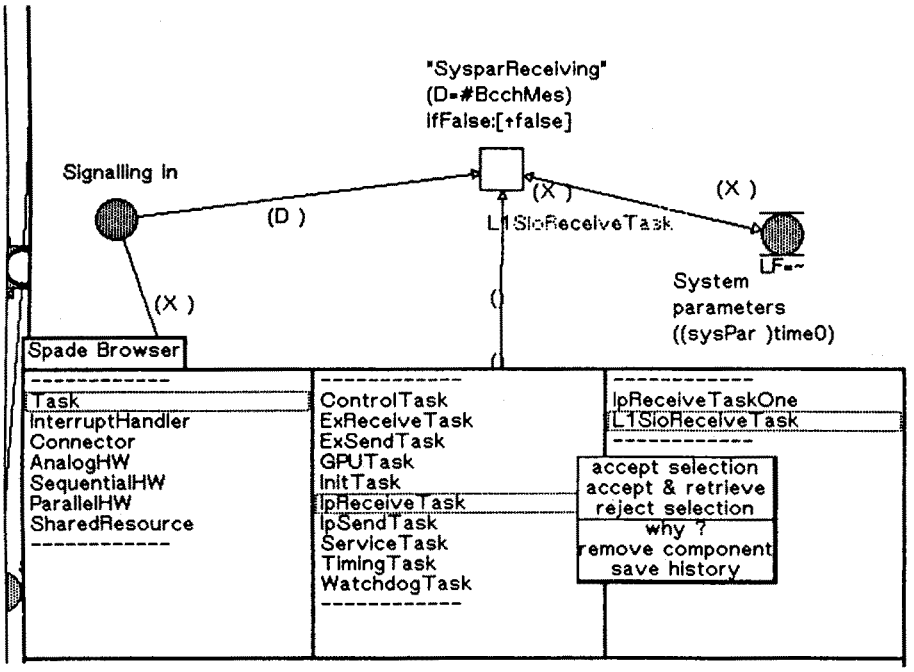

Figure 5. Spade Browser in operation

A number of alternative allocations of Layer 1 was studied using Spade Browser, after the executable version of its specification had been constructed. The experimental nature of both Espex and Spade restricted in practice the number of alternatives that could be studied during a reasonable period of time. We now make an overview of the allocation of a set of Layer 1 functions, as an example of the design alternatives investigated. The parts of Layer 1 on which we focus here are the data sending and receiving functions. The handling of system parameters is also addressed to some extend.

One of the alternative allocations of the Layer 1 specification that was designed using Spade includes separate real-time processing units for receiving and sending data to implement the corresponding functions in the specification. No distinct processing unit for handling system parameters is, however, provided. The corresponding functions in the specification are in part divided between the receiving unit and a system control unit that takes care of main controlling of Layer 1.

Timing requirements result, however, in the need of allocating some system parameter handling functions to the processing units of Layer 1. The reason is that during the initialization phase it is the responsibility of Layer 1, and not the upper layers, to obtain system parameters. Therefore, parts of the system parameter handling functions had to be allocated to the data receiving unit in this design alternative.

Another allocation alternative was produced and compared with the above solution. This design includes concurrent processing units for sending and receiving data, but also a distinct unit for handling system parameters. System parameters are now received always via the receiving unit. They are no longer updated by the upper layer functions or by the system control unit.

The benefits of this allocation, compared with the former one, raise from the fact that the handling of system parameters actually consists of two functionally separate activities that are now appropriately allocated to distinct processing units. One of the functions deals with data communication towards the central (now allocated to the data receiving unit) station and the other takes care of interpreting the received parameters (now allocated to the system parameter handling unit).

This design alternative is simpler also because a duplicated data store can be removed. An extra store would be needed in the first allocation, to save the particular system parameters needed for the processing

units of Layer 1 themselves. Another copy of the parameters would be managed by one of the upper layers.

