In Proceedings of the Sixth International Conference on Software Engineering and Knowledge Engineering, June 1994, pages 358 -- 365.

Cooperation of KBS Development Environments and CASE Environments*

A. Inkeri Verkamo
University of Helsinki, Department of Computer Science
P.O.Box 26 (Teollisuuskatu 23), FIN-00014 Helsinki, Finland

Abstract

We compare the process of knowledge based system development to the software engineering approach of the more traditional application development. Many similarities can be identified in the process structures. The most important differences are in the knowledge acquisition and conceptual modelling of the KBS which have no direct counterpart in the traditional software life cycle. We also study the cooperation of the KBS development environment and ordinary CASE tools. We develop a scheme to support this cooperation. The method is based on transformations between KBS development tools and the repository supporting the CASE toolset. For the case where no repository is available, the scheme can be modified to use transformations between each KBS development tool and the corresponding CASE tool. Since the phase structures of a KBS project and a SE project are fairly similar, the interface may be relatively simple.

1 Introduction

In an artificial intelligence application, such as an expert system, the knowledge base and the inference mechanism are only a part of the application, while large parts consist of traditional software, relatively independent of the AI capabilities. The application can be very complex, requiring skill and experience both in knowledge based system (KBS) development and more traditional software engineering (SE). An important example of the latter is designing the user interface of an application which requires state-of-the-art methods and people specialized in user interface development. KBS applications also have interfaces to other traditional software components such as databases, data communication facilities and interface support.

There is a need of systematic support for the development of KBSs, similar to what is already available for more conventional application design, e.g., data management applications. Some projects aiming at building tools or entire environments for KBS development are already under way (e.g., KADS and KADS II [20], and VITAL [18]).

The purpose of this study is to compare the tools contained in such a KBS development environment to a collection of typical CASE tools, in order to define the most essential similarities and differences of the tools, and to show how their coexistence could add functionalities to both environments. Here we focus our attention mainly to *workbenches*, i.e., collections of cooperating tools, typically supporting a particular life cycle model for software development. We use the close correspondence of the KBS and CASE tools to describe a simple scheme that can be used to provide cooperation between these tools. We further point out the most useful phases for this cooperation and present some examples to enlighten the problems involved. We close by describing some lessons that the developers of each type of environment can learn from the experiences of the others.

For brevity, we use the terms *KBS project – SE project* and *KBS life cycle – SE life cycle* throughout the presentation to refer to the respective projects and development life cycles.

2 KBS life cycle vs. SE life cycle

In principle, the KBS life cycle is very similar to the SE life cycle. Both consist of three main phases: *analysis*, *design*, and *implementation* (Figure 1). Compared to the analysis needed in the SE life cycle, the analysis phase of the KBS life cycle is more complex and consists of two (or three) phases: *requirements*

*This research was supported by the Technology Development Center (TEKES) and in part by the Commission of the European Communities under ESPRIT-II project 5365 VITAL. This paper reflects the opinions of the author and not necessarily those of the consortium. The e-mail address of the author is verkamo@cs.Helsinki.FI

specification, and *knowledge modelling*, where the latter includes both *knowledge acquisition* and *conceptual modelling*.

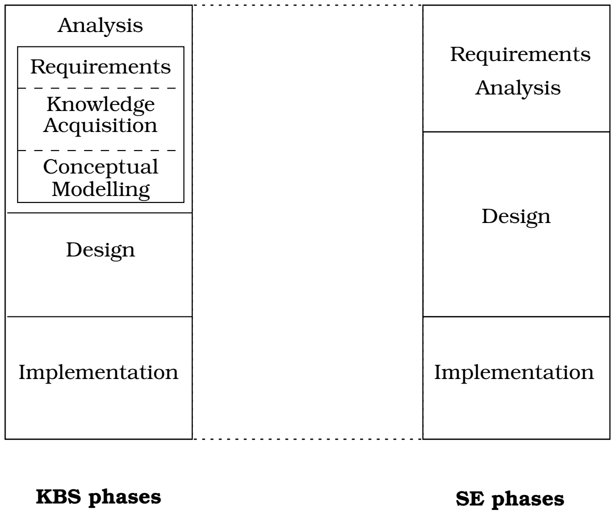

Figure 1: KBS and SE phases.

