ARCHITECTURAL ANALYSIS OF METACASE
A STUDY OF CAPABILITIES AND ADVANCES

by

HOSEIN ISAZADEH

A thesis submitted to the

Department of Computing and Information Science

in conformity with the requirements for

the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

June 1997

IN MEMORY OF MY FATHER

Abstract

MetaCASE is a generic approach to computer-aided software engineering (CASE). In recent years MetaCASE tools have been developed both commercially and in research centers. Their usage domain varies from an all-purpose CASE to a software engineering teaching tool. MetaCASE is a large field with many different viewpoints. There has been no extensive effort towards formal organization of this field and consistent analysis of different research results. Some researchers claim that their version of MetaCASE provides solutions to key problems surrounding adoption and use of traditional CASE tools. However, there has been very little work towards the examination and analysis of these claims.

This dissertation presents an in-depth examination of CASE and MetaCASE. It organizes the MetaCASE field, clarifies confusing aspects, and provides an extensive bibliography of the available MetaCASE publications. More importantly, it introduces a novel framework of study consisting of an architectural definition of MetaCASE tools, a modeling of a sample method, and a grouping of existing or proposed MetaCASE tools. The practicality of this framework is established by an extensive survey of the field. Results of the survey identify the open problems in the existing MetaCASE tools and prove to be an important outcome of this research.

Acknowledgments

For the original idea of this dissertation, for numerous readings and advises, and for allowing me to work as I pleased I am deeply indebted to my supervisor, Dr. Lamb. For careful readings of various versions of this manuscript and suggestions, not always followed, I thank Laurie Ricker and Talib Hussain. For their patience in listening to my half-baked ideas and their kind comments I owe thanks to the members of Software Technology Laboratory, and especially to my dear friend Homayoun Dayani-Fard. I also thank Dr. de Caen, Dr. Chan, Dr. Martin, and Dr. Blostein for their careful examination of this research.

I am indebted to my family for their continuous support over the years, especially to my brother, Ayaz, without whose encouragements and suggestions I would not have been here. I also thank my girlfriend, Rose, for her support and understanding.

Careful acknowledgments is also made to the Queen's University and the Information Technology Research Center for the financial support of this research.

Trademarks

The following names used in this dissertation are registered trademarks of the respective companies: UNIX: AT&T, IPSYS and Toolbuilder: Lincoln Software Ltd., QuickSpec: Meta Systems Ltd., MetaEdit and MetaEdit+: MetaCase Consulting Co., Excelerator and Customizer: Intersolve Inc., 4thought: IBM, ObjectMaker: Mark V Systems., Paradigm Plus: Platinum Technology Inc., Graphical Designer: Advanced Software Technologies., Refine: Leverage Technologists Inc.

Some of the other names of tools and components used in this dissertation (identified by capital letters) may also be commercially registered. For further legal rights please refer to the cited references.

Contents

List of Figures

Chapter 1

Introduction

Building large-scale software systems is a difficult task. A discipline is required to guide teams of software developers[1] towards building correct systems that are delivered on-time and within budget. Software Engineering provides such a discipline by devising methodologies to be used throughout the software process, but they are not widely adopted. Automation of the methodologies, known as computer-aided software engineering (CASE), promised to make this discipline usable by easing the tasks of software engineers but its success is questionable [10, 46, 89].

CASE tools have been large, complex, very labour-intensive, and extremely costly to produce and adopt. They have provided less capability than they promised and what they have provided has not been viable. It is no wonder that the use of CASE tools is not as widespread as was once expected. Examination of this problem reveals that weak methodology support is a key issue. CASE tools support a fixed number of methodologies but software development organizations dynamically change their

[1] For a definition of some of the terms used in this dissertation please see the Glossary at the end.

employed methodologies. MetaCASE technology approaches the methodology automation from a dynamic perspective.

MetaCASE tools allow definition and customization of CASE tools that support arbitrary methodologies. First, a CASE tool builder specifies the desired methodology and customizes the corresponding CASE tool, then a software developer uses that CASE tool to develop software systems. An advantage of this approach is that it allows the use of the same tool with different methodologies, which in turn, reduces the learning curve and consequently the cost of building software. In this approach, many desired methodologies can be automated or modified by the developing organization which provides a dynamic capability in today's dynamic and competitive world. From another perspective, this technology can be used as a practical teaching tool considering the shortened length of development and learning times that suit academic course periods.

1.1 Motivation

Although MetaCASE is a recent technology, it has attracted the interest of many industrial and research organizations. Many commercial CASE tool vendors and research organizations have developed tools that they call MetaCASE. These vendors and organizations have their own standards and often develop their own unique terminologies. As a result, developed tools and components often appear different when they are directly related and sometimes seem similar when they are essentially different. As an example, the following different terms are used by various researchers to refer to the same MetaCASE concept: *CASE Shell* [11], *metasystem* [31], *meta-environment* [87], or the most widely used MetaCASE [62].

With a large amount of research material described using various terminologies, MetaCASE field can be mystifying for a newcomer. It is difficult to read the literature and distinguish between different concepts that appear to be similar. There are numerous publications that uniformly examine CASE tools and provide frameworks and comparison methods. Due to fundamental differences between CASE and MetaCASE, provided CASE examination methods are not applicable to MetaCASE tools. As a result, industrial organizations have a very hard time adopting and using the various MetaCASE tools.

To help MetaCASE research, frameworks specially designed for examining MetaCASE tools are greatly needed. MetaCASE frameworks must also be used in providing comprehensive surveys of the existing MetaCASE tools. There are no up-to-date and comprehensive surveys of MetaCASE tools available. Such a survey can be used by developers in evaluating and selecting suitable tools for various projects.

One of the aspects of MetaCASE research is the architectural examination of MetaCASE tools. For the purpose of this dissertation, I use Garlan and Shaw's definition of architecture as being the 'components of a system and the interconnections among them' [33]. An architectural and component-based examination of MetaCASE tools is a worthwhile endeavor since it provides an in-depth insight into the existing tools. Architectural analysis of the tools is an aspect which has not previously been explored. It allows recognition of missing components and identification of similar ones. It also permits the study of MetaCASE inter-component relationships.

A major issue that MetaCASE researchers face is to identify the open problem areas in MetaCASE tools. The existing MetaCASE research material does not provide a general list of the open problems. The efforts of various MetaCASE researchers are

focused in different directions. Some of them are in areas that may not be of any key importance to the MetaCASE state of the practice. A list of open problems can be used together with MetaCASE user requirements to identify the key issues. Thus researchers may focus their efforts to address the identified important issues.

The above observations justify research in the MetaCASE area. It is important to examine the methods and abilities of an architectural analysis of MetaCASE tools and explore their capabilities and shortcomings.

1.2 The Thesis

This dissertation examines MetaCASE tools from an architectural point of view and identifies their capabilities and shortcomings. I claim that three important open problems in the existing MetaCASE tools are:

1. *The limitation of methodology specification to data-capture*
2. *The lack of analysis facilities such as simulation/animation*
3. *The difficulty of customization and unavailability of CASE user guidance*

To demonstrate the above claims, I provide an overview of the field and introduce a framework of study which is based on a novel architectural definition of MetaCASE tools and a new categorization method. I organize MetaCASE tools and examine representative ones based on this framework. A major part of this dissertation is a survey of the existing MetaCASE tools. It is based on the available literature and my practical experiences during the modeling of a sample method in the examined tools. As the sample method, I have selected a variation of finite state machines (FSM)

with the added hierarchy to allow various experiments with the examined tools. My overview of the field, the in-depth architectural analysis of MetaCASE tools, and the included extensive bibliography of over 100 related publications provide sufficient evidence that I have identified the capabilities and open problem areas of the existing MetaCASE tools.