The details of the demonstration, including a complete record of an allocation session produced by Spade, are discussed elsewhere. We describe here shortly the recorded search of reusable software components for implementing the data sending functions of Layer 1. In this case the search for a unit to which to allocate the data sending function has failed, as shown in Figure 6. There is no component available that would meet the given performance requirement. The requirement is therefore relaxed by increasing the maximum availability time of a response. The results of the new search are successful. Spade Browser now proposes the "L1SioSendTask" unit and displays it to the designer.

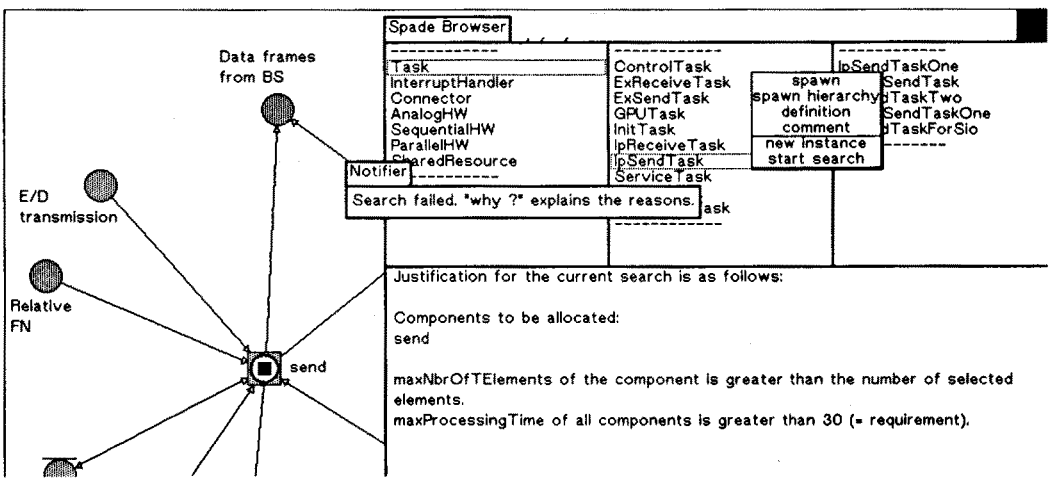

Figure 6. A failed search of reusable components

When the choice is accepted, the corresponding design decision is recorded into a history file. The size of the memory and the processing capacity of the MCU processor are decreased by the amount consumed in this allocation decision, which is an additional implementation constraint for the subsequent allocations.

The allocation information is denoted by the name of the unit written next to the data sending function. The design agenda is also updated and shows that eighteen lowest-level data transformations that describe the data sending function in detail become allocated. We now discuss the design history recording, design explanation and agenda mechanisms of Spade in more detail.

2.6 THE DESIGN AGENDA, EXPLAINER AND RECORDER

The design agenda of Spade, called *Spade Agenda*, corresponds in part to the agenda mechanisms in such knowledge-based design tools as Pride [Mittal *et al.* 1986] and IDeA [Lubars 1986]. Spade Agenda is, however, not an active scheduler of design goals or activities as agendas in Pride or IDeA. Rather, it is a bulletin board that shows the status of the design process being conducted and important information about the resulting designs.

This is of practical value when dealing with real-life designs and was clearly observed in the design of Layer 1. Most commercial CASE tools do not include any aspects related to agendas, simply because

they leave the design process completely implicit and focus only on design artifacts. Figure 7 shows Spade Agenda, after one of the alternative allocations of Layer 1 has been completed.

