Zlín 2016

Tomas Bata University in Zlín
Faculty of Applied Informatics

MODERN THEORY OF INFORMATION - STRUCTURED AND KNOWLEDGE BASED SYSTEM DESIGN

Said Krayem

Tomas Bata University in Zlín
Faculty of Applied Informatics

MODERN THEORY OF INFORMATION - STRUCTURED AND KNOWLEDGE BASED SYSTEM DESIGN

SAID KRAYEM

ZLÍN 2016

ABSTRACT

This keynote will address the challenges in the digital engineering field with a specific focus on analyzing security and harnessing the gaps that exist and the progress that needs to be made in the coming decade.

The state–based Event-B formal method relies on the concept of correct step-by-step development ensured by discharging corresponding proof obligations.

The CommonKADS methodology is a collection of structured methods for building knowledge-based systems. A key component of CommonKADS is the library of generic inference models which can by applied to tasks of specified types. These generic models can either be used as frameworks for knowledge acquisition or to verify the completness of models developed by an analysis of the domain.

Key words: System, Knowledge Representation, Expert System Design, Knowledge-based Systems, Knowledge Engineering, Formal Methods, events, action, machine and context

ACKNOWLEDGEMENT

The most important person I would like to thank is my room-mate Susan. Having Susan as a room-mate gives you the freedom to pursue your individual interest. Furthermore, she gives you the feeling that you can do something useful.

CONTENTS

INTRODUCTION

Vital, a four-and-a-half-year ESPRIT II research and development project that involves nine organizations in five countries is discussed herein. It addresses the problems of effective process modeling for knowledge-based systems, providing guidelines on when to use various knowledge-engineering methods and techniques - and, reducing the bottleneck in acquiring expert knowledge by providing both methodological and software support for developing large, industrial, knowledge-based system applications. The project goals, approach, and workbench are outlined, and a case-study is described. Our aim is to provide a survey for the design of knowledge-base systems using knowledge aquisition analysis and design (KADS-Europian Standard) and the formal Event-B. Event-B method is a mathematical approach for developing a formal method of systems (Abrial and Hallerstede (2006)). An Event-B model is constructed from a collection of modelling elements. These elements include invariants, events, guards and actions. The modeling elements have attributes that can be based on set theory and predicate logic. Set theory is used to represent data types and the manipulation of data. Logic is used to apply conditions to the data. The development of an Event-B model goes through two stages - abstraction and refinement. The abstract machine specifies the initial requirements of the system. Refinement is carried out in several steps, with each step adding more detail to the system - generally, but not exclusively, in a top-down manner.

1 System Definition [1,3]

System: An integrated set of interoperable elements, each with explicitly specified and bounded capabilities, working synergistically to perform valuable processing to enable a User to satisfy mission-oriented operational needs in a prescribed operating environment with a specified outcome and probability of success. As we know, a **mathematical model** is a description of a system using mathematical concepts and languages.

Let us examine each part in detail:

- By “an integrated set,” we mean that a system - by definition, is composed of hierarchical levels of physical elements, entities, or components.

- By “interoperable elements,” we mean that elements within the system’s structure must be compatible with each other in form, fit, and function - for example. System elements include equipment (e.g., hardware and system, system, facilities, operating constraints, support), maintenance, supplies, spares, training, resources, procedural data, external systems, and anything else that supports mission accomplishment.

- By each element having “explicitly specified and bounded capabilities”, we mean that every element should work to accomplish some higher-level goal or purposeful mission. System element contributions to the overall system performance must be explicitly specified. This requires that operational and functional performance capabilities for each system element be identified and explicitly bounded to a level of specificity that allows the element to be analyzed, designed, developed, tested, verified, and validated — either on a stand-alone basis or as part of the integrated system.

- By “working in synergistically”, we mean that the purpose of integrating the set of elements is to leverage the capabilities of individual element capabilities to accomplish a higher level capability that cannot be achieved by stand-alone elements.

- By “value-added processing”, we mean that factors such operational cost, utility, suitability, availability, and efficiency demand that each system operation and task add

value to its inputs availability, and produce outputs that contribute to the achievement of the overall system mission outcome and performance objectives.

- By “enable a user to predictably satisfy mission-oriented operational needs”, we mean that every system has a purpose (i.e., a reason for existence) and a value to the user(s). Its value may be a return on investment (ROI) relative to satisfying operational needs; or to satisfy system missions and objectives.

- By “in a prescribed operating environment”, we mean that for economic, outcome, and survival reasons, every system must have a prescribed — that is, bounded — operating environment.

- By “with a specified outcome”, we mean that system stakeholders (Users, shareholders, owners, etc.) expect systems to produce results. The observed behavior, products, by-products, or services for example, must be outcome-oriented, quantifiable, measurable, and verifiable.

- By “and probability of success”, we mean that accomplishment of a specific outcome involves a degree of uncertainty or risk. Thus, the degree of success is determined by various performance factors, e.g. reliability, dependability, availability, maintainability, sustainability, lethality, and survivability.

We need at least four types of agreement on working level definitions of a system:

1. A personal understanding
2. A program team consensus
3. An organizational (e.g., System Developer) consensus; and…
4. Most importantly, a contractual consensus with one´s customer

Why? Of particular importance is that you, your program team, and your customer (i.e., a User or an Acquirer as the User’s technical representative) have a mutually clear and concise understanding of the term. Organizationally, you need a consensus of agreement among the System Developer team members. The intent is to establish continuity across contracts and organizations as personnel transit between programs.

National and international standards organizations - as well as different authors, have their own definitions of a system. If one analyzes these, one will find a diversity of viewpoints - all tempered by their personal knowledge and experiences. Moreover, achievement of a "one-size-fits-all" convergence and consensus by standards organizations often results in wording that is so diluted that many believe it to be insufficient and inadequate. Examples of organizations which have standard definitions include:

- The International Council on Systems Engineering (INCOSE)
- The Institute of Electrical and Electronic Engineers (IEEE)
- The American National Standards Institute (ANSI)/Electronic Industries Alliance (EIA)
- The International Standards Organization (ISO)
- The US Department of Defense (DoD)
- The US National Aeronautics and Space Administration (NASA)
- The US Federal Aviation Administration (FAA)

You are encouraged to broaden your knowledge and explore definitions by these organizations. You should then select one that best fits your business application. Depending on your personal viewpoints and needs, the definition stated in this text should prove to be the most descriptive characterization.

When people develop definitions, they attempt to create content and grammar simultaneously. People typically spend a disproportionate amount of time on grammar and spend very little time on substantive content. We see this in specifications and plans, for example. Grammar is important - since it is the root of our language and communications. However, grammar has no value if it lacks substantive content.

You will be surprised how animated and energized people become over wording exercises. Subsequently, they throw up their hands and walk away. For highly diverse terms such as a system, a good definition may sometimes be simply a bulleted list of descriptors concerning what a term is - or, perhaps, is not. So, if you or your team attempts to create your own definition, perform one consensus step at a time.

Obtain consensus on the key elements of the substantive content. Then, structure the statement in a logical sequence and translate the structure into grammar.

As an abstraction, we symbolically represent a system as a simple entity by using a rectangular box - as shown in Figure 1-1. In general, inputs such as stimuli and cues are fed into a system that processes the inputs and produces an output. As a construct, this symbolism is acceptable; however, the words need to more explicitly identify WHAT the system performs. That is, the system must add value to the input in producing an output.

We refer to the transformational processing that adds value to inputs and produces an output as a "capability". You will often hear people refer to this as the system's "functionality"; this is partially correct. Functionality only represents the ACTION to be accomplished; not HOW WELL - as is characterized by performance. This text employs capability as the operative term that encompasses both the functionality and performance attributes of a system.

The simple diagram presented in Figure 1-1 represents a system. However, from an analytical perspective, the diagram is missing critical information that relates to how the system operates and performs within its operating environment. Therefore, we have expanded the diagram to identify these missing elements. The result is shown in Figure1-2. The attributes of the construct - which include desirable/undesirable inputs, stakeholders, and desirable/undesirable outputs - serve as a key checklist to ensure that all contributory factors are duly considered when specifying, designing, and developing a system [3].

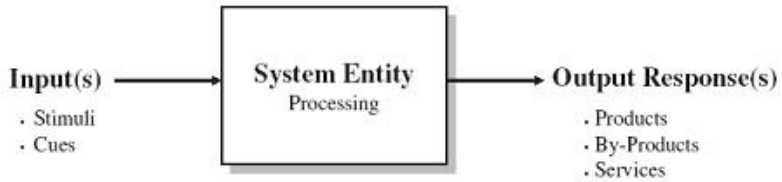


Figure 1-1: Basic System Entity Construct

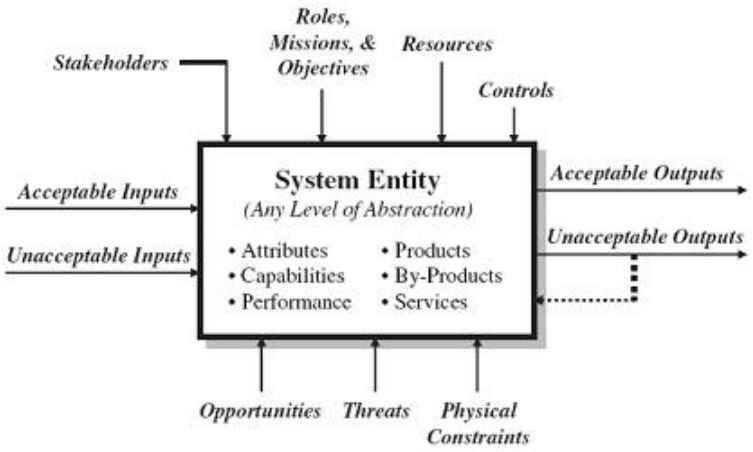


Figure 1-2: General Construct

Examples of various types of systems are workflow-based systems that produce systems, products, or services such as schools, hospitals, banking systems, and manufacturers. As such, they require insightful, efficient, and effective organizational structures, supporting assets, and collaborative interactions.

Some systems require the analysis, design, and development of specialized structures, complex interactions, and performance monitoring that may have an impact on the safety, health, and well-being of the public as well as the environment, thus the engineering of these systems may be required. As you investigate WHAT is required to analyze, design, and develop both types of systems, you will find that they both share a common set of concepts, principles, and practices. Business systems, for example, may require the application of various analytical and mathematical principles to develop business models and performance models to determine profitability and Return On Investment (ROI) and statistical theory for optimal waiting lines or weather conditions, for example. In the case of highly complex systems - analytical, mathematical, and scientific principles may have to be applied. We refer to this as the engineering of systems, which may require a mixture of engineering disciplines such as system engineering, electrical engineering, mechanical engineering, and software engineering. These disciplines may only be required at various stages during the analysis, design, and development of a system, product, or service.

This text provides the concepts, principles, and practices that apply to the analysis, design, and development of both types of systems. On the surface, these two categories imply a clear distinction between those that require engineering - and those that do not. So, how do you know when the engineering of systems is required?

Actually, these two categories represent a continuum of systems, products, or services that range from making a piece of paper - which can be complex, to developing a system as complex as an aircraft carrier or NASA ’s International Space Station (ISS). Perhaps the best way is to address the question: “What is system engineering?”

1.1 System Engineering[3,4]

Explicitly, System Engineering (SE) is the multi-disciplinary engineering of systems. However - as with any definition, the response should eliminate the need for additional clarifying questions. Instead, the engineering of a system response evokes two additional

questions: What is engineering? What is a system? Pursuing this line of thought, let's explore these questions further.

I. Defining Key Terms

Engineering students often graduate without being introduced to the root term that provides the basis for their formal education. The term, engineering originates from the Latin word ingenerare, which means "to create." Today, the Accreditation Board for Engineering and Technology (ABET), which accredits engineering schools in the United States, defines the term as follows:

- Engineering "The profession in which knowledge of the mathematical and natural sciences gained by study, experience, and practice is applied with judgment to develop ways to utilize economically the materials and forces of nature for the benefit of mankind." (Source: Accreditation Board for Engineering and Technology [ABET]).
- There are a number of ways to define System Engineering (SE), each dependent on an individual's or organization's perspectives, experiences, and the like. System engineering means different things to different people.
- You will discover that even your own views of System Engineering (SE) will evolve over time. So, if you have a diversity of perspectives and definitions, what should you do? What is important is that you, program teams, or your organization:

    1. Establish a consensus definition
    2. Document the definition in organizational or program command media to serve as a guide for all

For those who prefer a brief, high-level definition that encompasses the key aspects of System Engineering (SE), consider the following definition:

- System Engineering (SE) - The multidisciplinary application of analytical, mathematical, and scientific principles to formulating, selecting, and developing a solution that has acceptable risk, satisfies user operational need(s), and minimizes development and life-cycle costs while balancing stakeholder interests.

This definition can be summarized in a key System Engineering (SE) principle:

II. System Engineering BEGINS and ENDS with the User

System Engineering (SE) - as we will see, is one of those terms that requires more than simply defining WHAT System Engineering (SE) does; the definition must also identify WHO/WHAT benefits from System Engineering (SE). The ABET definition of engineering, for example, includes the central objective "to utilize - economically, the materials and forces of nature for the benefit of mankind."

Applying this same context to the definition of System Engineering (SE), the User of systems, products, and services symbolizes humankind. However, mankind's survival is very dependent on a living environment that supports sustainment of the specifics. Therefore, System Engineering (SE) must have a broader perspective than simply "for the benefit of mankind." System Engineering (SE) must also ensure a balance between humankind and the living environment without sacrificing either.

A big system may be seen as a set of interacting smaller systems - known as sub-systems, or functional units, each of which has its defined tasks. All these work in coordination to achieve the overall objective of the system. System Engineering requires the development of a strong foundation in understanding how to characterize a system, product, or service in terms of its attributes, properties, and performance.

As discussed above, a system is a set of components that work together to achieve some goal. The basic elements of the system may be listed as 1-1-1 Resources.

1.1.1 Resources

Every system requires certain resources for the system to exist. Resources can be hardware, software or liveware. Hardware resources may include a computer - its peripherals, stationery, etc. Software resources would include the programs running on these computers and the liveware would include the human-beings required to operate the system and make it functional.

Thus, these resources make up an important component of any system. For instance, a Banking system cannot function without the required stationery - like cheque-books, pass books, etc. Such systems also need computers to maintain their data and trained staff to operate these computers and cater to customer requirements.

1.1.2 Procedures

Every system functions under a set of rules that govern the system in order to accomplish the defined goal of the system. This set of rules defines the procedures for the system to operate with. For instance, Banking systems have their predefined rules for providing interest at different rates for different types of accounts.

1.1.3 Data/Information

Every system has some predefined goal. To achieve the goal, the system requires certain inputs - which are converted into the required output. The main objective of a System is to produce some useful output. Output is the outcome of processing. Output can be of any nature - e.g. goods, services or information.

1.1.4 Intermediate Data

Various processes process a system's Inputs. Before it is transformed into Output, it goes through many intermediary transformations. Therefore, it is very important to identify the Intermediate Data. For example, in a college - when students register for a new semester, the initial form submitted by a student goes through many departments. Each department adds their validity checks to it.

Finally, the form gets transformed - and the student gets a slip that states whether the student has been registered for the requested subjects or not - this helps in building the System in a better way. Intermediate forms of data occur when there is a lot of processing on the input data. So, intermediate data should be handled as carefully as other data, since the output depends upon it.

1.1.5 Processes

Systems have some processes that make use of resources to achieve the set goal under the defined procedures. These processes are the operational element of the system.

For instance, in a Banking system, there are several processes that are carried out. Consider, for example, the processing of a cheque as a process. A cheque passes through several stages before it actually gets processed and converted. These are some of the processes of the

Banking system. All these components taken together make a complete functional system. Systems also exhibit certain features and characteristics.

1.1.6 Objective

Every system has a predefined goal or objective towards which it works. A system cannot exist without a defined objective. For example, an organization has the objective of earning maximum possible revenues - for which, each department and each individual, has to work in coordination.

1.1.7 Standards

This is the acceptable level of performance for any system. Systems should be designed to meet standards. Standards can be business-specific or organization-specific.

For example, take a sorting problem. There are various sorting algorithms; but each has its own complexity. So, such algorithm should be used that gives the most optimum efficiency. So there should be a standard or rule for using a particular algorithm. It should be seen whether that algorithm is implemented in the system.

1.1.8 Environment

Every system - whether it is natural or man-made, co-exists with an environment. It is very important for a system to adapt itself to its environment. Also - for a system to exist, it should change according to the changing environment. For example, we humans live in a particular environment. As we move to other places, there are changes in the surroundings - but our bodies gradually adapt to the new environment. If it were not the case, then it would have been very difficult for humans to survive for so many thousands of years.

1.1.9 Feed-back

Feed-back is an important element of all systems. The output of a system needs to be observed and the feedback from the output used so as to improve the system and make it achieve the set standards. Figure 1-2 shows how a system takes in input. It then transforms it into output. Also, some feedback can come from customers (regarding quality) or it can be some intermediate data (the output of one process and the input for another) that is required to produce the final output.

1.1.10 Boundaries and Interfaces

Every system has defined boundaries within which it operates. Beyond these limits, the system has to interact with other systems. For instance, the Personnel system in an organization has its work-domain - with defined procedures. If the financial details of an employee are required, the system has to interact with the Accounting system to get the required details.

Interfaces are another important element through which a system interacts with the outside world. A system interacts with other systems through its interfaces. Users of the systems also interact with it through interfaces. Therefore, these should be customized to users´ needs. These should be as user-friendly as possible.

So, we now have a firm knowledge of various system components and characteristics. There are various types of system. To have a good understanding of these systems, these can be categorized in many ways. Some of the categories are open or closed, physical or abstract and natural or man-made information systems, which are explained below.

Classification of systems can be done in many ways.

1.2 Physical and Abstract Systems

Physical systems are tangible entities that we can feel and touch. These may be static or dynamic in nature. For example, take a computer center. The desks and chairs are the static elements that assist in the daily working of the center. Static parts don't change. Dynamic systems are constantly changing. Computer systems are dynamic systems. Programs, data, and applications can change according to user needs.

Abstract systems are conceptual. These are not physical entities. They may be formulas, representations or models of a real system.

1.3 Open / Closed Systems

Systems interact with their environment to achieve their targets. Things that are not part of the system are environmental elements for the system. Depending upon the interaction with the environment, systems can be divided into two categories - open and closed.

Open Systems: Systems that interact with their environment. Practically speaking, most systems are open systems. An open system has many interfaces with its environment. It can also adapt to changing environmental conditions. It can receive inputs from, and deliver output to, the outside of a system. An information system is an example of this category.

Closed Systems: Systems that do not interact with their environment. Closed systems exist in concept only.

2 Information Systems[6]

The main purpose of information systems is to manage data for a particular organization. Maintaining files, producing information and reports are just a few of its functions. An information system produces customized information - depending upon the needs of the organization. These are usually formal, informal, and computer-based.

Formal Information Systems: These deal with the flow of information from top management to lower management. Information flows in the form of memos, instructions, etc. - but feedback can also be given by lower authorities to top management.

Informal Information Systems: Informal systems are employee-based. These are made to solve day-to-day work-related problems.

Computer-Based Information Systems: This class of systems depends on the use of computers for managing business applications. These systems are discussed in detail in the next section.

In the previous section, we studied about various classifications of systems. Since - in business, we mainly deal with information systems, we shall further explore these systems and talk about different types of information systems that are prevalent in industry.

An information system deals with an organization´s data. The purposes of an information system are to process input, maintain data, produce reports, handle queries, handle on-line transactions, generate reports, and other outputs. These require one to maintain huge databases, handle hundreds of queries etc. The transformation of data into information is the primary function of any information system.

These types of systems depend upon computers to deliver their objectives. A computer-based business system involves six interdependent elements. These are: hardware (machines), software, people (programmers, managers or users), procedures, data, and information (processed data). All six elements interact to convert data into information. System Analysis relies heavily on computers to solve problems. For these types of systems, an analyst should have a sound understanding of computer technologies.

In the following section, we explore three of the most important information systems - namely, transaction processing systems, management information systems and decision support systems, and examine how computers assist in maintaining Information systems.

Information systems differ in their business needs. Also - depending upon different levels in the organization, the information systems also differ. The three major information systems are: Transaction Processing Systems, Management Information Sstems, and Decision Support Systems.

Figure 2.1 shows the relation of an information system to the levels of an organization. The information needs are different at different organizational levels. Accordingly, the information can be categorized as: Strategic Information, Managerial Information and Operational Information.

Strategic Information is the information needed by the top-most management level for decision-making. For example, the trends in revenues earned by the organization are required by the top management for setting the policies of the organization. This information is not required by the lower levels in the organization. The information systems that provide these kinds of information are known as Decision Support Systems.

Figure 2-1: Relation of Information Systems to Levels of an Organization

The second category of information required by the middle management level is known as Managerial Information. The information required at this level is used to make short-term decisions and plans for the organization. Information like a Sales Analysis for the past quarter, or yearly production details etc., fall under this category. The Management Information System (MIS) caters to such information needs of the organization. Due to its abilities to fulfill the managerial information needs of organizations, Management Information Systems have become a necessity for all big organizations. And due to their vastness, most big organizations have separate MIS departments to look into the related issues and proper functioning of the system.

The third category of information is related to the daily or short-term information needs of an organization - such as attendance employee records, for instaance. This kind of information is required at the operational level to carry out day-to-day operational activities. Due to its ability to provide information for processing the transaction of the organization, the information system is known as Transaction Processing System - or Data Processing System. Some examples of information provided by such systems are: processing orders, posting entries in bank, evaluating overdue purchaser orders, etc.

2.1 Transaction Processing Systems

TPS process the business transactions of the organization. A transaction can be any activity of the organization. Transactions differ from organization to organization. For example, take a railway reservation system. Booking, canceling, etc., are all transactions. Any query made to it is a transaction. However, there are some transaction -, which are common to almost all organizations. Like employees, new employees, maintaining their leave status, maintaining employee accounts, etc.

This provides high-speed and accurate processing of record-keeping of basic operational processes. These include calculation, storage and retrieval.

Transaction Processing Systems provide speed and accuracy, and can be programmed to follow the routine functions of the organization.

2.2 Management Information Systems

These systems assist lower management in problem-solving and making decisions. They use the results of transaction processing as well as some other information. They are a set of information processing functions. They should handle queries as quickly as they arrive. An important element of any MIS is its database.

A database is a non-redundant collection of interrelated data items that can be processed through application programs and is available to many users.

2.3 Decision Support Systems

These systems assist higher management to make long-term decisions. These type of systems handle unstructured or semi-structured decisions. A decision is considered unstructured if there are no clear procedures for making the decision, and if not all the factors to be considered in the decision can be readily identified in advance.

These are not of a recurring nature. Some recur infrequently or occur only once. A Decision Support System must very flexible. The user should be able to produce customized reports by giving particular data and format specific to particular situations.

2.3.1 Decision Tree[8,9]

The analysis of complex decisions with significant uncertainty can be confusing because:

1) The consequence that will result from selecting any specified decision alternative cannot be predicted with certainty
2) There is often a large number of different factors that must be taken into account when making a decision
3) It may be useful to consider the possibility of reducing the underlaying uncertainty in the decision by collecting additional information; and…
4) A decision-maker's attitude toward risk-taking can impact the relative desirability of different alternatives.

Decision Trees are excellent tools for helping to choose between several courses of action.

They provide a highly effective structure within which you can lay out options and investigate the possible outcomes of choosing those options. They also help you to form a balanced picture of the risks and rewards associated with each possible course of action.

2.3.2 Drawing a Decision Tree

You start a Decision Tree with a decision that you need to make. Draw a small square to represent this towards the left-handside of a large piece of paper.

From this box, draw out lines towards the right for each possible solution and write that solution along the line. Keep the lines as far as apart possible so that you can expand your thoughts.

At the end of each line, consider the results. If the result of taking that decision is uncertain, draw a small circle. If the result is another decision that you need to make, draw another square. Squares represent decisions and circles represent uncertain outcomes. Write the decision or factor above the square or circle. If you have completed the solution at the end of the line, just leave it blank.

Starting from the new decision squares on your diagram, draw out lines representing the options that you could select. From the circles, draw lines representing possible outcomes. Again, make a brief note on the line describing what it means. Keep on doing this until you

have drawn out as many of the possible outcomes and decisions as you can see leading on from the original decisions.

An example of an example of something you will end up with, is shown in Figure 2-2:

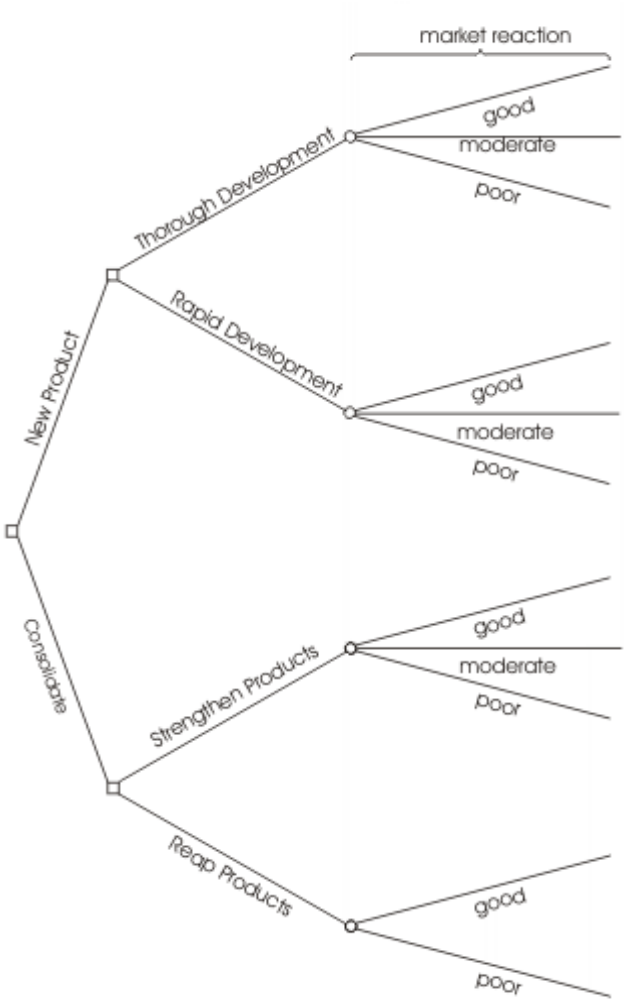


Figure 2-2: Example Decision Tree: Should be to develop a new product or to consolidate

Once you have done this, review your tree diagram. Challenge each square and circle to see if there are any solutions or outcomes you have not considered. If there are - draw them in. If necessary, redraft your tree if parts of it are too congested, or untidy. You should now have a good understanding of the range of possible outcomes of your decisions.

2.3.3 Evaulating a Decision Tree

Now you are ready to evaluate the decision tree. This is where you can work out which option has the greatest worth to you. Start by assigning a cash value - or score, to each possible outcome. Estimate how much you think it would be worth to you if that outcome came about.

So we can say that a decision tree is a decision-support tool that uses a tree-like graph or model of decisions and their possible consequences - including chance event outcomes, resource costs, and utility. It is one way to display by an algorithm.

Decision trees are commonly used in Operations research - specifically, in decision analysis in order to help identify a strategy (that is) most likely to reach a goal.

A decision tree is a flowchart-like structure in which each internal node represents a "test" on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each class-leaf node represents a label (decision taken after computing all attributes). The paths from root to leaf represent classification rules.

In Decision Analysis, a decision tree - and the closely related influence diagram, are used as a visual and analytical decision-support tool where the expected values (or expected utility) of competing alternatives are calculated.

A Decision Tree consists of 3 types of nodes:

1. Decision nodes – commonly represented by squares
2. Chance nodes – represented by circles
3. End nodes – represented by triangles

Decision trees are commonly used in Operations Research - specifically, in decision analysis to help identify a strategy most likely to reach a goal. If decisions in practice have to be taken online with no recall under incomplete knowledge, a decision tree should be paralleled by a probability model as a best choice model - or online selection model algorithm. Another use of decision trees is as a descriptive means for calculating conditional probabilities.

Decision trees Influence Diagrams, Utility Functions, and other decision-analysis tools and methods are taught to Business, Health Economics, and Public Health undergraduate students and are examples of Operations Research or Management Science methods.

2.4 Decision Tree Building Blocks

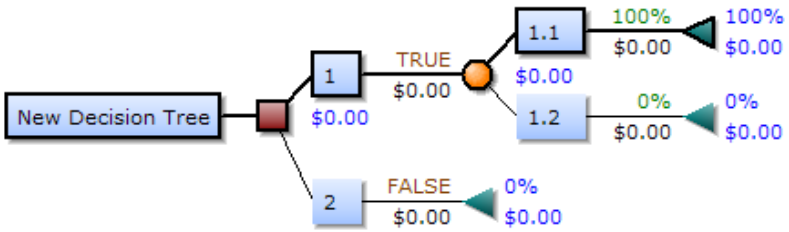


Figure 2-3: Decision Tree Element Blocks

Drawn from left to right, a decision tree has only “burst nodes” (i.e. splitting paths), but no “sink nodes” (i.e. converging paths). Therefore, used manually - they can grow very expansively and are then often hard to draw in full, by hand. Traditionally, decision trees were created manually - as the aside example shows – although increasingly, specialized software is employed.

2.4.1 Decision Rules

The decision tree can be linearized into decision rules - where the outcome is the contents of the leaf node, and the conditions along the path form a conjunction in the “if clause”. In general, the rules have the form: *if* condition1 *and* condition2 *and* condition3 *then* outcome.

Decision rules can also be generated by constructing association rules - with the target variable on the right.

2.4.2 Decision Tree Using Flowchart Symbols

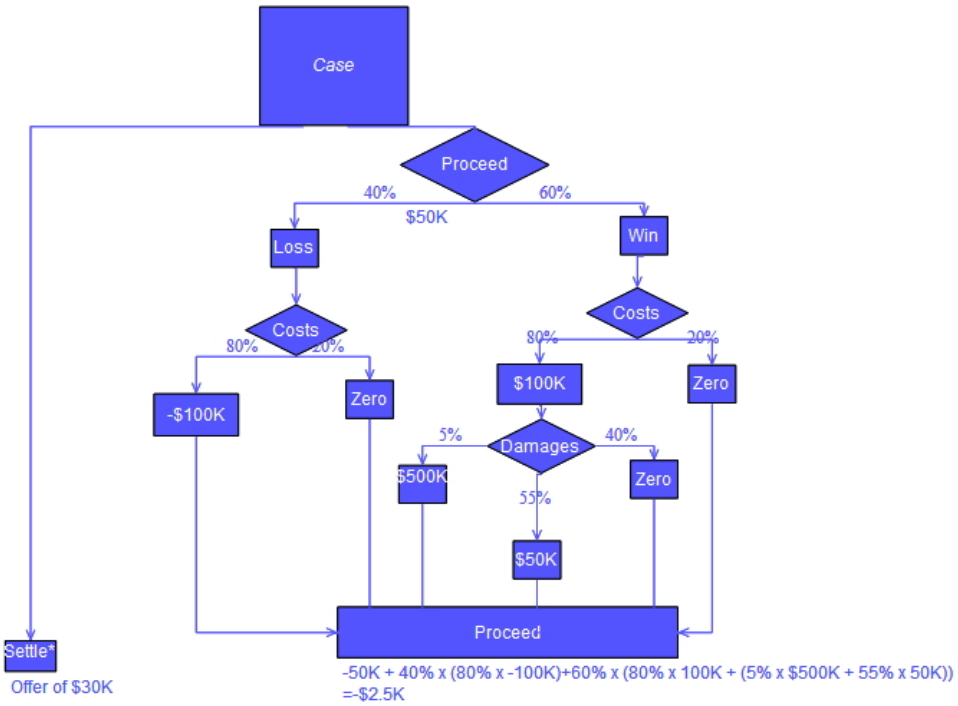


Figure 2-4: Tree Scheme

Commonly, a decision tree is drawn using flowchart symbols since it is easier to read and understand.

2.4.3 Analysis Example

Analysis can take into account the decision maker's (e.g. the company's) preference or utility function**;** for example, risk analysis:

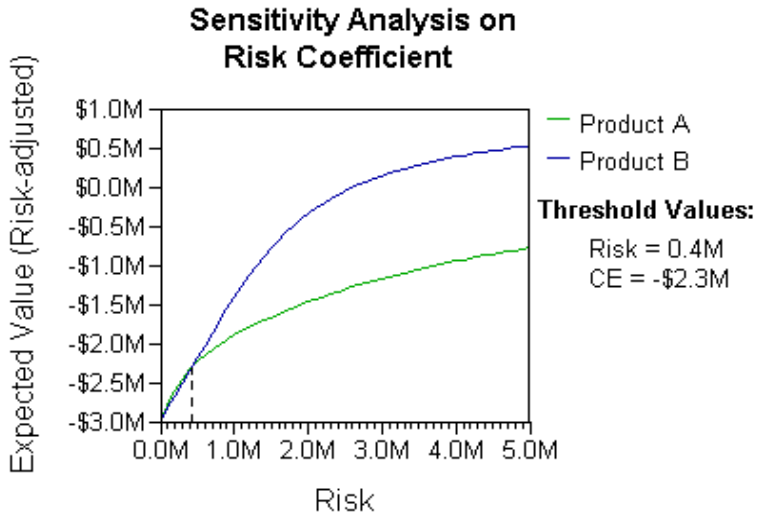


Figure 2-5: Risk Analysis

The basic interpretation in this situation is, that the company prefers B's risk and payoffs under realistic risk preference coefficients (i.e. greater than $400 K - in that range of risk aversion, the company would need to model a third strategy - "Neither A nor B").

2.4.4 An Influence Diagram[10]

Much of the information in a decision tree can more compactly be represented as an Influence Diagram**;** focusing attention on the issues and relationships between events.