1.3 Organization

This dissertation is organized according to the chart of Figure 1.1 in the next page.

The current chapter provides an introduction to this dissertation, establishes the motivations behind my research, and defines the thesis statement.

Chapter 2 provides the necessary concepts and terminology. It includes a discussion of software engineering environments, CASE tools, MetaCASE technology, and the existing survey works of the area.

Chapter 3 introduces a framework for studying MetaCASE tools. This framework involves an architectural definition of MetaCASE tools, the problem statement of a sample method to be modeled, and a categorization of the existing tools.

Chapter 4 contains the review of representative tools selected from different categories noted in chapter 3. This review is based on the introduced framework and includes the modelings of the sample method (FSM).

Chapter 5 summarizes the key features of the reviewed tools, reports on my observations during the modeling of FSM, and identifies the major shortcomings of each approach.

Finally, chapter 6 concludes with the contributions and limitations of this dissertation and outlines the future research directions.

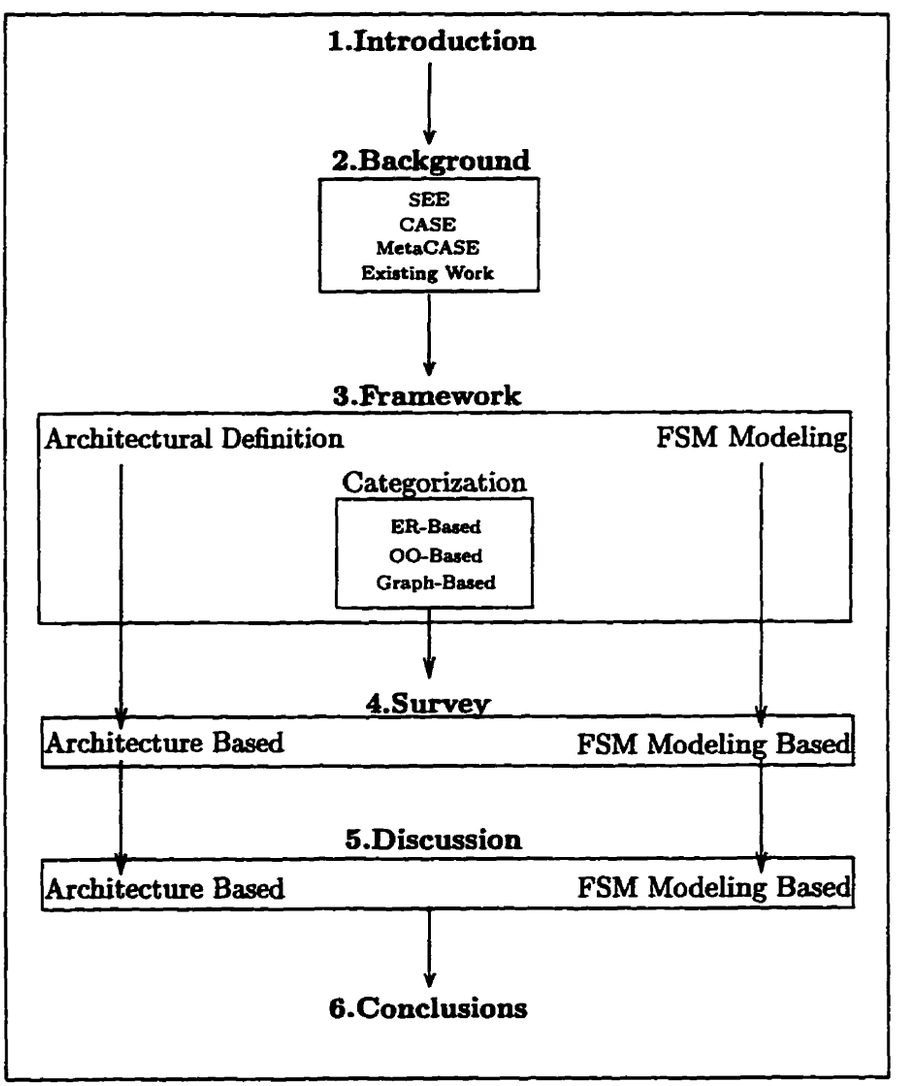


Figure 1.1: Dissertation Organization Diagram

Chapter 2

Background

In this chapter I examine the field of software engineering environments which is defined in section 2.1. The aspect of software engineering environments that I focus on is CASE. In section 2.2, various models of methodologies and CASE tools are examined and various CASE environments are discussed. MetaCASE as an extension of CASE is defined in section 2.3 where various claims are investigated. The final section of the chapter examines the existing surveys of MetaCASE and establishes the need for another review of the field.

2.1 Software Engineering Environments (SEE)

In this dissertation, I will use Charette's comprehensive definition of *software engineering environment*[1] which is the integration of "software process", "methods", and "automation" [13]. *Software process*[2] is defined to be the foundation of any

[1] Sometimes called Software Development Environment (SDE).
[2] Also referred to as Software Lifecycle.

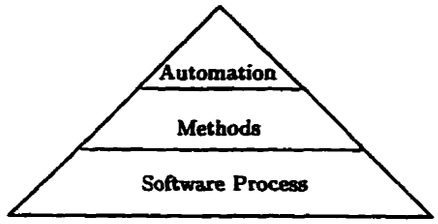


Figure 2.1: Charette's Model of Software Engineering Environment (From [13])

environment describing the sequence of events required to develop the software system. *Methods* are used to define, abstract, modify, refine, and document the software system. *Automation* is the computer implementation of the methods necessary to develop the software system. Figure 2.1 shows Charette's model of the software engineering environment. Ideally this model should be a rectangle, where all the methods required by the software process are automated. However, in practice only a few methods support a software process and only a few of those methods are ever automated. Therefore, software engineering environment is represented by a pyramid.

2.1.1 Software Process

Represented by process models or paradigms, software processes originate in the *monolithic waterfall model* [74]. This model regards development as one large process with successive phases as seen in figure 2.2. Each phase signifies activities that are distinct but whose boundaries are fuzzy. Typically, a project begins with an *opportunity* or *feasibility study* which is the recognition of a problem and the identification of possible solutions. Next is the formulation of the customer requirements and needs which produces a document called the *requirements specification*. This document is written using a formal or an informal language that may be textual or graphical

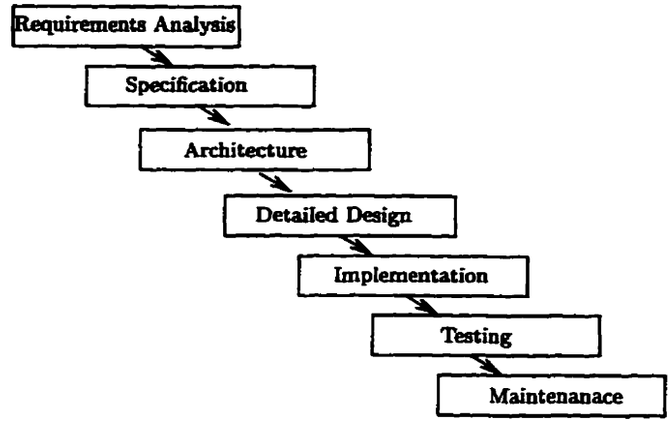


Figure 2.2: Waterfall Model of Software Process

[38]. The *design* phase starts with the planning of all aspects of the project from the labour management and budget plans to software configuration management. The actual design includes an architectural design of the software system followed by a detailed design phase. The outcome is a *design specification* document which specifies the modules of the software system and their interfaces. *Implementation* and *testing* deal with the development and verification or validation of the code. *Maintenance* refers to the evolution of the software system after the delivery, and it consumes a surprising 40 or more percent of total software effort in its lifetime [68]. A detailed discussion of the activities involved in the software process can be found in [49].