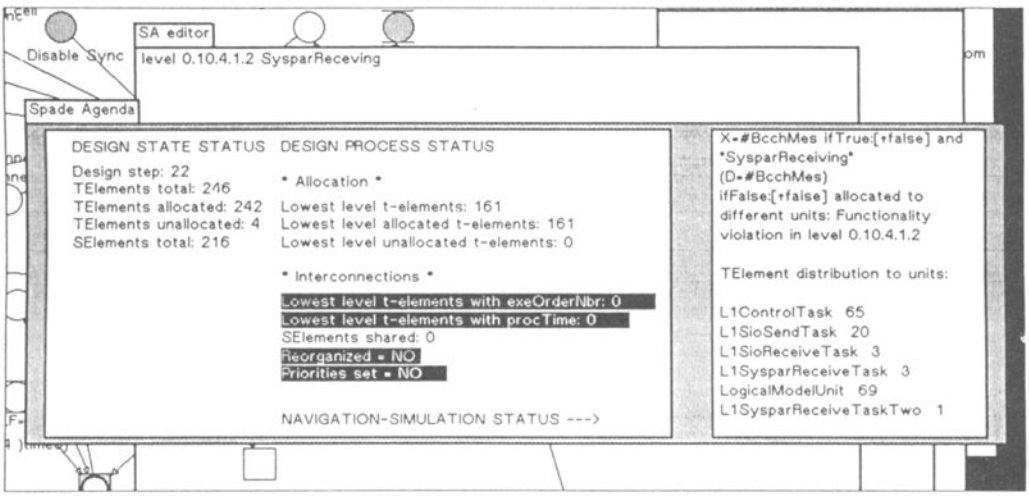

Figure 7. A snapshot of Spade Agenda in operation

The navigation status of the agenda informs about the progress made by the component proposer mechanism. Navigation and simulation status fields are mutually exclusive. Only one of them is active, depending on whether design or analysis is performed. The navigation status field of Figure 7 shows a warning on the violation of the "Functionality criterion" design principle, but also information about the distribution of T-elements (alias data transformations) allocated to chosen reusable processing units.

The design process status field of the agenda shows the number of lowest-level unallocated data transformations of a specification, to make the status of allocation explicit. The zero value of this field in the figure indicates that allocation is complete. Similar information about the status of the interconnection and interoperation results is provided. The reversed display, as shown in the figure, indicates pending design activities.

The design state status field shows information about the current design. The number of design steps that resulted in the current design is displayed. An operation that update the design editor window is counted as a design step. The size of the design is also indicated, for example its total number of S-elements (i.e., data flows and data stores).

The simulation status field of the agenda maintains knowledge about the completeness of design analyses and summarizes their results. Spade integrates the existing simulation tracing facilities of Espex with the agenda mechanism so that the analysis status is updated dynamically as graphical animation-based simulation proceeds. We show an example of the use of the simulation status field of Spade Agenda in the next section.

The design history recorder of Spade can be used to activate the recording of design sequences and to view any recorded sequence. A straightforward recording mechanism has been integrated with Spade Browser that makes it possible to store a design history into a text file, as illustrated in Figure 8. The history may be idealized or not, depending on the decision to also record backtrackings.