Influence Diagrams are hierarchical, and can be defined either in terms of their structure, or in greater detail - in terms of the functional and numerical relation between diagram elements. An ID that is consistently defined at all levels - structure, function, and number - is a well-defined mathematical representation, and is referred to as a Well-Formed Influence Diagram (WFID). WFIDs can be evaluated using reversal and removal operations to yield answers to a large class of probabilistic, inferential, and decision questions. More recent techniques have been developed by the Artificial Intelligence community, with their works relating to Bayesian Network Inference, (Belief Propagation).

An Influence Diagram, having only uncertainty nodes (i.e. a Bayesian Network) is also called a Relevance Diagram. This is perhaps a better use of language than “influence diagram”. An arc connecting node A to B implies not only that "A is relevant to B"; but also, that "B is relevant to A" - (i.e. relevance is a symmetrical relationship). The word “influence” implies more of a one-way relationship which is reinforced by the arc having a defined direction. Since some arcs are easily reversed, this "one-way" thinking that, somehow, "A influences B" is incorrect - (the causality could be the other way around). However, the term “Relevance Diagram” is never adopted in the larger community - and the world continues to refer to an Influence Diagram.

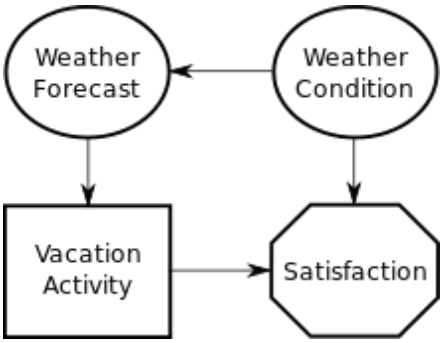


Figure 2-6: A Simple Example about Vacation Activity

A simple influence diagram for making decisions about vacation activity.

Consider the simple influence diagram (above), representing a situation where a decision-maker is planning her vacation.

- There is 1 Decision Node (Vacation Activity), 2 Uncertainty Nodes (Weather Condition, Weather Forecast) and 1 Value Node (Satisfaction)
- There are 2 Functional Arcs (… ending in Satisfaction), 1 Conditional Arc (… ending in Weather Forecast), and 1 Informational Arc (… ending in Vacation Activity)
- Functional Arcs ending in Satisfaction indicate that Satisfaction is a utility function of both Weather Condition and Vacation Activity. In other words, her satisfaction can be quantified if she knows what the weather will be like and what her choice of activity is. (Note that she does not value Weather Forecast directly)
- The Conditional Arc ending in Weather Forecast indicates her belief that Weather Forecast and Weather Condition can be dependent/dependable
- The Informational Arc ending in Vacation Activity indicates that she will only know the Weather Forecast … not the Weather Condition when making her choice. In other words, the actual weather will be known after she makes her choice; and only the "forecast" is what she can count on at this stage
- It also follows - semantically for example, that Vacation Activity is independent of (irrelevant to) Weather Condition; given that the Weather Forecast is known.

2.4.5 Applicability in Value Information

The above example highlights the power of an Influence Diagram to represent an extremely important concept in decision analysis - known as "Information Value". Consider the following three scenarios:

- Scenario 1: The decision-maker could make her *Vacation Activity* decision while knowing what the *Weather Condition* will be like. This corresponds to adding an extra Informational Arc from *Weather Condition* to *Vacation Activity* in the Influence Diagram above
- Scenario 2: The original influence diagram - as shown above
- Scenario 3: The decision-maker makes her decision … without even knowing the *Weather Forecast*. This corresponds to removing the Informational Arc from *Weather Forecast* to *Vacation Activity* in the Influence Diagram above.

Scenario 1 is the best possible scenario for this decision situation - since there is no longer any uncertainty about what she cares about (i.e. *Weather Condition*) when making her decision. Scenario 3 however, is the worst possible scenario for this decision situation since she needs to make her decision - without any hint (i.e. *Weather Fore*cast) on what she cares about – the (*Weather Condition*) will turn out to be.

The decision-maker is usually better off (definitely no worse off) if she moves from Scenario 3 to Scenario 2 through the acquisition of new information. The most she should be willing to pay for such a move is called the "Information Value" of the *Weather Forecast* - which is, essentially, a value of imperfect information on *Weather Condition*.

Likewise, it is best for the decision-maker to move from Scenario 3 to Scenario 1. The most she should be willing to pay for such a move is called the "Perfect Information Value" on *Weather Condition*.

The applicability of this simple ID - and the value of the Information Concept is tremendous; especially in Medical Decision-making, when most decisions have to be made with imperfect information about patients, diseases, etc.

Much of the information in a decision tree can more compactly be represented as an Influence Diagram; focusing attention on the issues and relationships between events.

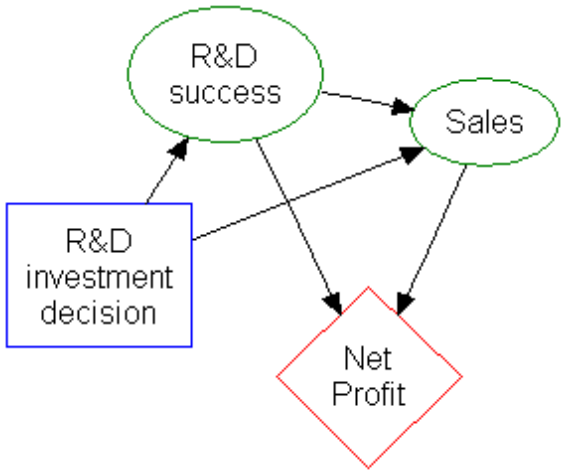


Figure 2-7: SquaresRepresent Decision; Ovals Represent Action; the Diamond Represents Results

2.5 Conclusion — Advantages and Disadvantages

Among Decision Support tools, Decision Trees (and Influence Diagrams), have several advantages; e.g. Decision Trees:

- Are simple to understand and interpret. People are able to understand decision tree models after a brief explanation

- Have value - even with little hard data. Important insights can be generated, based on experts describing a situation - e.g. (its alternatives, probabilities, and costs), and their preferences for outcomes.
- To allow the addition of new possible scenarios
- To help determine the worst, best - and expected values for different scenarios
- To use a White Box model; if a given result is provided by a model
- These can be combined with other decision techniques

Disadvantages of Decision Trees:
For data including categorical variables with a different number of levels, information gain in decision trees are biased in favor of those attributes with more levels.
Calculations can get very complex - particularly if many values are uncertain … and/or, if many outcomes are linked.

2.6 Summary of Information Systems

They substitute computer-based processing for manual procedures; they deal with well-structured processes; they include record-keeping applications; they provide input to be used in the managerial decision process; they deal with supporting well-structured decision situations. Typical information requirements can be anticipated; they provide information to managers who must make judgements about particular situations; they support decision-makers in situations that are not well structured.

3 Knowledge Base[22,23]

Knowledge Base: In general, is a centralized repository for information: e.g. a public library or a database of related information about a particular subject. In relation to Information Technology (IT), a knowledge-base is a machine-readable resource for the dissemination of information - generally on-line, or with the capacity to be put online. An integral component of Knowledge Management systems, a knowledge-base is used to optimize information collection, organization, and retrieval for an organization; or for the general public.

A Knowledge Base (KB), is a technology used to store complex structured information used by a computer system. The initial use of the term was in connection with “Expert Systems”, which were the first knowledge-base systems

The original use of the term “knowledge-base” was to describe one of the two sub-systems of a knowledge-based system. A knowledge-based system consists of a knowledge-base that represents facts about the world - and an inference engine that can reason about those facts and use rules and other forms of logic to deduce new facts or to highlight inconsistencies.

The term "knowledge-base" was used to distinguish it from the more common, widely-used term, “database”. At the time, (the 1970s), virtually all large Management Information Systems stored their data in some type of hierarchical or relational database. At this point in the history of Information Technology, the distinction between a database and a knowledge-base was clear and unambiguous. A database had the following properties:

- Flat Data. Data was usually represented in a tabular format, with strings or numbers in each field
- Multiple Users. A conventional database must support more than one user or system logged into the same data at the same time
- Transactions. An essential requirement for a database was/is to maintain integrity and consistency among data that is accessed by concurrunt users. These are the so-called “ACID” properties: Atomicity, Consistency, Isolation, and Durability
- Large, Long-lived Data. A corporate database needs/ed to support not just thousands but hundreds of thousands or more rows of data. Such a database usually needs to persist past the specific uses of any individual program; it needs to store data for years and decades, rather than for the life of a program

The first knowledge-based systems had data needs that were the opposite of these database requirements. An expert system requires structured data. Not just tables with numbers and strings, but pointers to other objects which, in turn, have additional pointers. The ideal representation for a knowledge-base is an object model (often called an “Ontology” in AI literature), with classes, subclasses, and instances.

Early expert systems also had little need for multiple users or the complexity that comes with requiring transactional properties on data. The data for the early expert systems was used to arrive at a specific answer - a medical diagnosis, the design of a molecule, or a response to an emergency. Once the solution to the problem was known, there was not a critical demand to store large amounts of data backed-up in a permanent memory store. A more precise statement would be, that given the technologies available, researchers compromised and did

without these capabilities because they realized they were beyond what could be expected - and they could develop useful solutions to non-trivial problems without them. Even from the very beginning, the more astute researchers realized the potential benefits of being able to store, analyze, and reuse knowledge.

The volume requirements were/are also different for a knowledge-base compared to a conventional database. The knowledge-base needed to know facts about the world; for example, to represent the statement that: "All humans are mortal". A database typically could not represent this general knowledge but, instead, would need to store information about thousands of tables that represented information about specific humans. Representing that all humans are mortal, and being able to reason about any given human that they are mortal, is the work of a knowledge-base. Representing that George, Mary, Sam, Jenna, Mike... and hundreds of thousands of other customers are all humans with specific ages, sexes, addresses, etc. - is the work of a database.

As expert systems moved from being prototypes to systems deployed in corporate environments, the requirements for their data-storage rapidly started to overlap with the standard database requirements for multiple, distributed users - with support for transactions. Initially, the demand could be seen in two different but competitive markets. Object-oriented Databases such as Versant emerged from the AI and Object-Oriented communities. These were systems designed from the ground-up, to not only have support for object-oriented capabilities - but also, to support standard database services as well. On the other hand, large database vendors like Oracle, added capabilities to their products that provided support for knowledge-base requirements - such as, for instance: class-subclass relations and rules.

The next evolution for the term “knowledge-base”, was the Internet. With the rise of the Internet, documents, hypertext, and multimedia support were now critical for any corporate database. It was no longer enough to support large tables of data or relatively small objects - that lived primarily in computer memory. Support for corporate web-sites required persistence and document transactions. This created a whole new discipline - known as “Web Content Management”. The other driver for document support was the rise of Knowledge Management vendors like Lotus Notes. Knowledge Management actually predated the Internet – but, with the Internet, there was great synergy between the two areas. Knowledge Management products adopted the term "knowledge-base" to describe their repositories - but the meaning had a subtle difference. In the case of previous knowledge-based systems, the

knowledge was primarily for the use of an automated system - to reason about; and draw conclusions about, the world. With Knowledge Mmanagement products, the knowledge was primarily meant for humans; for example, to serve as a repository of manuals, procedures, policies, best practices, reusable designs, code, etc. Of course, in both cases, the distinctions between the uses … and kinds of system were ill-defined. As the technology scaled-up,it was rare to find a system that could really be cleanly classified as "knowledge-based" in the sense of an expert system that performed automated reasoning – and, "knowledge-based" in the sense of Knowledge Management that provided knowledge in the form of documents and media that could be leveraged by humans.

3.1 Knowledge-based Reasoning[14]

The first reasoning systems were theorem provers;, systems that represent axioms and statements in First Order Logic, and then use rules of logic – e.g. "modus ponens", to infer new statements. Another early type of reasoning system was: "general problem-solvers". These were systems like the General Problem Solver designed by Newell and Simon. General problem solvers attempted to provide a generic planning engine that could represent and solve structured problems. They worked by decomposing problems into smaller, more manageable sub-problems, solving each sub-problem - and assembling the partial answers into one final answer. Another general problem solver example was the SOAR Family of Systems.

In practice, these theorem-provers and general problem-solvers were seldom useful for practical applications and required specialized users able to utilize a knowledge of logic. The first practical application of automated reasoning was "expert systems**"**. Expert Systems focused on much more well-defined domains than general problem-solving - such as, for instance, medical diagnosis or for analyzing faults in an aircraft. Expert systems also focused on more limited implementations of logic. Rather than attempting to implement the full range of logical expressions, they typically focused on "modus-ponens", implemented via IF-THEN rules. Focusing on a specific domain, and allowing only a restricted subset of logic, improved the performance of such systems so that they were practical for use in the real world … and not merely as research demonstrations; as most previous automated reasoning systems had been. The engines used for automated reasoning in expert systems were typically called "inference engines**"**. Those used for more general logical inferencing are typically called "theorem provers**"**.

With the rise in popularity of expert systems, many new types of automated reasoning were applied to diverse problems in government and industry. Some, e.g. Case-based Reasoning, were off-shoots of expert systems research. Others, e.g. Constraint Satisfaction Algorithms, were also influenced by fields like Decision Technology and LLinear Programming. Also, a completely different approach - one not based on symbolic reasoning, but on a connectionist model, has also been extremely productive. This latter type of automated reasoning is especially well suited for pattern-matching and signal-detection types of problems like text-searching and face-matching. In Information Technology**,** a reasoning system is a software system that generates conclusions from the available knowledge**,** using logical techniques like deduction and induction. Reasoning systems play an important role in the implementation of artificial intelligence and knowledge-based systems.

According to the everyday-usage and definition of the phrase, all computer systems are reasoning systems in that they all automate some type of logic or decision. However, in typical use in the Information Technology field, the phrase is usually reserved for systems that perform more complex kinds of reasoning. For example, not for systems that perform fairly straight-forward types of reasoning like calculating a sales tax or customer discount, but for making logical inferences about a medical diagnosis or a mathematical theorem. Reasoning Systems come in two modes: Interactive and Batch-processing. Interactive Systems interface with the user by asking clarifying questions, or otherwise allowing the user to guide the reasoning process. Batch Systems take in all the available information at once, and generate the best answer possible without user-feedback or guidance.

Reasoning Systems have a wide field of application including scheduling, business rule processing, problem solving, complex event processing, intrusion detection, predictive analytics, robotics, computer vision, and natural language processing.

3.2 Use of Logic[21]

The term “reasoning system” can be used to apply to just about any kind of sophisticated decision system - as illustrated by the specific areas described below. However, the most common use of the term “reasoning system” implies the computer representation of logic. Various implementations demonstrate significant variations in terms of systems of logic and formality. Most reasoning systems implement variations of propositional and symbolic (predicate) logic. These variations may be mathematically-precise representations of formal

logic systems; or extended and hybrid versions of those systems. Reasoning systems may explicitly implement additional logic types (e.g., modal, deontic, temporal logics). However, many reasoning systems implement imprecise and semi-formal approximations to recognized logic systems. These systems typically support a variety of procedural and semi-declarative techniques in order to model different reasoning strategies. They emphasise pragmatism over formality and may depend on custom extensions and attachments in order to solve real-world problems.

Many reasoning systems employ deductive reasoning to draw inferences from available knowledge. The inference engines use "modus ponens" to support forward-reasoning or backward-reasoning, and to infer conclusions. The recursive reasoning methods they employ are termed 'Forward Chaining' and 'Backward Chaining', respectively. Although reasoning systems widely support deductive inference, some systems employ abductive, inductive, defeasible, and other types of reasoning. Heuristics may also be employed to determine acceptable solutions to intractable problems.

Reasoning systems may employ the Closed World Assumption (CWA) or Open World Assumption (OWA). The OWA is often associated with ontological knowledge representation and the Semantic Web. Different systems exhibit a variety of approaches to negation. As well as logical or bitwise complement**s**, systems may support existential forms of strong and weak negation- including negation-as-failure**,** and 'inflationary' negation (i.e. negation of non-ground atoms). Different reasoning systems may support monotonic or non-monotonic reasoning, stratification and other logical techniques.

3.3 Reasoning Under Uncertainty[19]

Many reasoning systems provide capabilities for reasoning under uncertainty. This is important when building situated reasoning agents which must deal with uncertain representations of the world. There are several common approaches to handling uncertainty. These include the use of Certainty Factors, Probabilistic Methods like Bayesian Inference**,** or the Dempster-Shafer Theory, or multi-valued ('fuzzy') logic, and various other connectionist approaches.

3.4 Types of Reasoning System

This section provides a non-exhaustive and informal categorization of common types of reasoning systems. These categories are not discreet. They overlap to a significant degree, and share a number of techniques, methods and algorithms.

3.4.1 Constraint Solvers

Constraint Solvers solve Constraint Satisfaction Problems (CSPs). They support Constraint Programming. A constraint is a condition which must be met by any valid solution to a problem. Constraints are defined declaratively, and applied to variables within given domains. Constraint solvers use search, back-tracking and constraint propagation techniques to find solutions and to determine optimal solutions. They may employ forms of linear and nonlinear programming. They are often used to perform optimization within highly combinatorial problem spaces. For example, they may be used to calculate Optimal Scheduling, or to design efficient integrated circuits or to maximize productivity in a manufacturing process.

3.4.2 Theorem Provers

Theorem provers use automated-reasoning techniques to determine proofs of mathematical theorems. They may also be used to verify existing proofs. In addition to academic use, typical theorem prover applications include the verification of the correctness of integrated circuits, software programs, engineering designs, etc.

3.4.3 Logic Programs

Logic Programs (LPs), are software programs written using programming languages whose primitives and expressions provide direct representations of constructs drawn from mathematical logic. Prolog is an example of a general-purpose logic programming language. LPs represent the direct application of logic-programming to solve problems. Logic Programming is characterized by highly declarative approaches based on formal logic, and has wide application across many disciplines.

3.4.4 Rule Engines[19,20]

Rule engines represent conditional logic as discrete rules. Rule sets can be managed and applied separately to other functionalities. They have wide applicability across many domains. Many rule engines implement reasoning capabilities. A common approach is to implement production systems to support forward or backward chaining. Each rule ('production'), binds a conjunction of predicate clauses to a list of executable actions. At run-time, the rule engine matches production against facts and executes ('fires') the associated action-list for each match. If those actions remove or modify any facts, or assert new facts, the engine immediately re-computes the set of matches. Rule engines are widely used to model and apply business rules, to control decision-making in automated processes and to enforce business and technical policies.

3.4.5 Deductive Classifiers

Deductive Classifiers arose slightly later than rule-based systems and were a component of a new type of artificial intelligence knowledge representation tool - known as "frame languages". A frame language describes the problem domain as a set of classes, subclasses, and relations among the classes. It is similar to an object-oriented model. Unlike object-oriented models however, frame languages have formal semantics based on first-order logic. They utilize semantics to provide input to the deductive classifier. The classifier in turn, can analyze a given model, (known as an "ontology"), and determine if the various relations described in the model are consistent. If the ontology is not consistent, the classifier will highlight the declarations that are inconsistent. If the ontology is consistent, the classifier can then do further reasoning and draw additional conclusions about the relations of the objects in the ontology. For example, it may determine that an object is actually a subclass, or instance of additional classes, than those described by the user. Classifiers are an important technology in analyzing the ontologies used to describe models in a Semantic Web.

3.4.6 Machine Learning Systems[20]

Machine Learning Systems evolve their behavior over time, based on experience. This may involve reasoning over observed events, or example data provided for training purposes. For example, machine learning systems may use inductive reasoning to generate hypotheses for

observed facts. Learning systems search for generalised rules or functions that yield results in line with observations, and then use these generalisations to control future behavior.

3.4.7 Procedural Reasoning Systems

A Procedural Reasoning System (PRS), uses reasoning techniques to select plans from a procedural knowledge-base. Each plan represents a course of action for the achievement of a given goal. The PRS implements a "belief-desire-intention" model by reasoning over facts ('beliefs') in order to select appropriate plans ('intentions') for given goals ('desires'). Typical PRS applications include management, monitoring and fault detection systems.

3.4.8 Case-based Reasoning [17,18,19]

Case-Based Reasoning (CBR), systems provide solutions to problems by analyzing similarities to other problems for which known solutions already exist. They use analogical reasoning to infer solutions based on case histories. CBR systems are commonly used in customer/technical support and call centres.

History of the CBR Field

The roots of Case-Based Reasoning in AI are found in the works of Roger Schank on Dynamic Memory and the central role that a reminder of earlier situations (e.g. episodes, cases) and situation patterns (i.e. scripts, MOPs) have in problem-solving and learning. Other ventures into the CBR field have come from the study of analogical reasoning, and - further back, from theories on concept formation, problem-solving and experiential learning in the Philosophy and Psychology fields. For example, Wittgenstein observed that 'natural concepts', i.e., concepts that are part of the natural world - like bird, orange, chair, car, etc., - are polymorphic. That is, their instances may be categorized in a variety of ways, and it is not possible to come up with a useful classical definition in terms of a set of necessary and sufficient features for such concepts. An answer to this problem is to represent a concept extensionally, defined by its set of instances – or cases.

The first system that might be called a case-based reasoner was the CYRUS system, developed by Janet Kolodner, at Yale University (Schank's Group). CYRUS was based on Schank's Dynamic Memory Model and the MOP Theory of Problem-solving and Learning. It was basically a question answering system with some knowledge of the various travels and

meetings of former US Secretary of State Cyrus Vance. The case memory model developed for this system later served as thebasis for several other case-based reasoning systems.

Another basis for CBR - and another set of models were developed by Bruce Porter and his group at the University of Texas, Austin. They initially addressed the machine-learning problem of concept learning for classification tasks; this led to the development of the PROTOS system which emphasized integrating general-domain knowledge and specific-case knowledge into a unified representation structure.

The combination of cases with a general domain has developed into a strong environment for this type of CBR. In Gerhard Strube's group at the University of Freiburg, the role of Episodic Knowledge in cognitive models was investigated in the EVENTS project which lead to the group's current research profile of Cognitive Science and CBR. Currently, the CBR activities in the United States - as well as in Europe, are spreading. Germany seems to have taken a leading position in terms of its number of active researchers and several groups of significant size and activity level have recently been established. Japan and other Asian countries also have activity points - e.g. in India. In Japan, the interest is, to a large extent - on focus.

- ***Main CBR Methods Types***

The CBR paradigm covers a range of different methods for organizing, retrieving, utilizing and indexing the knowledge retained in past cases. Cases may be kept as concrete experiences, or a set of similar cases may form a generalized case. Cases may be stored as separate knowledge units; or split-up into sub-units and distributed within the knowledge structure. Cases may be indexed by a prefixed or open vocabulary, and within a flat or hierarchical indexstructure. The solution from a previous case may be directly applied to the present problem, or modified according to differences between the two cases; or the matching of cases, adaptation of solutions, and learning from approaches (e.g. Protos' Methods). Basically, this is an “on-generalization” approach to the concept-learning problem addressed by classical, inductive machine-learning methods.

- ***Memory-based Reasoning***

This approach emphasizes the concept of a collection of cases as a *large memory*, and reasoning as a process of accessing and searching this memory. Memory organization and access is a focus of case-based methods. The utilization of *parallel processing* techniques is a characteristic of these methods - and distinguishes this approach from others. The access and

storage methods may rely on purely syntactic criteria - as in the MBR-Talk System, or they may attempt to utilize general-domain knowledge, like the work done in Japan on *massive parallel memories*.

The typical case-based reasoning methods have some characteristics that distinguish them from the other approaches listed herein. Firstly, a typical case is usually assumed to have a certain degree of information richness contained within it, and a certain complexity with respect to its internal organization. That is to say, a feature vector holding some values and a corresponding class is not what we would call a typical case-description. What we refer to as typical case-based methods also have another characteristic property: They are able to modify - or adapt, a retrieved solution when applied to a different problem-solving context. Paradigmatic case-based methods also utilize general background knowledge - although its richness, degree of explicit representation and role within the CBR processes varies. Core methods of typical CBR systems borrow a lot from Cognitive Psychology theories.

- ***Analogy-based Reasoning***

This term is sometimes used as a synonym for Case-based Reasoning to describe the typical case-based approach just described above. However, it is often also used to characterize methods that solve new problems based on past cases from a *different domain*; while typical case-based methods focus on indexing and matching strategies for single-domain cases. Research on analogy reasoning is therefore, a subfield concerned with mechanisms for the identification and utilization of cross-domain analogies. The major focus of study has been on the *re-use* of a past case - called the "mapping problem": i.e. finding a way to transfer, or map, the solution of an identified analogue (called "source" or "base") to the present problem (called "target" on reusing a previous case, and *retaining* the new experience by incorporating it into the existing knowledge-base (or case-base). The four processes each involve a number of more specific steps that will be described in the task model. This cycle is illustrated in Fig. 3-1.

An initial description of a problem (see top of Fig. 3-1) defines a *new case*. This new case is used to RETRIEVE a case from the collection of *previous cases*.

The *retrieved case* is combined with the new case - through REUSE - into a *solved case*, i.e. a proposed solution to the initial problem. Through the REVISE process, this solution is tested for success - e.g. by being applied to the real-world environment or evaluated by a teacher; and repaired if failed. During RETAIN, useful experience is retained for future reuse, and the

case-base is updated by a new *learned case*, or by modification of some existing cases As indicated in the Figure, general knowledge usually plays a part in this cycle, by supporting the CBR processes.

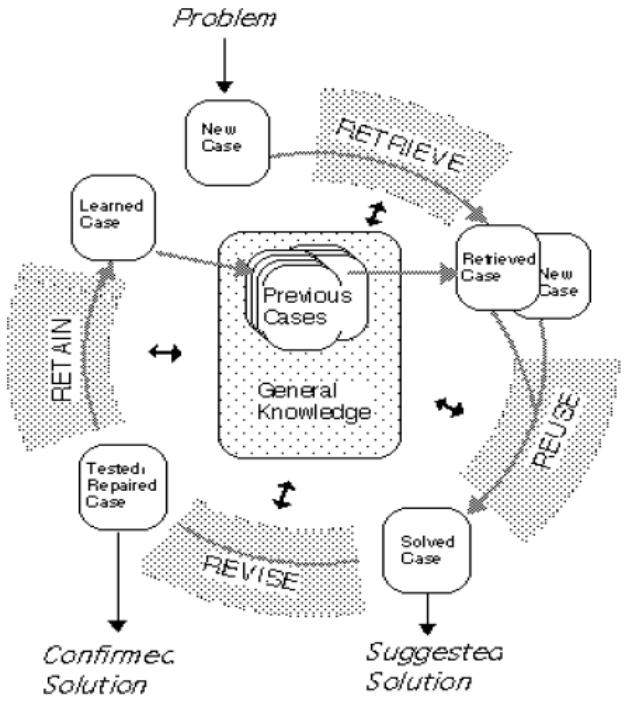


Figure 3-1 The CBR Cycle

This support may range from very weak (or none) to very strong, depending on the type of CBR method. By general knowledge, we here mean general domain-dependent knowledge; as opposed to specific knowledge embodied in cases. For example, in diagnosing a patient by retrieving and reusing the case of a previous patient, a model of anatomy together with causal relationships between pathological states may constitute the general knowledge used by a CBR system. A set of rules may have the same role.

At the highest level of generality, a general CBR cycle may be described by the following four processes:

1. RETRIEVE the most similar case or cases
2. REUSE the information and knowledge in that case to solve the problem
3. REVISE the proposed solution
4. RETAIN the parts of this experience likely to be useful for future problem solving

A new problem is solved by *retrieving* one, or more, previously experienced cases; *reusing* the case in one way or another; and *revising* the solution.

Case-Based Reasoning (**CBR**), broadly construed, is the process of solving new problems based on the solutions of similar past problems. An auto mechanic who fixes an engine by

recalling another car that exhibited similar symptoms is using case-based reasoning. A lawyer who advocates a particular outcome in a trial based on legal precedents, or a judge who creates case law, is using case-based reasoning. So too, is an engineer copying working elements of nature (i.e practicing biomimicry), treating nature as a database of solutions to problems. Case-based reasoning is a prominent kind of analogy-making.

It has been argued that case-based reasoning is not only a powerful method for computer reasoning, but also a pervasive behavior in everyday human problem-solving; or, more radically, that all reasoning is based on personally experienced past cases. This view is related to Prototype Theory - which is most deeply explored in the Cognitive Science field.

Case-Based Reasoning has been formalized for computer reasoning purposes as a four-step process:

1. **Retrieve:** Given a target problem, retrieve from memory cases relevant to solving it. A case consists of a problem, its solution; and, typically, annotations about how the solution was derived. For example, suppose Fred wants to prepare blueberry pancakes. Being a novice cook, the most relevant experience he can recall is one in which he successfully made plain pancakes. The procedure he followed for making the plain pancakes, together with justifications for decisions made along the way, constitutes Fred's retrieved case.
2. **Reuse:** Map the solution from the previous case to the target problem. This may involve adapting the solution as needed to fit the new situation. In the pancake example, Fred must adapt his retrieved solution to include the addition of blueberries.
3. **Revise:** Having mapped the previous solution to the target situation, test the new solution in the real world (or a simulation), and - if necessary, revise. Suppose Fred adapted his pancake solution by adding blueberries to the batter. After mixing, he discovered that the batter had turned blue – an undesired effect. This suggests the following revision: delay the addition of blueberries until after the batter has been ladled into the pan.
4. **Retain:** After the solution has been successfully adapted to the target problem, store the resulting experience as a new case in memory. Fred, accordingly, records his new-found procedure for making blueberry pancakes; thereby enriching his set of stored experiences, and better preparing him for future pancake-making demands.

Comparison to Other Methods

At first glance, CBR may seem similar to the rule-induction algorithms of machine learning. Like a rule-induction algorithm, CBR starts with a set of cases, or training examples; it forms generalizations of these examples - albeit implicit ones, by identifying commonalities between a retrieved case and the target problem.

If, for instance, a procedure for plain pancakes is mapped to blueberry pancakes, a decision is made to use the same basic batter and frying method; thus implicitly generalizing the set of situations under which the batter and frying method can be used. The key difference however, between the implicit generalization in CBR and the generalization in rule induction lies in when the generalization is made. A rule-induction algorithm draws its generalizations from a set of training examples before the target problem is even known; that is, it performs eager generalization.

For instance, if a rule-induction algorithm were given recipes for plain pancakes, Dutch apple pancakes, and banana pancakes as its training examples, it would have to derive - at training time, a set of general rules for making all types of pancakes. It would not be until testing time that it would be given, say, the task of cooking blueberry pancakes. The difficulty for the rule-induction algorithm is in anticipating the different directions in which it should attempt to generalize its training examples. This is in contrast to CBR, which delays (implicit) generalization of its cases until testing time; a “lazy generalization” strategy. In the pancake example, CBR has already been given the target problem of cooking blueberry pancakes; thus it can generalize its cases exactly as needed to cover this situation. CBR therefore tends to be a good approach for rich, complex domains in which there are myriad ways of generalizing a case.

CBR critics argue that it is an approach that accepts anecdotal evidence as its main operating principle. Without statistically relevant data for backing, and implicit generalization, there is no guarantee that the generalization is correct. However, all inductive reasoning where data is too scarce for statistical relevance is inherently based on anecdotal evidence. There is recent work that develops CBR within a statistical framework and formalizes Case-Based Inference as a specific type of probabilistic inference; thus, it becomes possible to produce case-based predictions equipped with a certain level of confidence.

CBR traces its roots to the work of Roger Schank and his students at Yale University in the early 1980s. Schank's Model of Dynamic Memory was the basis for the earliest CBR systems: Janet Kolodner's CYRUS and Michael Lebowitz's IPP.

Other schools of CBR and its closely allied fields emerged in the 1980s, directed at topics such as Legal Reasoning, Memory-Based Reasoning (a way of reasoning from examples on massively parallel machines), and combinations of CBR with other reasoning methods. In the 1990s, interest in CBR grew internationally - as evidenced by the establishment of an International Conference on Case-Based Reasoning in 1995, as well as European, German, British, Italian, and other CBR workshops.

CBR technology has resulted in the deployment of a number of successful systems; the earliest being Lockheed's CLAVIER, a system for laying-out composite parts to be baked in an industrial convection oven. CBR has been used extensively in help-desk applications like the Compaq SMART system, and has found a major application area in the Health Sciences.

3.4.9 Case-Based Reasoning and Logical Reasoning

Informally, two kinds of **logical reasoning** can be distinguished, in addition to formal deduction: induction and abduction. Given a precondition or *premise*, a conclusion or *logical consequence*, and a rule or *material conditional* that implies the *conclusion* given the *precondition*, one can explain that:

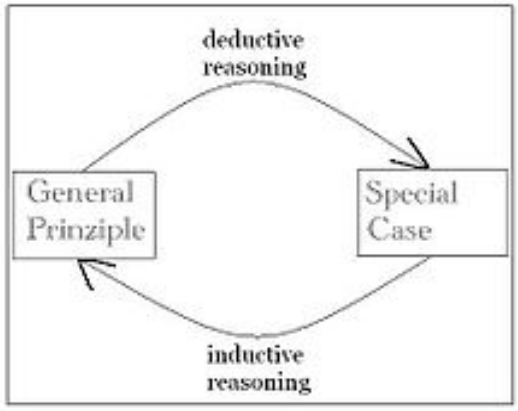


Figure 3-2 Inductive and Deductive Reasoning

- **Deductive Reasoning**: Figure 3-2 determines whether the truth of a *conclusion* can be determined for that *rule*, based solely on the truth of the premises. Example: "When it rains, things outside get wet. The grass is outside, therefore: when it rains, the grass

gets wet." Mathematical logic and philosophical logic are commonly associated with this type of reasoning.