The design phase can be divided into subphases in several ways. In SE projects the typical subphases are *product design* for designing the general structure of the application, the modularization of the system, and the interfaces between the main components of the system, and *module design* for designing the internal structure of each component and the actual solutions (algorithms, data structures) to be used. Another substructure sometimes used in KBS projects is to separate the user-oriented *functional design*, and the implementation-oriented *technical design*. However, the tasks performed during the design phase of a KBS project and a SE project are very similar. The implementation phase of the KBS life cycle produces *executable code*; consequently it corresponds to the implementation phase of the SE life cycle.

In the SE world, several *process models* are proposed for software development. They differ in the way they manage the phases of the life cycle, and in the amount of overlapping and iteration of phases that they permit. KBS development is typically seen as a sequential process, very much like the classical *waterfall model* for software development [17]. Presumably, minor iteration between consecutive phases is expected, but the movement is mainly in one direction. In SE projects, a strict waterfall model is currently considered unrealistic (see, e.g., [23]): it doesn't represent the way software projects are actually run, nor the way efficient software projects should be run. In reality, there is a strong need for iteration during several phases. For the requirements specification vs. knowledge modelling phases of the KBS life cycle, even iterative development is hardly enough: these phases will have to be performed in parallel, since the requirements are tightly coupled with application specific knowledge [4].

In a waterfall type process model the project is seen as a group of activities producing a number of *process products* or *deliverables*. The four basic process products of a KBS project are the *requirements document*, the *conceptual model*, the *design document*, and the *executable code*. In addition to these, a SE project usually produces some additional documents, such as *test documentation*, *user manuals*, etc. These documents should similarly be considered in a KBS project.

In the SE world, this *document-driven* or *code-driven* approach to software development is recommended if the development process is well established. However, in the case of innovative projects, a *risk-driven* approach is more appropriate [2], to reduce the potential losses in case of an unsuccessful project, and also to emphasize the quality of all factors affecting the development process. The risk-driven approach requires identifying, at each stage of development, the major sources of risk that the project is facing and resolving these, before advancing to the less risk-prone parts of the development. The most significant risks may involve functional requirements (e.g., operations that the system must accomplish) as well as quality factors (e.g., reliability or safety aspects). For a risk-driven approach, more appropriate process models may be based on *evolutionary development* or *prototyping*.

Prototyping is used as a means to resolve various types of risks and specifically to evaluate the accuracy of the problem formulation [13]. This is the way prototypes are typically used within the knowledge acquisition phase of a KBS project. Prototypes may be the easiest way to find out how the application is supposed to interact with the environment. This is the approach often used in SE projects for specifying the user interface requirements: even a simple prototype may greatly help in determining what the user actually wants to do with the system. Prototypes are further useful for exploring the range of acceptable solutions [13]: a prototype is built to represent certain essential, possibly high-risk parts of the system, in order to minimize the potential costs involved if the plans turn out being infeasible or impractical [2]. Defining what the customer actually wants is one of the goals of prototyping, but the feasibility and robustness of the solution and the performance effects are at least as important. This way of using prototypes has usually not been suggested for KBS development. Efficient prototyping environments may produce graphical visualisations of the behavior of the application or support execution of partial descriptions [13].

The following comparison of KBS development tools and CASE tools is structured along the phase structure of a KBS project. Instead of choosing a particular representative from the wide and inconsistent market of CASE tools we use the idea of a generic integrated CASE environment consisting of tools and features that are typical to present advanced CASE environments. Typical examples of such environments are Foundation of Andersen Consulting, Software through Pictures of Interactive Development Environments, and Application Development Workbench of KnowledgeWare (for more information on integrated CASE environments, see, e.g., [16]).

In addition to phase specific tools, a CASE environment contains tools to handle planning and management functions of the project, such as estimating and assessing resources and schedules, and controlling changes. These tools are applicable throughout the life cycle and offer means for treating the requirements for the entire development process.

3 Requirements specification

The requirements specification of a KBS project describes the *organizational*, *functional*, *human factor*, and other requirements of the KBS under development.

The organizational requirements include constraints imposed by the environment where the application will be working, and general characterization of the objectives of the application. The functional and human factor requirements state the core of the requirements for the KBS. In addition to these, there are requirements for the entire development process (such as *verification and validation*, and *quality control plan*), and requirements for some of the process products (such as *usage plan* and *user manual*). Thus the principal contents of the requirements specification do not differ from those produced in a SE life cycle.