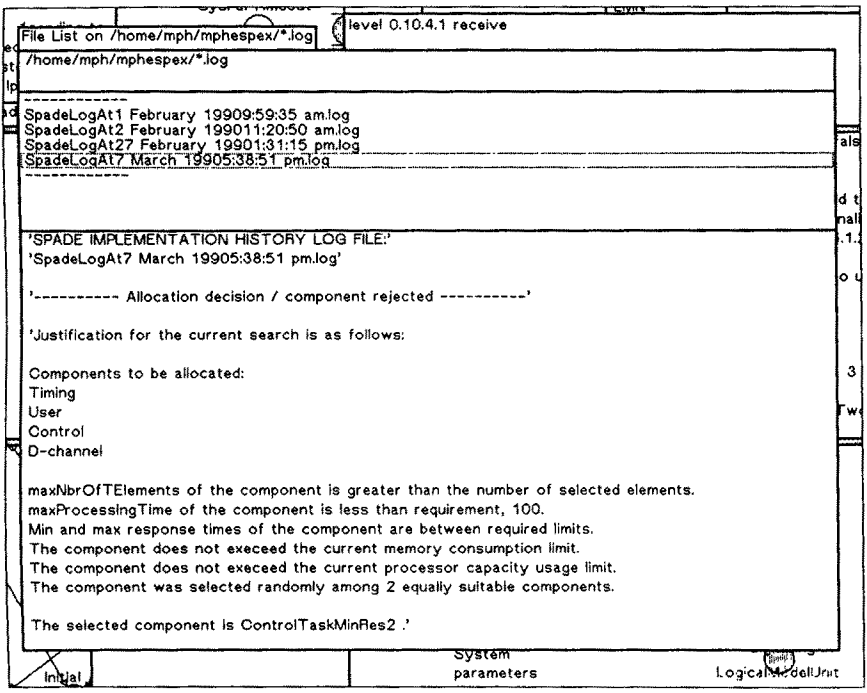

Figure 8. A section of a recorded design history

Most existing CASE tools do not include mechanisms to record design histories. In many cases the reason is that the design process is not modelled in the first place. Design decisions, if modelled explicitly, can be used to record design histories [Biggerstaff 1989, Potts & Bruns 1988].

In highly mechanical design tools, such as code generators for real-time systems [Anon. 1987], automation is considered as a solution to the problem. The implicit justification is that it would always be easier to produce a new implementation from scratch than to replay design histories. However, in order to explain the behaviour of a design system during a design session, it is necessary to make design criteria explicit independently of the number of decisions that are made mechanically.

A design explanation facility has therefore also been implemented in Spade, as an integrated part of Spade Browser. Both the alternative classes of reusable components available and the reasons for choosing a particular component can be justified. The explainer is thus a design-time view of some sections of a design history. The explainer is simple, however. Further research is needed to study the role of explanations in the design of software systems.

2.7 SIMULATION-BASED DESIGN ANALYZER

Design guidance benefits from the analysis of design results. Spade has been implemented as a part of the Espex simulation tool for this reason. Espex also benefits from Spade because it does not itself possess any assistance in the choice of alternative designs, before they are actually generated.

Espex covers the simulation of specifications and architectural designs. The former are simulated by executing them using the Ward scheduling scheme [Ward 1986], as represented in [Pulli 1988]. The latter are executed by the kind of a pre-emptive scheduling scheme which is used in most existing real-time operating systems [Lintulampi 1990]. The two scheduling schemes can be integrated, so that parts of a

specification not yet allocated to concurrent processing units are considered as a single "logical unit" scheduled using the Ward scheme. This is convenient for the incremental implementation and prototyping of large real-life specifications.

We briefly introduce the simulation features of Espex as a counterpart to the properties of Spade. Espex combines three types of time-based simulation techniques [Lintulampi 1990]:

- In time driven simulation systems stimuli and internal data are collected to a list and sorted according to the time. A simulation clock is incremented to the next time in the list. After each increment the behaviour of the [simulated] system is analyzed.

- In data driven simulation systems time is connected to the data. Each data flow, store and buffer of the simulated system has a delay time which a data item has to wait before it is available for a transformation.

- In a process driven approach a simulated system is divided to indivisible processes or transformations to which processing times are connected. Data tokens have time stamps that are used by the simulation system in the activation of the processes.

A unique time stamp is attached to each data token in a design represented as an object Petri net. This stamp is compared with the global simulation time, managed by the simulation system, to decide whether the token fires an object Petri net transition or not. Delays and lifetimes that may be indefinite are attached to data flows and buffers, to simulate communication time and the availability of data.

Processing times need to be estimated by the designer for the lowest-level data transformations allocated to processing units. A processing time is relative to the global simulation time. Highest-priority processing units are assumed to be interrupt handlers.

Mailboxes that provide communication between processing units are modelled as S-elements of object Petri nets. The timing of waits in mailboxes must be modelled explicitly. A piece of shared code between processing units is modelled as a software resource that can be executed only by one unit at time. Shared memories are modelled as stores whose content is not physically available while being read or written by some unit.