- **Inductive Reasoning**: Attempts to support a determination of the *rule*. It hypothesizes a *rule* after numerous examples are taken to be a *conclusion* that follows from a *precondition* in terms of such a *rule*. Example: "The grass got wet numerous times when it rained, therefore: the grass always gets wet when it rains." While they may be persuasive, these arguments are not deductively valid; see the problem of induction. Science is associated with this type of reasoning.
- **Abductive Reasoning**: a.k.a. *inference to the best explanation*, selects a cogent set of *preconditions*. Given a true *conclusion* and a *rule*, it attempts to select some possible *premises* that, if true also, can support the *conclusion*, though not uniquely. Example: "When it rains, the grass gets wet. The grass is wet. Therefore, it might have rained." This kind of reasoning can be used to develop a hypothesis, which in turn can be tested by additional reasoning or data. Diagnosticians, detectives, and scientists often use this type of reasoning.

3.4.10 Common-sense Reasoning

Commonsense Reasoning is the branch of Artificial Intelligence concerned with simulating the human ability to make deductions about the kind of ordinary situations they encounter every day. These include judgements about the physical properties, purpose, intentions and possible behavior of all ordinary things, e.g. people, cups, water, blocks, clouds, animals, and so on. A machine which exhibits common-sense reasoning will be capable of drawing conclusions that are similar to a human's folk psychology (i.e. the innate human ability to reason about other people's intentions and mental states) or naive physics (e.g. the understanding of the physical world which every normal human child exhibits).

There are several components to this problem, including:

- Developing adequately broad and deep commonsense knowledge bases
- Developing reasoning methods that exhibit the features of human thinking, including the ability to:
    - Reason with knowledge that is true by default
    - Reason rapidly across a broad range of domains
    - Tolerate uncertainty in your knowledge

    - Take decisions under incomplete knowledge and perhaps revise that belief or decision when complete knowledge becomes available
- Developing new kinds of cognitive architectures that support multiple reasoning methods and representation.

4 Knowledge-Based Systems[14]

A **Knowledge-Based System** (**KBS**), is a computer program that reasons and uses a knowledge-base to solve complex problems. The term is broad and is used to refer to many different kinds of systems. The one common theme that unites all knowledge-based systems is an attempt to represent knowledge explicitly via tools like ontologies and rules, rather than implicitly via code, the way a conventional computer program does. A knowledge-based system has two types of sub-systems: a knowledge-base and an inference engine. The knowledge-base represents facts about the world, often in some form of subsumption ontology. The inference engine represents logical assertions and conditions about the world, usually represented by IF-THEN rules.

Knowledge-Based systems were first developed by Artificial Intelligence researchers. These early knowledge-based systems were primarily expert systems. In fact, the term is often used synonymously with expert systems. The difference is in the view taken to describe the system. The term “Expert System” refers to the type of task the system is trying to solve, to replace, or to aid a human expert in a complex task.

A knowledge-based system refers to the architecture of the system; that it represents knowledge explicitly, rather than as procedural code. While the earliest knowledge-based systems were almost all expert systems, the same tools and architectures can - and have since been, used for a whole host of other types of systems; i.e., virtually all expert systems are knowledge-based systems - but many knowledge-based systems are not expert systems.

The first knowledge-based systems were rule-based expert systems. One of the most famous was Mycin- a medical diagnosis program. These early expert systems represented facts about the world as simple assertions in a flat data-base, and used rules to reason about – and, as a result, add to these assertions. Representing knowledge explicitly via rules had several advantages:

- Acquisition & Maintenance: Using rules meant that domain experts could often define and maintain the rules themselves rather than via a programmer
- Explanation: Representing knowledge explicitly allowed systems to reason about how they came to a conclusion and use this information to explain results to users. For example, to follow the chain of inferences that led to a diagnosis and then use these facts to explain the diagnosis
- Reasoning: Decoupling the knowledge from the processing of that knowledge enabled general purpose inference engines to be developed. These systems could develop conclusions that followed from a data set that the initial developers may not have even been aware of

As knowledge-based systems became more complex, the techniques used to represent the knowledge-base became more sophisticated. Rather than representing facts as assertions about data, the knowledge-base became more structured - representing information using similar techniques to object-oriented programming, such as – for instance, hierarchies of classes and subclasses, relations between classes, and the behavior of objects. As the knowledge-base became more structured, reasoning could occur - both by independent rules, and by interactions within the knowledge-base itself. For example, procedures stored as objects could replicate the chaining behavior of rules.

A further advancement was the development of special-purpose automated reasoning systems - called “classifiers”. Rather than statically declare the subsumption relations in a knowledge-base, a classifier allows the developer to simply declare facts about the world, and let the classifier deduce the relations. In this way, a classifier also can play the role of an inference engine.

The most recent advancement of knowledge-based systems has been to adopt the technologies used for the development of systems that use the Internet. The Internet often has to deal with complex, unstructured data that can't be relied on to fit a specific data model. The technology of knowledge-based systems - and especially the ability to classify objects on demand, is ideal for such systems. The model for these kinds of knowledge-based Internet systems is known as the Semantic Web.

4.1 Knowledge-Based System Structure

Structured methologies for the development of knowledge-based systems are now in practical use in many places in Europe and elsewhere.

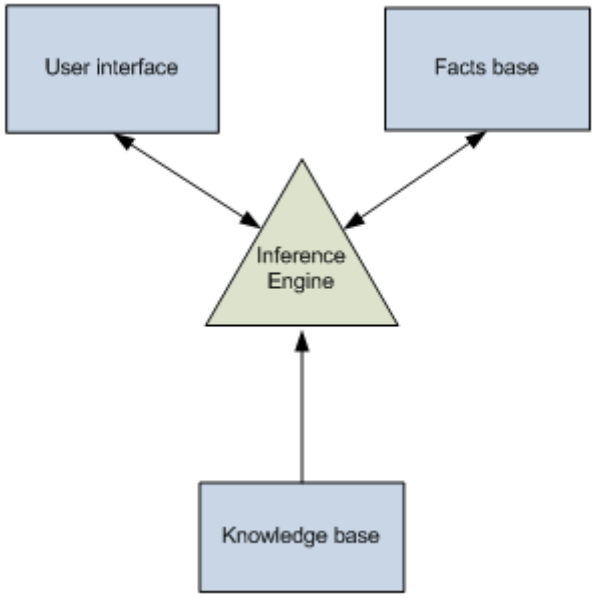


Figure 4-1 A Knowledge-based Schema

Knowledge Base

A KBS contains human expert knowledge in a formalized and structured form; this is what is known as a knowledge-base. Some Knowledge-Based Systems include "**meta-knowledge"**; or, knowledge about knowledge - that is to say, the ability to search the knowledge-base and find the solution to a problem in an intelligent manner, using different resolution strategies, with their particular conditions. This means that some criteria are defined, by which the system decides on one or the other strategy based on the initial data. The knowledge-base can be represented as predicate calculus, lists, objects, semantic networks; and/or, production rules. Most frequently, it is implemented using rules and objects.

An Inference Engine

This is also called a "rule interpreter", and its goal is to search for, and select, the correct rule to be applied in the reasoning process.

The Facts Base

This is like a temporal-auxiliary memory that stores the user-data, initial problem data, hypothesis, and intermediate results during the inference process. Through it, we can

determine the current system status, and how it was reached. The best way to store this information is in relational databases, rather than in other rudimentary systems.

An Interface

Allows communications with the UML and KBS data input and output.

4.2 Knowledge Engineering: Principles and Methods

This section provides an overview of the development of the field of Knowledge Engineering over the last 15 years. We discuss the paradigm shift from a “transfer view” to a “modeling view” and describe two approaches which considerably shaped Knowledge Engineering research: Role-limiting Methods and Generic Tasks. To illustrate various concepts and methods which have evolved in recent years, we describe three modeling frameworks: CommonKADS, MIKE and PROTÉGÉ-II. This description is supplemented by discussing some important methodological developments in more detail: specification languages for knowledge-based systems, problem-solving methods and ontologies. We conclude by outlining the relationship of Knowledge Engineering to Software Engineering, Information Integration and Knowledge Management.

4.3 KADS: A Modelling Approach to Knowledge Engineering [27]

The KADS Methodology (Schreiber et al. 1993; Tansley & Hayball 1993), and its successor - CommonKADS (Wielinga et al. 1992), have proved to be very useful approaches for modeling the various transformations involved between eliciting knowledge from an expert and encoding this knowledge in a computer program. These transformations are represented in a series of models. While it is widely agreed that these methods are excellent approaches from a theoretical viewpoint, the documentation provided herewith concentrates on defining what models should be produced; with only general guidance on how the models should be produced. This has the advantage of making KADS and CommonKADS widely applicable; but, it also means that considerable training and experience is required to become proficientin using them. The KADs approach in knowledge engineering is the development of a knowledge-based system (KBS-Modeling Activity). A KBS is not a container filled with knowledge extracted from an expert, but an operational model that exhibits some desired behavior that can be observed in terms of real-world phenomena. Five basic principles underly the KADS approach, namely: (i) The introduction of partial models as a means to cope with the complexity of the knowledge engineering process; (ii) the KADS four-layer

framework for modelling the required expertise; (iii) the re-usability of generic model components as templates supporting top-down knowledge acquisition; (iv) the process of differentiating simple models into more complex ones; and, (v) the importance of structure - preserving transformation of models of expertise into design and implementation.

The development of Knowledge-Based Systems is not sufficiently standardized. Nevertheless, there is an emerging European methodology - funded as an **ESPRIT** project, to develop Knowledge-Based Systems. It is called KADS (*Knowledge Acquisition Design System*). For KADS, building a Knowledge-Based System is basically a modeling activity. One of the most important characteristics in KADS is the construction of a complete knowledge-model that is independent of the implementation. Finally, KADS is a results-oriented methodology; the results obtained as a product of each development activity are the only objective criteria for project control and direction.

4.4 The CommonKADS Methodology [28, 29]

The aim of CommonKADS is to fill the need for a structured methodology for KBS projects by constructing a set of engineering models built with organization and application in mind. We provide a brief overview of the CommonKADS methodology - while paying special attention to “expertise modeling”; an aspect of KBS development that distinguishes it from other types of software development.

CommonKADS is the leading methodology in support of KBS Engineering. It has been developed and tested by many companies and universities in the context of the European ESPRIT program. It is now the European, de-facto standard, for Knowledge-Based System development; and has been adopted in companies in Europe - as well as the US and Japan. CommonKADS also provides methods to perform a detailed analysis of knowledge tasks and processes. The image below depicts the Knowledge-Based Systems methodologies evolution.

Figure 4-2 Knowledge Evolution

CommonKADS originated from the KADS-I project, a longer - and more technically and better staffed project; though this methodology lacked formalization. In the winter of 1990, development began on a new methodology that was commercially viable - and covered the entire KBS life-cycle. The result was a new methodology, called CommonKADS.

4.5 Model Sets

"A model reflects - through detail abstraction, system characteristics in the real world. Each model highlights certain system characteristics and abstracts other ones". [DeMarco, 82]

The model set provides task decomposition in knowledge engineering - which reduces the complexity. When a model is being built, other aspects can be left for later. The commonKADS models are shown in the image below:

Figure 4-3 KADS Model

4.5.1 Organization Model

This describes and analyzes the main activities of an enterprise.

4.5.2 The Task Model

Analyzes the organization's global sub-process scheme: input, output, preconditions, performance criteria, resources, and competencies.

4.5.3 TheAgent Model

This describes Agent characteristics as task executors: competencies, authorizations, and restrictions.

4.5.4 The Communication Model

A conceptual description of agent transactions involved in a task.

4.5.5 The Knowledge Model

Describes knowledge types and structures used in a task and the role of these knowledge components in the task resolution, but *implementation is independent.*

4.5.6 The Design Model

Starting with the previous models, this describes the technical specifications – e.g. architecture, implementation platform, software modules, etc., in order to attain the functionality specified in the Knowledge and Communication models. Based on these models, we will focus on the ***Knowledge Model*** - which has the following structure, shown below:

Figure 4-4 Application Knowledge

4.5.7 Domain Knowledge

This details the entities and concepts regarding the application domain (i.e. knowledge domain), independently of how they have been used for other knowledge types.

4.5.8 Inference Knowledge

This deals with the different types of inferences (e.g. reasoning elements used by experts in the task´s solution). An inference is defined by its input and output - which are the domain roles. An inference does not allow subsequent decompositions.

4.5.9 Task Knowledge

Specifies the task goal and the method to use to solve it.

4.5.10 Problem-solving Methods

These describe any task-solving method, specifying the subtasks for recursive decomposition, and the order in which to execute them.

4.5.11 Strategic Knowledge

This specifies the task plan. Considerationof strategic knowledge allows more flexible system design.

4.6 Roles

Like any other software project, there must be a human structure to organize, manage, and develop the Knowledge-Based Systems. In KBS development, there are six relevant roles involved in the system´s construction. The most representative ones are depicted below, and explained in the following lines:

Figure 4-5 Knowledge Roles

4.6.1 A Knowledge-Provider/Specialist

This is the owner of human "knowledge", typically an expert in the application domain, but they could be another person in the organization who does not have expert status.

4.6.2 A Knowledge Engineer/Analyst

One important problem for a Knowledge Engineer is to find the real experts - as reffered to above. The term "Knowledge Engineer" is usually reserved for system-analysis work. These people could be also called a "Knowledge Analyst"; therefore, these two terms can be considered interchangeable. CommonKADS offers the Knowledge Engineer a range of methods and tools that make the analysis of a standard knowledge-intensive task relatively straightforward.

4.6.3 A Knowledge System Developer

Knowledge System Developers are responsible for design and implementation. The developer must have a basic background of analysis methods. In knowledge system development, the

main knowledge problems have to be solved by the knowledge analyst. Therefore, this role must have someone with software designer skills.

4.6.4 A Knowledge User

A Knowledge User makes use - directly or indirectly, of a knowledge system. Their interaction with the KBS is important for project development and validation.

4.6.5 A Project Manager

Manages the project - especially, the Knowledge Engineer and Knowledge System Developer.

4.6.6 A Knowledge Manager

Is a person at the top of the hierarchy who acts as a Project Manager - but at higher levels; it's a role like a knowledge strategist, cooperating, defining, and distributing the knowledge to coordinate all the other roles.

4.6.7 Views on Knowledge Acquisition [47]

During the knowledge acquisition process, the knowledge that a knowledge-based system (KBS) needs in order to perform a task is defined in such a way that a computer program can represent and adequately use that knowledge. Knowledge Acquisition involves - in our view, at least the following activities: eliciting the knowledge in an informal – usually verbal – form, interpreting the elicited data using some form of conceptual framework, and formalising the conceptualisations in such a way that the program can use the knowledge. In this chapter, we will mainly focus on the interpretation and formaližation activities in knowledge acquisition.

Traditionally, the knowledge acquisition process was viewed as a process for extracting knowledge from a human expert and transferring the extracted knowledge into the KBS. In practice, this often means that the expert is asked what rules are applicable in a certain problem situation and the Knowledge Engineer translates the natural language formulation of these rules into the appropriate format.

The expert, the Knowledge Engineer and the KBS should share a common point-of-view on the problem-solving process and a common vocabulary in order to make knowledge transfer a viable way of knowledge acquisition. If the expert looks at the problem or the domain in a different way than the Knowledge Engineer, asking for rules or similar knowledge

structures and translating them into the knowledge representation formalism of the system does not work.

A different view on knowledge acquisition is that of a modelling activity. A KBS is not a container filled with knowledge extracted from an expert, but an operational model that exhibits some desired observed behavior, or specified in terms of real-world phenomena. The use of models is a means of coping with the complexity of the development process.

Constructing a KBS is seen as building a computational model of the desired behaviour. This desired behaviour can coincide with some behaviour as exhibited by an expert. If one wants to construct a KBS that performs medical diagnoses, the behaviour of a physician in asking questions and explaining a patient´s problem may be a good starting-point for a description of the intended problem-solving behaviour of the KBS. However, a KBS is hardly ever the functional and behavioural equivalent of an expert. There are a number of reasons for this. Firstly, the introduction of information technology often involves new distributions of functions and roles for the agents. The KBS may perform functions which are not part of the experts´ repertory. Secondly, the expert´s underlying reasoning process can often not be made fully explicit. Knowledge, principles and methods may be documented in a domain, but these are aimed at a human interpreter and are not descriptions of how to solve problems in a mechanical way. Thirdly, there is an inherent difference between the capabilities of machines and humans. The decision is guided by the fact that - for a machine, it presents no problem to store a large number of hypotheses in short-term memory; whereas for humans, this is impossible.

So, in the modelling view, knowledge acquisition is essentially a constructive process in which the Knowledge Engineer can use all sorts of data about the behaviour of the expert; but in which the ultimate modelling decisions have to be made by the Knowledge Engineer in a constructive way. In this sense, Knowledge Engineering is similar to other design tasks: the real world only provides certain constraints on what the artefact should provide in terms of functionality; the designer has to aggregate the bits and pieces into a coherent system.

In KADS, we have adopted the modeling perspective of knowledge acquisition. The KADS approach can be characterised through a number of principles that underlie the process to building knowledge-based systems, namely:

- The introduction of multiple models as a means for coping with the complexity of the Knowledge Engineering process
- The KADS four-layer framework for modelling the required expertise
- The reusability of generic model components as templates supporting top-down knowledge acquisition
- The process of differentiating simple models into more complex ones
- The importance of the structure-preserving transformation of expertise models into the design and implementation phases

4.6.8 Principle 1: Multiple Models

The construction of a knowledge-based system is a complex process. It can be viewed as a search through a large range of knowledge-engineering methods, techniques and tools. Numerous choices have to be made with regard to elicitation, conceptualisation and formalisation. Knowledge Engineers are thus faced with a jungle of possibilities and find it difficult to navigate through this space.

The idea behind the first principle of KADS is that the knowledge-engineering range of choices and tools can - to some extent, be controled by the introduction of a number of models. Each model emphasises certain aspects of the system to be built, and abstracts from others. Models provide a decomposition of knowledge-engineering tasks: while building one model, the Knowledge Engineer can temporarily neglect certain other aspects. The complexity of the knowledge-engineering process is thus reduced through a “divide-and-conquer” strategy.

In this section, we discuss a number of models, namely: (i) the Organizational Model; (ii) the Application Model; (iii) the Task Model; (iv) the Cooperation Model; (v) the Expertise Model; (vi) the Conceptual Model; and, (vii) the Design Model.
We use the term *Knowledge Engineering* in a broader sense to refer to the overall process of KBS construction (i.e. the construction of all of these models and artefacts) and the term *Knowledge Acquisition* in a more restricted sense - to refer to those parts of this construction process that are concerned with the information about the actual problem-solving process. The scope of the present chapter is limited to the Knowledge Acquisition aspects. Other Knowledge Engineering aspects are only briefly addressed.

4.6.9 The Organizational Model, Application Model and Task Model

In KADS, we distinguish three separate steps in defining the goals of KBS construction, namely:

- Defining the *problem* that the KBS should solve in the organization
- Defining the *function* of the system with respect to future users (which can be either humans or, possibly, other systems)
- Defining the actual *tasks* that the KBS will have to perform

In this section, we discuss three models that address parts of this three-step process. The first two are discussed briefly - since these are outside the scope of this chapter.

The Organizational Model: An organizational model provides an analysis of the socio-organizational environment in which the KBS will have to function. It includes a description of the functions, tasks and bottlenecks *in the organization*. In addition, it describes (predicts) how the introduction of a KBS will influence the organization and the people working in it.

The result may be that the final system is aimed at solving a problem that no longer exists in the organization. We are convinced that the organizational viewpoint is important throughout the KBS construction process.

The Application Model: An application model defines what problem the system should solve in the organization - and what the function of the system will be in this organization. For example, the daily operation and fault-handling of an audio-system can pose serious problems for people who are not familiar with - or just not interested in, more than the superficial “ins-and-outs” of such a system. A potential solution to this problem could be the development of a knowledge-based system. The function of this system would be to ensure that the owner of the audio-system is supported in the process of correcting operational malfunctions of the audio-system.

In addition to the function of the KBS and the *problem* that it is supposed to solve, the application model specifies the *external constraints* that are relevant for the development of the application. Examples of such constraints are: the required speed and/or efficiency of the KBS, and the use of particular hardware or software.

The Task Model: A task model specifies how the function of the system (as specified in the application model), is achieved through a number of tasks that the system will perform. Establishing this relationship between function and task is not always as straightforward as it may seem. For example, consider a problem like the medical care of patients with acute infections of the bloodstream. One approach to solve this problem is to perform the following tasks: (i) Determine the identity of the organism that is causing the infection; and, (ii) Select, on the basis of that diagnosis, the optimal combination of drugs to administer to the patient. In real-life hospital practice however, the recovery of the patient is the primary concern. So, if identification of the offending organism proves difficult - e.g., because no laboratory data are available, a therapy will have to be selected on other grounds. In fact, some doctors show little interest in the precise identity of the organism causing an infection - as long as the therapy works. Stated in more general terms: Given a goal that a system should achieve; there may be several alternative ways in which that goal can be achieved. Which alternative is appropriate in a given application depends on the characteristics of that application; on the availability of knowledge and data; and on the requirements imposed by the user - or by external factors.

With respect to the content of the task model, we can distinguish three facets: (i) Task Decomposition, (ii) Task Distribution, and (iii) Task Environment.

Task Decomposition: A task is identified that would achieve the required functionality. This task is decomposed into sub-tasks. A technique like Rational Task Analysis is often used to achieve such a decomposition. We call the composite top-task a "real-life task", as it often represents the actual task that an expert solves in the application domain. The sub-tasks are the starting-point for further explorations, e.g. the modeling of expertise and cooperation. A simple decomposition of a real-life task in the audio domain is shown in Fig. 4-6.

Each separate task is described through an input/output specification; where the output represents the goal that is achieved with the task and the input is the information that is used to achieve this goal. What constitutes the goal of a task is not always self-evident.
Even for a seemingly well-understood task - such as diagnosis for instance, it is not always clear what a diagnosis of a faulty system means. A diagnosis could be the identification of a subsystem (e.g. a component of an audio-system) that is malfunctioning, or it could be a full causal model of how a malfunction came about. Similarly, the result of a design task could be a detailed description of the structure of a system (e.g. a device for monitoring patients in an

intensive care unit) or it could be a description of the functionality, structure and use of the device.

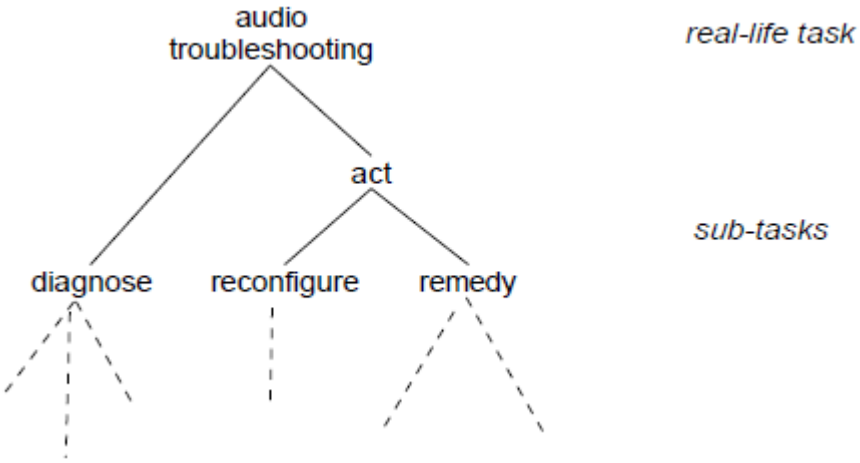


Figure 4-6 Task Decomposition of the Audio Example

Task Distribution: Task distribution involves the *assignment* of tasks to *agents.* Example agents are the KBS, the user - or some other system. The last two agents are called *external* agents. Given the task decomposition, the knowledge engineer has to decide what sub-tasks to assign to the system and what tasks to the user. These decisions essentially constitute cognitive engineering problems: they should be made on the basis of an analysis of the user requirements and expectations, the knowledge and skills that the user has, and the potential capabilities and limitations of the system.

Task Environment: The nature of the task-domain itself usually enforces a number of constraints on how the task can be performed. We call these constraints the *task environment*. For example, the task environment of a support system for handling malfunctions in an audio system could consist of the following constraints:

- The KBS is not a physical part of the audio system
- It has no sensors to make observations - (and thus depends on the user to do this)
- It has no robot arm to perform recongurations and/or repairs (and thus again, depends on the user to do this)
- The KBS users will be novices who are not expected to be able to understand technical terms, or to examine the interior parts of the audio-system

The constraints posed by the task environment influence both the scope and the nature of the expertise and cooperation models (see further).

4.6.10 The Cooperation Model

The task model consists of a decomposition of the real-life task into a number of primitive tasks … and the distribution of tasks among the agents. The cooperation model contains a specification of the functionality of those sub-tasks in the task model that require a cooperative effort. These tasks can - for instance, be data acquisition tasks activated during problem-solving or various types of explanation tasks. Such tasks are called *transfer* tasks, since they involve transferring a piece of information from the system to an external agent or vice-versa.

There is thus a clear dependency between the cooperation model and the expertise model. Some of the sub-tasks will be achieved by the system, others may be realised by the user. For example, in the diagnostic task in the audio example, the system may suggest certain tests to be performed by the user; while the user will actually perform the tests and will report the observed results back to the system. Alternatively, the user may want to volunteer a solution to the diagnostic problem; while the system will criticise that solution by comparing it with its own solutions.

The result is a model of cooperative problem-solving in which the user and the system, together, achieve a goal in a way that satisfies the various constraints posed by the task environment, the user and the state of the art of KBS technology.

4.6.11 The Expertise Model

Building an expertise model is a central activity in the KBS construction process. It distinguishes KBS development from conventional system development. Its goal is to specify the problem-solving expertise required to perform the problem-solving tasks assigned to the system.

One can take two different perspectives on modeling the expertise required from a system. The first perspective – one that is often taken in AI – is to focus on the computational techniques and the representational structures (e.g. rules, frames), that will provide the basis of the implemented system. The second perspective focuses on the behavior that the system should display and on the types of knowledge that are involved in generating such behavior - by abstracting this from the details of how the reasoning is actually realised in the implementation.

We hold to the second perspective and view the expertise model as being a knowledge-level model. The expertise model specifies the desired problem-solving behavior for a target KBS through an extensive categorization of the knowledge required to generate this behavior. The model thus fulfills the role of a *functional specification* of the problem-solving part of the artefact. As stated previously, it is not a cognitive model of the human expert. Although the construction of the expertise model is usually guided by an analysis of expert behaviour, it is biased as to what the target system should - and can do.

In modeling expertise, we abstract from those sub-tasks that specify some form of cooperation with the user. In the audio domain for example, we could identify two tasks that require such interactions: *performing a test* and *carrying out a reconguration*. In the expertise model, such interactions or transfer tasks are specified, more-or-less, as a black box (see Sec. 4.6.14). The detailed study of the nature of these transfer tasks is the subject of the modeling of cooperation.

As the expertise model plays a central role in KBS development, its details are discussed extensively in Sec. 4.6.14.

4.6.12 The Conceptual Model = Expertise Model + Cooperation Model

Taken together, the expertise model and the cooperation model provide a specification of the behavior of the artefact to be built. The model that results from the merging of these two models is similar to what is called a *conceptual model* in database development. Conceptual models are abstract descriptions of the objects and operations that a system should know about, formulated in such a way that they capture the intuitions that humans have of this behavior. The language in which the conceptual models are expressed is not the formal language of computational constructs and techniques, but is the language that relates real-world phenomena to the cognitive framework of the observer. In this sense, conceptual models are subjective - they are relative to the cognitive vocabulary and framework of the human observer. Within KADS, we have adopted the term "conceptual model" to denote a combined, implementation-independent model of both expertise and cooperation.

4.6.13 The Design Model

The description of the computational and representational techniques that an artefact should use to realise the specified behavior is not part of the conceptual model. These techniques are

specified as separate design decisions in a design model. In building a design model, the knowledge engineer takes into consideration external requirements - like speed, hardware and software, into account. Although there are dependencies between the conceptual model specifications on the one hand and the design decisions on the other, we have experienced that building a “conceptual model” model without having to worry about system requirements makes life easier for knowledge engineers.

The separation between conceptual modeling on the one hand and a separate design step on the other has been identified as both the strength - and the weakness, of the KADS approach.

The main advantage lies in the fact that the knowledge engineer is not biased during conceptual modeling by the restrictions of a computational framework. KADS provides a more-or-less universal framework for modeling expertise (see the next section); and, although computational constraints play a role in the construction of such models, experience has shown that this separation enables knowledge engineers to come up with more comprehensive specifications of the desired behaviour of the artefact. The disadvantage lies in the fact that the knowledge engineer - after having built a conceptual model, is still faced with the problem of how to implement this specification.

Fig. 4.7 (see previous chapter), summarises the different roles which the conceptual model and the design model play in the knowledge engineering process. An observer (knowledge engineer) constructs a conceptual, knowledge-level model of the artefact by abstracting from the behaviour of experts. This abstraction process is aided by the use of an interpretational framework, e.g. generic models of tasks classes or task-domains. The conceptual model is real-world oriented in the sense that it is phrased in real-world terminology and can thus be used as a communication vehicle between a knowledge engineer and an expert. The conceptual model does not take into account detailed constraints with regard to the artefact. The design model - on the other hand, is a model that is phrased in the terminology of the artefact: it describes how the conceptual model is realized with particular computational and representational techniques.

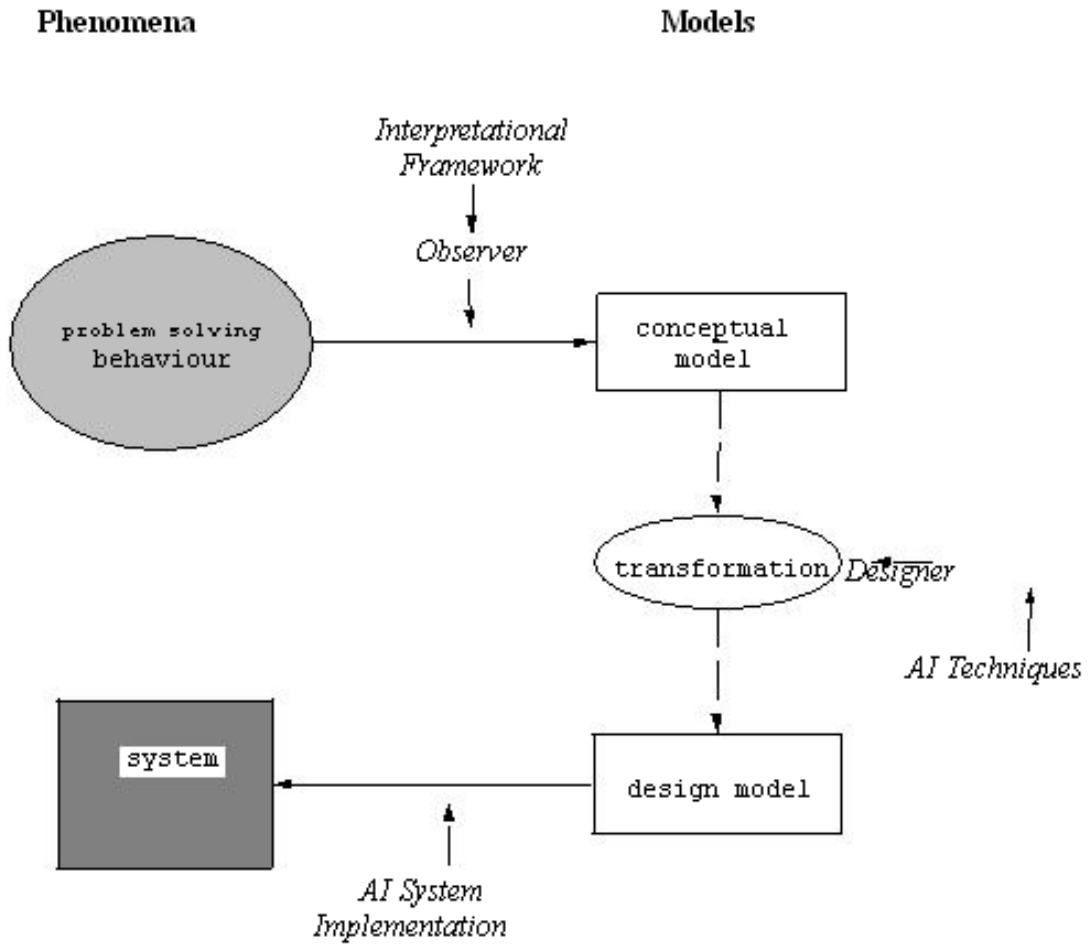


Figure 4-7 Role of the Conceptual (Knowledge-level) Model and the Design (Symbol Level) Model in the Development of Knowledge-based Systems

Fig. 4-8 shows the dependencies between the models discussed in this section. Connections indicate that information from one model is used in the construction of another model. The actual activities in the construction process do not necessarily have to follow the direction from organization model to system. In fact, several life-cycle models have been developed, each depending on various phases and activities in building these models. The first life-cycle model developed in KADS was of the “water-fall” type. At the end of the KADS project, a new life-cycle was defined - based on the concept of a spiral model.