The functional requirements in KBS and SE projects are typically represented using natural language and data-flow diagrams. These are suitable for representing functional requirements informally, and are supported by diagramming tools and textual editors. However, they are not well suited for describing non-functional requirements, such as performance requirements, or time constraints and dependencies of the functions. When this type of requirements are essential some additional techniques or extensions of the previous techniques are available (e.g., software performance engineering [19], state diagrams, and real-time data flow diagrams [16].

If requirement specifications are presented using informal languages they tend to be both incomplete and ambiguous. For more precision, formal languages should be used. Rigorous use of a formal specification language also supports the verification and validation of the requirements. Several formal specification languages have been around for ten or twenty years, and there are automated or semi-automated tools available to support them (for a survey, see [21]). Even when informal languages are used in the beginning of the requirements specification phase, transportation from informal to formal representations at a later stage should be considered (see, e.g., [8]).

4 Knowledge acquisition and modelling

Acquiring and defining the knowledge of the application domain and the problem solving behavior is the task that most notably makes a difference between a KBS project and a SE project. In a SE project, related activities are conducted as part of the requirements analysis phase, while in a KBS project these tasks are performed in separate knowledge acquisition and conceptual modelling phases. Most of the tools specific for KBS development are knowledge acquisition tools.

Knowledge acquisition can be supported by *generalised directive models* (GDM) [18]. GDMs are general models describing problem-solving strategies. Since GDMs are domain independent, a library of GDMs is a useful way of reusing high level models across application domains. During KBS development, an appropriate GDM is chosen (e.g., from the GDM library) and then refined to suit the domain knowledge of the application. A similar idea of high level reuse could be applied in all phases of KBS development. The entire development process could be built on reusing models of development and modifying them to suit the needs of the particular application domain [9]. A reuse-driven approach to software development results in higher reliability and productivity. Efficient reuse requires tools for arranging, searching and manipulating reusable elements.

Knowledge acquisition methods are either heuristic classification methods or constructive problem solving methods using synthesis. Typical examples of KBS development tools for knowledge acquisition are laddering grid, card sort, repertory grid, machine learning tools, and protocol editors [3]. As similar support is not available in CASE development environments, exporting knowledge acquired by KBS tools might prove to be useful to all software development [4].

The results of the knowledge acquisition phase are presented as a conceptual model which is a model of the expertise required to perform the task. Such a conceptual model can be viewed as consisting of three components: *domain data*, *domain knowledge*, and *problem solving behavior* [18]. The first two components are closely related to the data modelling concepts that are used within the information engineering approach to software development, e.g., entity-relationship modelling. The tools that are used for representing ER-models and other data-driven designs in CASE environments are suitable for treating these components.

The third component, the problem solving behavior, is specific to the KBS development process, and thus requires more discussion. It describes the tasks that are carried out by the KBS. Although a domain independent GDM can well be the starting point for modelling the problem solving behavior, it is not clear whether the refined model can actually be in any way independent of the requirements imposed by the application domain. In some cases, it may be difficult to separate description of the tasks (included in the conceptual model) from the requirements and constraints imposed on them (treated in the requirements specification). Thus conceptual modelling and requirements specification need to be performed in cooperation. Tools suitable for representing problem solving behavior may be data flow diagrams or other process-driven design tools of CASE environments.

5 Design and implementation

In the design phase of a KBS project, the *functional components* of the system and its *interface* to the user and the environment are designed. The technical details of the components are further elaborated, including the implementation-dependent features. Similar tasks are performed during the design phase of a SE project.

The basis of this design lies in the conceptual model. In addition, the requirements document may contain additional information describing the users' expectations of the interface they will eventually be using. Similarly, if the application must interact with some existing systems (e.g., other software systems) the interface of the systems is conceptually comparable to the requirements imposed by the hardware interface. Therefore, to some extent, this interface should be considered in the requirements specification to have the necessary influence on the design phase of the KBS project.

Similarly, designing the implementation-dependent features of the system requires adjusting to the hardware and the non-functional requirements defined for the system, such as its performance, safety, etc. If they are critical for the acceptability of the application they must be considered early in the life cycle since they cannot be added into the system late in the design phase. A risk-driven approach is appropriate for deciding the right phase to consider these features.