With the waterfall model, the current position of the software system in the process is easily known. However, activities such as verification and validation, that cover the entire process, are left outside this framework. Furthermore, customer feedback is not possible until the completion of the development process.

For large software systems, an *incremental process* can be used to allow quick prototyping and customer feedback at every increment of the way [82]. Figure 2.3 illustrates this incremental approach in a very crude way. The introduction of high-

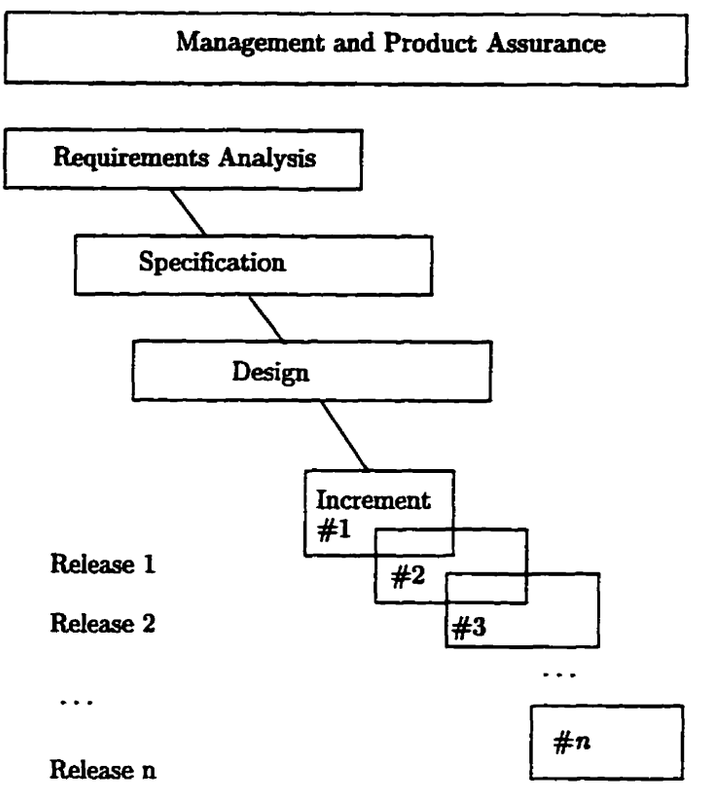


Figure 2.3: Incremental Model of Software Process (Adapted From [82])

level 4th generation languages, such as Tcl-Tk [66], now allows the prototype to be used as the basis of the developed software system, reducing the development time even further.

Today there are many software process models. In particular, the *knowledge-based* model is worth mentioning. An example of this model is a ESPRIT[3] project called ASPIS [4]. In this approach, application domain-specific knowledge is captured and used to support the software system development process. Figure 2.4 shows a model of the knowledge-based approach as outlined by Anderson Consulting [20]. In

[3]European Strategic Program for Research and Development in Information Technology.

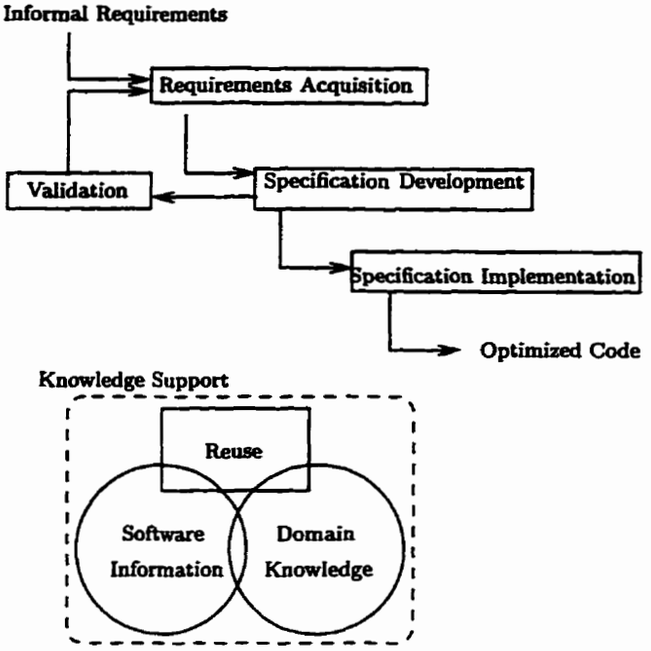


Figure 2.4: Knowledge-based Model of Software Process (Adapted From [20])

this model, different reasoning assistants (Requirements, Specification, and Implementation) communicate with the developer while using software information and domain-specific knowledge. The current state-of-practice is far from a completely automatic and artificially intelligent knowledge-based process. This is largely due to problems in managing the large knowledge base involved in this type of process.

2.1.2 Methods

A model is defined to be a simple representation of a system. It provides insight about particular instances and collections of instances by abstracting away the nonessential details while generalizing the essential ones into components [67]. Methods are explicit steps and rules that are used to develop a model. They are required by the process model to provide reliability, efficiency, modifiability, or understandability

when building a software system.

The ideal objective of a software engineer is to be able to use any method in conjunction with any other. However methods have implicit and conflicting rules in them which may not allow their integration. Hence, there has been a tremendous effort to choose, create, and combine integratable methods into methodologies that are useful throughout the software process. Therefore, methodologies can be defined as 'organized collections of methods' which is a definition that I will assume in this dissertation [13].

Examples of methodologies include: Jackson System Development (JSD) [41], Structured Systems Analysis and Design (SSAD) [99, 100], Booch Methodology [9], Jacobsen's Object-Oriented Software Engineering (OOSE or Use Cases Methodology) [42], Rumbaugh's Object Modeling Technique (OMT) [75], and Shlaer-Mellor's Object-Oriented Analysis (OOA) [83]; there are many others. An important problem is the huge variety of different types of methodologies. Each methodology has its own extensive set of specific notations, process rules, and guidelines. Therefore, learning about various methodologies and switching between them is a very time-consuming and costly process. A recent ongoing effort by three of the leading methodologists (Booch, Jacobson, and Rumbaugh) is the creation of a standard unified notation for object-oriented development [71]. However, even if this notation could be used generically, non-object-oriented methodologies like JSD are left out of this framework.

2.1.3 Automation

Having discussed the software process and the supporting methods, we now focus on the practicality of software engineering environments. Although methods to improve

the practice of software development have been available for over two decades, it is only in the last decade that the daily practice of organizations has been changed [16, 46]. For many years the labour-intensive nature of such methods outweighed the improvements they produced. Automation has changed all that. In this context, automation differs from a single tool; it is the computer implementation of methods and an integral part of the total process. Automation reduces the labour cost, increases productivity and creativity by putting the focus of the developers on the task. It also helps learning and communication and allows deployment of methods that are clerically impossible for manual use.