Espex supports simulation by the animation of designs, but also by providing a set of execution monitoring features. The animator simulates graphically the movement of input and output data tokens in a specification or a design, represented as an object Petri net. A snapshot of the simulation of the system parameter handling unit of Layer 1 in one of the alternative allocations is shown as an example in Figure 9.

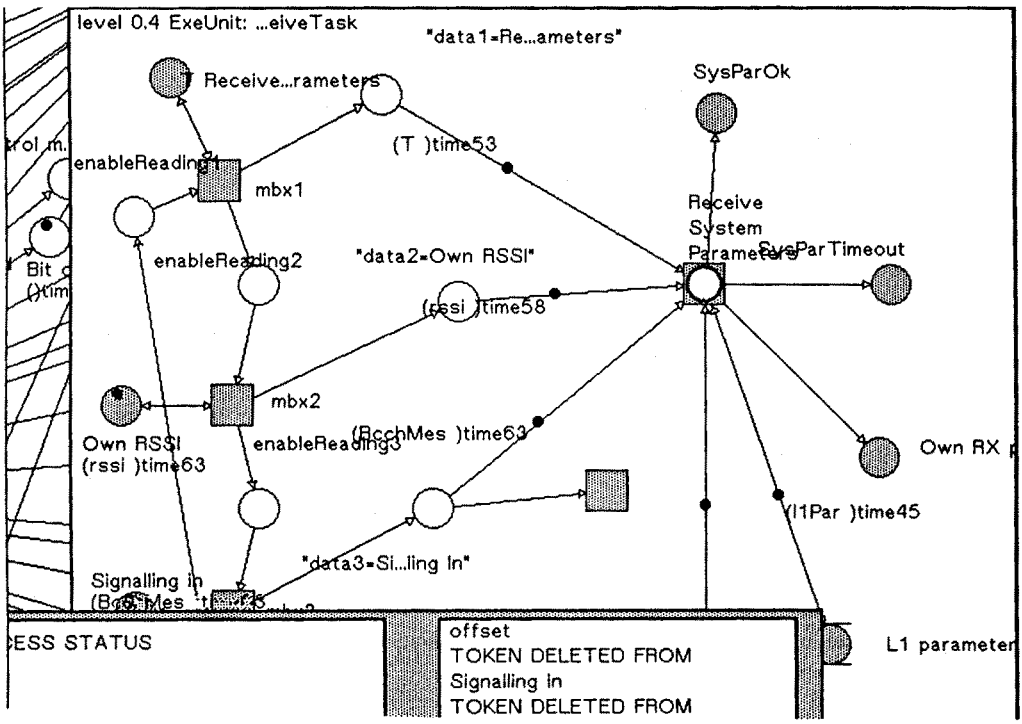

Figure 9. Simulation of Layer 1 using Espex

Layer 1 reads system parameters from the central station at the end of its initialization phase. Parameters are transmitted via the data receiving unit into a timer-controlled mailbox so that each data flow in the interface between the units has its own mailbox. Tokens, shown as small dots in the figure, that move during simulation indicate when data is being received by the unit.

The simulation status field of Spade Agenda now displays simulation results. For example, the message "Token deleted from Signalling in" indicates that a new data element has arrived into the "Signalling in" data store, before an earlier data element has been read. Because the element is of the type that is always written over, the earlier data is lost. This indicates that there are problems in the synchronization of the processing units. The system parameter handling unit does not get enough processing time to read and process the incoming data.

The execution monitoring facility of Espex allows for the off-line investigation and interpretation of simulation results. The following features can be detected [Lintulampi 1990]: buffer overflows (indicates problems in allocation), the overwriting of discrete data and control flows (indicates that the interoperation of processing units is not correct), events not accepted by control transformations (indicates problems in interoperation), conflicts in the reservation of shared resources (indicates problems in either allocation or interoperation). The contents of active data and control flows and buffers can also be recorded, in order to monitor the operation of unit interconnections. The recording of the activation and inactivation times of a data transformation provides a foundation for efficiency measurements.