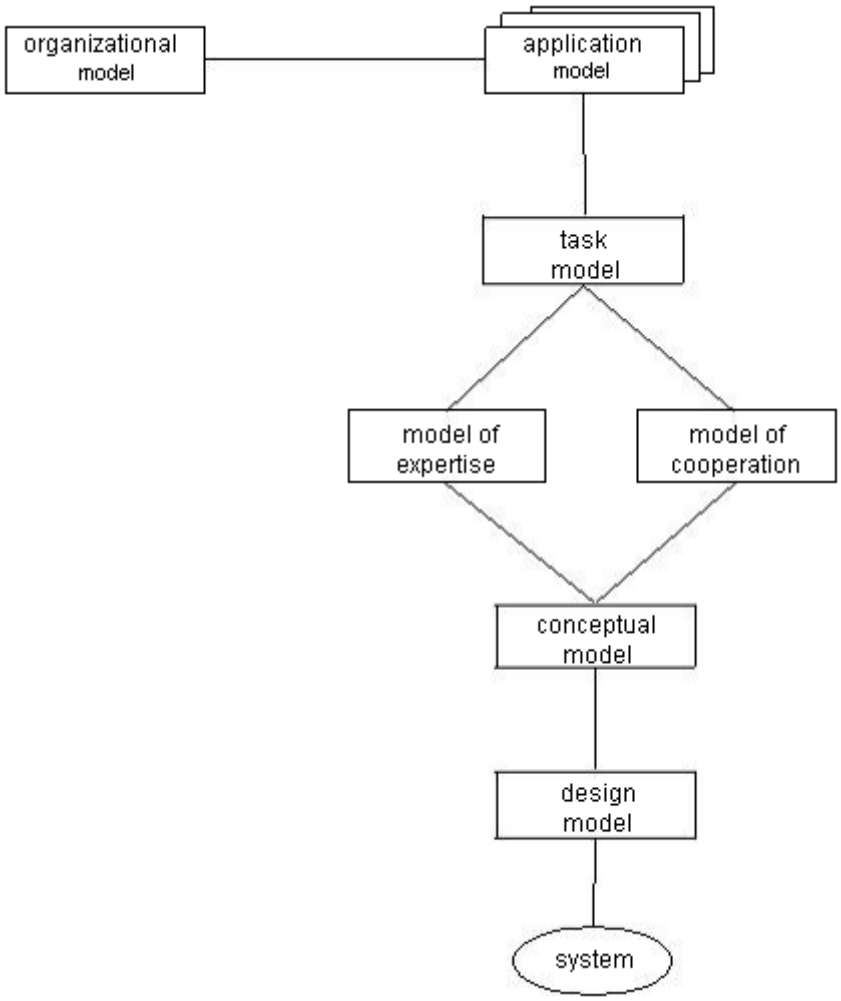


Figure 4-8 Principle 1: Models Provide a Decomposition of the Knowledge-engineering Task

The nature of knowledge engineering thus becomes a process that bridges the gap between required behavior, and a system that exhibits that behavior through the creation of a set of models. In summary, we can say that the KADS modeling view of knowledge acquisition gives rise to a methodology that involves the construction of a variety of models in the course of the knowledge engineering process. Each model represents a particular view of the KBS. They allow the knowledge engineer to cope with the complexity of the knowledge engineering process through a "divide & conquer"strategy.

The remainder of this chapter mainly focuses on the expertise model, since it plays such a central role in KBS development.

4.6.14 Principle 2: Modeling Expertise

The major challenge for any modeling approach to KBS construction is to find an adequate answer to the question of “how to model” expertise. It is this aspect of the system that

distinguishes KBS development from conventional systems development As discussed above, we require of the resulting expertise model that it is independent of a particular implementation. A modeling expertise framework is outlined in this section. It is a slightly different version of this KADS approach to modeling expertise (usually called the "four-layer model").

Two basic premises underly the ideas presented herein. First, we assume that it is possible - and useful, to distinguish between several generic types of knowledge according to the different *roles* that knowledge can play in reasoning processes. Second, we assume that these types of knowledge can be organised in several *layers* - which have only limited interaction. A first distinction that is often made is the one between *domain knowledge* and *control knowledge*. Here, we will take such the separation of knowledge into two layers one step further and will argue for a refined distinction of different types of control knowledge at three levels.

The categories in which the expertise knowledge can be analyzed and described are based on epistemological distinctions: they contain different types of knowledge. We distinguish between:

1. Static Knowledge - describing a declarative *theory* of the application domain; (i.e. Domain Knowledge)
2. Knowledge of different *types of inferences* that can be made in this theory; (the first type of Control Knowledge)
3. Knowledge representing *elementary tasks*; (the second type of Control Knowledge)
4. *Strategic Knowledge*; (the third type of Control Kknowledge)

Each of these knowledge categories is described at a separate level; the separation reflects different ways in which the knowledge can be viewed and used. In the following sections, each of the four knowledge categories distinguished in KADS is discussed in more detail.

The distinction between different types of knowledge is not new. Several authors have reported ideas which pertain to the separation of Domain and Control Kknowledge, and have proposed ways of increasing the flexibility of Control in Expert Systems.

4.6.15 Domain Knowledge for a Particular Application

Domain Knowledge embodies the *conceptualisation* of a domain for a particular application in the form of a *domain theory*. The primitives that we use to describe a domain theory are

based upon the epistemological primitives proposed by concepts, properties, two types of relations, and structures:

The Concept: *Concepts* are the central objects in Domain Knowledge. A concept is identified by means of its name (e.g. amplifier)

The Property/Value: Concepts can have properties. Properties are defined by their name and a description of the values that the property can take. For example: an *amplifier* has a property *power* with values *on/off* as possiblities.

The Relation between Concepts: A first-type relation is the relation between concepts, for example: an *amplifier is a component*. The most common relations of this type are the "sub-class" relation and the "part-of" relation. Several variants of these two relations exist - each with its own semantics.

The Relation between Property Expressions: A second-type relation is the relation between expressions about property values. An expression is a statement about the value(s) of a property of a concept, e.g. **amplifier: power = on**. Examples of this type of relation are causal relations and time relations. An example of a type of causal relation in the audio domain could be: **amplifier: power-button = pressed CAUSES amplifier: power = on**

The Structure: A structure is used to represent a complex object: an object consisting of a number of objects/concepts and relations. For example, the audio-system as a whole can be viewed as a structure - consisting of several components and relations (part-of, wire connections) between these components.

The choice of this set of primitives is in a sense arbitrary, and probably somewhat biased by the types of problems that have been tackled with KADS. The problem is to find a sub-set that provides the knowledge-engineer with sufficient expressive power. One could consider including additional special-purpose primitives - like mathematical formulae.
Primitives are used to specify what we call a *domain schema* for a particular application. A domain schema is a description of the *structure* of the statements in the domain theory. It is roughly comparable to the notion of a signature in logic. For example, in a domain schema, we could specify that the domain theory contains **part-of** relations between **component** concepts without worrying about the actual types of this relation. We prefer to use the term "schema" rather than "ontology" to stress the fact that Domain Theory is the product of

Knowledge Engineering and thus, does not necessarily describe an inherent structure in the domain - (as the word "ontology" would suggest).

The Domain Schema specifies the main decisions that the knowledge engineer makes with respect to the conceptualisation of the domain. For example, when a domain schema for a diagnostic domain is constructed, a decision has to be made whether "correct" or "fault" models (or both) are part of the domain theory. Parts of a domain schema often reappear in similar domains - and can be reused (see Sec. 4.16.20 for a more detailed discussion of reusability). The domain schema also provides convenient handles for describing the way in which Inference Knowledge uses domain theory. Issues related to the interaction between Domain Knowledge and Inference Knowledge are discussed in the next section. A language for describing domain schemata is presented in Ch. 6.

An example domain schema of a simple domain theory for diagnosing faults in an audio-system is shown in Table 4-1. Two types of concepts appear in this theory: *components* and *tests*. Both components and tests can have properties: respectively, a *state-value* and a *value*. Two relations are defined between concepts of a "component" type: *"is-a"* and *"sub-component-of"*. In addition, two relations between property expressions are defined: (i) A *causal* relation between the state values of components; while, (ii) *"an"* indicates the relation between test values and state values.

Fig. 4-9 shows some domain knowledge in the audio domain. The domain knowledge description follows the structure defined in the domain schema in Tab. 4-1.

Domain Knowledge can be viewed as a declarative theory of the domain. In fact, adding a simple deductive capability would enable a system in theory - (but, given the limitations of theorem-proving techniques, not in practice), to solve all problems solvable by the theory. The domain knowledge is considered to be relatively "task neutral", i.e. represented in a form that is independent of its use by particular problem-solving actions. There is ample evidence that experts are able to use their domain knowledge in a variety of ways, e.g. for problem-solving, explanations, teaching etc. Separating Domain Knowledge embodying the theory of the domain from its use in a problem solving process is the first step towards the flexible use and reusability of Domain Knowledge.

4.6.16 Inference Knowledge from the Domain Theory

At the first Control Knowledge layer, we abstract from the domain theory and describe the inferences that we want to make in this theory. We call this layer the *"Inference Layer"*. An inference specified at the inference level is assumed to be *primitive* - in the sense that it is

fully defined through its name, an input/output specification, and a reference to the Domain Knowledge it uses.

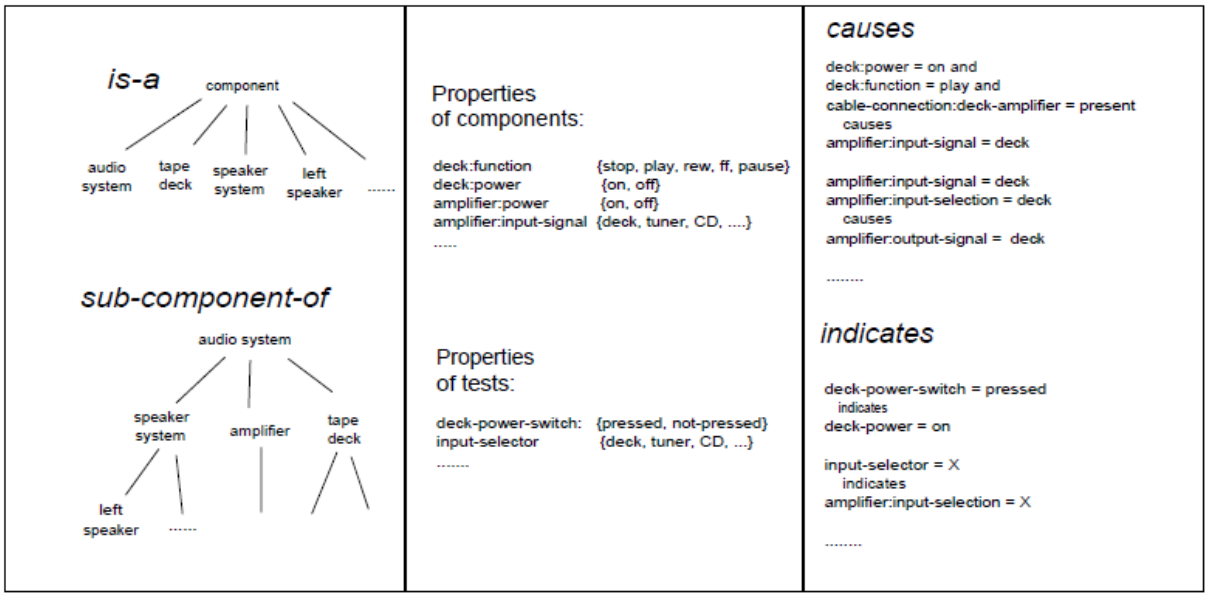


Figure 4-9 The Domain Knowledge of an Audio-system Using the Described Schema

The actual way in which the inference step is carried out is assumed to be irrelevant for the purposes of modeling expertise. From the viewpoint of the expertise model, no control can be exercised on the internal behaviour of the inference. One could look upon the inference as applying a simple theorem prover.

Note that the inference is only assumed to be primitive with respect to the expertise model. It is very well possible that such a primitive inference is realised in the actual system through a complex computational technique.

Primitive	Name	Description
Concept	component	The elements of the audio system
Relation between concepts	component IS-A component	Sub-type hierarchy of components of the audio system
Relation between concepts	component SUB-COMPONENT-OF component	Part-of hierarchy of components of the audio system
Property	component:state-value	Components have properties describing the state that components are in at some moment in time.
Relation between expressions	component:state-value CAUSES component:state-value	Causal relations that specify how normal state-values of components are causally related to each other.
Concept	test	Test that can be performed to establish a state of an audio system.
Property	test:value	Possible outcomes of a test.
Relation between expressions	test:value INDICATES component:state-value	A relation describing which internal state is indicated by a particular test outcome.

Table 4-1: A Domain Schema for Diagnosing Faults in an Audio-system

In the KADS expertise model, we use the following terms to denote the various aspects of a primitive inference:

The Knowledge Source: The entity that carries out an action in a primitive inference step is called a *Knowledge Source*. A knowledge source performs an action that operates on some input data - and has the capability of producing a new piece of information ("knowledge") as its output. It uses Domain Knowledge in this process. The name of the knowledge source is supposed to be indicative of the type of action that it carries out.

Meta-class: A knowledge source that operates on data elements and produces a new data element. We describe those data elements as *Meta-classes*. A meta-class description serves a dual purpose:

(i) It acts as a *placeholder* for domain objects, describing the *role* that these objects play in the problem solving process; and …

(ii) It points to the *type(s)* of the domain objects that can play this role

Domain objects can be linked to more that one meta-class. For example, a particular audio-system component could play the role of a *hypothesis* at one point in time - and the role of a solution at some other point. The name "meta-class" is inspired by the fact that it provides a "meta" description of objects in a domain "class". An input data element of a knowledge

source is referred to as an *input* meta-class; the output as an *output* meta-class. Each meta-class can be the input and/or output of more than one knowledge source.

The Domain View: The *domain* view specifies how particular parts of the domain theory can be used as a "body of knowledge" by the knowledge source.

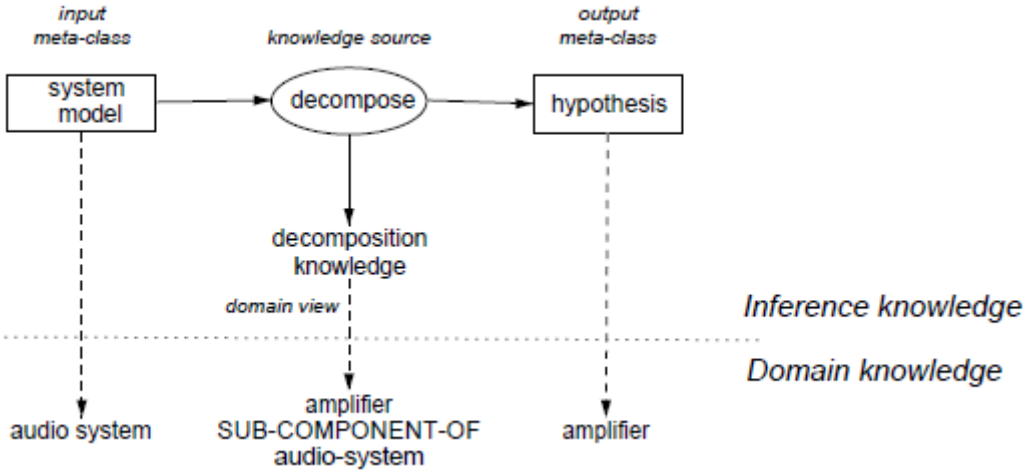


Figure 4-10: A Primitive Inference Performing a Decomposition Action

Fig. 4-10 describes a primitive inference in the audio domain with example references to domain knowledge. A *decomposition* inference is specified at the inference level. The action that is performed in this inference is the decomposition of a composite model of the audio-system into sub-models. The *System model* and *hypothesis* are examples of meta-classes. They describe the role that domain objects like **audio-system** and **amplifier** can play in the problem-solving process. The *decompose* knowledge source achieves its goal - the generation of a new hypothesis, through the application of decomposition knowledge. The domain view of this inference specifies that types of the **SUB-COMPONENT-OF** relation in the domain theory can be used as decomposition knowledge. Fig. 4-10 shows one applicable type of this relation.

A somewhat more formal specification of the decompose inference is given below. The arrow specifies how Inference Knowledge maps onto Domain Knowledge.

```
knowledge-source decompose
  input-meta-class:
      system-model → component
  output-meta-class:
      hypothesis → component
  domain-view:
      decomposition(system-model, hypothesis) → sub-component-of(component, component)
```

Note that this specification only refers to elements of the domain theory schema. Both *system model* and *hypothesis* are place-holders of objects of the "component" type and describe the

role these objects play in the inference process. In this particular example, the domain view refers to just one type of knowledge in the domain theory - namely the **SUB-COMPONENT-OF** relation. In principle however, there could be several of these mappings.

There are distinct advantages to separating the domain theory from the way it is viewed, and used by, the inferences:

- The separation allows multiple use of essentially the same domain knowledge. Imagine for example, a knowledge source *aggregate*, that takes as its input a set of components and aggregates them into one composite component. This knowledge source could use the same **SUB-COMPONENT-OF** relation, but view it differently - namely as Aggregation Knowledge. Such an inference could very well occur in a system that performs audio-system congurations
- Domain knowledge that is used in more that one inference is specified only once - thereby, knowledge redundancy is prevented
- It provides a dual way to *name* domain knowledge: both user-independent and user-specific. Knowledge engineers tend to give domain knowledge elements names that already reflect their intended use in inferencing and keep changing the names when their usage changes. We would argue that both types of names can be useful and should be known to the system - for example, for explanation purposes.
- The scope of the domain theory is often broader than what is required for problem-solving. For example, explanatory tasks (defined in KADS as the cooperation model), often require deeper knowledge than is used during the reasoning process itself.

This is not to say that we claim that a domain theory can - in general, be defined completely independent of its use in the problem-solving process. The scope and structure of the domain knowledge has to meet the requirements posed by the total set of inferences. In many applications, there are interactions between the process of conceptualising a domain and specifying the problem-solving process. We are convinced however, that it is useful to *document* them separately at least.

As stated previously, the primitive inference steps form the building blocks for an application problem-solver. They define the basic inference actions that the system can perform and the roles the domain objects can play. The combined set of primitive inferences specifies the basic inference capability of the target system. The set of inference steps can be graphically

represented in an *inference structure*. The inference structure thus specifies the problem-solving *competence* of the target system.

Fig. 4-11 presents such an inference structure for the audio domain. The inferences specify a top-down and systematic approach to a sub-model of an audio system that is behaving inconsistently. The following inferences appear in the inference structure:

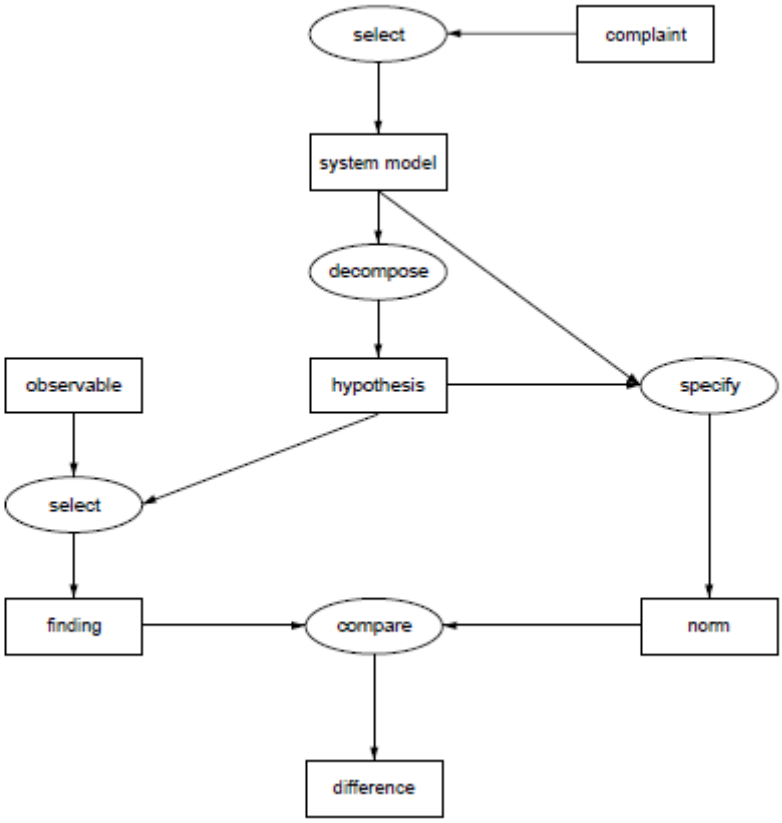


Figure 4-11 An Inference Structure for Diagnosing Faults in an Audio-system - Rectangles represent meta-classes; while ovals represent knowledge sources. Arrows are used to indicate input-output dependencies.

- A selection of a (sub-part) of the audio-system (*system model*) on the basis of a *complaint*
- A decomposition of some part of the system into a number of sub-components that play the role of *hypotheses*
- A prediction of a *norm*-value for a hypothesis. The norm is a value of a test that is consistent with the normal state of the hypothesis
- A specification of an observable element for which a value is to be obtained (the *finding*)
- A comparison of the observed *finding* and the predicted *norm*

The Inference Structure defines the vocabulary and dependencies for control - but not the control itself. This latter type of knowledge is specified as Task Knowledge.

4.6.17 Task Knowledge

The third category includes knowledge about how elementary inferences can be combined to achieve a certain goal. The prime knowledge type in this category is the *task*. Tasks can achieve a particular *goal*. The relations between tasks and goals are - in principle, many-to-many. Task Knowledge is usually characterized by a vocabulary of control terms – e.g., indicating that a finding has been processed or a hypothesis has been verified.

Tasks represent fixed strategies for achieving problem-solving goals. Several researchers have pointed out that Task Knowledge is an important element of expertise. The Competence Model of the diagnostic strategy of **Neomycin** is an example of what we call task knowledge. Clancey describes the sub-tasks of this strategy by using meta-rules. The main difference between his approach and our approach is that he refers directly in these meta-rules to the Domain Knowledge. In KADS, tasks only refer to inferences - and not explicitly to Domain Knowledge.

We use the following constructs to describe Task Knowledge:

Task: A task is a composite problem-solving action. It implies decomposition into sub-tasks. The application of the task to a particular (sub)-problem results in the achievement of a goal.

Control Terms: This is the control vocabulary used. A control term is nothing more than a convenient label for a set of meta-class elements. The label represents a term used in the control of problem-solving, e.g. "differential" or "focus". Each control term is defined through the specification of a mapping of this term onto sets of meta-class elements (e.g. the differential is the set of all active hypotheses).

Task Structure: Tasks are decomposed into sub-tasks and the control dependencies between these sub-tasks are specified. Decomposition can involve three types of sub-tasks:

1. Primitive problem-solving tasks: inferences specified in the inference layer
2. Composite problem-solving tasks: a task specified in the task layer. In principle, this could be a recursive invocation of the same task
3. Transfer tasks: tasks that require interaction with an external agent - usually the user

The dependencies between the sub-tasks are described as a structured-English procedure such as is used in conventional software engineering, with selection and iteration operators.

The conditions in these procedures always refer to control terms and/or meta-class elements, e.g. "**if** *the differential is not empty* **then**…"

There is interaction between the task knowledge in the expertise model on the one hand and the cooperation model on the other, with respect to the specification of the transfer tasks. Transfer tasks are more-or-less specified as a black box in the expertise model. We may distinguish four types of transfer tasks:

1. *Obtain*: The system requests a piece of information from an external agent. The system has the initiative
2. *Present*: The system presents a piece of information to an external agent. The system has the initiative
3. *Receive*: The system gets a piece of information from an external agent. The external agent has the initiative
4. *Provide*: The system provides an external agent with a piece of information. The external agent has the initiative

An example of task-knowledge specification for our audio domain is shown below. It consists of three tasks. The first task is *systematic-diagnosis*. The goal is to find a sub-system with inconsistent behaviour at the lowest aggregation level. The task works under the single-fault assumption. An applicable system model is selected on the basis of a complaint. This selection task corresponds to the “knowledge source select” specified in the inference layer. Subsequently, hypotheses in the differential are generated by means of the *generate-hypotheses* sub-task. In the *test-hypotheses* sub-task, these hypotheses are then tested to find an inconsistent sub-system. This hypothesis then becomes the focus for further exploration. The generate-and-test process is repeated until no new hypotheses are generated (i.e. the differential is empty).

```
task systematic-diagnosis
  goal:
      find the smallest component with inconsistent behaviour, if one.
  input:
      complaint
  output:
      inconsistent-sub-system: sub-part of the system with
        inconsistent behaviour
  control-terms:
      differential: set of currently active hypotheses
  task-structure:
      systematic-diagnosis(complaint → inconsistent-sub-system) =
        select(complaint → system-model)
        generate-hypotheses(system-model → differential)
        REPEAT
          test-hypotheses(differential → inconsistent-sub-system)
          generate-hypotheses(inconsistent-sub-system → differential)
        UNTIL differential = ∅
```

For readability purposes, the names of knowledge-sources are italicised in the task structure. The arrows in the task structure describe the relation between input and output of the sub-task. Note that all arguments of tasks and conditions are either explicitly declared (*differential*) or are meta-class names.

The *generate-hypotheses* task is very simple. It just executes the *decompose* knowledge source, until it produces no more solutions.

```
task generate-hypotheses
  goal:
      generate new set of hypotheses through decomposition
  input:
      system model
  output:
      hypothesis-set: set of newly generated hypotheses
  control-terms:
      hypothesis: device component

task-structure:
    generate(system-model → hypothesis-set) =
      REPEAT
        decompose(system-model → hypothesis)
        hypothesis-set := hypothesis ∪ hypothesis-set
      UNTIL no more solutions of decompose
```

The *test-hypotheses* task tests the hypotheses in the differential sequentially until an inconsistency is found, (*difference* = *true*). Testing is done through a kind of experimental validation: a norm-value is predicted, and this value is compared with what is actually

observed. *Obtain (observable, finding)* is an example of a transfer task that starts an interaction with the user to obtain a test value. How the transfer task is carried out, should be specified in the Cooperation Model.

```
task test-hypotheses
  goal:
      test whether a hypothesis in the differential behaves inconsistently
  input:
      differential
  output:
      hypothesis: element of the differential with inconsistent behaviour
  control-terms: -
  task-structure:
      test(differential → hypothesis) =
        DO FOR EACH hypothesis ∈ differential
          specify(hypothesis → norm)
          specify(hypothesis → observable)
          obtain(observable → finding)
          compare(norm + finding → difference)
        UNTIL difference = true
```

If one abstracts from the control relations between sub-tasks - and assumes a fixed task decomposition - the set of task structures can be graphically represented as a tree. The tree for systematic diagnosis is shown in Fig. 4-12. Such a decomposition of a task assigned to the system is - in fact, a further refinement of the decomposition specified in the task model (see Sec. 4.6.8).

4.6.18 Strategic Knowledge

The fourth category of knowledge is Strategic Knowledge. Strategic Knowledge determines what goals are relevant to solve a particular problem. How each goal is achieved is determined by the *knowledge* task. Strategic Knowledge will also have to deal with situations where the knowledge categories mentioned above fail to produce a partial solution. For example, the problem-solving process may reach an impasse because information is not available - or because contradictory information arises. In such cases, the strategic reasoning process should suggest new lines of approach or attempt to introduce new information e.g., through assumptions.

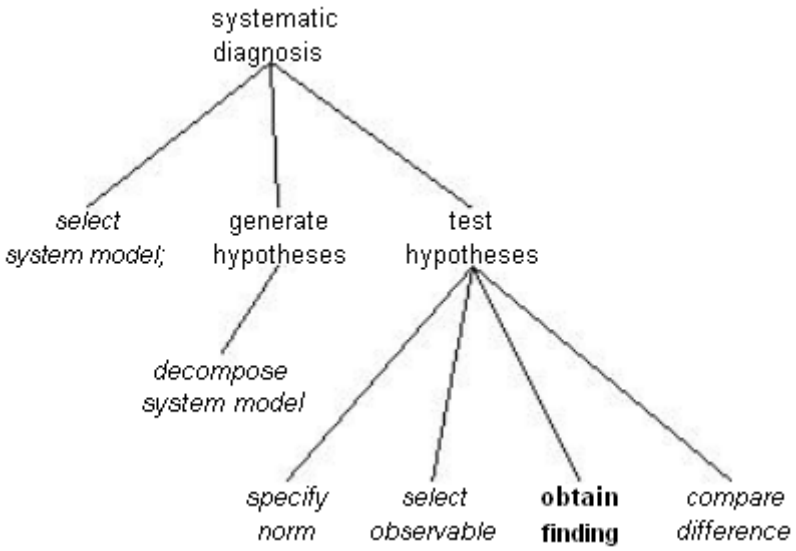


Figure 4-12: Systematic Diagnosis Task-tree. The leaves of such a tree are either knowledge sources or transfer tasks.

Strategic Knowledge concerns - among other things, the dynamic planning of task execution. However, most systems developed with the KADS approach used only fixed task decompositions, and had little or no Strategic Knowledge. In our opinion, this does not mean that Strategic Knowledge is unimportant - or superfluous. When knowledge engineers have to construct more complex and flexible knowledge-based systems than is presently usually the case; we think a much more detailed exploration of Strategic Knowledge will be necessary. We have recently started to work on an ESPRIT project called REFLECT where the central topic is the exploration of Strategic Knowledge. Apart from dynamic planning, strategic knowledge can also enable a system to answer questions like "Can I solve this problem?" For the moment however, the study of the nature of Strategic Knowledge remains to be mainly a research topic.

4.6.19 Synopsis of the Expertise Model

The four knowledge categories (Domain, Inference, Task and Strategic Knowledge), can be viewed as four levels with meta-like relations in the sense that each successive level interprets the description at the lower level. In Fig. 4-13, these four levels and their interrelations are summarised.

The four-layer framework is a structured, but informal framework. This means that the specifications are sometimes not as precise as one might want them to be - and thus, may be

interpreted in more than one way. This has led to research aimed at designing a formal framework for representing expertise models. The price paid for this greater amount of precision in formal specifications is, however, a reduction in conceptual clarity. In our view, there is a place for both informal and formal representations in the knowledge engineering process. The use of both informal and formal model representations is a major topic of research in the KADS-II project.

The four-layer framework used for knowledge modeling has been successfully used as a basis for the structured acquisition and description of knowledge at an intermediate level between the expertise data obtained from experts, text-books, etc., and the knowledge representation in an implemented system. From a knowledge-level view-point, the present four-layer model captures knowledge categories that are quite similar to those encountered in other models in the literature. However, differences in opinion exist about where to situate particular types of knowledge.

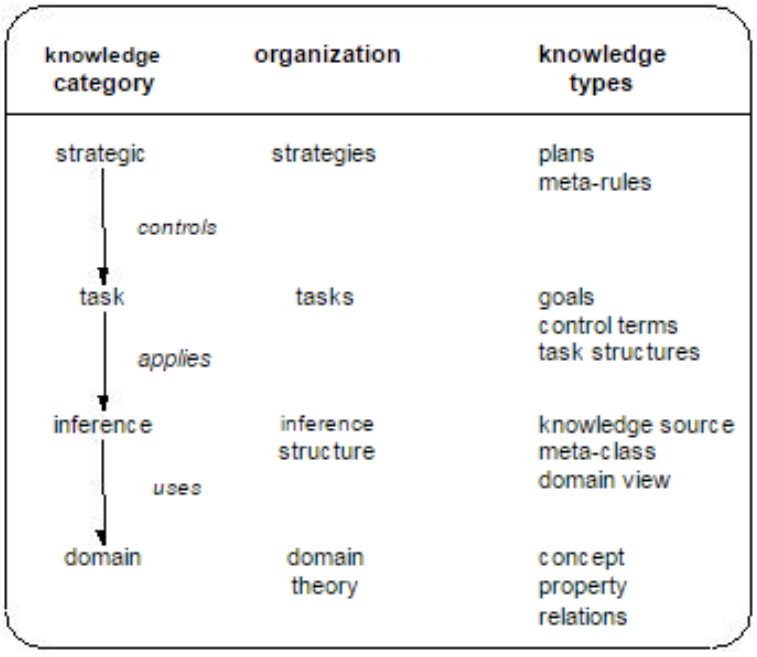


Figure 4-13 Synopsi of the KADS Four-Layer Model.

4.6.20 Principle 3: Reusable Model Elements

There are several ways in which expertise models can be used to support the knowledge acquisition process. A potentially powerful approach is to reuse (structures of) model elements. When modeling a particular application, it is usually already intuitively clear that

large parts of the model are not specific for this application - but reoccur in other domains and/or tasks. KADS (as with most other knowledge modeling approaches) makes use of this observation by providing a knowledge engineer with predefined sets of model elements. These libraries can be of great help to a knowledge engineer. They provide them with ready-made building blocks and prevent them from "re-inventing the wheel" each time a new system has to be built. In fact, we believe that these libraries are a *conditio sine qua non* for improving the “state-of-the-art” in Knowledge Engineering.

In this section, two ways of reusing elements of the expertise model are discussed: (i) *typologies* of primitive inference actions (knowledge sources); and, (ii) *interpretation models*. In principle however, the reusability principle holds for all models in the KBS construction process.

4.6.21 Knowledge Sources Typologies

We have defined a tentative typology of primitive problem-solving actions (knowledge sources) which has formed the basis of a considerable number of models. The typology is based on the possible operations one can perform on the epistemological primitives defined in KL-ONE. This set of primitives consists of:

- Concept
- Attribute (of concept)
- Value (of attribute)
- Instance (of concept)
- Set (of concepts)
- Structure (of concepts)

In the typology of inferences, we view these primitives not as data-structures - but as epistemological categories. Their actual representation in a system may be quite different (e.g. in terms of logical predicates rather than KL-ONE-like constructs).

Operation type	Knowledge source	Arguments
Generate concept/instance	instantiate classify generalise abstract specify select	concept ! instance instance ! concept set of instances ! concept concept ! concept concept ! concept set ! concept
Change concept	assign-value compute	attribute ! attribute-value structure ! attribute-value
Dierentiating values/structures	compare match	value + value ! value structure + structure ! structure
Structure manipulation	assemble decompose transform	set of instances ! structure structure ! set of instances structure ! structure

Table 4-2 A Typology of Knowledge Sources

Table 4-2 gives an overview of the typology of knowledge-sources used in KADS. The inferences are grouped on the basis of the type of operation that is carried out by the knowledge source: *generate concept/instance, change concept, differentiate values/structures* and *manipulate structures*. A detailed description of the inferences mentioned in Table 4-2 is given.

Although this typology has been a useful aid in many analyses of “expertise”, it has a number of important limitations:

- The selected set in Table 4-2 is - in a sense, arbitrary. For example, we could have added other operations on sets, e.g. *join, union, or merge*

- The ontology on which the typology is based is of a very general nature - and hence, weak. The operations are defined more-or-less independent of the tasks and/or domains. Often, it is difficult for a knowledge engineer to identify how an inference in a particular application task must be interpreted

- A more serious limitation is that some inferences cannot be adequately classified because they require another ontological framework. For example, operations on causal relations such as *abduction* and *differentiation* cannot be represented in a natural way.