2.2 CASE Tools

One of the earliest and simplest attempts to automate certain aspects of the software process is the familiar UNIX *make* utility [28]. The early efforts were mostly focused on building a language-specific programming support environment with facilities for error checking, debugging, compilation, linking, version control, and other programming supports [91]. This is still an active research area of software engineering. What is changed, however, is the focus which has moved from a particular phase of software process, namely the implementation phase, to cover the entire process [24].

In the last decade, various process frameworks and enhancements to software development models have gained acceptance. In addition, methods and methodologies have been developed and practiced in software development projects, and well-established techniques have been adopted from engineering disciplines. Many software engineering support tools have been used to aid the system development throughout the entire process from the analysis and design phase to generation of

code and testing. Since these tools allow the employment of well-known software engineering methods, the term Computer Aided Software Engineering (CASE) has been coined [25].

Chikofsky notes: 'CASE is a production oriented integration technology that ties methods and tools into effective commercially viable environments' [16]. In the context of this dissertation, effectiveness of a tool refers to the satisfaction of the tool users and the quality of the tool-assisted products [12]. In general, CASE offers graphical tools with manipulation capabilities. These tools allow information capture, exchange, storage, and transformation. Information is captured using a variety of graphical and textual tools. Captured information is stored in some form of a database, dictionary, or repository. For the purposes of this dissertation these terms refer to the storage place of analysis and design objects such as requirements statements, structured diagrams, and source code.

Most often, software system characteristics can be viewed from three perspectives. The first and most studied is the static structure of the software system which refers to the data structure perspective. The second perspective deals with the function of the software system. The third perspective examines the dynamic behavior and the control in the software system [40]. CASE tools provide capabilities to represent these characteristics. In most cases, Entity-Relationship (ER) diagrams and structured textual descriptions are used to represent the static structure [15]. The functions are often represented by data flow diagrams [97]. The dynamic behavior of the software system is usually captured by state transition diagrams [68].

2.2.1 The RPG and SMP Models

A recent model defines methodologies as attempts in coding experiences that have been found to produce good software system structures [72]. This transfer of expertise can be useful if methodologies contain the following three parts: "representation", "process", and "guidelines" (RPG).

The *representation* part consists of a set of abstractions in textual or graphical forms. They are used to describe the components of a model of the system under construction. For example, the OMT methodology provides specific notations for modeling the dynamic behavior of the software system [75]. The behavior is modeled using system states and transitions that change the states. The OMT notation consists of a set of graphical facilities for representing the states (boxes) and transitions (arrows) as well as textual capabilities for labels and descriptions.

The *process* part provides the sequence of steps to be taken during development. It prescribes the systematic process involved in building software systems. In modeling the dynamic behavior of a system using OMT, the process rules indicate the definition of the states before the transitions.

Guidelines refer to the set of heuristics, rules of thumb, and general directions that guide the developer in using the representation part and enforcing the process rules. A good example of a guideline is the recognition of objects from the customer requirements specification (which is usually in a textual form). Many object-oriented methodologies have guidelines that describe the extraction of frequent nouns from the requirements specification which become the object classes of the modeled software system.

On the other hand, CASE tools are defined to have three interrelated components:

"structures", "mechanisms", and "policies" (SMP) [67].

The first component relates to *structures* such as filesystems, abstract syntax trees or graph structures, project databases, and repositories that represent the basic software artifacts and other related information. The representation part of methodologies are usually supported by the structures of CASE tools. As an example, CASE tools supporting object-oriented methodologies define structures for representing classes, properties, and associations [81].

The second component deals with *mechanisms* such as languages and facilities that operate on the structures. They may be visible to the CASE users or hidden as low level support tools. Mechanisms encode information from all the three parts of methodologies (representation, process, and guidelines). For example, most CASE tools provide textual languages that allow modeling of software systems according to the representation format prescribed by methodologies. These languages also enforce some of the process rules and guidelines of the methodologies. However, the emphasis here is on encoding the representation part more than the other two parts of methodologies.

The third component of CASE tools relates to the *policies* which are rules, strategies, and guidelines imposed on the developers. The representation and process parts of methodologies often have rules that are supported by the policies of CASE tools. For example, CASE tools may have rules that ensure the correctness of certain aspects of a diagramming representation notation or the ordering of the modeling process. In addition, guidelines prescribed by methodologies are also addressed by the policies component of CASE tools but there is very little emphasis on them. Some researchers believe that none of the existing CASE tools support the guidelines prescribed by their

underlying methodology [81].

Based on my preliminary studies of the OMT methodology, guidelines range from simple general directions to difficult and tedious tasks. For example, selecting meaningful names for classes is easily done by developers but it may be difficult to accomplish by CASE tools. On the other hand, selecting qualified names for objects can be a tedious task for developers but it is a simple job for CASE tools (by using an automatic naming convention). Some of the guidelines are technically difficult to implement at this point. An example is extracting nouns from requirements specified in a natural language and mapping them onto classes of the modeled software system. Understanding natural language is an active research area and I believe CASE research should also place more emphasis on the application of this type of research in developing better CASE tools that support more of the guidelines.

2.2.2 Various Kinds of CASE Environment

Research in CASE tools has been a continuous focus of the software engineering community. An early classification of CASE is based on the supported process phases [97]. *Upper CASE*, or front-end tools, are used during analysis and design. *Lower CASE*, or back-end tools, are used during implementation and testing. This way of looking at CASE tools changed as soon as total process support tools were introduced. An example of a CASE tool that supports the entire process is SoftDA [39]. As seen in figure 2.5, this tool has seven subsystems, two of which provide support for structured analysis and design [21]. Another two subsystems deal with detailed design and testing while the other three provide databases and reuse capabilities.

Later, CASE research shifted from ensuring that a CASE tool works to making

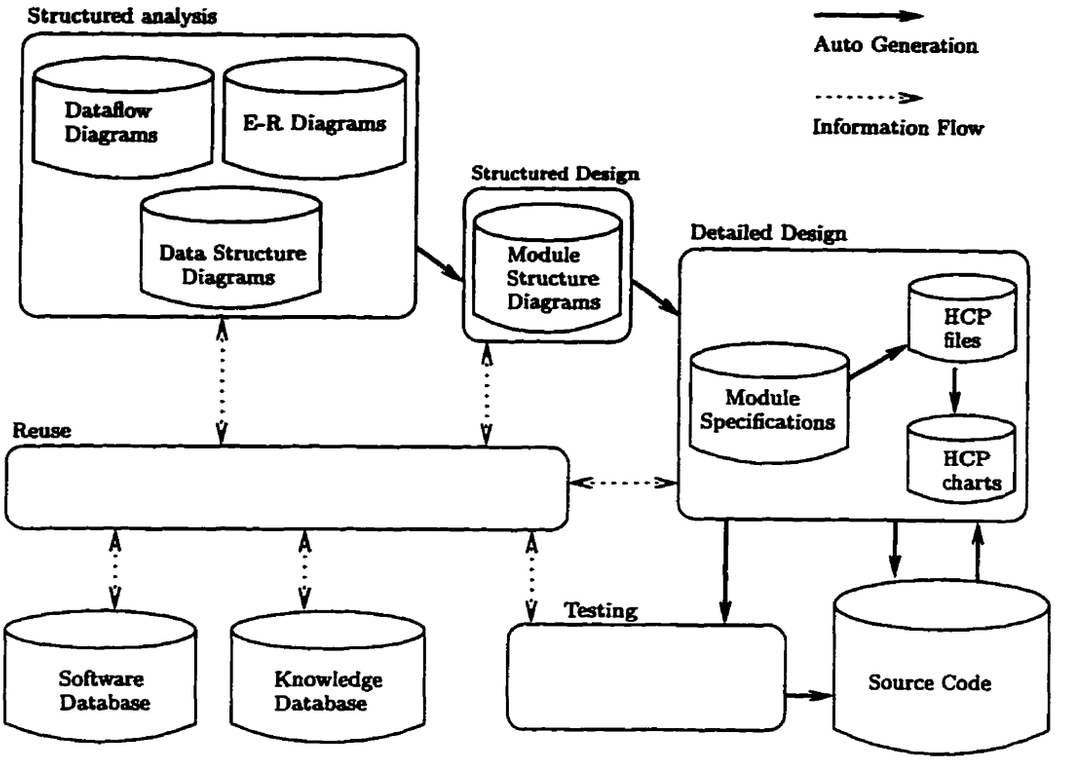


Figure 2.5: SoftDA CASE Tool Functionalities (Adapted From [39])