3. SUMMARY OF THE EXPERIENCE

A considerably large real-life specification was allocated in a number of different ways to a set of reusable concurrent processing units, using Spade as a design assistant. The results were analyzed using the simulation features of Espex.

As a summary of the demonstration, we discuss our experience from the use of a database of reusable software components, the role of non-functional design requirements, the design guidance knowledge and methods applied in Spade, the clerical support provided for the design process and the graphical simulation of designs.

- A database of reusable components is an effective basis for evaluating alternative designs in any type of software development.

Although only a limited repository of reusable components was built for the demonstration of Spade, its use as a storage of alternative designs was very convenient. Most real-life software designs, not only of embedded systems, tend to become large and complex. A good approach for managing the complexity is to partition a design into a set of smaller entities and to search for existing solutions to the individual subproblems in the component database.

The possibility of having logical units (i.e., parts of system specifications) integrated with concurrent processing units (i.e., parts of software designs) is a practical means to help such partitioning. An approach to develop and incrementally mix high-level specifications, architectural designs, detailed designs and true software implementations seems very attractive and powerful on the basis of our experience with Spade and Espex.

We did not find a single alternative design produced and documented for any system that we analyzed in several case studies for the construction of Spade (only Layer 1 is reported in this paper as one of the examples that we investigated). Most of the systems that we studied had been designed using modern CASE tools. One of the reasons for this lack is that it is overly time-consuming to try to build alternative designs using such tools. Using Spade, it was quite easy to partition designs into reusable software componenets, to store them and to construct alternative allocations from different configurations of these components.

One explanation for this is that in any types of hierarchically organized design representations a considerable number of lower-level elements often become treated automatically, as a consequence of a design operation performed at a higher level. For example, in the case of the data sending function of Layer 1 discussed above, eighteen data transformations become allocated automatically. A system like Spade makes it thus possible to eliminate plenty of unnecessary manual work.

- Integration of non-functional requirements and functional specifications eliminates design revisions.

Only one non-functional requirement had been documented as part of the original structured specification of Layer 1. There are no good means for expressing such requirements when using current CASE tools. This effectively prevents from making explicit various types of non-functional quality requirements for software-intensive systems at present.

As an example of the importance of non-functional requirements, consider the timing information that was added to the functional elements of the specification of Layer 1. It provided a realistic basis for estimating the efficiency of different allocations that we studied. In many cases making such information explicit and verifiable is crucial for the success of the design of a real-time system.

Even the kinds of simple implementation constraints available in Spade made it possible to follow the consumption of such implementation resources as memory space and the sufficiency of the processing power in a systematic way. There are no such means available for the designer in the CASE tools used routinely at present. The consequences are well illustrated by the kinds of design revisions that have to be

made regularly in software development projects. What is still needed in design tools like Spade is a link from code-based software metrics to the quality indicators of specifications and designs.

- Design guidance is needed in the implementation of real-life system specifications.

The amount of navigation knowledge encoded in Spade at the time of writing this is limited. However, the architecture of the tool makes it easy to extend its knowledge. The number of most critical design criteria for certain domains of applications may to our experience not be very large. It is also not difficult to capture and customize modelling-specific navigation knowledge from textbooks and past design models. The existing technologies used by a production organization form a sound basis for making explicit technology-specific design criteria.

For inexperienced designers several types of knowledge are useful, especially application-specific design criteria and knowledge of well-established implementation techniques, such as the construction of an interface to a serial data communication line. One of the clear benefits of making design knowledge explicit is to recognize readily inapplicable solutions.

- Clerical design support decreases errors in routine activities and ensures the completeness and consistency of the results.

Most modern CASE tools include some clerical design support and means for ensuring the consistency of the results. The design agenda of Spade, however, provides a considerably stronger approach to keep track of the completeness of a design process. It forces the designer to attack unfinished activities and constantly displays information about design and analysis results. The design recording and explanation facilities provide additional means to display and store design information.