The implementation phase of the KBS life cycle produces executable code using the design models as input. This is very similar to the implementation in a SE project. If the design models are presented in a formal language, this production can be based on a (possibly) user-assisted transformation from the design language to the implementation language.

Typical CASE tools for design and implementation support formal and informal presentation of the design, and code generation. The market of CASE tools for design and implementation is wide and contains tools reaching from simple editors, compilers and linkers to high-level design and implementation toolsets. The most advanced tools also include facilities supporting the *testing* of the system, i.e., automatic test case generators or testing environments to automate test runs and verification.

6 Information transfer

When developing an application containing a KBS as well as traditional software it is useful to take advantage of the tools already available. As we have seen, the KBS and SE life cycles have a lot in common. The most notable difference in the life cycle is in the knowledge acquisition and conceptual modelling phases where advanced tools have been developed for KBS development. On the other hand, the design and implementation phases are extensively covered by the CASE environments. We contend that simultaneous use of KBS and SE environments can greatly help in KBS development. To make such a simultaneous use possible, one has to be able to transfer information from one environment to the other. The conceptual modelling phase is the most essential point where information transfer between KBS and CASE tools is needed (Figure 2). The domain knowledge acquired and modelled using KBS tools can be used as a basis for further development using CASE tools [10].

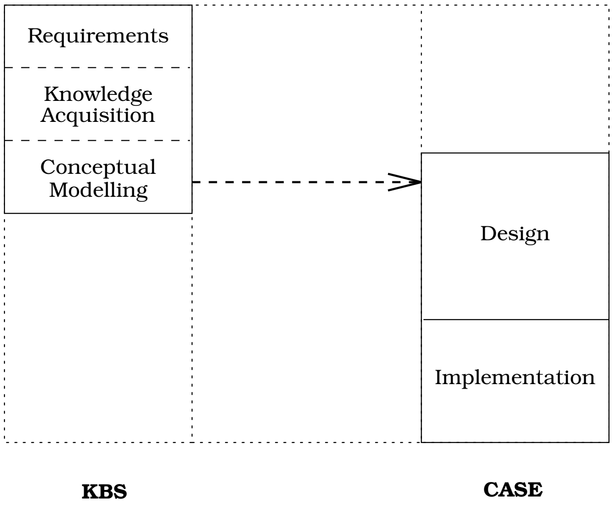

Figure 2: KBS development using KBS and CASE tools.

There are some other situations where the two environments need to cooperate. If the KBS is only a small part of the application it may be reasonable to use both environments in parallel, KBS tools for the KBS part and CASE tools for the non-KBS part, resulting in a (limited) need of information transfer in both directions. Furthermore, if software development is to be based on massive reuse of software components this requires information transfer from CASE tools to KBS tools, as most of the present software has been developed in CASE environments.

The practical solution to the information transfer problem depends on the way information is stored in each of the environments. Integrated CASE environments typically rely on a common *repository* containing both technical and managerial information. The repository is a collection of data and the necessary interface routines for manipulating this data. The data may contain, e.g., diagrams, software components, project task plans, and text. The data is stored using one or few standard formats, and the variety of possible representations is achieved through the interface routines. A natural platform for the repository is a database management system. Since the elements often have a complex stucture involving both textual and graphical information and rules for consistency and validation, an object-oriented database system seems well suited for the implementation of repositories [11].

The repository is used both within the development of one application (one project) and as the effective "memory" of the entire enterprise. The repository contains the current versions of the process products, the accumulated information on the application, and the technical and managerial information collected during several projects (ideally, all the projects of the enterprise). As such, the repository is a common basis for information sharing between the developers and efficient reuse of software components of all kind. The interface routines enable efficient transportation of knowledge from one representation to another.

Several standards for CASE repositories have been suggested. These include *Portable Common Tools Environment (PCTE)* adopted by ECMA [7], *Information Resource Dictionary System (IRDS)* approved by ISO and ANSI [12], as well as several industry standards, e.g., the repository within AD/Cycle of IBM [5], and CDD/Repository of DEC [6].

Transferring information between KBS tools and CASE tools typically requires transformation from one representation to another. If the CASE environment uses a repository, all the representations available for the CASE tools are stored in the repository and are manipulated by the repository tools (Figure 3a). Thus the transformation from KBS representation to CASE representation or vice versa requires building an interface between the respective KBS tool and the repository.