sure that different tools work together. Hence *integrated CASE tools* gained popularity to the point of having dedicated workshops and conferences. A major effort in this area is the categorization of tool integration into five types by Wasserman: platform, presentation, process, data, and control integrations [96]. Tool integration means following open architecture principles, writing program interfaces, using file formats or database schemas, building data sharing mechanisms, and doing all of those within a common CASE user interface.

Of other buzzwords involving CASE tools, three are worth mentioning. *Intelligent CASE tools* refer to the application of AI to CASE in the form of built-in domain-specific knowledge that may assist the developer [52]. These tools, although

promising and ambitious, lack performance when the domain is wide; they are not practical as general tools. *Repository-based CASE tools* offer enterprise-wide and project-wide repositories that integrate various tools for different phases of development [63]. They are very dependent on the employed methodology and can support development of only certain type of applications. MetaCASE is the most recent approach to computer-aided software engineering which I will discuss next.

2.3 MetaCASE Tools

Traditionally CASE tools only allowed the employment of a certain type of software engineering methodology which was fixed for the CASE users. In this approach, the developers are provided with the expressive powers that the underlying methodology permits. Furthermore, traditional CASE tools have a certain type of data dictionary or repository that is used to store all the development artifacts. Object exchange is only allowed within the compatible and offered tools. Finally in the traditional approach graphical or textual editors are usable only when the rules and guidelines of the methodology are followed. As an example, Teamwork/OOAD is heavily based on the Shlaer-Mellor methodology [32, 83].

This traditional approach is a source of great difficulty. It must be understood that every development company has its own organizational software process. CASE tool developers not only base their products on a particular methodology, they also adopt the underlying software process dictated by the methodology. This makes each CASE tool a special purpose tool that is useful for a particular type of organization developing a particular type of a software system. However, software developing organizations are different from one another and evolve over time. They change

their product lines, management styles, and manufacturing procedures to adapt to their customer needs and to maintain a competitive edge. As a result, since software processes and development methodologies change, traditional CASE tools are not able to provide realistic solutions.

Many CASE developers are now moving towards CASE tools that are capable of providing support for several different methodologies [27]. Often a group of related methodologies are supported by these tools without any real data or control integration facility. The CASE user selects a methodology and follows the enforced notation and rules. Although this approach is better than building tools that support a single methodology, it does not provide the solution to the problem of dynamically changing software development methodologies that require modifiable methodology support. One solution is to build generic CASE tools that provide capabilities for dynamic production of toolsets for different methodologies.

Customizing a tool to the needs of an organization is not a new concept. Many vendors have been providing this service to the large companies with a long software process who can afford the high cost of customization. This high cost is due to the fact that CASE tools are large, complex, and very labour-intensive to adapt. High cost solutions are not feasible for small or mid-sized companies in need of CASE tools. What is really needed is to provide the ability to capture the specifications of the required CASE tool within hours and then to generate that CASE tool from its specification as automatically as possible. This is what MetaCASE technology attempts to offer.

The term 'meta' is used by many researchers very loosely. Often the context of usage explains the meaning. Meta refers to the use of a representation facility or a

mechanism in describing a similar type of a representation facility or a mechanism. As an example, meta-data is data that describes other data like the classes in object-oriented models. Models are also meta-data that describe the modeled things rather than the actual things. To clarify any confusion, consider relational database management systems that use meta-tables to store table definitions and the tables themselves that store the actual data [75]. Meta-models are data modeling techniques such as those based on entity-relationships. They are used in MetaCASE tools to capture the underlying data models of different methodologies.

MetaCASE technology dates back to the introduction of structured editors and the possibility of customizing tools that manipulate them [1]. An example is Hotdraw, a graphics framework that allows creation of editing applications [44]. An application, such as a state transition diagram editor, has a set of figures, for representation of states and transitions and a set of tools for creating and manipulating the figures. Today, MetaCASE makes full use of structured editors and has added many more technology-specific capabilities, such as correctness checking, through the introduction and use of underlying data modeling techniques that suit software engineering applications.

In general, MetaCASE 'includes mechanisms to define a CASE tool for an arbitrary method or a chain of methods' as defined by Bubenko [11]. As a simple example of this technology, Alderson (who is associated with IPSYS, developers of Toolbuilder MetaCASE tool), describes compiler-compiler systems [1]. In his description, first the syntax of a language is described to the system using a meta language. Then the system generates syntax and lexical analysis tables which parameterize a generic compiler to create a compiler for that specific language.

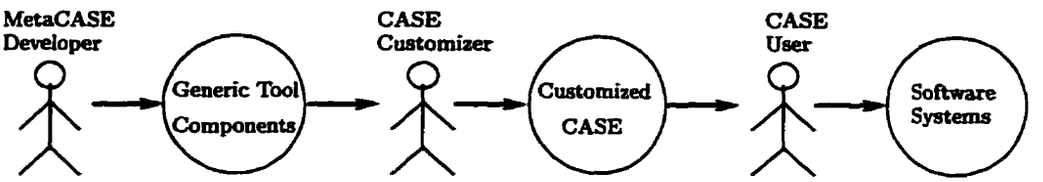


Figure 2.6: The Three Levels of MetaCASE Development and Usage

Alderson believes MetaCASE tools must have three components: a specification component, a generation component (which transforms the specification into parameters for the generic tool), and a run-time generic tool. A similar division is shared by Sorenson and Tremblay of the Metaview project [86]. In their opinion, there are three levels of specification in a MetaCASE domain: meta level, environment level, and user level. Figure 2.6 shows a high-level view of a typical MetaCASE tool with the three levels of development and usage.

At the top level, MetaCASE developers build the generic components of the tool and describe the basic structure of one or more meta-modeling techniques to be used in capturing the representation information present in a methodology. In addition, mechanisms and languages are developed that would allow definitions of structures of the meta-models (as discussed in the SMP model of section 2.2.1). This would be the specific implementation of the meta-modeling technique. The difficulty in implementing a meta-modeling technique is twofold. If the meta-model is simple then it can be used to model most methodologies but may not be sufficient in dealing with sophisticated methodologies. On the other hand, complicated meta-models are difficult to work with and might make the modeling of a simple or a new methodology too complex.