The agenda was very useful in the design of Layer 1 and eliminated a large number of manual follow-up activities. The size of the specification of Layer 1, as the specification of any non-trivial computer system, is much bigger than what a person can comprehend at the same time without difficulty. The agenda mechanism keeps the designer well-informed on the status of the design process and its results.

- Integrated analysis is a must for incremental and iterative software development.

It is unrealistic to expect that automated tools would make it possible to construct perfect specifications and designs. Rather, they should make the experimental and evolutionary aspects of software design more explicit than at present.

This being the case, a computerized design assistant must have means both to construct designs and to analyze them. The integration of Spade and Espex proved to be effective with this respect and resulted in more than the sum of the two tools together.

It is straightforward to generate alternative allocations and to simulate the results the very next moment. Without the possibility of simulations Spade would provide means to produce alternative designs in a controlled way, but lack a practical measuring stick for analyzing the behaviour of the results.

Acknowledgements: The research project carried out for implementing Spade and Espex was supported financially by the Technology Development Centre of Finland, the Technical Research Centre of Finland, and by the following industrial partners of the project: Embe Systems, Insoft, Kone Elevators and Nokia Mobile Phones.

4. REFERENCES

Agresti, W. (ed.). 1986. Tutorial: new paradigms for software development. Washington D.C., IEEE Computer Society Press. 295 p.

Anon. 1987. The state of automatic code generation. Part 2: embedded systems. CASE Outlook 1, 6, pp. 11 - 22.

Anon. 1988. CASE tools for real-time analysis and design. CASE Outlook 2, 1, pp. 1 - 20.

Arango, G. 1988. Domain engineering for software reuse. Irvine, California, University of California, Department of Information and Computer Science. Ph.D. thesis. 195 p.

Balzer, B., Cheatham, T.E. Jr. and Green. C. 1983. Software technology in the 1990's: using a new paradigm. IEEE Computer 16, 11, pp. 39 - 45.

Biggerstaff, T.J. 1989. Design recovery for maintenance and reuse. IEEE Computer 22, 7, pp. 36 - 49.

Heikkinen, M. 1990. Prototyping of executable SART models. Oulu, Finland, University of Oulu, Department of Electrical Engineering. Diploma thesis (in Finnish).

Lintulampi, R. 1990. A specification of the simulation of SA/SD-RT models. Oulu, Finland, Technical Research Centre of Finland, Computer Technology Laboratory. Research report. 25 p.

Lubars, M.D. 1986. A knowledge-based design aid for the construction of software systems. Urbana-Champaign, University of Illinois, Department of Computer Science. Report UIUCDCS-R-86-1304. 202 p.

Mittal, S., Dym, C.L. and Morjaria, M. 1986. Pride: An expert system for the design of paper handling systems. IEEE Computer 19, 7, 1986, pp. 102 - 114.

Potts, C., Bruns, G. 1988. Recording the reasons for design decisions. Proc. of the 10th International Conference on Software Engineering. Singapore, Thailand, 11 - 15 April 1988. Washington D.C., IEEE Computer Society Press. Pp. 418 - 427.

Pulli, P. 1988. Execution of Ward's transformation schema on the graphic specification and prototyping tool Specs. Proc. of the CompEuro'88 conference. Brussels, Belgium, 11 - 14 April 1988. Washington D.C., IEEE Computer Society Press. Pp. 53 - 56.

Puncello, P.P. Torrigiani, P., Pietri, F., Burlon, R., Cardile, B. and Conti, M. 1988. ASPIS: a knowledge-based CASE environment. IEEE Software, March, pp. 58 - 65.

Seppänen, V. 1990. Acquisition and reuse of knowledge to design embedded software. Espoo, Finland, Technical Research Centre of Finland. Publications 66. 218 p.

Ward, P.T. 1986. The transformation schema: an extension of the dataflow diagram to represent control and timing. IEEE Transactions on Software Engineering SE-12, 2, pp. 198 - 210.