If the CASE tools do not use a common repository, a transformation must occur between each pair of tools (Figure 3b). This makes the situation much more complicated. As CASE repository standards become more generally accepted, it is advisable for tool builders to conform to these standards, in order to facilitate the cooperation between their tool and others. For similar reasons, such standards should be used and encouraged within KBS tool development. This may be even more important, since conformance to CASE repository standards opens the choice for a large number of CASE tools that can be efficiently utilized in KBS development. Thus, in the most advanced situation, both the KBS tools and the CASE environment use a similar repository standard, making the information transfer straightforward (Figure 3c).

7 Example of information transfer

Our example illustrates information transfer from a KBS conceptual modelling tool to a CASE design tool. Let us assume that we have produced a conceptual model of an application using a KBS conceptual

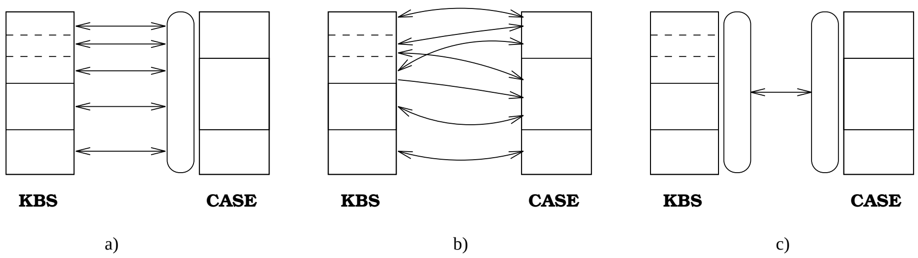

Figure 3: Information transfer between workbenches, with or without repositories.

modelling tool and that we want to transfer its information contents to a CASE design tool for further application development. The KBS tool that we used is the OCML (Operationalizable Conceptual Modelling Language) tool of VITAL [15]. The tool includes capabilities for presenting coarse level (GDM) models as well as more refined models that include domain data and knowledge as well as problem solving behavior. The CASE tool chosen is Foundation Design of Andersen Consulting [1]. However, the characteristics of these particular tools are fairly similar to any of the advanced tools that are used for conceptual modelling or design in the KBS or CASE life cycle.

The problem that we focus on in our example is the Sisyphus II elevator design problem introduced in [14, 22]. The Sisyphus II problem involves a KBS for configuring an elevator system and includes customer demands and constraints. It is solved using a propose-and-revise strategy: an initial design model is produced and repeatedly modified until all constraints are satisfied. Some of the constraints (such as safety regulations) are strict, while some can be fixed; some fixes may be more desirable than others. For more detailed presentation of the Sisyphus II problem, see [14, 22].

The conceptual model presented in Figure 4 is a reprint of the model presented in [15]. This model is the result of several refinement steps up to the point where each task (ellipse) of the model is an inference that cannot be refined without using domain specific information. It is therefore the final GDM of the solution [15]. The model in Figure 4 contains only tasks (or processes) and data collections and does not involve control or timing. Such a conceptual model can be seen as a standard data flow diagram, and hence it is straightforward to transform it into a corresponding representation within a CASE environment with data flow diagram capabilities. This is an example of a simple transformation between the environments: each object in the source model has a distinct correspondent in the resulting model.

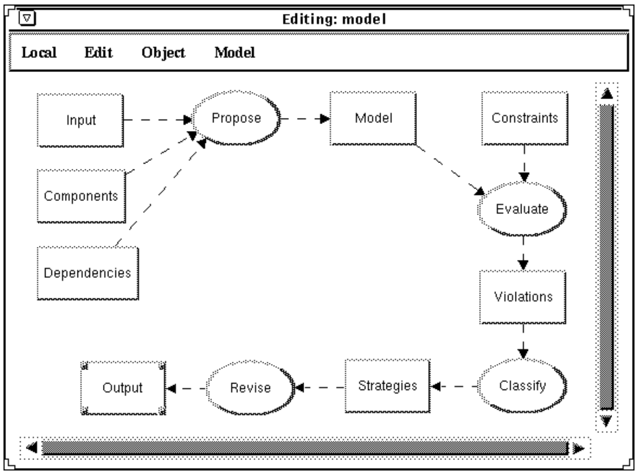

Figure 4: GDM of the solution for the Sisyphus II elevator problem [15].