At the second level, CASE customizers use the provided meta-modeling techniques and the generic components to build a customized CASE tool. They define structures

of the particular CASE tool and encode the methodology prescribed representation and process rules (policies of SMP model). Customization of the CASE tool is done using the mechanisms and languages defined by MetaCASE developers. This process can take only a few hours which is much shorter and less costly than the traditional vendor customization approach [2].

At the last level, software developers use the customized CASE to develop software systems. An interesting approach to showing the powers of MetaCASE would be to compare development of the MetaCASE, both initially and using MetaCASE itself. This has been done in the case of Toolbuilder [29].

2.4 Existing Reviews

There exists a large amount of work on CASE tools including a number of books [59], various conference proceedings (SEE, SDE and CASE, and CAiSE), and special issues of journals and magazines. There are also many investigative and comparative papers [18, 64, 70, 94], and proposed general requirement models of CASE tools [24, 61]. However, with respect to generic environments and MetaCASE tools there are only a few publications.

MetaCASE is a relatively young field without much available analytical data about it. The little information that we have are research results of various academics. It was only recently (1995) that a conference was dedicated to MetaCASE technology [62]. In this conference, six papers reported on the experiences during MetaCASE adoption and use. They mention the cost as the primary MetaCASE selection criterion. On the other hand, MetaCASE has also opened avenues of research in automation of such tedious tasks as the collection of measurement data, metrics, and assessments

[23, 48]. This capability is due to the generic nature of MetaCASE which allows methodology independent data collections, building of generic metric engines, and quality assessments.

In the following two sections, I will examine the only two published and readily available reviews of meta-environments and MetaCASE tools.

2.4.1 Karrer and Scacchi's Review

Karrer and Scacchi's paper focuses on the broad area of the generic software engineering environments[4][45]. They categorize more than sixty of the related tools and technologies based on the type of adopted software process and the services of the environment. They divide meta-environments into the following five classes with examples from each class: environment frameworks, customizable environments, process modeling, process programming, and tool integration. The following sections provide an overview of these classes.

Environment Frameworks

Environment frameworks are those that support a set of low level services. Object Management Service (OMS) is an example of a low level service that provides for persistent objects and relationships as opposed to the traditional filesystems. User Interface Service (UIS) is another example that provides mechanisms for defining tool user interfaces and associating environment objects. The Portable Common Tool Environment (PCTE) is a typical environment framework with an Entity-Relationship-Aggregate (ERA) model based OMS and a set of UIS primitives [92]. PCTE is used

[4] Also known as meta-environments [26, 47] or environment generators[98].

as a basis for integrating tools.

Customizable Environments

Customizable environments provide high-level services such as software configuration management facilities. They also allow the users to extend and customize the provided core capabilities of the environment. The main difference between customizable environments and the environment frameworks is that customizable environments allow the modeling and customization of a small portion of the environment while fixing the rest. For example, meta-programming environments assist in creation of parsers and related tools which manipulate a particular language.

Most MetaCASE tools are hybrids of the frameworks and customizable environments. They offer low level services, such as the ER-based OMS of Metaview [86] or the graphical UIS of Toolbuilder [1], as well as customizability of a portion of the environment and adaptability to various methodologies.

Process Modeling

Karrer and Scacchi believe that the above two classes adopt particular software processes and consequently cannot be useful for other processes. The process modeling class of meta-environments attempts to overcome this problem. Here, the provided software process model can be instantiated to specify the activities, developers, resources, artifacts, and their relationships which together form the environment. I believe the generality of this process model reduces the possibility of providing focused and useful tools.

Process Programming

Process programming uses a programming language (like Ada) to describe the process and define the capabilities of the environment. It is most useful in providing programming support environments but in general may not be expressive enough to model various language-independent software processes.

Tool Integration

The final class, tool integration, deals with integrating various environment tools. This is done either by adopting a set of standards during development of new tools or by integrating existing non-standard tools. Although not entirely successful, it has been a desired objective of any meta-environment to provide integratable tools and services.

2.4.2 Martiin et al's Review

A more focused and detailed comparative review of MetaCASE is the research result of Martiin et al from the MetaPHOR project [53, 56]. The acronym MetaPHOR stands for Meta-modeling, Principles, Hypertext, Objects, and Repositories. These words are used to show the objectives of this project: to build a configurable CASE tool by applying object-oriented modeling philosophies in a distributed computing environment while using modern hypertext-based user interfaces. The result of this project is the MetaEdit tool [60].

Martiin et al's comparative review is based on a framework addressing two issues: the properties and the effectiveness of MetaCASE tools. MetaCASE tools are divided

based on their linguistic properties, functional properties, and mechanisms. Linguistics refer to the description languages used in defining conceptual structure of models and their textual or graphical representations (meta-models). Functional properties relate to the essential data management facilities like query and report definitions. Mechanisms cover CASE user interface, data communication, and operating system issues. Effectiveness of MetaCASE tools is studied in the meta-model, MetaCASE user interface, and design task areas.

The selection of MetaCASE tools for the study is based on classifying the tools according to the style of customization. Four styles are identified. Database-oriented tools use a meta-modeling language to define the methodologies, e.g., MetaPlex [14], Metaview [86], and QuickSpec [69]. Interface-oriented tools have generic graphical notations for modeling the environment, e.g., RAMATIC [5]. Extension kits involve extending an existing tool to include new meta-modeling languages, e.g., Excelerator and its Customizer [19, 27]. Finally, knowledge-oriented tools like ConceptBase have a meta-model based on logical rules [73].

Martiin et al discuss the first three of these categories and review a representative tool from each category. They compare the tools based on properties and effectiveness as outlined before. Their review contains a good deal of detail about each tool and a modeling of a sample method in the reviewed tools.

2.4.3 Concluding Remarks

In general, Karrer and Scacchi's discussion of the sixty reviewed tools is not deep. It is merely a brief categorization and listing of various tools. This work can only be used as a preliminary framework for studying the existing meta-environments.

Martiin et al's discussion contains a good deal of detail but only covers three tools. The classification of MetaCASE tools does not provide any insights to the architecture of the tools. Finally, their review is outdated and does not cover the recent developments in MetaCASE technology.

An unpublished paper by Stilwell (from the Metaview project) attempts to address some of the fundamental issues surrounding MetaCASE state of practice [88]. He examines five tools that are similar to the Metaview system. The main criterion for the comparison is the functionality provided by the Metaview system. His study, although detailed, fails to address many of the issues that are not addressed by Metaview system. Examples include the capability of tools in modeling the dynamic behavior of the software system, the process coverage, the level of provided support, and architecture related issues.

As a result of these observations, we can conclude that there are still issues surrounding MetaCASE which are not addressed by researchers. To address these issues we need to develop a framework for studying MetaCASE tools. Following the framework, a detailed architecture-based survey of the current MetaCASE tools is necessary and timely.

Chapter 3

The Framework of Study

My approach to examining MetaCASE tools is based on studying their architectures. Results of this examination would be beneficial for researchers of the field as well as the industrial developers. Analysis of the components of MetaCASE tools and their interconnections can help us identify the common parts and recognize the weak points of the tools. It can also be useful in evaluating the performance of the tools. To perform this analysis, I introduce a common architectural framework for MetaCASE tools.

Section 3.1 describes the components of this architecture in detail. In section 3.2, I define a sample method (a variation of finite state machines) to be modeled in the surveyed tools. The final section of the chapter presents my view on how MetaCASE tools should be categorized based on their underlying data modeling techniques. This categorization includes a brief discussion of various example tools and the selection of the representative ones for this study.