We consider the study of more adequate taxonomies of inferences to be a major research issue. Potentially, taxonomies are very powerful aids for a knowledge engineer. In a new

research project, (KADS-II), we are exploring the possibility of describing taxonomies that are specific for classes of application domains - such as technical diagnosis, for instance. These taxonomies will be based on a much more task-specific ontology.

It is interesting to see that, from a different angle, the "Fireghter" project is aiming at similar results. An important goal of this project is to look at what they call *mechanisms* that are used in various applications, to detect commonalities between these mechanisms, and to construct a library of mechanisms that can be reused in other applications. These mechanisms appear to have the same grain-size as the knowledge-sources in KADS. The main difference is that *mechanisms* are more computational in nature.

4.6.22 Iterpretation Models

Typologies of elements of an expertise model - like a typology of knowledge-sourcesfoor example, represent a first step into the direction of reusability. A further step would be to supply *partial models* of expertise, e.g. models without all of the detailed domain knowledge filled in. Such partial models can be used by a knowledge engineer as a template for a new domain and thus support top-down knowledge acquisition. In KADS, such models are called *interpretation models* because they guide the interpretation of verbal data obtained from the expert.

The KADS interpretation models are expertise models, with an empty domain layer. Interpretation models describe typical inference-knowledge and task-knowledge for a particular task. As these descriptions are phrased in domain-independent terminology, they are prime candidates for reuse in other domains. For example, the inference and task description of the audio domain could very well be applied to another domain where some device is being diagnosed. A large number of tasks are present in interpretation models. One of these is the model for systematic diagnosis that is presented here.

An Example of an Interpretation Model: The monitoring task is another model in this library. This model has been used in applications ranging from Process-Control to Software Project Management. It is also interesting since it illustrates how different tasks can apply the same set of inferences in different ways.

The inference structure of the monitoring task interpretation model, (shown in Fig. 4-14), depicts the following inferences:

- The selection of a system parameter
- The instantiation of the normal value of the parameter (the norm)
- The selection of a corresponding observable
- A comparison of observed and expected values leading to a difference description
- A classication of the difference into a discrepancy class – for example, a minor or major disturbance. Often, data from previous monitoring cycles is used in this inference.

Two typical tasks (fixed strategies) were identified for *monitoring*. One could view them as two different ways of "going through" the inference structure of Fig. 4-14.

The first, the, *model driven monitoring* task, describes a monitoring approach where the system has the initiative. This type of task is usually executed at regular points in time. The system actively acquires new data for some selected set of parameters, and then checks whether the observed values differ from the expected ones.

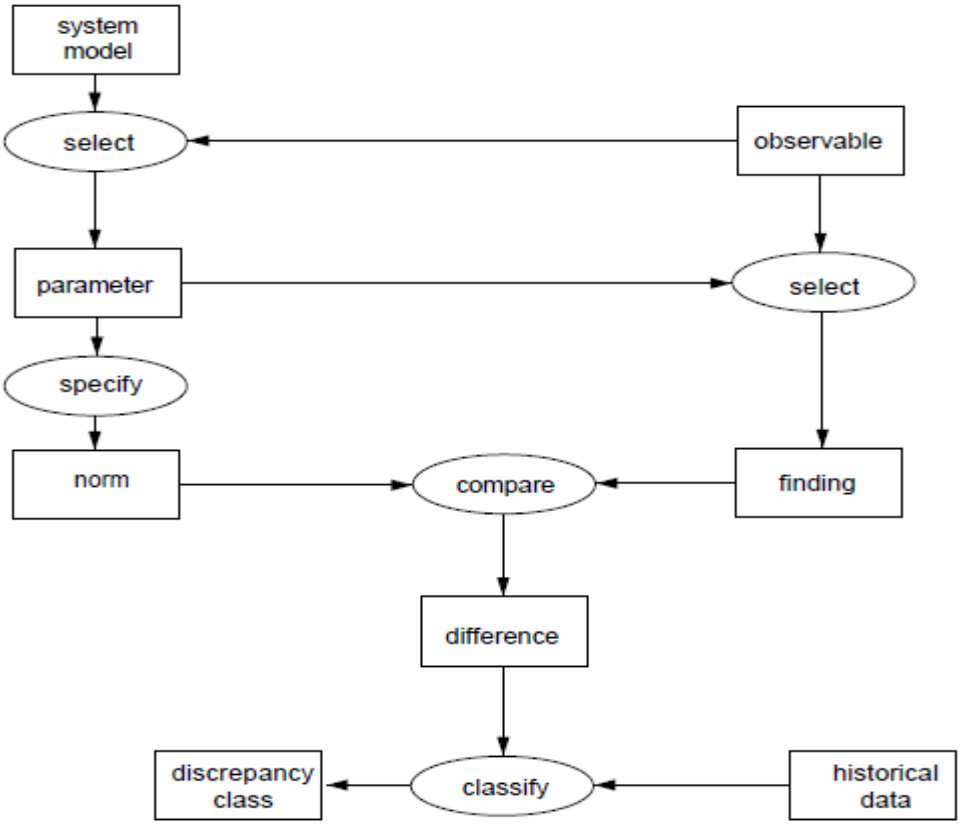


Figure 4-14 Inference Structure of the Interpretation Model for *monitoring*

```
task model-driven monitoring
  goal:
      execute a monitoring cycle in which the system actively acquires new data
  input: -
  output:
      discrepancy
  control-terms:
      active-parameters:set of parameters
  task-structure:
      monitor(discrepancy) =
        select(system-model, active-parameters)
        DO FOR EACH parameter ∈ active-parameters
          specify(parameter → norm)
          select(parameter → observable)
          obtain(observable → finding)
          compare(norm + finding → difference)
          classify(difference + historical-data → discrepancy)
```

The second task - *data-driven monitoring*, is initiated by incoming data. It contains a *receive statement* representing a transfer task in which an external agent (a human user or another system) has the initiative. The values received are checked against the expected values for the observables concerned. The resulting differences are subsequently classified in discrepancy classes.

```
goal:
    execute a monitoring cycle when a new value of an observable
    is received by the system
input: -
output:
    discrepancy
task-structure:
    monitor(discrepancy) =
      receive(observable-set → finding)
      DO FOR EACH observable ∈ observable-set
        select(observable + system-model → parameter)
        specify(parameter → norm)
        compare(norm + finding → difference)
        classify(difference + historical-data → discrepancy)
```

Selecting an Interpretation Model from the Library: The interpretation model library consists of a number of models that can be used to describe the reasoning process in various applications. A knowledge engineer is guided in deciding which interpretation model to choose for a particular application by a decision tree. Part of this tree is shown in Fig. 4-15. Index Interpretation Model Library, Model Selection.

The decision-tree is based on the taxonomy of task types. This taxonomy is a modified and extended version of the Clancey's description of problem types. The decision points in this tree concern features of the solution-space, the problem-space and the required domain knowledge types.

The first decision point concerns the availability of information about the structure of the system involved in a task. The term "system" refers here, to the central entity in the application domain, e.g. an audio-system in the audio domain; a patient in the medical domain; a device in the technical domain, etc. Other decision-points concern - for example, the type of solution (e.g. state, category, types of categories, etc.), and the nature of the domain knowledge, (fault-model or correct-model of the system). The branches of the decision-tree are associated with one - or more, interpretation models that specify typical inference and task knowledge for modeling this task. For example, the interpretation model for monitoring - presented earlier above, is associated with the *monitoring* task in Fig. 4-15. This model is chosen if: (i) The structure of the system is given; (ii) The solution is a category; and, (iii) This solution category is not a fault category nor a decision class, but a simple discrepancy between the observed and expected behaviour.

It should be noted that in many real-life applications, the task is a compound one: it consists of several basic tasks. For example, in the expertise model for the audio domain, we focused only on the diagnostic sub-task. In actual practice, the repair/remedy task also needs to be addressed. This may result in a combination of (parts of) two or more interpretation models.

A number of researchers have developed knowledge-acquisition tools that are based on the notion of a generic model of the problem-solving task. This model prescribes what domain knowledge is needed to build an actual expert system. The conceptual model in OPAL is not just a model of the problem-solving process (i.e. the upper three layers in the KADS framework) but also contains templates of the domain knowledge needed. As a consequence, OPAL can present the expert with detailed forms that they can fill in with the details of an application domain. Although this approach is very powerful indeed, it has limitations in scope and applicability.

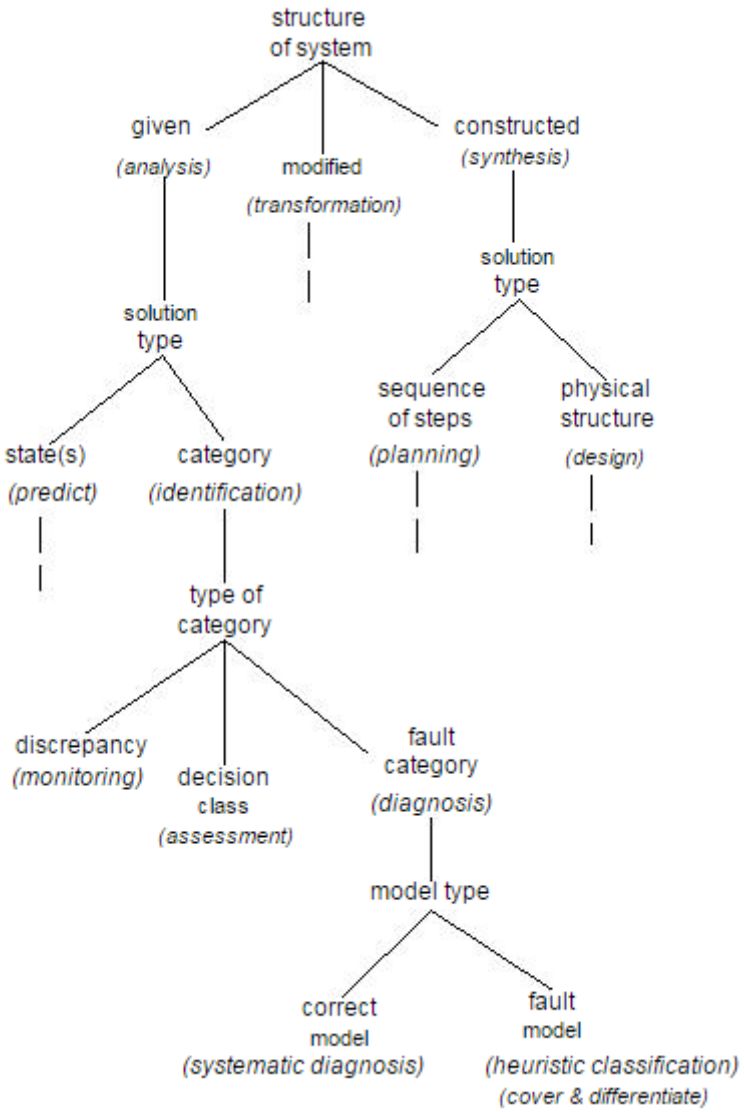


Figure 4-15: An Inference Structure of the Interpretation Model for *monitoring*

4.6.23 The Knowledge Acquisition Process

The description of the various models can be seen as the *product* of KBS´ construction, With respect to the *process* of KBS construction, KADS provides two ways of support:

(i) A description of phases, activities, and techniques for Knowledge Engineering; and, (ii) computerised support tools. Both are discussed briefly in this section.

4.6.24 Phase, Activities and Techniques

A *phase* represents a typical stage in the Knowledge Engineering process. A phase is related to a number of *activities* that are usually carried out in this phase. One particular activity can occur in more than one phase. For example, "data collection" can occur in many different phases. The activities are the central entities in the process view of Knowledge Engineering. An activity is a piece of work that has to be carried out by a Knowledge Engineer. An activity produces a result. This result either directly constitutes a part of one or more models, or it represents some intermediate product that is used by other activities. An activity applies one or more techniques. For example, a "time estimation" activity can be carried out with an extrapolation technique. Life-cycle models predefine particular phases, activities, techniques and products – as well as their interrelations. We limit the discussion here to those activities that are related to building a First Model of Expertise. We distinguish two phases in building such a model: *knowledge identication* and *knowledge modeling*

Knowledge Identication is more-or-less a preparation phase before the actual construction of the expertise model can begin. The relevant activities of this phase are shown in Fig. 4-16, together with applicable products and techniques. The results include a task model as well as intermediate products that are used by activities in other phases - especially, in the knowledge modelling phase. Glossary and lexicon construction are two such example activities. A glossary and a lexicon provide a way of documenting the application domain, without committing to any formal conceptualisation.

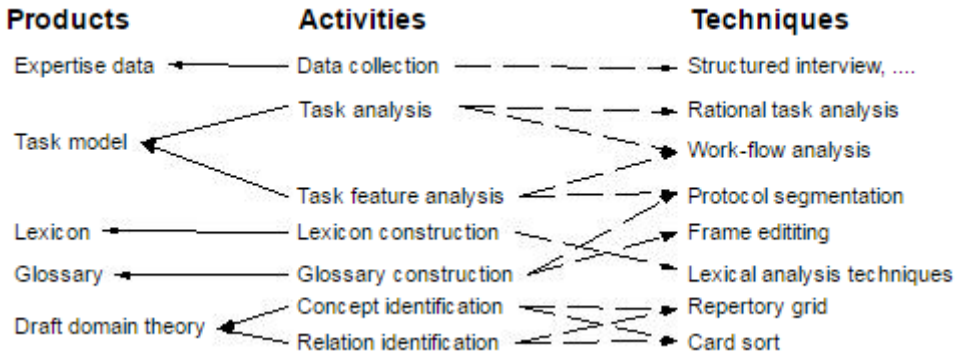


Figure 4-16 Knowledge Identication Activities and Related Products and Techniques

In the knowledge-modeling phase, a knowledge engineer constructs an expertise model. Fig. 4-16 summarises the main activities relevant for knowledge-modeling. The selection of an

interpretation model is crucial. This activity is supported through the decision-tree discussed in Sec. 4.6.20. The model-validation and model-differentiation activities often make use of Protocol Analysis techniques. Model validation can also be supported by transformation of the model into a functional prototype. This prototype can be seen as a simulator of the problem-solving aspects of the artefact. The KADS-II project is currently working on a tool to support this type of prototyping. Other activities deal with the definition of the Domain Conceptualisation. In KADS, we usually assume that - in the resulting model, the Domain Theory can be a partial one - but, with a fully-defined Domain Schema. Refinement and debugging of the domain theory is performed in a later phase, possibly with the use of automised techniques.

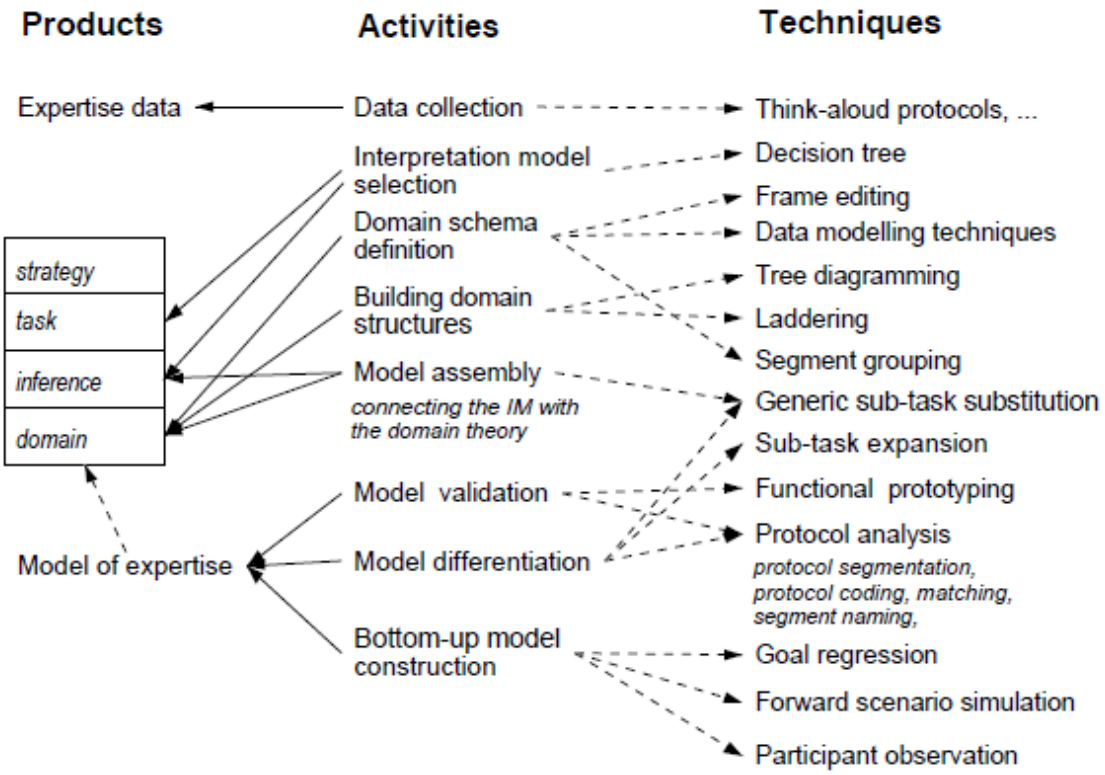


Figure 4-17 Knowledge Modeling Activities and Related Products and Techniques

4.6.25 Tools

The Shelley workbench was developed within the scope of the KADS project to support activities in the KBS life-cycle. Shelley contains an integrated set of computerised support tools. The user of the workbench is a knowledge engineer. Examples of support tools in Shelley for the knowledge-modeling phase are:

- *A Domain Text Editor* - Is a tool that allows the management and analysis of protocols or other texts; for example, through the creation of text-fragments of a particular type. These fragments can subsequently be linked to other objects, e.g. elements of the expertise model
- A *Concept Editor* - to create concepts and corresponding attributes
- An *Interpretation Model Library* - from which models can be selected
- An *Inference Structure Editor* - that supports the construction of the inference layer of the expertise model

Fig. 4.18 shows an example of the use of Shelley in the audio domain. The knowledge engineer has selected the Systematic Diagnosis Interpretation model from the library and inserted it into the Inference Structure editor. A "think-aloud" protocol is being analyzed.

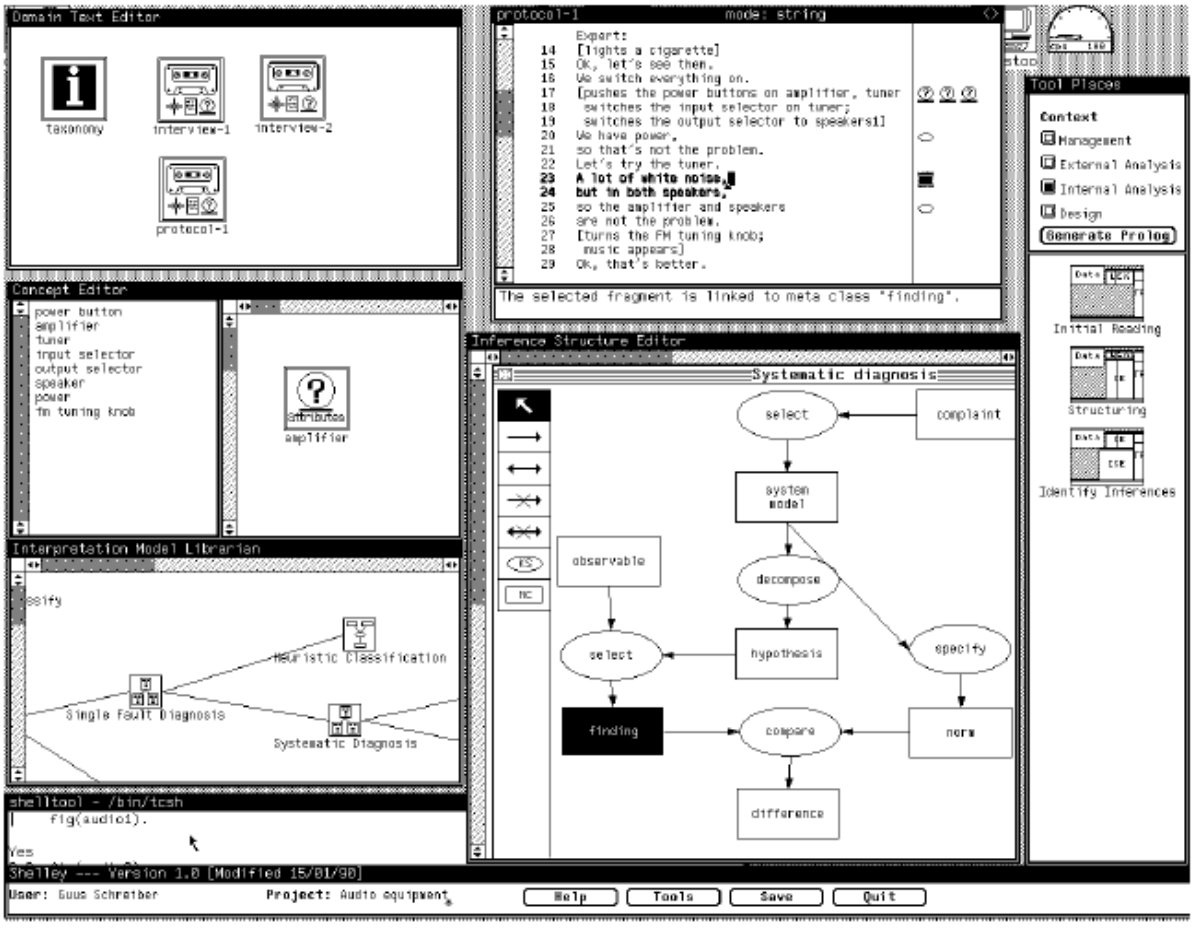


Figure 4-18 Example of a Shelley Workbench Session

The link of a particular fragment of the protocol to a meta-class in the Inference Structure editor is shown above.

4.7 A Case Study

Developing a KBS with Common KADS [48]

To illustrate the previous concepts, we chose to develop a small *literary assistant* application featuring an expert system. The basic idea of the assistant is to assign a particular book to a reader according their age, education, and interests. The application does not pretend to be a real-world system but could be a limited version of what it should be. Therefore, it could be used (on a large-scale) in a library or book-store, to advise readers which book they should read.

Domain Knowledge

This describes the concepts and concept-relationships involved in a domain (e.g. mechanic, medicine, etc.). The concept is the representative central entity in the domain knowledge. A concept is identified by its name - and refers to abstract entities (e.g. a patient); or specific entities (i.e. patient John). It is like the C++ or Java class concept. The concepts are described by their properties or attributes, and defined by their type and name. The property is the core of domain-knowledge representation. The domain knowledge is composed of the following three elements, they are:

4.7.1 A Class Diagram

We define the concepts and attributes using UMLfor example; although we could use other AI modeling languages like KIF, Ontolingua, etc.

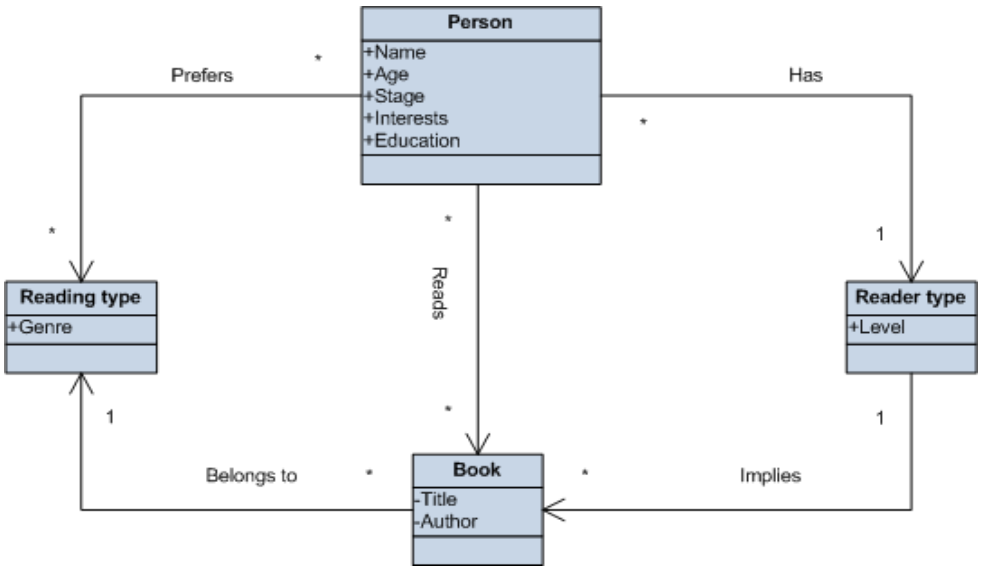


Fig. 4-19 A UML Model

After drawing the diagram, we will write it in CML (CommonKADS Conceptual Modeling Language).

```
Concept Person;
ATTRIBUTES:
Name: String;
Age: Int;
Stage: String;
Interests: String;
Education: String;
END CONCEPT Person;
```

```
CONCEPT Reading-type;
ATTRIBUTES:
Genre: String;
END CONCEPT;
```

```
CONCEPT Reader-
type;
ATTRIBUTES:
Level: String;
END CONCEPT;
```

```
CONCEPT Book;
ATTRIBUTES:
Title: String;
Author: String;
END CONCEPT;
```

Fig. 4-20 Conceptual Model Language

4.7.2 An Expression Relationship

These represent relations in an "*if...then*" rule form: a conditional expression in the antecedent and an assignment expression in the consequent. They suppose a cause-effect association. In the example, the relations are the following:

```
RULE-TYPE Abstraction-rules;
DESCRIPTION: Abstract the age of a person
ANTECEDENT: Person;

CARDINALITY: 1;
CONSEQUENT: Stage;
CARDINALITY: 1;
CONNECTION-SYMBOL: Abstracts;
END-RULE-TYPE Abstraction-rules;
```

Fig. 4-21 A Preference Sheme

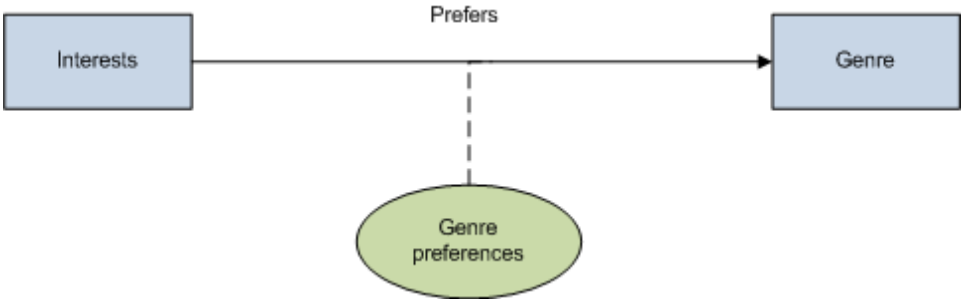


Fig. 4-22 A General Preference Scheme

```
RULE-TYPE Genre-preferences;
DESCRIPTION: Genre selection starting from the interests
ANTECEDENT: Interests;
CARDINALITY: 1;
CONSEQUENT: Genre;
CARDINALITY: *;
CONNECTION-SYMBOL: Prefers;
END-RULE-TYPE Genre-preferences;
```

Fig. 4-23 KADS Language

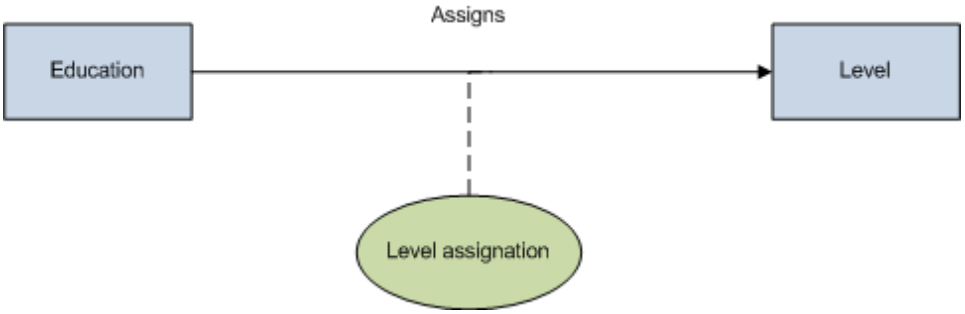


Fig. 4-24 Level Assignation

```
RULE-TYPE Level-assignation;
DESCRIPTION: Assigns a level according the education
ANTECEDENT: Education;
CARDINALITY: 1;
CONSEQUENT: Level;
CARDINALITY: 1;
CONNECTION-SYMBOL: Assigns;
END-RULE-TYPE Level-assignation;
```

Fig. 4-25 KADS Language

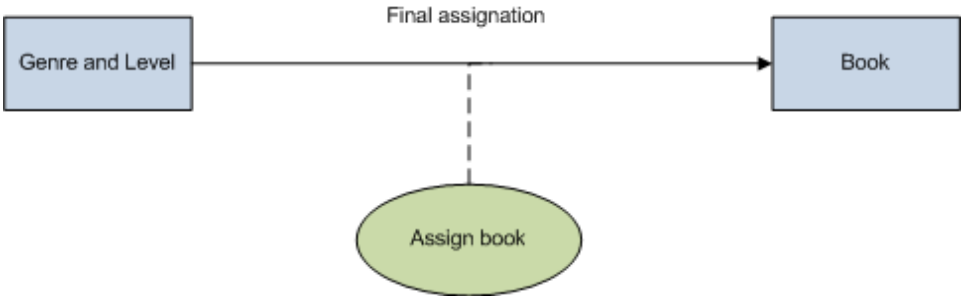
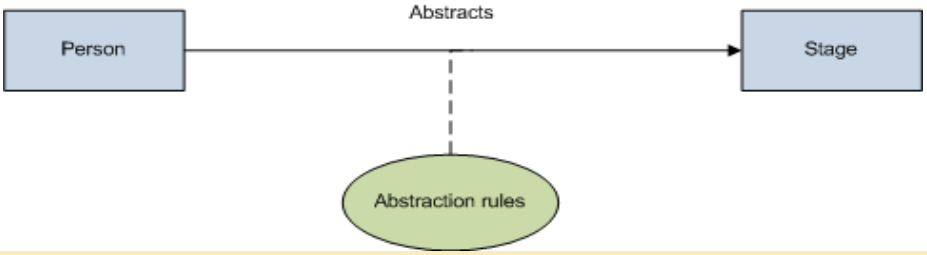


Fig. 4-26 Assignation Book

```
RULE-TYPE Assign-book;
```




```
DESCRIPTION: Assign books
ANTECEDENT: Genre and Level;
CARDINALITY: *;
CONSEQUENT: Book;
CARDINALITY: *;
CONNECTION-SYMBOL: Final assignation;
END-RULE-TYPE Assign-book;
```

Fig. 4-27 KADS Language

4.7.3 Knowledge Base

A knowledge-base acquires the pairs through different previously-defined generic rules. Shown below is a sample instance of the knowledge-base:

```
KNOWLEDGE-BASE Literary-assistant-base
USES
Abstraction-rules FROM literary-assistant-scheme;
Genre-preferences FROM literary-assistant-scheme;
Level-assignation FROM literary-assistant-scheme;
Assign-book FROM literary-assistant-scheme;

EXPRESSIONS

/* Select stage */

Person.age <= 13
ABSTRACT Person.stage = CHILD

Person.age > 13 AND Person.age <= 19
ABSTRACT Person.stage = TEENAGER

Person.age > 19 AND Person.age <= 40
ABSTRACT Person.stage = YOUNG

Person.age > 40
ABSTRACT Person.stage = ADULT

/* Discriminate interests */

Person.interests = "Music" OR Person.interests = "Painting" OR
Person.interests = "Cinema" OR Person.interests = "Sculpture"
PREFER reading-type.genre = ART
.
/* Adjust reader level */

Person.education = SECONDARY AND Person.stage = YOUNG
SELECT-LEVEL reader-type.level = ADVANCED
 Person.education = SECONDARY AND Person.stage = ADULT
SELECT-LEVEL reader-type.level = ADVANCED

Person.education = UNIVERSITARY
SELECT-LEVEL reader-type.level = ADVANCED
.
/* Science section */

reader-type.level = CHILD AND reading-type.genre = SCIENCE
SELECT-BOOK Book.title = "Arithmetics handouts"

reader-type.level = BASIC AND reading-type.genre = SCIENCE
SELECT-BOOK Book.title = "Basic arithmetics and geometry "

reader-type.level = INTERMEDIATE AND reading-type.genre = SCIENCE
SELECT-BOOK Book.title = "Derivatives and integrals"

reader-type.level = ADVANCED AND reading-type.genre = SCIENCE
SELECT-BOOK Book.title = "Differential equations"
.
 END KNOWLEDGE-BASE Literary-assistant-base;
```

Fig. 4-28 KADS Language

4.7.4 Inference Knowledge

Inference Knowledge refers to sub-task sets that do not need subsequent decomposition. They are the “reasoning primitives” and the “elemental reasoning steps” for task-resolution. They are described by specifying the performed function and its input and output. It is important to note that an inference description does not imply how it is performed - because it heavily dependent upon the particular application and domain.