The model in Figure 4 contains embedded iteration, as the steps *Propose*, *Evaluate*, *Classify*, and *Revise* are repeated until a satisfactory result is found or the process fails. The model is refined by introducing control to describe this iteration [15]. Each subtask is then further refined until an appropriate level of detail is achieved. The same tool can be used for refinement.

Figure 5 presents an early refinement step of subtask *Revise* [15]. In order to revise the model, there must exist some constraint violations in the elevator model that are fixable (by changing some parameter values). The model in Figure 5 presents a refinement where the fixable constraint violations (FCV) are collected in a pool to determine whether the process fails or continues.

The conceptual model in Figure 5 can no longer be seen solely as a data flow diagram: although it contains information on data used by the subtasks, it

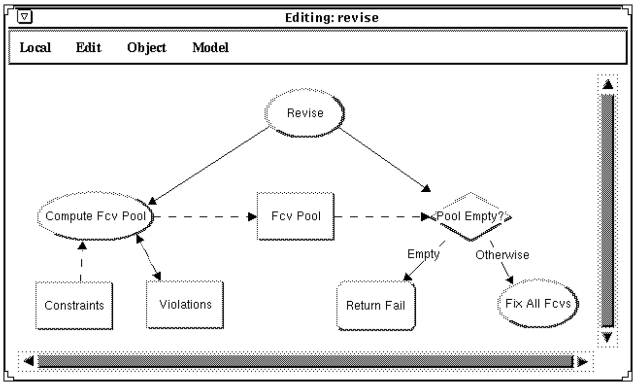

Figure 5: Refinement for subtask *Revise* [15].

also describes the procedural structure of *Revise*. For this purpose, it contains constructs describing sequentiality (*Compute FCV Pool* must be performed before *Pool Empty?*) and choice (*Pool Empty?* results in performing exactly one of the two tasks *Return Fail* and *Fix all FCVs*).

Transferring the information contents of this model into a CASE environment is not straightforward. Even though the CASE environment includes tools for both data flow diagramming and procedural structure diagramming, it may not be possible to transfer all the information in a way that it can be handled by both types of tools. The import facilities of CASE tools are not necessarily planned for this kind of information transfer. Figures 6 and 7 are both models that can be produced using Foundation Design. Figure 6 presents the data flow diagram corresponding to the conceptual model of subtask *Revise*. This diagram lacks the control features embedded in the conceptual model. Similarly, the procedure diagram presented in Figure 7 contains the control structure and the sequentiality but lacks the data flow aspects of the conceptual model.

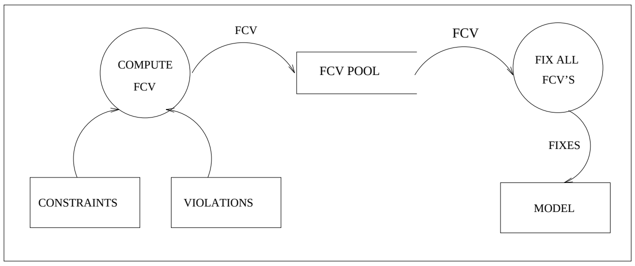

Figure 6: Data flow diagram of subtask *Revise*.

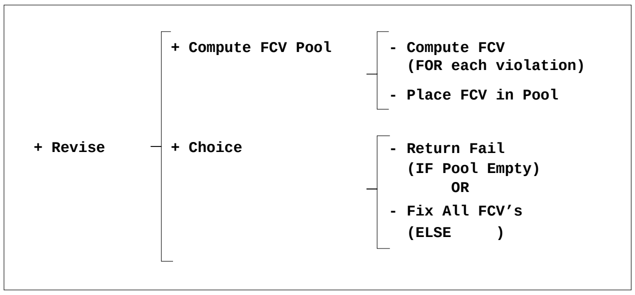

Figure 7: Procedural structure of subtask *Revise*.

Any features included in the conceptual model that cannot be presented using the design tool can, of course, be added as comments for further use. This would be particularly useful in case we plan to do a transformation back to the conceptual model at some later time. In a CASE environment with a common repository (such as Foundation Design) it is possible to handle the objects with any of the tools in the environment. However, it may still not be possible to import objects having multiple bindings in another environment and retain these bindings functional in the new environment. Only if both environments use a similar repository, the transfer is guaranteed to be easy and retain all information.