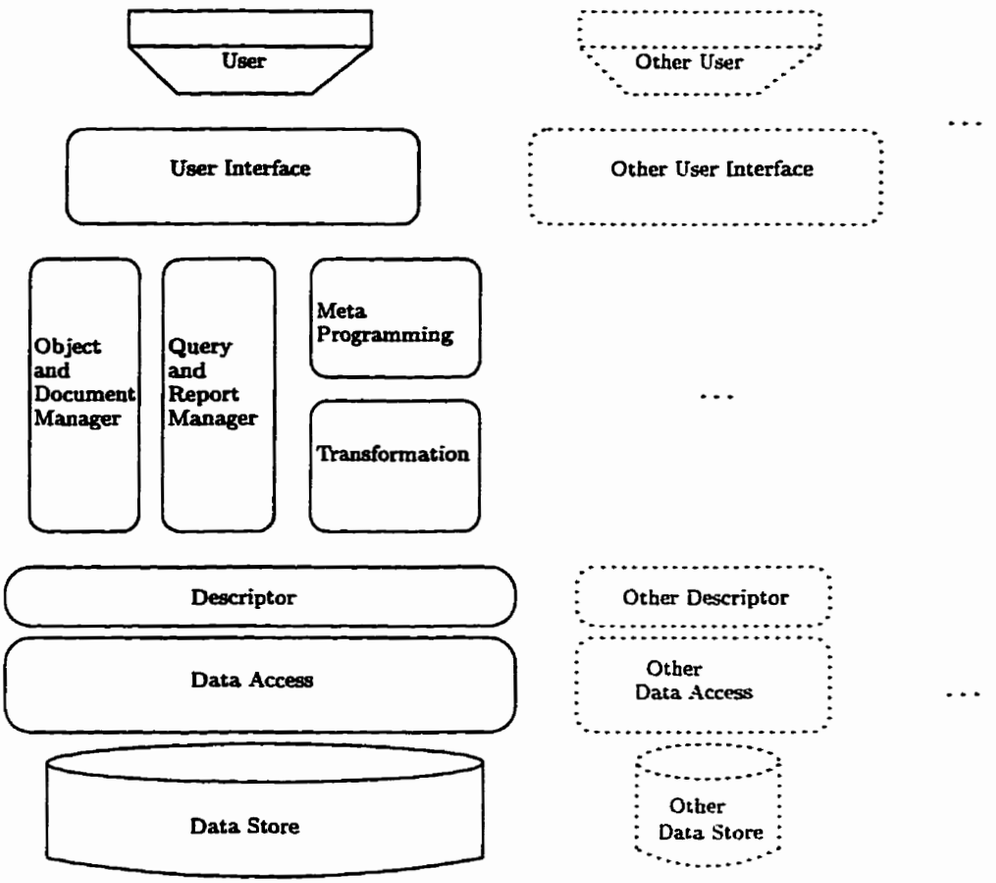


Figure 3.1: Components of a Typical MetaCASE Tool

3.1 Architectural Definition of MetaCASE tools

Figure 3.1 illustrates the typical model of the architecture of MetaCASE tools as defined in the framework of this study. In this figure, the core components are represented by boxes with solid lines; these components are parts of almost all MetaCASE tools. The other components are the possible additions and variations. As an example, a tool may be distributed with more than one data store or it may support group work with more than one user. The interface between these components is fuzzy and depends on the individual MetaCASE tool implementation. In an ideal

open architecture, each component should be replaceable by other compatible ones. An example is the 'Other User Interface' which may substitute the original one based on the application type or the user preference.

In the following sections I will describe the typical components of MetaCASE tools.

3.1.1 Data Descriptor, Access, and Storage Facilities

The data descriptor allows modeling of the methodology-related data at the conceptual level. Once descriptions are complete, models of software systems can be built, manipulated, and accessed according to the descriptor. By modeling, I mean using data structures to specify the methodology and the actual software systems. Descriptors are also called meta-models which is a term that I use frequently. Often one or more meta-modeling techniques are used in a MetaCASE to capture information about the methodology. I categorize meta-modeling techniques into the following three types:

1. **ER-based:** Some MetaCASE tools use traditional entity-relationship modeling techniques where concepts are represented as entities with relationships among them. This allows easy modeling of most of the concepts in structured analysis and design methodologies. Often the original implementation of these techniques in various tools does not allow the modeling of hierarchy of entities and the graphical representations of the artifacts. Therefore, the implementation has been extended to provide these functionalities but the success of these extensions is questionable (more detail in section 4.1).

2. **OO-based:** Other tools use more recent object-oriented modeling techniques. They differ from ER-based techniques in allowing object instantiation as well as the use of object modeling notations. Implementations of these techniques in MetaCASE tools permits easy modeling of most of the concepts in OO-based methodologies but requires extensions to include concepts such as aggregation and graphical representations. Implemented OO-based techniques often contain too much detail and lack a clear and predefined structure and functionality (details in section 4.3).

3. **Graph-based:** The third category of meta-modeling techniques involves hierarchical graphs. These techniques are more formal (set-theoretic or logic-based). They have visualization and querying capabilities with associated mechanisms and languages. Since the concepts of methodologies are often graph-like, these meta-modeling techniques match the natural form of the concepts and, therefore, are more expressive. In addition, implementations of these meta-modeling techniques include logical querying languages that provide predefined structures with more semantic powers (as described in section 4.4).

The storage of data involves low level data management issues such as file locking, concurrency control, data sharing, and mapping of data from physical to conceptual level. A data management system is often used to control the access of data in the storage and provides functionality such as locking that would allow multi-user access. Another issue, addressed by this component, is network computing and use of distributed databases.

There are further issues that need to be addressed by any MetaCASE tool. Examples include software evolution and reuse. Evolution refers to concepts such as

re-engineering, reverse engineering, and design recovery of the existing software systems. Reuse is concerned with providing knowledge storage and analysis facilities including graphical user interface (GUI) libraries or program-code components. These issues arise during the customization of the CASE tool and more importantly while using the CASE tool. Often they are methodology dependent; thus the implemented meta-modeling technique must provide the appropriate capabilities.

In reviewing the MetaCASE tools, I examine details of the meta-modeling techniques. I also briefly discuss some of the data storage and software evolution facilities of each tool but a detailed study remains to be conducted in the future.

3.1.2 User Interfaces

User interfaces make up the front-end of the MetaCASE tool. Often a combination of graphical and textual interface is used as a bridge between the MetaCASE system and the MetaCASE user. Since many of the concepts in software engineering involve graphical elements, there is a strong need for a GUI capability. Use of GUI facilities may help easy visualization and manipulation of graphical elements. However, some non-graphical software engineering elements are better understood and manipulated using a textual interface. Hence a combination is often desired.

Another UI factor to consider is the type of the MetaCASE user. MetaCASE is used in two different levels as tools for building information systems (CASE tools) as well as software systems (products). Information engineers at one level build meta-models of methodologies and produce customized CASE tools which are then used by developers in building software systems. Users at these two levels may have differing computer knowledge and expertise. A good user interface should provide tools and

services appropriate to its users. A current popular tendency in MetaCASE UI design is a common user interface. This is based on the assumption that information engineers, or CASE customizers, and the software engineers, or CASE users, are the same people. Here, developers customize their own CASE tools and use the customized tools to build the software systems.