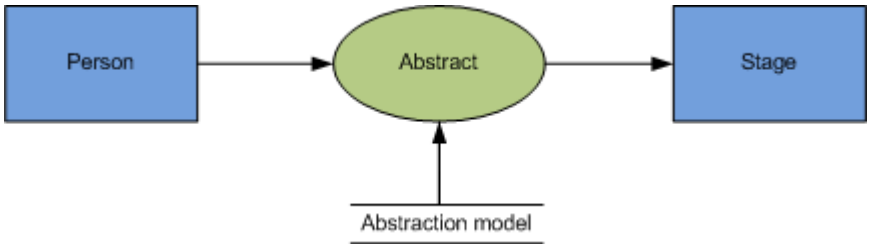


Fig. 4-29 An Abstraction Model Scheme

4.7.5 An Inference Scheme

As explained above, inference is the basic reasoning step; but depends on these domain roles:

- *Static Roles*: These are the domain elements that are used in the reasoning process, but are not affected by it
- *Dynamic Roles*: These are the inference inputs and outputs that sign the domain elements that will be used during the reasoning process

```
INFERENCE Abstract
ROLES
INPUT: Person;
OUTPUT: Stage;
STATIC: Abstraction-model;
SPECIFICATION:
"Abstracts the age of a person"
END INFERENCE Abstract;
```

Fig. 4-30: KADS language

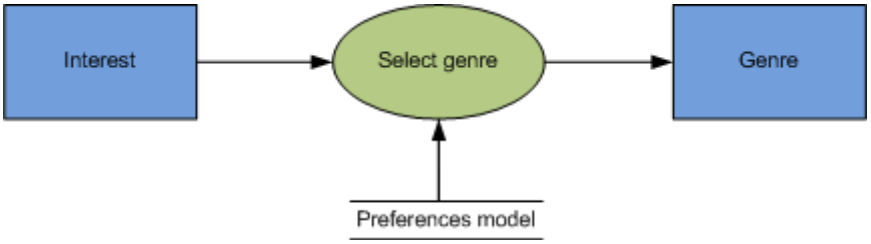


Fig. 4-31 A Preferences Model

```
INFERENCE Select-genre
ROLES
```

```
INPUT: Interest;
OUTPUT: Genre;
STATIC: Preferences-model;
SPECIFICATION:
"Selects a genre according the user's interests"
END INFERENCE Abstract;
```

Fig. 4-32 KADS Code

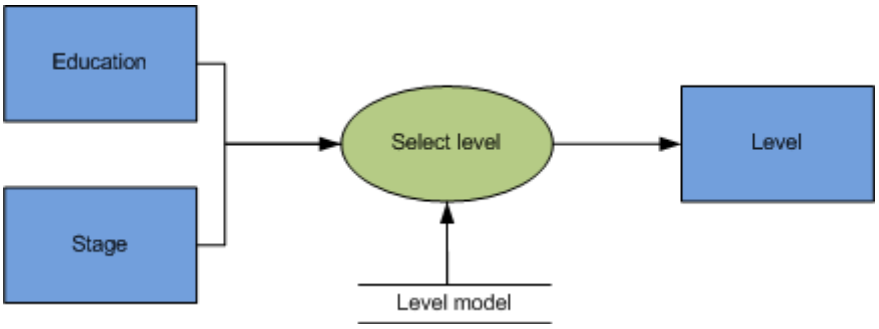


Fig. 4-33 A Level Model Scheme

```
INFERENCE Select-level
ROLES
INPUT: Education,Stage;
OUTPUT: Level;
STATIC: Level-model;
SPECIFICATION:
"Selects a level according the user's education and stage"
END INFERENCE Abstract;
```

Fig.4-34 KADS Code

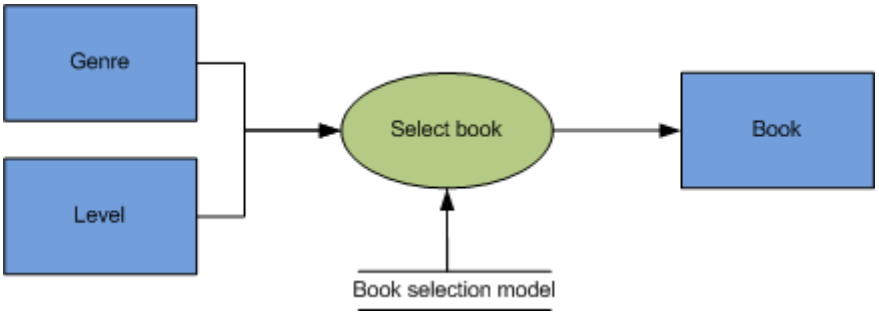


Fig. 4-35 A Book Model Scheme

```
INFERENCE Select-book
ROLES
INPUT: Genre, Level;
OUTPUT: Book;
STATIC: Book-selection-model;
SPECIFICATION:
"Assigns a book according the user's level and selected genres"
END INFERENCE Abstract;
```

Fig. 4-36: KADS Code

4.7.6 Domain Connection

After modeling the Inference Knowledge in question, it is necessary to describe its connection with the elements of the Knowledge Domain. In the example, the results are as follows:

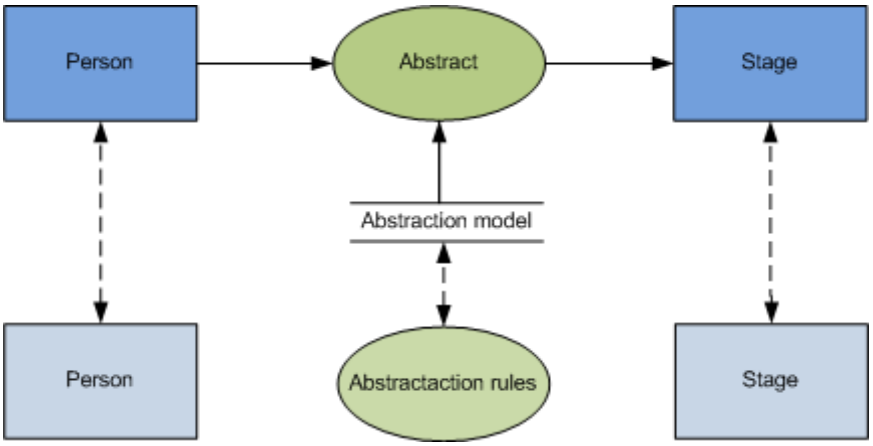


Fig. 4-37 An Abstraction Model Scheme

```
KNOWLEDGE-ROLE Person;
TYPE: DYNAMIC;
DOMAIN-MAPPING: Person;
END KNOWLEDGE-ROLE Person;
```

```
KNOWLEDGE-ROLE Abstraction-model;
TYPE: STATIC;
DOMAIN-MAPPING:Abstraction-rules
  FROM literary-assistant-scheme; ;
END KNOWLEDGE-ROLE Abstraction-model;
```

```
KNOWLEDGE-ROLE Stage;
TYPE: DYNAMIC;
DOMAIN-MAPPING: Stage;
END KNOWLEDGE-ROLE Stage;
```

Fig. 4-38 KADS Code

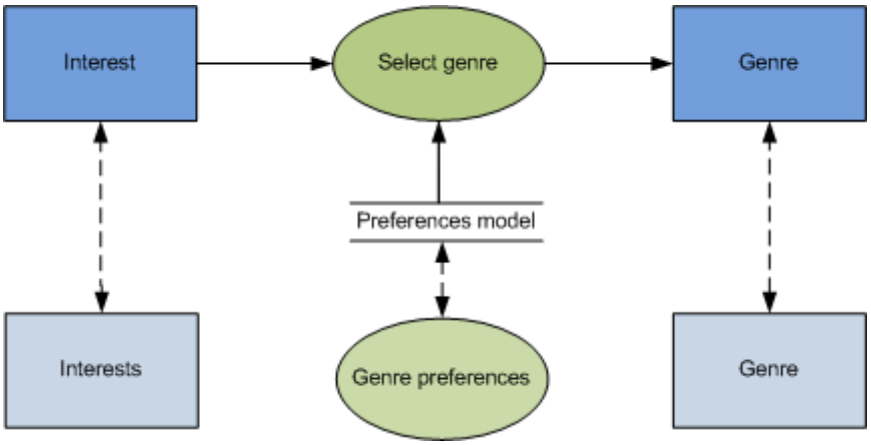


Fig. 4-39 A Preferences Model Scheme

```
KNOWLEDGE-ROLE Interest;
TYPE: DYNAMIC;
DOMAIN-MAPPING: Interest;
END KNOWLEDGE-ROLE Interest;
```

```
KNOWLEDGE-ROLE Preferences-model;
TYPE: STATIC;
DOMAIN-MAPPING:Genre preferences
   FROM literary-assistant-scheme; ;
END KNOWLEDGE-ROLE Preferences-model;
```

```
KNOWLEDGE-ROLE Genre;
TYPE: DYNAMIC;
DOMAIN-MAPPING: Genre;
END KNOWLEDGE-ROLE Genre;
```

Fig. 4-40 KADS Code

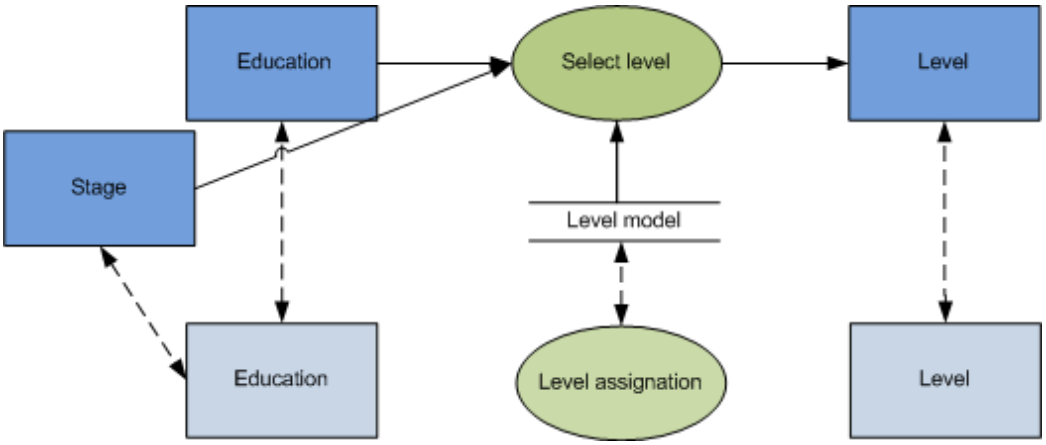


Fig. 4-41 A Level Model Scheme

```
KNOWLEDGE-ROLE Education;
TYPE: DYNAMIC;
DOMAIN-MAPPING: Education;
END KNOWLEDGE-ROLE Education;
```

```
KNOWLEDGE-ROLE Stage;
TYPE: DYNAMIC;
DOMAIN-MAPPING: Education;
END KNOWLEDGE-ROLE Education;
```

```
KNOWLEDGE-ROLE Level-model;
TYPE: STATIC;
DOMAIN-MAPPING:Level-assignation
   FROM literary-assistant-scheme;
END KNOWLEDGE-ROLE Level-model;
```

```
KNOWLEDGE-ROLE Level;
TYPE: DYNAMIC;
DOMAIN-MAPPING: Level;
END KNOWLEDGE-ROLE Level;
```

Fig. 4-42 KADS Code

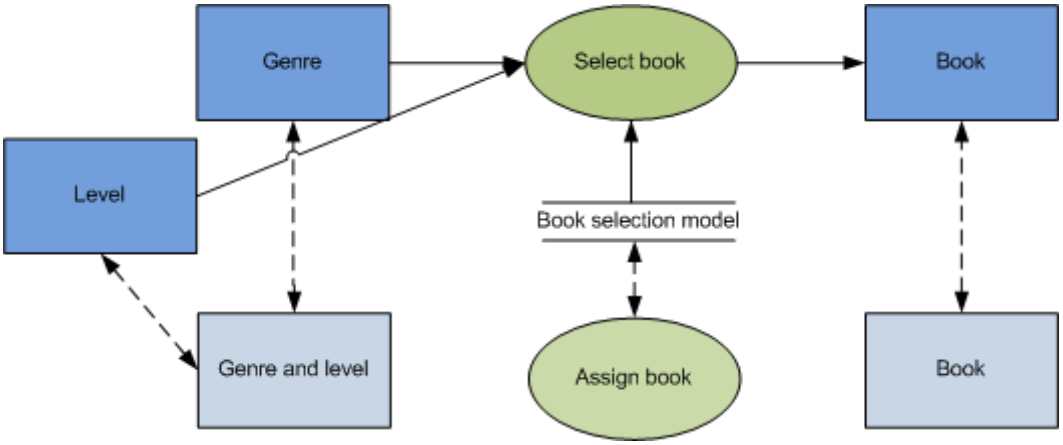


Fig. 4-43 A Book Selection Model Scheme

```
KNOWLEDGE-ROLE Genre;
TYPE: DYNAMIC;
DOMAIN-MAPPING: Genre-and-Level;
END KNOWLEDGE-ROLE Genre;
```

```
KNOWLEDGE-ROLE Level;
TYPE: DYNAMIC;
DOMAIN-MAPPING: Genre-and-Level;
END KNOWLEDGE-ROLE Level;
```

```
KNOWLEDGE-ROLE Book-selection-model;
TYPE: STATIC;
DOMAIN-MAPPING:Assign-book
   FROM literary-assistant-scheme;
END KNOWLEDGE-ROLE Book-selection-model;
```

```
KNOWLEDGE-ROLE Book;
TYPE: DYNAMIC;
DOMAIN-MAPPING: Book;
END KNOWLEDGE-ROLE Book;
```

Fig. 4-44 KADS Code

4.7.7 Task Knowledge

The task is a **general objective** (e.g. to diagnose, plan, etc.); and to get it, a **method** is used which implies a sub-task decomposition into more elemental tasks - and a **control** to sequence them in task-runtime. When dividing the sub-tasks recursively, we get down to the last level of elemental tasks that are not longer decomposable - and these are called inferences. The Knowledge Control Method is essential for the final implementation of a task. It specifies a clear, precise - and unequivocal procedure to link the inferences. In the following image, we can see the task diagram and its sub-task decomposition into inferences:

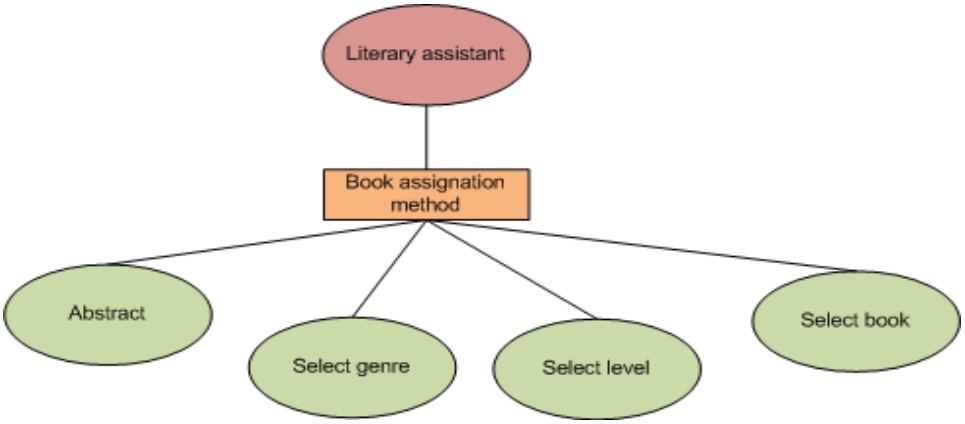


Fig. 4-45An Assignation Model Scheme

```
TASK Literary-assistant
    GOAL:
        "Advice a book to a reader";
    ROLES:
        INPUT:
            Person: "A person who wants to read a book";
        OUTPUT:
            Book: "Some books to read";

    SPEC:
    "Assign a book to a person according to its education and age";
END TASK Literary-assistant;

TASK-METHOD Literary-assistant-sequential;
    REALIZES:
        "Literary assistant";
    DECOMPOSITION:
    INFERENCES: Abstract, Select-genre, Select-level, Select-book;
    TRANSFER-FUNCTIONS:;
    ROLES:
    INTERMEDIATE:
        Abstraction-model:
        Preferences-model:
        Level-model:
        Book-selection-model:
    CONTROL-STRUCTURE:
        Abstract(+Stage,-Person,-Abstraction-model);
        Select-genre(+Genre,-Interest,-Preferences-model);
        Select-level(+Level,-Education,-Stage,-Level-model);
        Select-Book(+Book,-Genre,-Level,-Book-selection-model);
END-TASK-METHOD Literary-assistant-sequential;
```

Fig. 4-46 KADS Code

4.8 Implementation (A Sample)

We chose to use CLIPS - an expert system-tool that was created in 1985, to implement the example - with the following features:

- **Knowledge Representation**: CLIPS provides a cohesive tool for handling a wide variety of knowledge with support for three different programming paradigms: Rule-based, Object-oriented, and Procedural

- **Portability**: CLIPS is written in C for portability and speed - and has been installed on many different Operating Systems without code changes. Operating systems on which CLIPS has been tested include **Windows XP**, **MacOS X**, and **Unix**. CLIPS can be ported to any system which has an ANSI compliant C or C++ compiler. "CLIPS" comes with (its) complete source-code - which can be modified or tailored to meet user-specific needs
- **Integration/Extensibility**: CLIPS can be embedded within procedural code - called a sub-routine; and can be integrated with languages like **C**, **C++**, **Java**, **FORTRAN**, and **ADA**.
- **Verification/Validation**: CLIPS includes a number of features to support the verification and validation of expert systems - including support for modular-design and knowledge-base partitioning, or static and dynamic constraint-checking of slot values and function arguments; and, the semantic analysis of rule patterns to determine if inconsistencies could prevent a rule from excluding or generating an error, e.g.:
- Fully-documented
- Low-cost: CLIPS is public domain software

Here below is a sample of the complete source code - written in CLIPS:

```
; Select level according to education

(defrule select_level
    ?f1<-(start_level)
    (person (education ?edu)(stage ?stg))
    (level (education ?edu)(stage ?stg)(level ?lev))
    =>
    (assert (reader-type (level ?lev)))
    (retract ?f1)
)
; Find interests in interest list

(deffunction has_person (?mat $?interests)
    (return (member$ ?mat $?interests))
)
; Select genre according to interests

(defrule select_genre
    (interests)
    (person (interests $?interests))
    (matter (interest $?cat)(genre ?genre))
    =>
    (bind ?i 1)
    (while (<= ?i (length$ $?interests))
        (bind ?mat (nth$ ?i $?interests))
        (if (has_person ?mat $?cat) then
            (assert (reading-type (genre ?genre)))
        )
```

```
    (bind ?i (+ ?i 1))
    )
)
; Select book according to interests

(defrule recommend_book
    (reader-type (level ?level))
    (reading-type (genre ?genre))
    (category (level ?level)(genre ?genre)(book ?book))
    =>
    (printout t "According to this data, you should read: " ?book crlf)
)
```

Fig. 4-47 CLIPS Code

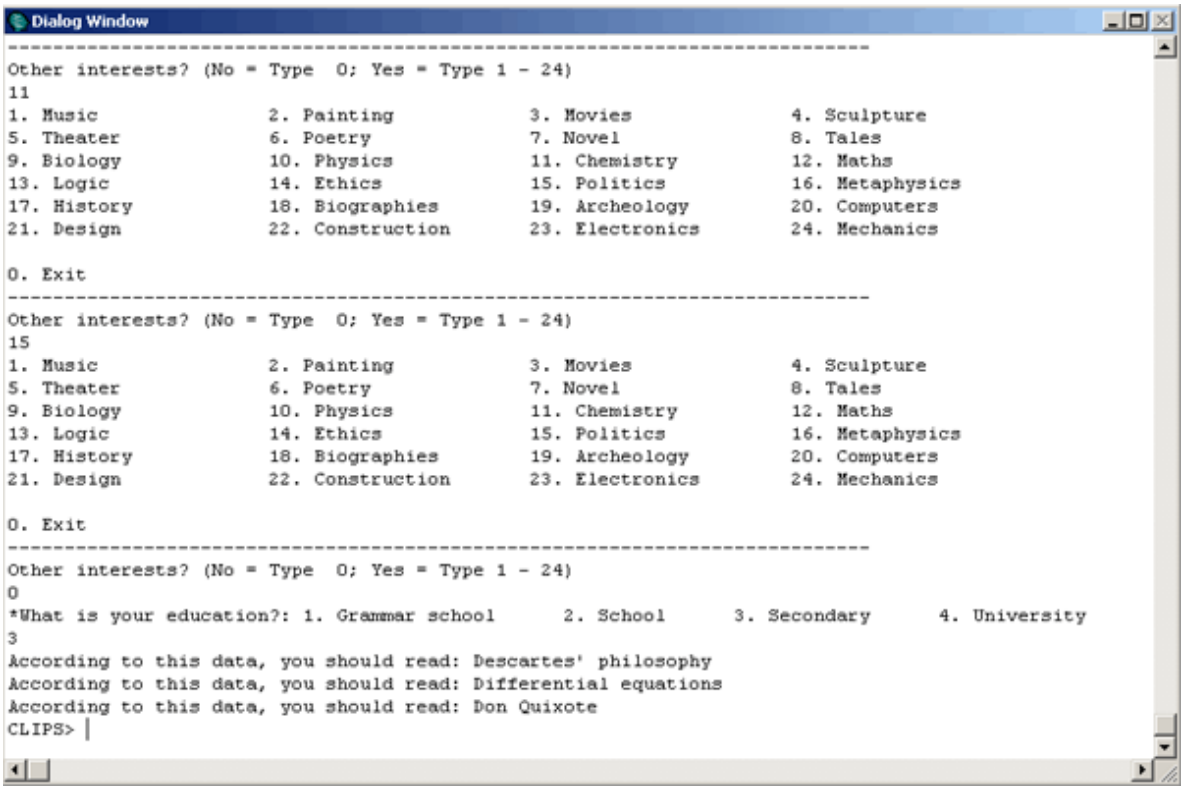


Fig. 4-48 A Dialogue Scheme

The image above presents the execution of the application in CLIPS. Note that if you type an age and an education that do not have a logical match, the system will not return any result. For example, if you type age: 50, and education: Child, the system will not return any suggestion.

4.9 C++ Embedding

The simplest form of embedding the CLIPS source into C++ is to include the header-file "*clips.h*" through the *externkeyword* in the project. Then, we call “InitializeEnvironment” to run the CLIPS engine and call “Load” to load the *.clp* file containing the source. Finally, we

call “reset and run” to start the application. Here below is the code snippet in order to be able to perform this:

```
#include <iostream>

extern "C"
{
    #include "clips.h"
}

using namespace std;

int main(int argc, char* argv[])
{
    InitializeEnvironment();
    Load("literary assistant.clp");
    // Insert the CLIPS source code here

    Reset();
    Run(-1L);
    cin.get();
    return 0;
}
```

Fig. 4-49 C++ Code

5 Formal Methods [31]

This section intent is to give you some insights on modeling and formal reasoning. These activities are supposed to be performed before undertaking the effective coding of a computer system - so that the system in question´s construction be correct.

In this section, we shall thus learn how to build models of programs - and, more generally, Discrete Systems - but this will be done with practice (praxsis) in mind. For this, we shall study a large number of examples from various computer system developments sources: e.g. sequential programs, concurrent programs, distributed programs, electronic circuits, reactive systems, etc.

You will understand that the model of a program is quite different from the program itself. And, you will learn that it is far easier to reason about the model - than about the program. You will be made aware of the very important notions of *abstraction* and *refinement*: the idea being that an executable program is only obtained as the final stage of a sometimes, long sequence consisting of gradually-accreting and more and more accurate models of the future program (i.e. think of the various blue-prints made by an architect).

We shall make it very clear what it is that we mean by “reasoning about a model”. This will be done by using some simple mathematical methods that will be presented to you - firstly, by

means of some examples; then, by reviewing classical Logic (i.e. Propositional and Predicate Calculus) and Set Theory. You will understand the necessity of performing proofs in a very rigorous fashion.

You will also understand how it is possible to detect the presence of inconsistencies in your models - just by the fact that some proofs cannot be attained. The failure of the proof will provide you with some helpful clues on what is wrong - or insufficiently defined, in your model.

The formalism we use is called “Event-B” [34]. It is a simplification - as well as an extension of the B formalism; which has been used in a number of large industrial projects. The formal concepts used in “Event-B” are - by no means, new.

They were proposed a long time ago, in a number of parent formalisms. e.g. Action Systems.

Here is an example - together with the necessary formalism; that will allow you to understand the mathematical concepts being used.

5.1 Examples[34]

"A Simple File Transfer Protocol": Here, we present a so-called “protocol” - to be used on a computer network by two agents. This is the very classical “two-phase handshake” protocol. A very nice presentation of this example can be found in L. Lamport´s book.

This example allows us to extend our usage of mathematical language with such constructs like Partial and Total Functions, Domain and Range Functions, and Function Restrictions. We shall also extend our logical language by introducing universally-quantified formulae and corresponding inference rules.

Nowadays, the term “formal method” leads to great confusion because its usage has been enlarged to cover many different activities. Typical questions that can be asked about such methods include the following: “Why use formal methods?”; “What are they used for?”; “When do we need to use such methods?”; “Is UML a formal method?”; “Are they needed in object-oriented programming?”; “How can we define formal methods?” …

We will answer these questions on a gradual basis. Formal methods have to be used by people who have recognized that the (internal) program development process they use is inadequate. There may be several reasons for such inadequacies: i.e. Failure, Cost, Risk.

The choice of formal method is not easy - in part because there are many formal method vendors; or, to be more precise - the adjective “formal” does not mean anything. Here are some questions one may ask a formal-method vendor. “Is there any theory behind your formal method?”; “What kind of language does your formal method use?”; “Does there exist any kind of refinement mechanism associated with your formal method?”; “What is your way of reasoning with your formal method?”; “Do you prove anything when using your formal method?”

People might claim that using formal methods is impossible because there are some intrinsic difficulties in doing-so. Here are a few of these claimed difficulties - one has to be a mathematician though. The proposed formalism is hard to master. It is not visual enough (i.e. boxes, arrows are missing); or, people will be unable to perform proofs.

We disagree with most of the points of view above - but recognize that there are some real difficulties - which … to our mind; are the following:

1. When using formal methods, one has to think a lot before coding … which is not - as we know, current practice
2. The use of formal methods has to be incorporated within a certain development process - and this incorporation is not easy. In industry, people develop their products under very precise guidelines - which they have to follow very carefully. Usually, the introduction of such guidelines in an industry takes a significant time before fully being accepted and diligently observed by engineers. Nowadays, changing such guidelines to incorporate using formal methods in them, is something that managers are very reluctant to do - because they are afraid of the time and cost this modification process will take.
3. Model-building is not a simple activity:

   One has to be careful not to confuse modeling and programming. Sometimes, people use some kind of “pseudo-programming” - instead of Modeling. To be more precise;, the initial model of a program describes the properties that the program must fulfil. It

does not describe the algorithm contained in the program - but rather, the way by which we can eventually judge that the final program is correct. For example, the initial model of a file-sorting program does not explain “how to sort”. Rather, it explains what the properties of a sorted file are - and which relationship exists between the initial non-sorted file we want to sort and the finally-sorted one

4. Modeling has to be accompanied by “reasoning”. In other words - the model of a program is not just a piece of text; whatever formalism is being used. It also contains proofs that are related to that text.

   For many years, formal methods have just been used as a means of obtaining abstract descriptions of a program we wanted to construct. Once again, descriptions are not enough. We must justify what we write by proving some consistency properties. Now, the problem is that software-practitioners are not used to constructing such proofs; whereas, people in other engineering disciplines are far more familiar with doing so. And one of the difficulties in making this become part of software engineering daily practice (praxsis) - is the lack of sufficient good-quality proving-tool support for proofs that could be used on a large scale.

5. Finally, one important difficulty encountered in modeling is the very frequent lack of good requirement documents associated with the programming job that needs to be performed. Most of the time, the requirement documents to be found in industry are either - almost inexistent; or far too verbose. In our opinion, it is vital - most of the time, to completely re-write such documents before starting any modeling. We shall come back to this point herein below.

The people in charge of the development of large - and complex computer systems, should adopt a point-of-view shared by all mature engineering disciplines; namely, the use of an artifact to reason about the future system, during its construction. In these disciplines, people use blueprints - in the wider sense of the term; which allow them to reason formally during the construction process itself. Here are a number of mature engineering disciplines: Avionics, Civil Engineering, Mechanical Engineering, Train Control Systems, (Air)Ship Construction, etc. In such disciplines, blue-prints are used and considered as very important parts of their engineering activity.

Let us analyze just what a blue-print is. A blue-print is a certain representation of a future system. It is not however a mock-up - because the basis is lacking: you cannot drive the blueprint of a car! The blueprint allows one to reason about the future system one wants to construct during its own construction process.

Reasoning about a future system means defining and calculating its behavior - and its constraints. It also allows you to gradually construct an architecture; based on some dedicated underlying theories: strength of material, fluid mechanics, gravitation, etc.

Now, it is possible to use a number of “blueprinting” techniques - which we are going to review. When blueprinting, one uses a number of pre-defined conventions - which help reasoning; while also allowing the sharing of blueprints between wide/large, communities. Blueprints are usually organized as sequences of more and more accurate versions (once again, think of blueprints made by architects) - where each (and every) more-recent version adds details that could not be visible in previous ones. Likewise, blueprints can be further decomposed in order to enhance their “readability”. It is also possible that - for some early blueprints, these would not be completely determined - thus leaving open options…. later to be refined (in further blueprints).

Finally, it is very interesting to have libraries of old blueprints where an engineer can browse in order to reuse something that has already been done. All this - Refinement, Decomposition, Reuse; clearly requires that blueprints are used with care so as to ensure that the entire blueprint development of a system is coherent. For example, one has to be sure that a more accurate blueprint does not contradict a previous - less precise one.

Most of the time - in our software construction engineering discipline, people do not use such blueprinting artifacts. This results in a very heavy testing phase on the final product - which is well-known to quite often happen too late. The blueprint drawing of our discipline consists of building models of our future systems. In no way is the model of a program, the program itself. But the model of a program - and more generally, of a complex computer system; although not executable which allows one to clearly identify the properties of the future system, and to prove that it will be present in it.

So, we can say that formal methods are the application of mathematics in order to model and verify software or hardware systems; (Storey (1996)). Mathematically-based languages can be

used to write specifications of systems, using precise rules. The specification can be interpreted unambiguously - and can be formally verified so as to ensure consistency and correctness.

Formal methods have been used to develop applications – e.g. Air Traffic Control (Hall (1996)); Railway Signaling, (Behm et al. (1999)); and transaction-processing systems, (Houstan and King (1991)). Formal verification involves the application of mathematical proofs to every possible behavior allowed by a specification (Abrial (1996)).
In a state-based specification, the behavior is a transformation of a system - moving from one state to another. Proof obligations are generated using the *specification* and *language* rules.
These proof obligations then need to be discharged by using the specification properties.

5.2 Examples of Formal Methods

5.2.1 The B-Method [34]

The B-Method is a software development method, based on B, a tool-supported formal method based on an abstract machine notation, used in the development of computer software. It was originally developed by Jean-Raymond Abrial in France and the UK. B is related to the Z notation (also by Abrial) and supports the development of programming language code from specifications. B has been used in major safety-critical system applications in Europe (e.g. the Paris Métro Line 14). It has robust, commercially-available tool support for specification, design, as well as proof and code generation.

Compared to Z, B is slightly more low-level and more focused on refinements to code rather than just formal specification — hence it is easier to correctly implement a specification written in B than one in Z; in particular - there is good tool support for this.

Recently, another formal method called **Event-B** has been developed. Event-B is considered to be an evolution of B (also known as Classical B). It is a simpler notation that is easier to learn and use. It comes with tool support in the form of the Rodin Tool.

5.2.2 The Main Components

B Notation depends on set theory and first-order logic in order to specify different versions of software that cover the complete project development cycle.

An Abstract Machine

In the first and the most abstract version - called “Abstract Machine”, a designer should specify the goal of the design.

Refinement

- Then - during a refinement step, they may pad the specification in order to clarify the goal; or to make the abstract machine more concrete by adding more details about data structures and algorithms that explain how the goal may be achieved
- The new version - called Refinement, should prove to be coherent and include all the properties of the Abstract Machine
- Designers may make use of many B libraries in order to visualize data-structure, or to include or import some components

Implementation

- The refinement in turn, may be refined one - or many times, to obtain a deterministic version - which is called “Implementation”
- The same notation is used during all of the development steps and the last version may be translated into Ada, C or C++ language.

5.2.3 Some B Method Characteristics

- Use the same language in specification, design and programming
- Mechanisms include encapsulation and data-locality
- A clear and close introduction for the Refinement Concept
- Originated in the 1980s by Jean-Raymond Abrial
- The B Method is a tool-supported formal method based on AMN (Abstract Machine Notation), used in the development of correct software
- The B Method has been used in some major safety-critical system applications in Europe (e.g. Paris Métro Line 14 and the Ariane 5 Rocket)

The **B-Toolkit**, developed by Ib Holm Sørensen *et al.*, is a collection of programming tools designed to support the use of the B-Tool - a set theory-based mathematical interpreter for the purposes of a formal software engineering methodology known as the “B Method”.

The toolkit uses a custom X Window Motif Interface for GUI management and runs primarily on the Linux, Mac OS X and Solaris operating systems. It was developed by the UK-based company B-Core (UK) Limited.

5.2.4 Atelier B

Developed by ClearSy, Atelier B is an industrial tool that allows the operational use of the B Method to develop defect-free, proven software (i.e. formal software). Two versions are available: The Community Edition - available to anyone without restriction; and the Maintenance Edition, for maintenance contract-holders only.

It is used to develop safety automatisations for the various subways installed throughout the world by Alstom and Siemens, as well as for Common Criteria certification and the development of system models by ATMEL and STMicroelectronics.

5.2.5 The Event-B Method [34]

Event-B is a mathematical approach for developing formal models of systems (Abrial and Hallerstede, (2006)). An Event-B model is constructed from a collection of modeling elements. These elements include invariants, events, guards and actions.

The modeling elements have attributes that can be based on Set Theory and Predicate Logic. Set Theory is used to represent data-types and to manipulate the data. Logic is used to apply conditions to the data. The development of an Event-B model goes through two stages; *abstraction* and *refinement*.

The abstract machine specifies the initial requirements of the system. Refinement is carried out in several steps - with each step adding more detail to the system; generally - but not exclusively, in a top-down manner. Reactive Systems (Harel and Pnueli, (1985)), are systems that continually respond to changes in their environment.