8 Concluding remarks

The developers of KBSs face many problems that are fairly similar to the ones already treated in traditional SE. If AI is used in an application, more profound phases of knowledge acquisition and conceptual modelling are needed than would be necessary in a traditional software project. Nevertheless, the life cycles of KBS and SE projects are similar enough to employ almost identical process models, and even to use KBS tools and CASE tools during different phases of the same project. The experience that has been gained in developing applications with traditional SE techniques should be employed in KBS development. Similarly, the tools developed for CASE environments are available for KBS development, thus permitting KBS developers to concentrate on tools specific to KBS problems. On the other hand, the KBS tools for knowledge acquisition and conceptual modelling could be useful even in traditional software projects.

The most important interface where information transfer between the environments would be beneficial is from the conceptual modelling tools of KBS environments to the design tools of CASE environments. In addition to this, if previously developed components are reused it is necessary to transfer a limited amount of information in both directions. Information transfer requires a transformation between the participating representations. Currently we are investigating the characteristics of this transformation. The effort required is greatly reduced if both environments store the information using a standard repository. This should be strongly emphasized when developing further tools for KBS development.

References

[1] Andersen Consulting. *FOUNDATION Design, Analyze Application Requirements, Version 2.0*, 1993.

[2] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61 – 72, May 1988.

[3] John H. Boose. A survey of knowledge acquisition techniques and tools. *Knowledge Acquisition*, 1(1):3 – 37, March 1989.

[4] Alexander Borgida, Sol Greenspan, and John Mylopoulos. Knowledge representation as the basis for requirements specification. *Computer*, 18(4):82 – 91, April 1985.

[5] Debra Cameron. IBM's AD/Cycle and Repository. Technical report, Computer Technology Research, 1990.

[6] Digital Equipment Corporation. *Digital's Distributed Repository*, 1991.

[7] ECMA European Computer Manufacturers Association. *Portable Common Tool Environment (PCTE), Abstract Specification*, December 1990.

[8] Martin D. Fraser, Kuldeep Kumar, and Vijayk Vaishnavi. Informal and formal requirements specification languages: Binding the gap. *IEEE Transactions on Software Engineering*, 17(5):454 – 466, May 1991.

[9] Allen Goldberg. Reusing software developments. In *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 107 – 119, December 1990.

[10] Sol Greenspan. On the role of domain knowledge in knowledge-based approaches to software development. In *Proceedings of International Workshop on the Software Process and Software Environments*, pages 61 – 65, March 1985.

[11] R. Gupta and E Horowitz, editors. *Object-Oriented Databases with Applications to CASE, Networks and VLSI CAD*. Prentice-Hall, 1991.

[12] ISO International Organization for Standardization. *Information Resource Dictionary System (IRDS) Framework*, 1990.

[13] Luqi and Winston Royce. Status report: Computer-aided prototyping. *IEEE Software*, 9(6):77 – 81, November 1992.

[14] Sandra Marcus, Jeffrey Stout, and John McDermott. VT: An expert elevator designer that uses knowledge-based backtracking. *AI Magazine*, 9(1):95 – 111, March 1987.

[15] Enrico Motta, Kieron O'Hara, Nigel Shadbolt, Arthur Stutt, and Zdenek Zdrahal. A VITAL solution to the Sisyphus II elevator design problem. Technical report, Esprit Project P5365 Vital, 1993.

[16] Roger S. Pressman. *Software Engineering: a Practitioner's Approach*. McGraw-Hill, 1992.

[17] W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of WESCON*, August 1970.

[18] Nigel Shadbolt, Enrico Motta, and Alain Rouge. Constructing knowledge-based systems. *IEEE Software*, 10(6):34 – 39, November 1993.

[19] Connie U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, 1990.

[20] B. J. Wielinga, A. Th. Schreiber, and J. A. Breuker. KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1):5 – 53, 1992.

[21] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8 – 24, September 1990.

[22] Gregg R. Yost. Configuring elevator systems. Draft, AI Research Group, Digital Equipment Corporation, December 1992.

[23] Edward Yourdon. *Decline and Fall of the American Programmer*. Prentice-Hall, 1992.