The focus on atomic events in Event-B creates a representation of a reactive system (Jones, (2005)). The model transitions are triggered by changes in the state of the model - which can represent the system’s environment. The guard on an event will allow, or prevent, an event from occurring depending on the state of the model.

When none of the guards are true, the system is deadlocked. Event-B is designed for modeling distributed systems (Abrial and Hallerstede, (2006)). It implements the theory of discrete transition systems. Discrete Transition Systems - or Action Systems, model atomic actions that can be performed in parallel providing the actions do not affect the same state variables. One method for specifying concurrency in Event-B is to model each update as a group of potentially interleaving atomic events (Edmunds and Butler, (2008)).

This allows the model to specify how concurrent execution can be dealt with by the system being modeled. Specifying a distributed system in Event-B takes a global approach. Rather than creating a specification for each component of the system, it is modeled as a whole - along with its environment. The model is closed in that it reacts only to changes in its internal state. Initially, states are modeled abstractly with the events that describe the main goal of the system. Detail is added through refinement to describe the final distributed system. The ability to add new events and to refine single events into multiple concrete events, allows the functionality of the system to expand beyond that modeled in the abstract machine. Refinement ensures that the refined models are consistent with the abstract macro-systems (Abrial and Hallerstede, (2006)).

The interactive systems formalism is a recent approach for developing software systems using both structural state-based, as well as instruction–based, composition operators. One of the most interesting features of Interactive Formalism is the structuring of components interactions. In order to study whether a more structured interactive approach can significantly ease the proof-obligation effort needed for correct development in Event-B, we need to devise a way of integrating these formalisms and to define a refinement-base translation from Event-B to an interactive system. The following work-plan is for integrating Event-B and an interactive system formalism:

1. Define a notation of refinement in an interactive system model - based on the combination of the refinement of state-based systems and of the Broy-style refinement of data-flow-based interactive systems
2. Define a translation of EB2IS from Event-B models into structured IS models
3. Prove that the translation of EB2IS preserves refinement
4. Use one of the known translations to move from structured IS models to Event-B models
5. Define a refinement preserving the translation of IS2EB models
6. Use these EB2IS and IS2EB translations to:

a) Improve the discharging of proof-obligations in Event-B is based on IS structual operators and associated decomposition techniques.

b) Get tool-support to develop and analyze IS models

5.2.6 Example

An Event-B Model

Event-B is intended to be extensible. The approach is designed so that new modeling elements can be added without affecting the underlying method. Figure 1 showed how the basic modeling elements can be structured to form an Event-B machine. The first element is the context for the model. The context is an Event-B component that contains the static properties of the model.

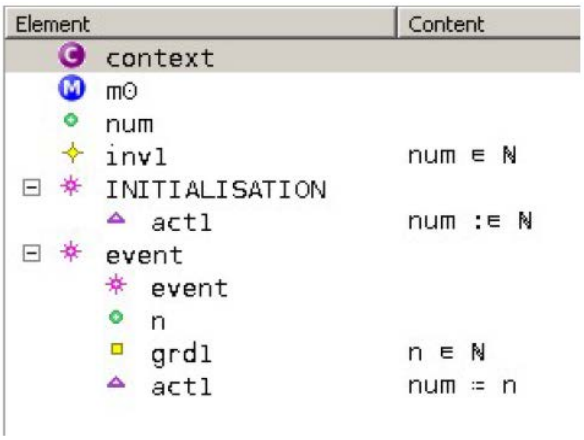


Fig. 5-1 Context

Modeling Elements

An element is the context for the model. The context is an Event-B component that contains the properties of the model.

The model is a refinement - and this is shown by the next modeling element, which indicates the abstract machine m0.

The third modeling element is a variable, *num* that models the state of the machine

The type of the variable, $num \in N$; is defined by the invariant modeling element that follows the variable modeling element.

Most machines will have multiple variable and invariant elements.

Descriptions

Invariants are used to define the necessary properties of a model. The modeling elements have specified the state of the system so far. The event modeling elements specify the actions that can be taken on the state.

The first event is the INITIALISATION event, which is present in all Event-B models. This event initialises the variables of the model.

Because the model only has one variable, there is only one action modeling element for the INITIALISATION event.

The action contains a generalized substitution that non-deterministically assigns an element of the set of natural numbers to the state variable *num*.

The next event has “refines clause, event-variable, and guard and action” modeling elements.

The “refines clause” states the name of the event in the abstract model that this event refines.

The content of the guard is a predicate that restricts the value of the event variable, *n*, to an element of the set of natural numbers,

Modeling Keywords

The content of the action is a generalized substitution that assigns the value of the event variable *n* to the state variable *num*.

Notation can be used to construct models that present the modeling elements in a textual format.

Event-B models will be presented - using the keywords ANY, WHERE, THEN and END

The event variables of an event will be written between ANY and WHERE.

The guards of the event will be written between WHERE and THEN and the actions of the event will be written between THEN and END.

Continue

The context also contains a constant modeling element labeled ***object<u>Id</u>***.

The axiom modeling element contains a predicate that defines the constant ***object<u>Id</u>*** as a total function between the carrier-set OBJECT and the set of natural numbers.

This specifies that each object has an ***object<u>Id</u>***.

The Safety and Liveness of the Event-B Models

Safety and liveness are properties of a formal model that ensure the correct and continuous processing of the model.

A *safety* property specifies that something bad will not happen; and a *liveness* property specifies that something desirable will eventually happen (Lamport, (1977)).
Safety, in Event-B, is specified through events and invariant conditions. *Liveness* in an Event-B model is based on the model being free from deadlock and divergence.

Refinement

The refinement of a model is the process of adding more detail, in a step-by-step manner to a model. The refinement of an Event-B abstract machine can be carried out in several steps.
More detail is added to the model at each step.
Classic step-by-step refinement proceeds in a top-down fashion (Back et al., (1998)).
In large developments, this is not always the case - and refinement can be an iterative process.
The advantage of using refinement is that it allows the model to be analyzed at an abstract level, where the complexity is reduced (Abrial and Hallerstede, (2006)).
The detail of the model can be introduced over several steps, with each step being available for separate analysis.
A refinement model can be proven to be consistent with the abstract model using formal verification.

Decomposition [36]

Decomposition makes it possible to manage the complexity of the models, which increases through the refinement process.
Decomposition splits the model into separate components that model individual parts of the system.
Decomposition is especially useful when modeling systems that contain complex subsystems, e.g. agents - as it can be re-applied at the different levels of the system hierarchy (Jennings, (1993)).
Another advantage of decomposition is that the components can be re-used (Pressman, (2000)).
Two forms of decomposition may be used in Event-B, *event-based* and *state-based*.

Event-based Decomposition

Event-based Decomposition separates the model into its events. The variables are encapsulated in the separate components, and the events or parts of the events that affect the variables are specified in that component model.

The events that have been split will then need to be synchronized in order to recreate the functionality of the original system model.

This can be achieved by exchanging inputs and outputs between the synchronized component events.

State-based Decomposition

In State-based Decomposition, the variables are split between the different components with some variables being shared by events in different components. Events are added to components to simulate how the shared variables are used in other components. The shared variables and events must be kept synchronized between the components during refinements. The system can be rebuilt into a single model later in refinement.

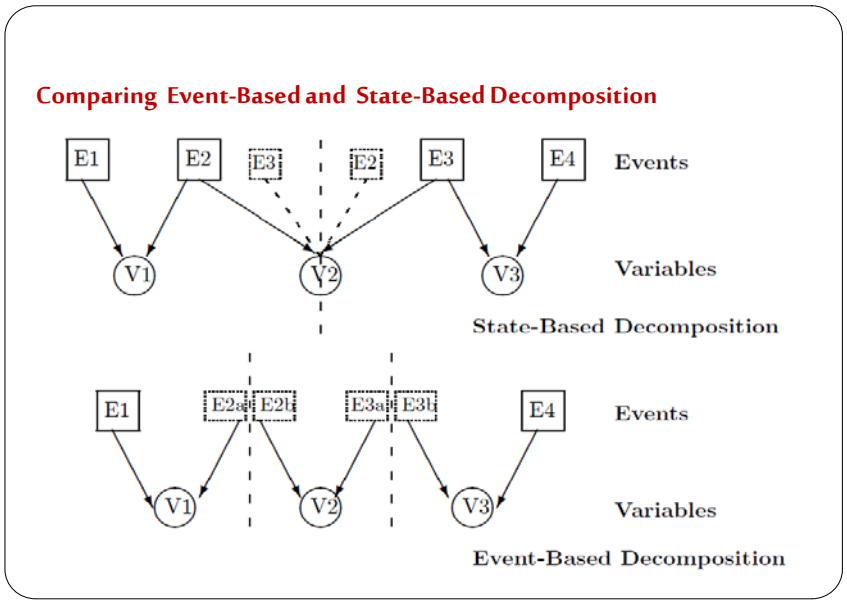


Fig. 5-2 Event-Based Decomposition

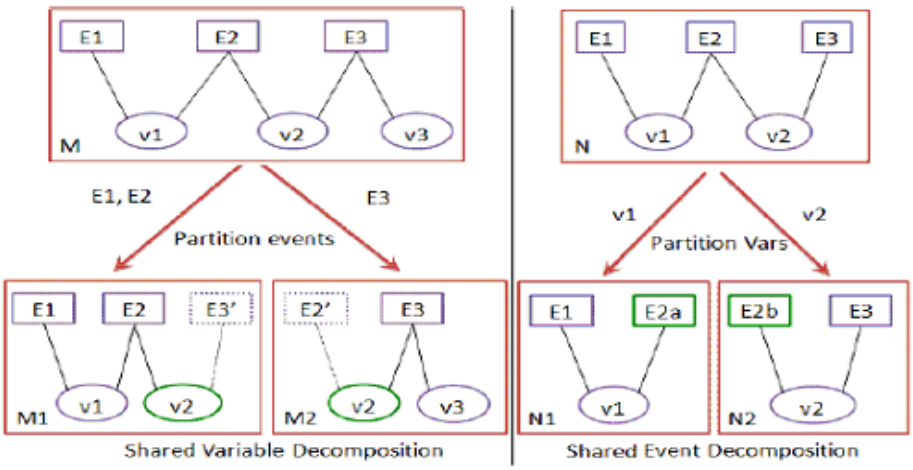


Fig. 5-3 Decomposition Types in Event-B

5.3 The Rodin Platform and Plug-in Installation [38]

The Rodin Platform is an Eclipse-based IDE for Event-B that provides effective support for refinements and mathematical proof. The platform is open-source and contributes to the Eclipse framework - and is further extendable with plugins.

The Rodin Tool supports the application of the Event-B formal method. It provides core functionality for the syntactic analysis and proof-based verification of Event-B models. Rodin also provides extension-points for a range of additional plug-ins that enrich the core functionality through support for features like model-checking, model-animation, graphical front-ends, additional-proof capabilities and code-generation.While the B Method - developed by Jean-Raymond Abrial in the early 1990s, is focused on supporting the formal development of software; Event-B broadens the perspective to cover systems - instead of just modeling software components, Event-B is intended for modeling and reasoning about systems that may consist of physical components, electronics and software. An essential diference between Event-B and the B Method is that Event-B admits a richer notion of refinement, in which new observables may be introduced in refinement steps; this means that complex interactions between subcomponents may be abstracted from in early-stage modeling, and then introduced through refinement in incremental stages.

At around the same time that Jean-Raymond was developing the concepts in Event-B, Mr. Michael Butler, Southampton, 2012 was involved in an initiative with the University of Newcastle (Alexander Romanovsky, C. Jones), Abo Akademi (Kaisa Sere, Elena Troubitsyna) and Jean-Raymond to put together an EU proposal on formal methods for dependable systems – wwhich later became the RODIN project (2004 to 2007); a key part of the project was the development of an open-source extensible tool-set to support refinement-based formal development. Many of Jean-Raymond's ideas on Event-B were worked into the requirements for the tool, and the development of the core tool platform was led by Jean-Raymond and Laurent Voisin (both then at ETH, Zurich). Thorough analysis was undertaken to determine that Eclipse was the right platform on which to build an open tool-set. The ease with which the core may be extended with plug-ins from a range of teams to provide seamless functionality indicates this was a good decision. The tools developed in the RODIN Project took on the name of the project and since it had a certain “cachet”, it was decided to retain the Rodin name for the tool after the project ended.

The RODIN Project was followed by the DEPLOY Project - which addressed the further development of the Rodin core and associated plug-ins in parallel with industrial-scale deployment of the Rodin tools. Exposing the tools to serious industrial users in DEPLOY drove the developers to implement significant improvements in the performance, usability and stability of Rodin and key plug-ins like ProB, the Theory plug-in, Camille and UML-B. Of course, as well as demanding improvements to the tool, industrial users also demanded documentation on the tool - which led to a handbook. Michael Jastram and the team in Dusseldorf have done an excellent job in pulling-together, extending and improving various sources of documentation on the Rodin Tool. Like the Rodin Tools, it will serve as a valuable resource that will continue to evolve beyond the DEPLOY project.

Name	Installation
Rodin Platform	• Requires Java 1.6 • Download the Core: Rodin Platform file for your platform. To install, just unpack the archive anywhere on your hard-disk and launch the "rodin" executable in it. • Start Rodin • Information on the latest release.
Plug-ins	• Plug-ins are installed from within Rodin by selecting Help/Install New Software. Then select the appropriate update site from the list of download sites. • Details on plug-ins. • Install the Atelier B Provers plugin from the Atelier B Provers Update site to take full advantage of Rodin proof capabilities • Install the ProB plugin from the ProB Update site for powerful model checking and animation
User-manual and Tutorial	• Rodin Handbook

Tab 5-1: Rodin Installation – requirements above

6 Case-studies

Event-B is a formal modeling language - a successor to Abrial's classical B. It was developed as part of the RODIN[1] and earlier EU projects. The DEPLOY[2] project - along with its industrial partners, is currently focused on deploying this into industry. Event-B - a state-based language, is based on set-theory and first-order logic, and allows the specification and verification of reactive systems.

The correctness of a model is defined by invariant properties on its state which must be preserved by all transitions in a system - called “events”. An event is enabled when certain pre-conditions on the event - called guards, become true.

Verification conditions (known as Proof Obligations or POs) concerned with model consistency - i.e., invariant preservation, are generated and discharged by proof-support tools. Event-B is further supported by the integrated Rodin toolkit – comprised of editors, theorem-provers, and an animator and a model checker.

Every Event-B model contains a machine and multiple contexts. The machine specifies the behavioural - or dynamic part of the system, and the context contains static data which includes sets, constants, axioms and theorems. The sets denote types whereas the axioms give the properties of the constants – e.g. typing etc.

Theorems must be proved to follow from axioms - the machine sees context(s). The state is expressed by machine variables. Invariant predicates provide the variables and also specify the correctness properties that must always be upheld.

The state transition mechanism is accomplished through events which modify the variables. An event can have conditions - known as event guards, which must be true in order for the event to take place. It can also have parameters, (also known as local variables). The variables are initialized using a special event called *Initialization*, which is unguarded. An event has the following syntax:

$$e = Any\ t\ where\ G(t,v)\ then\ A(v,t)\ end$$

An event *e*, having parameters *t*, can perform actions *A* on variables *v* if the guards *G* on *t* and *v* are true. A model is said to be consistent if all events preserve the invariants. These invariant preservation properties - called Proof Obligations (POs), are the verification conditions automatically generated by the tool - and then discharged using theorem-provers to verify the correctness of the model.

1 RODIN: EU Project IST-511599. http://rodin.cs.ncl.ac.uk

2 DEPLOY: EU Project IST-214158. http://www.deploy-project.eu

6.1 Case-study I

Figure 6-1 shows an example of a complete Event-B model with a machine and its seen context. It has a variable *bal*, typed by the invariant *inv1* and initialized in the Initialization event. The *ACCOUNT* set is given in the context.

There are two events - i.e., *transfer* and *deposit*, which update the variable *bal* when their respective guards become true.

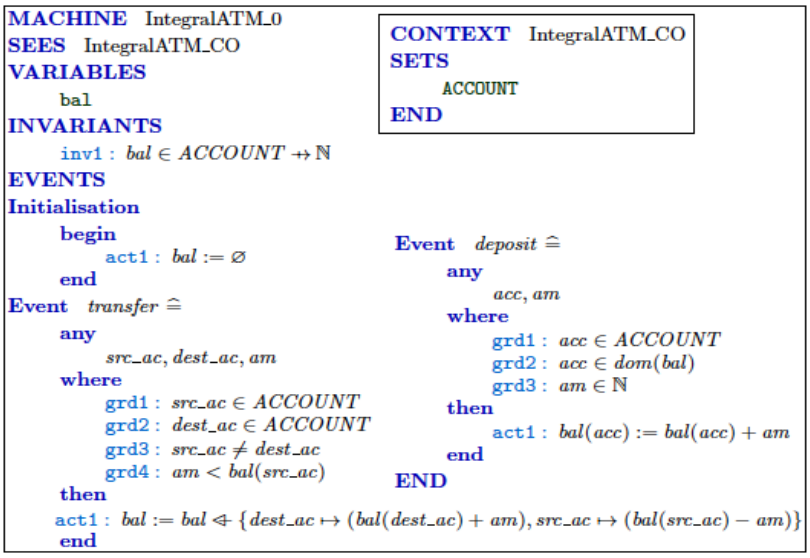


Fig. 6-1 Case-study – IntegralATM

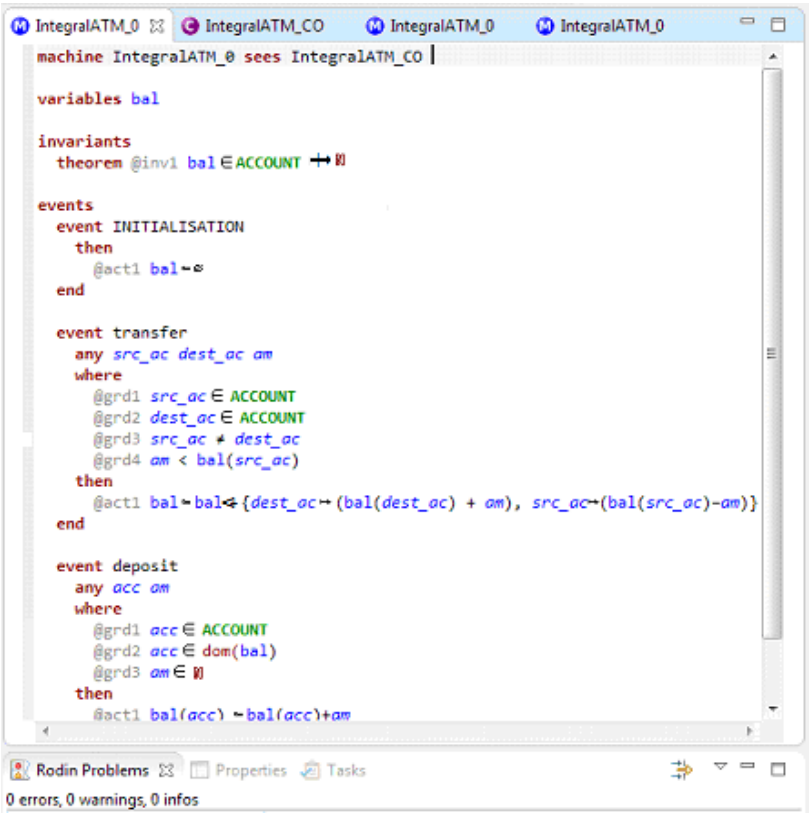


Fig. 6-2 Case-study – MACHINE IntegralATM_0 in the Rodin Tool, Camille Editor

Refinement is a top-down development method and is at the core of Event-B modeling. We start by specifying the system at an abstract level and gradually refine by adding further details in each refinement step until the concrete model is achieved. A refinement is a development step that guarantees that every behavior in the concrete model is one specified in the abstract model. It usually reduces non-determinism - and each refinement step must be proved to be the correct refinement of the abstract model by discharging suitable refinement POs. Typically, we classify the refinement into *horizontal* and *vertical* refinements. In horizontal refinement, we add more details to the abstract model to elaborate the existing specification - or to introduce further requirements of the system being modeled. In *vertical refinement* (a.k.a data refinement), the focus is on design decisions - i.e., transforming and enriching data types and the elaboration of algorithms. In *vertical refinement*, the state of a concrete model is linked to the abstract model using gluing invariants. It is usually harder to prove vertical refinements as compared to horizontal refinements since the gluing invariants increase the PO complexity. A model is vertically refined after the horizontal refinement has been performed to introduce all the requirements of the system.

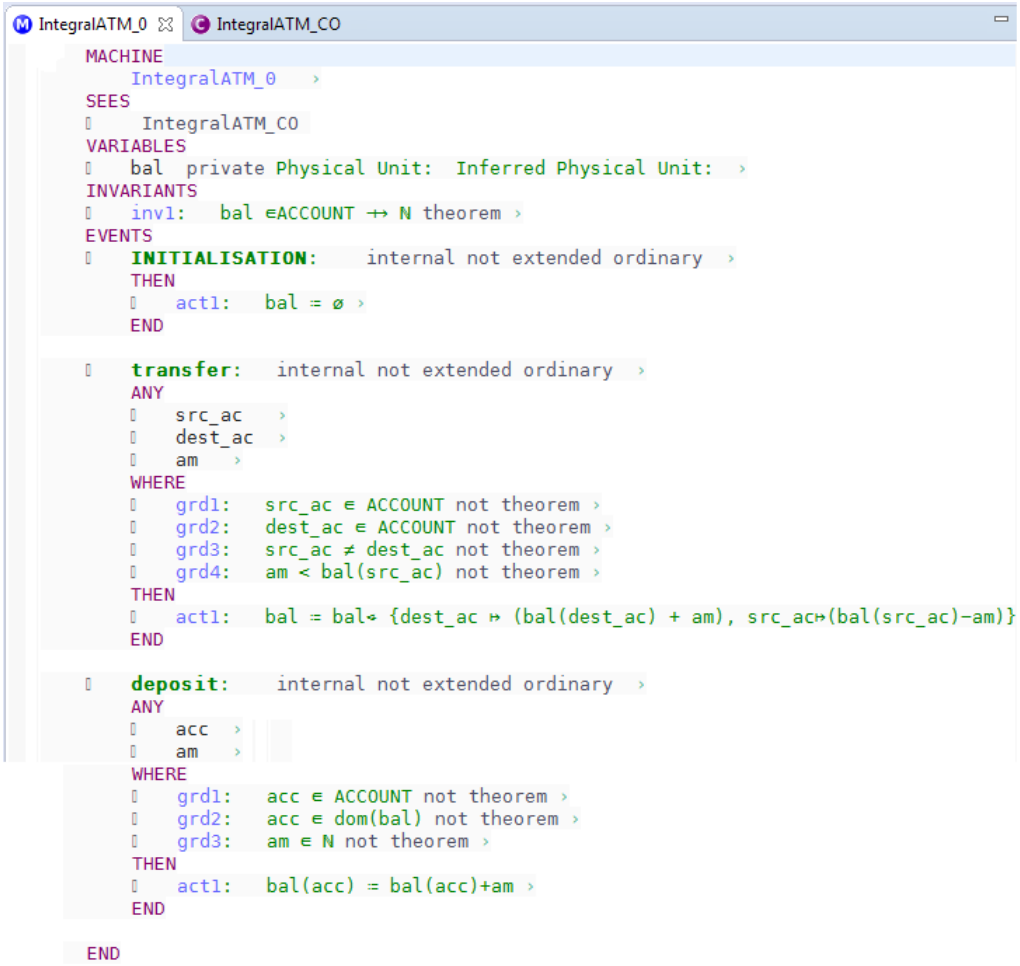


Fig. 6-3 Case-study – MACHINE IntegralATM_0 in the Rodin Tool, Rodin Editor

6.2 Case-study II [43]

6.2.1 The Invoice System

Analyzing the Text of the Case-study

The starting point of the incremental development of an event B model is the analysis of the requirements needed to extract the pertinent details; generally, the requirements are not very well-structured and it may be helpful to structure them - and then to derive logical and mathematical structures of the problem: *sets*, *constants* and *properties* over *sets* and *constants*. The mathematical landscape is produced through requirements elicitation.

B Guidelines: *The concept of set is a central one in the B methodology; each basic object is a set; relations and functions should be considered primarily as sets.*

The lines of the case-study are numbered; numbers will be used when we analyze requirements. We will interleave questions asked either by the customer, the specifier, or… and we will answer these questions.

I. The subject of the case-study is to invoice an order

In fact, an order is a member of a set -, namely, the set of orders. We define a set of orders by the name ALL_ORDERS; we do not yet know if it is a quantity which may be modified. It is the set of all possible orders.

This is explained later in the text.

II. To invoice is to change the state of an order (i.e. to change it from the state pending to invoiced).

To invoice an order is an action or an event which models a modification of the status of an order. The status of an order is either *invoiced* or *pending*; the action should modify the status from *pending* to *invoiced*.

The action or the event is called an "*invoice order*", and is triggered for each pending order. The full condition is defined later in Item 5. However, let us detail the status of an order. An order is either *pending* or *invoiced* and the *action invoice* allows us to modify the state from *pending* to *invoiced*. It is then clear that we should be able to express the state of orders in our model - and that the state may change. We can use a set *STATUS* with two elements *invoiced* and *pending*; the *variable orders state* can be a function from the set of orders called ALL_ORDERS to the set STATUS (*orders*

state ∈ ALL ORDERS → STATUS); the orders state is a function because an order has - at most, only one possible status and is a total function because an order has at least one status. In fact, we can use a set called "*invoiced orders*" containing the invoiced orders, which is a subset of the set of orders ALL_ORDERS.

III. On an order - we have one and only one, reference to an ordered product of a certain quantity. The quantity can be different between orders.

The structure of an order is not clearly given; in fact, no new information on the orders is available. We have one and only one reference to a product of a certain quantity. This means that you can not have two different sets of information for the same product on the same order. If you want to order 4 products **p** and 5 products **p**; either you need to order 9 products **p**, or you order 4 products **p** *and* 5 products **p**, but you will have two different orders in the set of orders.

It seems that there is only one ordered product on an order. *The quantity can be different from other orders* means that the quantity is related to an order - and not to a product.

A set of orders can not be a subset of PRODUCTS×N_* (N_* is the set of non-zero natural numbers) because two orders can have the same product - and the same quantity. We can have a sequence of PRODUCTS×N_*, but it is not a good idea in a first abstraction. The simplest way to define the set of orders is as follows: We suppose that ALL ORDERS is the abstract set which contains all orders (invoiced, pending and future), and orders is the set of existing orders.

We have the following safety property:

$$\boldsymbol{orders \subseteq \text{ALL ORDERS}.}$$

Access operations are defined through the following functions:

$$\boldsymbol{reference \in orders \rightarrow \text{PRODUCTS}}$$

Where, *reference* assigns a product to each order and is a function - because an order is related to one and only one, ordered product.

$$\boldsymbol{quantity \in orders \rightarrow N_*}$$

Where, *quantity* assigns a quantity to each ordered product, and we assume that if a product is ordered, the quantity is at least 1.

Another possible choice is to combine the two previous functions into a single one, as follows:

$$\textit{Reference\_quantity} \in \textbf{orders} \rightarrow \textbf{PRODUCTS} \times \mathbf{N}_*$$

Where, the reference quantity is a function for defining the set of pairs (product, quantity) of the current orders.

IV. The same reference can be ordered on several different orders

You can order 4 bottles of wine and you (or someone-else) can order yet 4 more bottles of wine - so there are two different orders with the same reference (bottle) and the same quantity.

V. The same reference can be ordered on several different orders

The state of the order will be changed into *invoiced* if the ordered quantity is either less than, or equal to, the quantity which is in-stock according to the reference of the ordered product.

When you invoice an order, you should check that there is enough quantity in stock. The text provides us the guard of the *invoice order* event - and the expression of the guard requires us to model the stock. The *stock* variable is a state variable because the stock will evolve according to the occurrences of the *invoice order* event and it assigns to each product the current quantity of available products in the stock:

$$\textit{stock} \in \textbf{PRODUCTS} \rightarrow \mathbf{N}$$

Another possible choice is to define *stock* as a partial function; but the *invoice order* event is more complex to write, since we should first check the definability of the function.

VI. You have to consider the two following cases:

(a) **Case 1**

All the ordered references are references in stock. The stock, or the set of the orders, may vary:

• Due to the entry of new orders or canceled orders

• Due to having a new entry of quantities of products in stock in the warehouse

You do not have to take these entries into account. This means that you will not receive two entry flows (i.e. orders, entries-in-stock). The stock and the set of orders are always given in an up-to-date state.
We state that new events maintain the current invariant over variables and we do not care about the way the events are modifying the variables. We keep the invariant.

(b) **Case 2**
You do have to take into account the entries of:
• New orders
• Cancellations of orders
• Entries of quantities in the stock

All ordered references are references in stock. Item 5 already states this fact. In fact, we want to model the first case-study model, Case 1; and then derive by refinement the second case-study model Case 2. We will explain this process later. Perhaps the customer says that some order can arrive with an unreferenced product. It is not really difficult to handle, since such orders can be filtered in the next refinement.

VII. Decision

The mathematical structure is the set of all possible orders - denoted ALL ORDERS, and the state variables of the system are: *orders*, *stock*, *invoiced orders*, *reference*, *quantity*; the first case-study model, Case 1, explicitly states that: *The stock or the set of the orders may vary*, and we can now confirm the state-variables, They satisfy the following properties:

ALL_ORDERS = Ø: the set of all possible orders is not empty

orders ⊆ ALL ORDERS: the set of current existing orders is a subset of the set of all possible orders

Invoiced_orders ⊆ orders: The set of invoiced orders is a subset of the existing orders

Pending_orders ⊆ orders: The set of pending orders is a subset of the existing orders

Invoiced_orders ∪ pending_orders = orders and invoiced_orders ∩ pending_orders= Ø are the two safety properties that link the three variables:

- Reference ∈ orders → PRODUCTS
- Quantity ∈ orders → N*
- Stock ∈ PRODUCTS → N

The text has already defined the *invoice_order* event; the item (a) defines two new events: a first event (*new_orders*) adds new orders and a second one (*cancel_orders*) cancels orders. Moreover, the stock may vary, and new quantities of products may be added to the stock: the *delivery to stock* event.

In fact, the set of current pending orders is defined by *orders−invoiced_orders*, and we will understand later why we do not use the variable *pending_orders*.
The first event B model, namely: Case 1, is sketched in the following lines; the model is not yet completed and the events should be defined.

An event B model encapsulates variables defining the state of the system; the state should conform to the invariant and each event can be triggered when the current state satisfies the invariant. An abstract model has a name *m*; the clause SETS contains definitions of sets; the clause CONSTANTS allows one to introduce information related to the mathematical structure of the problem to solve, and the clause PROPERTIES contains the effective definitions of the constants: it is very important to carefully list the properties of the constants in a way that can be easily used by the Click'n'Prove tool.

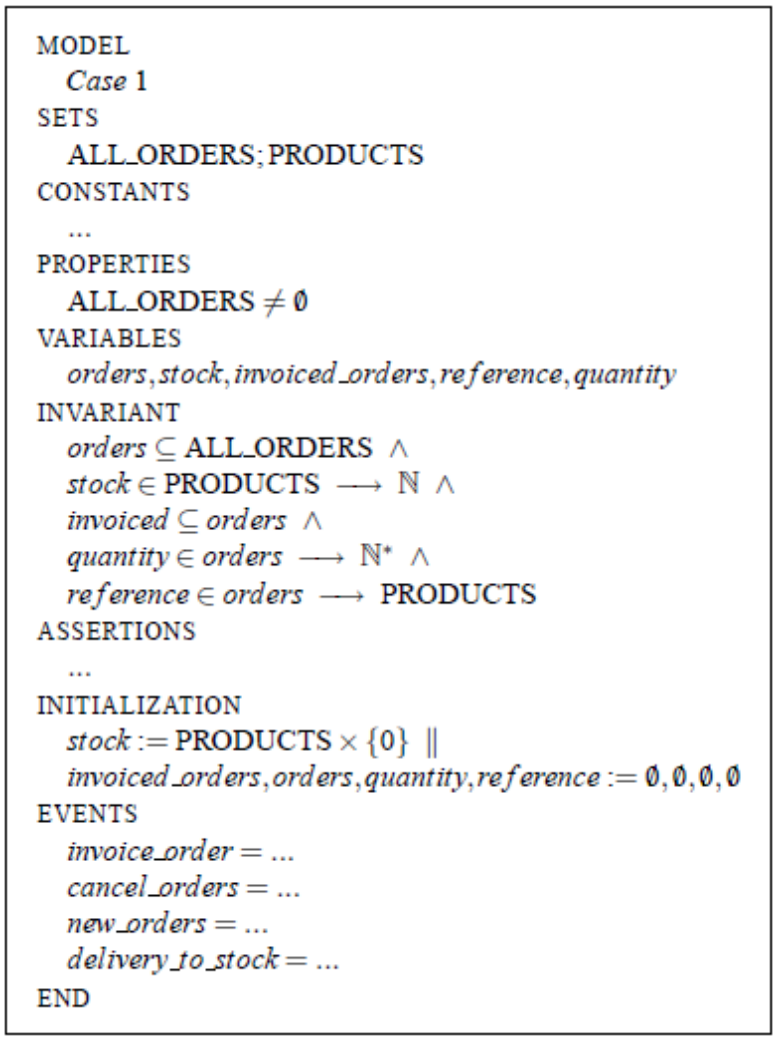


Fig. 6-4 Case 1 – Invoice System Model

The second part of the model defines the dynamic aspects of state variables and properties over variables using the *invariant* generally called "*inductive invariant*", and using *assertions* generally called "*safety properties*".

The invariant *I*(*x*), types the variable *x*, which is assumed to be initialized with respect to the initial conditions - namely, *Init*(*x*); and which is supposed to be preserved by events (or transitions) enumerated in the EVENTS clause.

Conditions of verification, called "*proof obligations*", are generated from the text of the model using the SETS, CONSTANTS and PROPERTIES clauses to define the mathematical theory; and the INVARIANT, INITIALISATION and INVARIANT clauses to generate proof obligations for the preservation, (when triggering events), of the invariant - and proof obligations stating the correctness of safety properties with respect to the invariant.

B Guidelines: *The requirements should be re-structured; basic sets should be identified.*

Modeling the First Event B Model - Case 1

The construction of an event B model is based on an analysis of data which is then manipulated; each B model is organized according to the clauses; and the requirements of the case-study are incrementally added into the B model. In Section 2, we analyzed the requirements and we derived a "first sketch" of an event B model. Events should now be completed - and the model should be internally validated. The internal validation checks that proof obligations hold and are made with the help of the Click'n'Prove tool.

In the text of the description of the system, we use the following information: all of the ordered references are *in-stock* references and we derive the invoice_order event, which is triggered when there are enough items of a given reference in the current stock. Let o be a pending order ($o \in$ orders−invoiced_orders). If the quantity in stock of the product whose reference is: $reference(o)$, is greater than the ordered one: ($quantity(o) \leq stock(reference(o))$, then the order is invoiced, using:

$$\boldsymbol{Invoiced\_orders := invoiced\_orders \cup \{o\}}$$

… and the stock is updated:

$$\boldsymbol{stock(reference(o)) := stock\ (reference(o)) - quantity(o)}$$

```
invoice_order =
  ANY
    o
  WHERE
      o ∈ orders − invoiced_orders    ∧
      quantity(o) ≤ stock(reference(o))
  THEN
      invoiced_orders := invoiced_orders ∪ {o} ||
      stock(reference(o)) := stock(reference(o)) − quantity(o)
  END
```

Fig. 6-5 Case 1 – An Invoice_Order

The next three events model the state changes for the variables attached to the stock and to the orders: *the stock or the set of the orders may vary*.

First of all, the text expresses that the stock and the set of orders may vary; either the variable *invoiced orders* is not modified since the invoiced orders are processed orders; or, we can cancel invoiced orders since it action is possible over orders. We can only modify the set *orders−invoiced_orders*.

The modifications are to add a new order to the current set of orders, and to set the order into the pending set; or, to cancel a pending order from the orders set, or to modify the stock variable by incrementing the quantity of a product.

The specification text tells us that variables are modified and they are less precise than those we suggest. So we propose to require that the three events modify variables; while the invariant is preserved - but the variable *invoiced_orders* is not modified by these events.

The event *cancel_orders* and the event *new_orders* modify the variables *orders, quantity, reference* and the next values of these variables should satisfy:

$$\left(\begin{array}{l} orders \subseteq \text{ALL\_ORDERS} \ \wedge \\ invoiced\_orders \subseteq orders \ \wedge \\ quantity \in orders \longrightarrow \mathbb{N}^* \ \wedge \\ reference \in orders \longrightarrow \text{PRODUCTS} \end{array}\right).$$

Fig. 6-6 Case 1 – Invoice_Order, Invariants

We do not provide details on the possible modifications and we do not care to follow the first case.

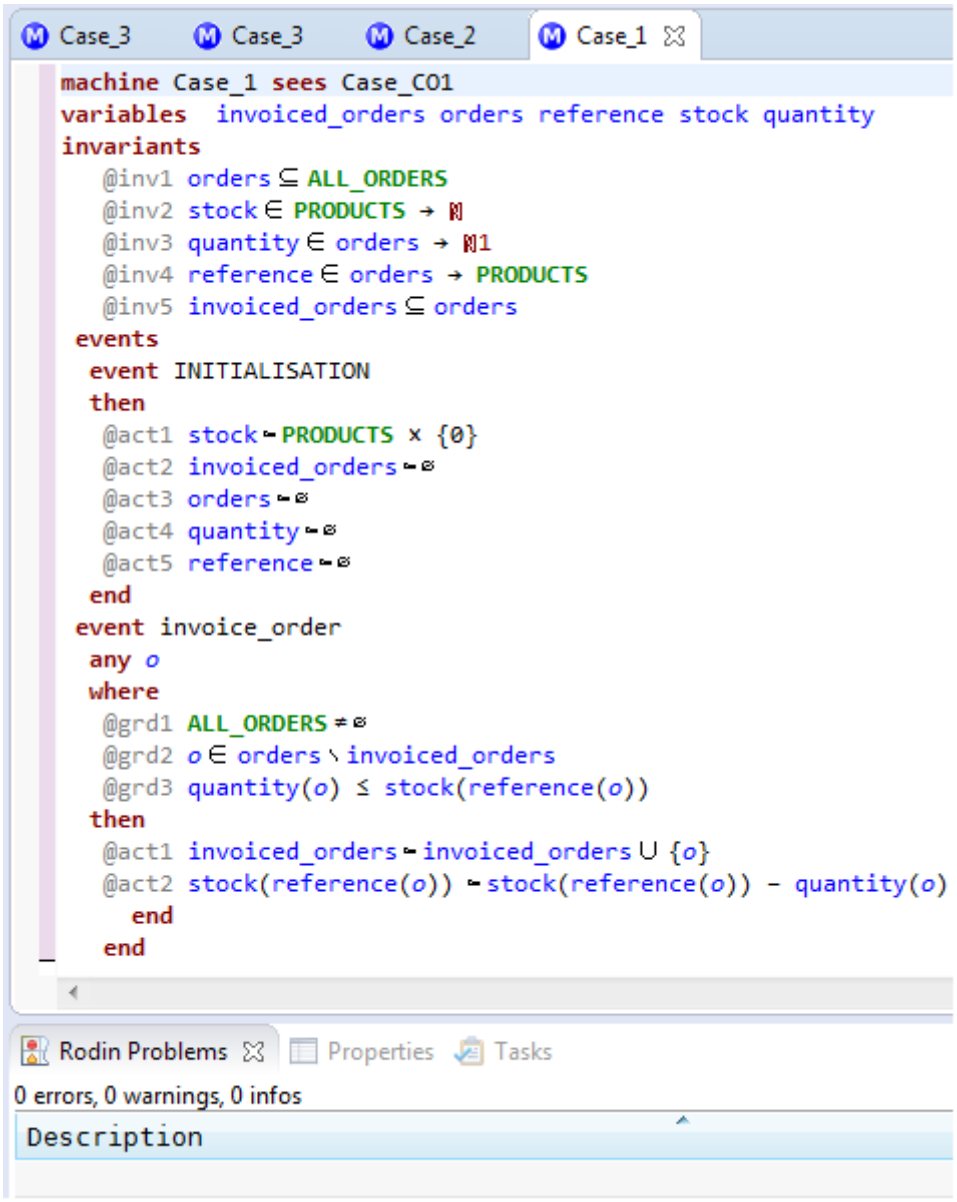


Fig. 6-7 First case – An Invoice System, the Camille Editor

These events are hidden in the “first case” but are explicitly mentioned. They will be refined in the “second case” because the second case provides more details about those events. Finally, they illustrate the *keep* concept, which expresses a possible change with respect to the invariant and also simplifies the refinement.

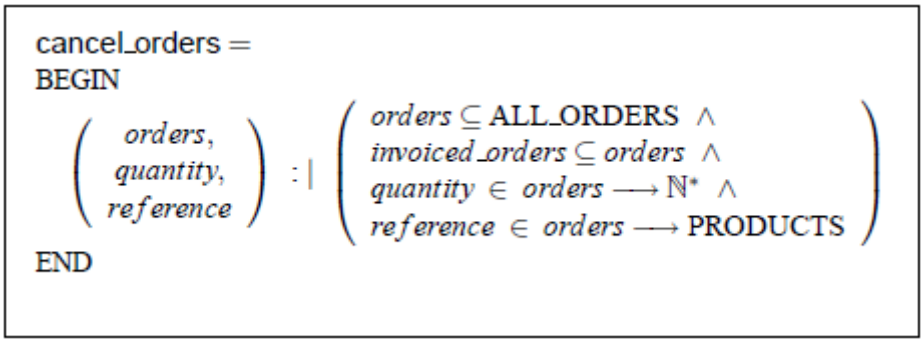


Fig.6-8 Event – Cancel_Orders

The event *delivery_to_stock* changes the value of *stock* and does not change other variables. We do not know how the stock is modified and we only express that a modification is possible.

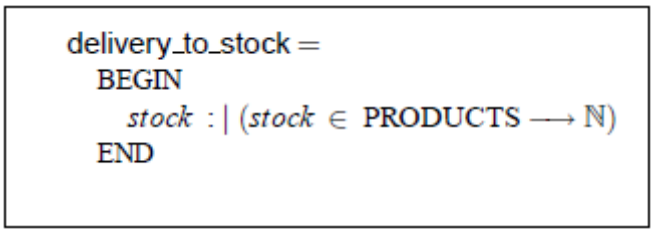


Fig.6-9 Event – Delivery-to-Stock

Checking internal consistency is established by proving nine proof obligations stating that the invariant is initially true and that each event is maintaining the invariant. Each proof obligation is automatically discharged by the Click'n'Prove tool.

The client may be interested by an animation and an animator can be used for testing the possible behaviors of the global model.

The set of variables of the model is the frame of the model; no other variable can be modified; if a variable is not explicitly modified, it is not changed. We assume that the model is closed.

Model Refinement

The refinement of a formal model allows us to enrich a model by using a *step-by-step* approach. Refinement provides a way of constructing stronger invariants as well as adding details to a model.

It is also used to transform an abstract model into a more concrete version by modifying the state description. This is essentially done by extending the list of state variables, (or, possibly,

suppressing some of them), by refining each abstract event into a corresponding concrete version and by adding new events.

The abstract state variable, *x*, and the concrete one, *y*, are linked together by means of a so-called *gluing invariant J*(*x*,*y*). A number of proof obligations ensure that: (1) Each abstract event is correctly refined by its corresponding concrete version; (2) Each new event refines *skip*; (3) No new event takes control forever; and, (4) Relative deadlock-freeness is preserved (i.e. the relative deadlock-freeness states that the concrete model is not more blocked than the abstract one!).

We suppose that an Abstract Model *AM* with variables *x* and invariant *I*(*x*), is refined by a Concrete Model *CM* with variables *y* and gluing invariant *J*(*x*, *y*).
The first proof obligation states that the initial concrete states implies that there is at least one initial abstract state that satisfies the abstract initial condition and is related to the initial concrete state by the gluing invariant:

$$\mathbf{INIT(y) \Rightarrow \exists x.(Init(x) \wedge J(x,y))}$$

If *BAA*(*x*,*x*′) (standing for Before-After Abstract event) and *BAC*(*y*,*y*′) (standing for Before-After Concrete event) are, respectively, the *abstract* and *concrete before-after* predicates of the same event; we have to prove the following statement:

$$\mathbf{I(x) \wedge J(x,y) \wedge BAC(y,y') \Rightarrow \exists x' \cdot (BAA(x,x') \wedge J(x',y'))}$$

This says that, under the abstract invariant I(x) and the concrete one J(x,y), a concrete step BAC(y,y′) can be simulated ($\exists$ x′) by an abstract one BAA(x,x′) in such a way that the gluing invariant J(x′,y′) is preserved. A new event with the *before-after predicate* BA(y,y′) must refine *skip* (x′ = x). This leads to the need to prove the following statement:

$$\boldsymbol{I(x) \wedge J(x,y) \wedge BA(y,y') \Rightarrow J(x,y')}$$

Moreover, we must prove that a variant *V*(*y*) is decreased by each new event (i.e. to guarantee that an abstract step may occur). Thus, we have to prove the following for each new event with the *before-after predicate BA*(*y*,*y*′):

$$\boldsymbol{I(x) \wedge J(x,y) \wedge BA(y,y') \Rightarrow V(y') < V(y)}$$

Finally, we must prove that the concrete model does not introduce more deadlocks than the abstract one. This is formalized by means of the following proof obligation:

$$I(x) \wedge J(x,y) \wedge \text{grds}(AM) \Rightarrow \text{grds}(CM)$$

Where grds(*AM*) stands for the disjunction of the guards of the events in the abstract model; and grds(*CM*) stands for the disjunction of the guards of the events in the concrete one.

Modeling the Second Event B Model - Case 3 by Refinement of Case 2

According to the text of the specification, the second case-study model *takes into account the entries of*:

- *New orders*
- *Cancellations of orders*
- *Entries of quantities into stock*

The behavior of Case 2 is more specialized than Case 1. In Case 1, we do not express how the variables are modified. We state that variables are modified by maintaining the invariant and it is clear that Case 2 is more deterministic than Case 1:

• *Orders* may change by adding new orders

• *Orders* may change by removing pending orders from the current *orders*

• *Stock* changes by adding new quantities of products into stock

No new event is added.

Decision:

The last three events of Case 1 should be refined so as to handle the modifications of the variables: *orders*, *quantity*, *re f erence* and *stock*, according to the last three items:

• *New orders*: The New Orders event modifies *orders*, *quantity*, *reference*; it adds a new order called *o* - which does not yet exist in the current set of orders called: *orders*; *quantity* and *reference* are updated according to the ordered quantity *q* and reference *p*;

• *Cancellations of orders*: The Cancel Orders event modifies *orders*, *quantity*, *reference*; it removes an order called *o*, which is pending in the current set of orders called: *orders*; *quantity* and *reference* –which are updated

• *Entries of quantities into the stock*: The Delivery-to-Stock event adds a given quantity *q* of a given product *p* to the stock.

The text for Case 2 is very clear and it mentions specific ways of modifying the *orders*, *quantity*, *reference* and *stock* variables. Events will simply translate these expressions.

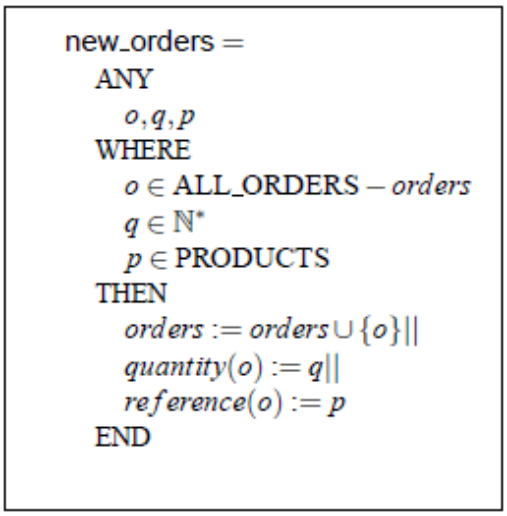


Fig.6-10 New_Orders Event

Let *o* be an order which is neither pending nor invoiced. It is a future order which is added to the current set of orders (*orders*) and the quantity of product is set to *q*; the identification of the product of the *o* order is set to *p*.

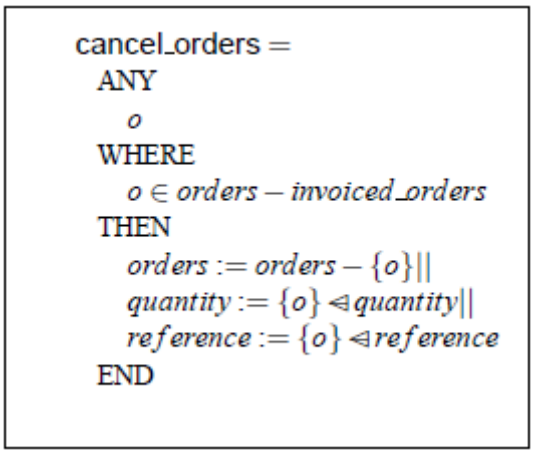


Fig.6-11 Cancel_Orders Event

Let *o* be a *pending* order. The event deletes the order from the *orders* set and the two functions: *quantity* and *reference* are updated by removing *o* from the set of orders which is the domain of those functions.

The refinement conditions generate a proof obligation like o ∈ orders ⇒ invoiced_orders ⊆ orders−{o} and it is clearly not provable without the guard o /∈ invoiced_orders.

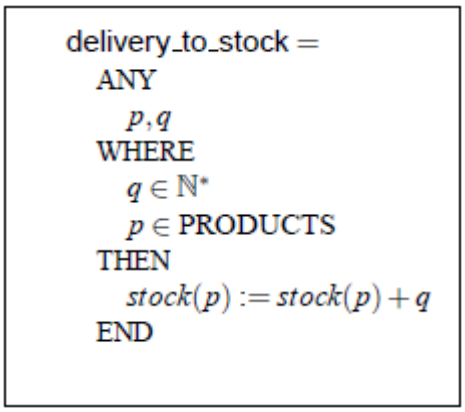


Fig. 6-12 Delivery_to_Stock Event

The stock can only be increased - and the event increases the quantity of the product *p* by *q* units. The stock for *p* is increased by *q*.

The concrete event only modifies the quantity of one product. The abstract event can also decrease quantities of products.

In the case-study, customers mentioned the following statement:

But, we do not have to take these entries into account. This means that you will not receive two entry flows (orders, entries in stock). The stock and the set of orders are always given to you in an up-to-date state.

The last question leads to a new case-study, called Case 3; it takes into account the flow of orders. The new model captures the notion of *flow* by a set; i.e.that the ordering of arrival is not expressed, for instance.

We can require some fairness assumption over some events in order to obtain a *deadlock* and *live-lock-free* model. It is clear that we can not state any kind of fairness in B and the reason is that B language does not provide this facility; it merely analyzes the extension of B´s scope with respect to *liveness* and *fairness* properties.
However, the key question is to refine the models, while the fairness constraints are stated – and, Cansell et.al., have proposed predicate diagrams to deal with these questions. In fact, it is possible that an order can remain “always pending”, and is never invoiced because there are always other orders which are being/have been processed. Another problem is that the quantity may be not sufficient for a while, and it is often insufficient for a given quantity of a given product.

If the referenced quantity changes in the stock, (i.e. the delivery_to_stock event), one can also invoice another order with the same referenced product. Modeling this fact in an abstract way requires strong fairness on the *delivery_to_stock* event. At first, the idea was to use a sequence of orders, and to invoice the first suitable order in the sequence. In this case, we have no lack if the *delivery_to_stock* event is correct enough. For the customer, this is not a good solution since the delay for stock deliveries is too long - and so, one can invoice other orders. We decided to add a time to each order so as to sort orders $(time \in orders \rightarrowtail \mathbb{N})$ and to invoice the most recent possible order.

The *new orders* event gives each new order its time - using the variable: t (t ∈ N), which always contains the next ordered time: $(\forall i \cdot (i \in \mathbf{ran}(time) \Rightarrow i \leq t)$.

The variable *time* records the time when the order was added, and the new condition strengthens the guard of the previous event: invoice_order:

$$\forall d \cdot \left(\begin{array}{l} d \in orders - invoiced\_orders \ \wedge \\ quantity(d) \leq stock(reference(d)) \\ \Rightarrow \\ time(o) \leq time(d) \end{array} \right)$$

Fig.6-13 Refinement of Invoice-order, Added-guard event

The *time* variable is a total injection from *orders* into N - which defines the total ordering over *orders*.

invoice_order =
  ANY
    o
  WHERE
    $o \in orders - invoiced\_orders \ \wedge$
    $quantity(o) \leq stock(reference(o)) \ \wedge$
    $\forall d \cdot \left(\begin{array}{l} d \in orders - invoiced\_orders \ \wedge \\ quantity(d) \leq stock(reference(d)) \\ \Rightarrow \\ time(o) \leq time(d) \end{array} \right)$
  THEN
    $invoiced\_orders := invoiced\_orders \cup \{o\} ||$
    $stock(reference(o)) := stock(reference(o)) - quantity(o)$
  END

Fig.6-14 An Invoice_Order – Refinement Event

The variable *t* is a new, shared variable which models the evolution of the time-stamps; we use the same variable to be sure that we can obtain a total ordering over orders. *Time* is updated according to the current value of the *t* variable.

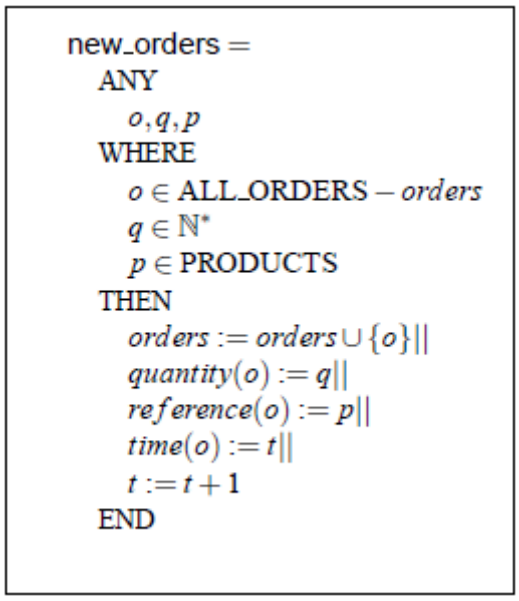


Fig.6-15 A New_Orders – Refinement Event

When one cancels an order, *time* should be updated by removing the canceled order from the *time* domain.

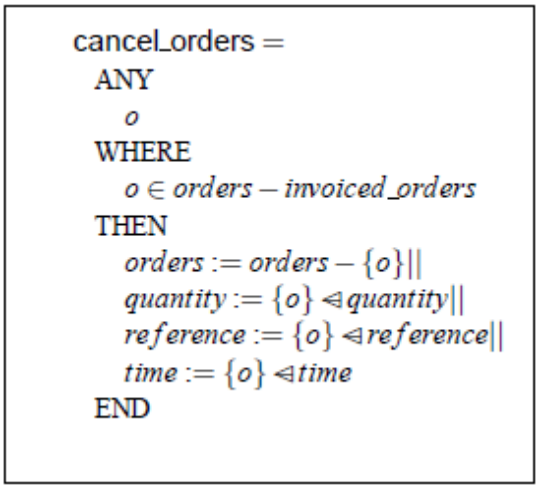


Fig.6-16 A Cancel_Orders – Refinement Event

Fig. 6-17 Second Case – An Invoice System, Added Events, the Camille Editor

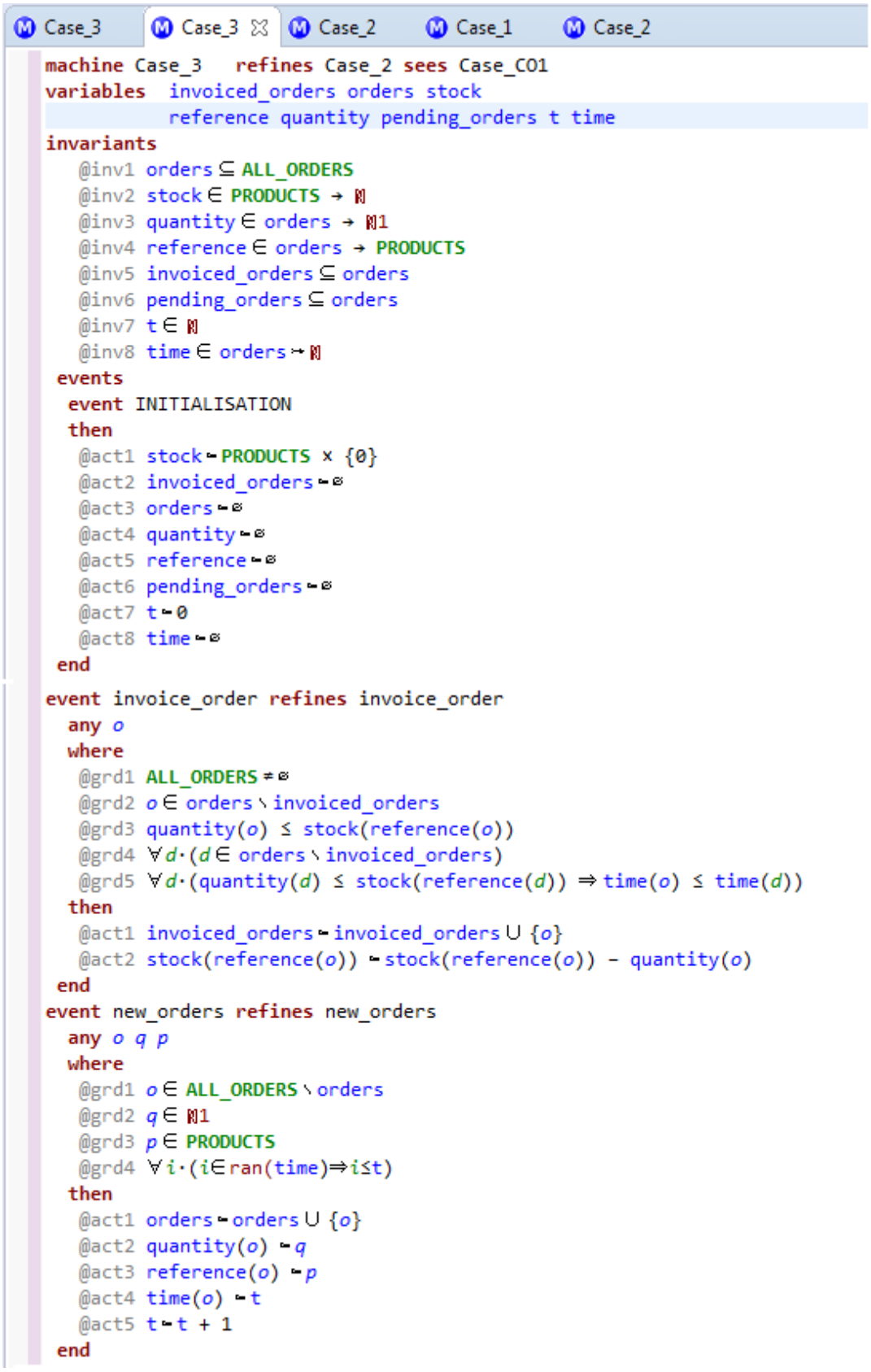


Fig. 6-18 Third Case – Events Refinement, (Timestamp added), the Camille Editor

```
event cancel_orders refines cancel_orders
  any o
  where
   @grd1 o ∈ orders ∖ invoiced_orders
  then
   @act1 orders ≔ orders ∖ {o}
   @act2 quantity ≔ {o} ⩤ quantity
   @act3 reference ≔ {o} ⩤ reference
   @act4 time ≔ {o} ⩤ time
 end
event delivery_to_stock refines delivery_to_stock
  any p q
  where
   @grd1 q ∈ ℕ1
   @grd2 p ∈ PRODUCTS
  then
   @act1 stock(p) ≔ stock(p) + q
 end
   end
```

Fig. 6-19 Third Case – Events Refinement (e.g. *cancel_orders*, *delivery_to_stock*), the Camille Editor

CONCLUSION

Our motivation in these lectures is based on introducing certain features of Event-B – and, its associated Rodin platform.Modeling in Event-B is semantically justified by proof obligations. Every update of a model generates a new set of proof obligations in the background. Without proving the required obligations, we cannot be sure of the correctness of a model. The effort to prove - and thus, encourages a developer to structure formal model development in such a way that manageable proof obligations are generated at each step.

CommonKADS enables one to spot opportunities and bottlenecks in the ways in which organizations develop, distribute and apply their knowledge resources - and so, provides us with Corporate Knowledge Management tools. CommonKADS also provides methods in order to perform a detailed analysis of “knowledge-intensive” tasks and processes. Finally, CommonKADS also supports the development of knowledge-systems that support selected parts of the business process.

REFERENCES

1. Alexander Backlund (2000). "The definition of system". In: *Kybernetes* Vol. 29 nr. 4, pp. 444–451.
2. **Charles S. Wasson,** *System Analysis, Design, and Development,* Wiley-Interscience, 2006, ISBN-13 978-0-471-39333-7, <http://samples.sainsburysebooks.co.uk/9780471728238_sample_387046.pdf>
3. *System Analysis and Design,* < http://www.freetutes.com/systemanalysis/sa001_2.3-systems-that-require-engineering.html>
4. *System Component And Characteristic,* <http://www.freetutes.com/systemanalysis/sa001_3-system-components-and-characteristics.html>
5. *Classification Of System* < http://www.freetutes.com/systemanalysis/classifications-of-system.html>
6. *Information Systems* < http://www.freetutes.com/systemanalysis/information-system.html>
7. *Types Of Information Systems* < http://www.freetutes.com/systemanalysis/types-of-information-system.html>
8. *Decision Tree* < http://www.saylor.org/site/wp-content/uploads/2012/06/Wikipedia-Decision-Tree.pdf>
9. *Decision tree using flow chart symbols* <https://wiki.eecs.yorku.ca/course_archive/2014-15/F/4412/_media/decision_trees.pdf>
10. *Decision Analysis – Influence diagrams* <http://www.agsm.edu.au/bobm/teaching/SGTM/id.pdf>
11. Ross D. Shachter, *Model Building with Belief Network and Influence Diagrams* <http://stanford.edu/~shachter/pubs/AdvancesDraft.pdf>
12. *Influence diagram* <http://www.conceptdraw.com/How-To-Guide/influence-diagram>
13. *Knowledge base* <*http://searchcrm.techtarget.com/definition/knowledge-base*>
14. *Knowledge base reasoning* <*http://web.itu.edu.tr/sonmez/lisans/dogus/come444/REASONING.pdf*>
15. *Reasoning About Object Affordances in a Knowledge Base Representation* <*http://vision.stanford.edu/pdf/zhu14.pdf*>
16. *Reasoning system* <*http://america.pink/reasoning-system_3694990.html*>

17. *Reasoning system – Symbolic, Statistical* <http://www.myreaders.info/04_Reasoning_Systems.pdf>
18. *Case Based Reasoning* <https://www.cs.auckland.ac.nz/~ian/CBR/>
19. *Artificial Inteligence – Case-Based Reasoning* http://artint.info/html/ArtInt_190.html
20. *Foundations of Artificial Inteligence* <http://www.cs.nott.ac.uk/~pszrq/files/9FAICBR.pdf>
21. *Logical thinking* <http://mesosyn.com/mental1-12.html>
22. *Knowledge Acquisition for Knowledge-Based Systems: Notes on the State-of-the-Art* <http://link.springer.com/article/10.1023%2FA%3A1022662924615#page-1>
23. *Knowledge – Based Systems for Development* <http://www.tmrfindia.org/eseries/ebookv1-c1.pdf>
24. http://www.ajrhem.com/EXPERT.pdf
25. https://www.google.cz/url?sa=t&rct=j&q=&esrc=s&source=web&cd=9&cad=rja&uact=8&ved=0ahUKEwiTrtKKtY3LAhVJyRQKHaklBM4QFghLMAg&url=https%3A%2F%2Facg6415.wikispaces.com%2Ffile%2Fview%2FKnowledge-based%2Bsystems.pptx&usg=AFQjCNHt6hJeJMhtKG-4c9Qi6B4fBQesuw&bvm=bv.114733917,d.d24
26. *Knowledge engineering* <http://www.thefullwiki.org/Knowledge_engineering>
27. *Applzing KADS to KADS: knowledge based guidance for knowledge engineering* <https://www.researchgate.net/publication/2418990_Applying_KADS_to_KADS_knowledge_based_guidance_for_knowledge_engineering>
28. *CommonKADS* <http://commonkads.org/>
29. *Knowledge-Based Systems with the CommonKADS Methodology* <http://www.codeproject.com/Articles/43474/Knowledge-Based-Systems-with-the-CommonKADS-Method>
30. *Asdfasdfas* <http://www.event-b.org/A_ch1.pdf>
31. *Formal Methods* <http://www.event-b.org/A_ch1.pdf>
32. *Modeling in Event-B Szstem and Software Engineering* <http://www.cambridge.org/us/academic/subjects/computer-science/programming-languages-and-applied-logic/modeling-event-b-system-and-software-engineering?format=HB>
33. **Charles S. Wasson,** *System Analysis, Design, and Development,* Wiley-Interscience, 2006, ISBN-13 978-0-471-39333-7,

34. **Jean-Raymond Abrial,** *Modeling in Event-B System and Software Engineering, june 2010, 612 pages, ISBN 9780521895569*
35. **Zoltán Istenes**, *Formal Methods in Software Engineering,* <*http://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0052_10_formal_methods_in_software_engineering/ch05.html*>
36. **Renaro Silva, Michael Hitler,** *Shared Event Composition/Decomposition in Event-B,* <*http://eprints.soton.ac.uk/272178/5/comp_decomp_paper.pdf*>
37. *Event-B and the Rodin Platform* <*http://www.event-b.org/*>
38. *Rodin Platform and Plug-in Installation* <*http://www.event-b.org/install.html*>
39. *Event-B and Rodin Documentation Wiki* <*http://wiki.event-b.org/index.php/Main_Page*>
40. *Composing Event-B Specifications-Case-Study Experience,* <*http://www.academia.edu/2860280/Composing_Event-B_Specifications-Case-Study_Experience*>
41. *Modelling Software-based Systems,* <*http://www.loria.fr/~mery/erasmusmaynooth/lecture5.pdf*>
42. **Dominique Méry**, *he Event-B Modelling Metod,* <*http://www.loria.fr/~mery/erasmusmaynooth/n1.pdf*>
43. **Dominique Méry, Dominique Cansell**, *The invoice case studz modelling in Event-B* <*https://hal.inria.fr/inria-00000857/document*>
44. https://www.edrawsoft.com/flowchart-symbols.php
45. http://www.breezetree.com/article-excel-flowchart-shapes.htm
46. http://www.smartdraw.com/flowchart/flowchart-symbols.htm
47. Schreiber A. Th., *Pragmatic of the Knowledge Level, University of Amsterdam, 255 pages* < *http://www.cs.vu.nl/~guus/papers/Schreiber92c.pdf*>
48. *Knowledge-based Systems with the Common KADS Methology,Carlos Jemenez de parga 12 NOV 2009*

Title	Modern Theory of Information - Structured and Knowledge Based System Design
Author	Prof. Dr. Eng. Said Krayem, Csc.
	Faculty of Applied Informatics
Published by	Tomas Bata University in Zlín
1st edition	
2016	
Published electronically	

ISBN 978 - 80 - 7454 - 597 - 9

ISBN 978-80-7464-507-9