The MetaCASE user interface must provide consistent facilities to reduce the learning curve. A MetaCASE user should be able to find and perform the common commands of different tools in a common and consistent way. Differences may arise, based on the requirements of different methodologies, but these should already be understood by the user. I believe this aspect of MetaCASE is very important. A consistent and easy to learn MetaCASE user interface can lead to customizing and using customized tools in a shorter time which can result in increased productivity.

3.1.3 An Object and Document Manager

The object and document manager is the direct back-end of the MetaCASE user interface. With this facility, graphical or textual objects are assembled for viewing or manipulation by the MetaCASE user through the user interface. Interaction with the MetaCASE user is the responsibility of the user interface but the management and organization of the artifacts is the responsibility of the object and document manager.

Documents are the artifacts created during the software engineering process. Examples of such documents include data flow or module specification diagrams, test plans, user manuals, and source code. Organization and management of such documents involves gathering all the related information from the data store, interpreting

and applying the enforced policies of the methodologies from the descriptor, and allowing MetaCASE user manipulations. In producing a document, the tool searches for the related artifacts, collects them while enforcing the descriptor's rules, creates the desired format by applying possible transformation techniques, and displays or outputs the result in the appropriate format.

A form of management process support falls within the responsibilities of Object and Document Manager. Process support is a way of modeling the activities involved in a software engineering process. An activity refers to an information system development or managerial task (or a composition of tasks). It uses, or produces, a deliverable or acts as a managerial event. Events are often significant dates of the project. Such a support is project-tailored and involves all the managerial aspects of the project. Therefore, implementing a management process support system requires a rule based triggering mechanism in the database to deal with the events, a version control for all the deliverables, and navigational as well as viewing capabilities to provide decision support.

3.1.4 A Query and Report Manager

The query and report managers deal with MetaCASE user queries and retrieval of respective information satisfying a query or a report request. Queries vary from simple requests of information about states of certain documents to more complicated inquiries about the methodology. Reporting the results of queries may involve searching and navigating around the documents, performing logical and arithmetic calculations, and displaying the results.

Queries may involve requesting information about the methodology during the

customization of the CASE tool. Often the CASE customizer would inquire about the consistency and correctness of the meta-model with respect to certain logical rules and conditions. This may require a meta-model which can be mathematically manipulated.

Queries may also be about the software systems under construction. The simplest query would be to ask about a certain input item or object of the software system. This involves searching and navigating through the documents. A more complicated query may deal with checking if the software system can be in two different states at the same time. This involves a simulation of the dynamic behavior of the software system and a search through the possible states [37]. Some of these queries are quality assurance issues such as consistency, completeness, and conformance to the rules of the adopted methodology.

The above facilities identify errors and problems in the form of reports. In addition, various other statistical information about the models can be generated using implemented metrics and produced as reports. The formatting and production of these reports are managed by the report manager.

3.1.5 Transformation and Meta-programming Tools

Transformation and meta-programming tools refer to the activities involved in 'lower CASE'. The ideal is to have the customized CASE automatically transform the requirement specification into the source code. Transformations should be mathematically provable, in which case, the source code would provably satisfy the requirements. However, in reality this is rare. Very few MetaCASE tools offer automatic code generators and those that offer this feature require extra information from the developers

to be input and checked manually at earlier stages in the process.

According to the used methodology, transformation may involve the automatic generation of one or more of the documents (e.g., the module specification document) from the previous ones (e.g., the data flow and state transition documents). Often the degree of automatic code generation is dependent on the degree of formality of the originating documents. Methodologies prescribing a formal specification of the design would provide transformation mechanisms that, if employed, can lead to the automatic generation of the code.

Meta-programming refers to the availability of programming support environment ranging from the choice of the language to the parsers, compilers, and debuggers specific to the language. In most cases, automatic transformation mechanisms provide the programmer with the module interfaces and program headers. Then, according to the specified behavior, the final coding is done and tested against the requirements, all within the rules of the specific captured methodology.

3.2 Sample Method: A Variation of FSM

To help us understand more of the issues involved in MetaCASE tools, I have selected finite state machines (FSM) as a sample modeling and experimentation method. This method is selected since it is sufficiently powerful to provide a realistic example and can easily be associated to a formal syntax and semantics. This method has roots in automata theory and is used in modeling of behavior in the construction of software systems [34]. Focusing on the dynamic behavior is a different approach than the other researcher's approach which is focused on modeling the data structure or functions of the software system (e.g., data flow diagrams) [56, 88].

The importance of this method is the role that modeling of behavior plays in MetaCASE tools. In order to simulate, analyze, or animate behavior, a MetaCASE tool must allow behavioral modeling. Therefore, experimenting with finite state machines can help us recognize behavior-related details and shortcomings in MetaCASE tools. In addition, it can help in understanding the expressiveness and usability of the implemented meta-modeling techniques used by the examined MetaCASE tools.

In the following few sections, I will discuss the various selected concepts and constraints of FSM. The same organization is used in the next chapter during the modeling of the FSM in various surveyed MetaCASE tools. First, I discuss the basic primitives followed by 'transition' relationship. Next, I introduce an added concept called 'hierarchy' (or 'depth') followed by various simple and complex constraints. Finally, I comment on the meta-modeling technique of each tool based on my observations.

3.2.1 Primitive Concepts

Based on classical automata definitions and Rumbaugh's usage, an FSM is represented by a state transition diagram (STD) [75]. It has two basic primitives: states and events. A *state* of a system is an abstraction of attribute values of the system which determines its behavior in response to input events. An *event* is informally defined to be an occurrence of something at a point in time which has an effect on the state of the system.

In defining the basic primitives, I use generic primitives and inheritance relationships. Inheritance allows definition of more specific elements from the generic ones

by maintaining the structural similarities and allowing the definition of added characteristics. As a result of this type of modeling, I examine inheritance capabilities of various implemented meta-modeling techniques.

Other concepts involved in FSM are transitions, attributes, and special initial or final states. I discuss transitions in the next section. Attributes are descriptive concepts relating to primitives. Initial state refers to a state where transitions of the FSM begins. Final states are sets of states with special significance after a sequence of transitions has taken place. Input and output events act on the initial and final states. To keep the method simple, I omit concepts like the input and output events from our study.

3.2.2 Transitions

Transitions are relationships among the basic primitives (states and events) which determine the next state of the FSM. In particular, given a state and an event of an FSM, the system transits into at most one next state of that FSM.

Figure 3.2 shows an STD of a simple pop-up menu with 'Idle' and 'Menu Visible' states (shown as boxes) and three transitions (represented by arrows). Once in 'Idle' state, a 'Button Down' event will take the system to the 'Menu Visible' state and a 'Button Up' event will take it back. In the 'Menu Visible' state, the 'Cursor Moved' event does not change the state (and it remains in the 'Menu Visible' state).

Most MetaCASE tools provide textual or form-based languages to define simple objects, like states, and more complicated ones, like transitions. I model the FSM concepts using the meta-modeling techniques of various MetaCASE tools. However, in order to keep it simple, I omit the definition of other associations and graphical

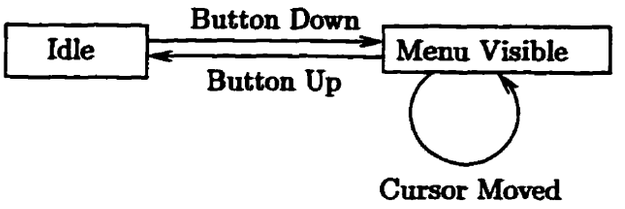


Figure 3.2: State-Transition Diagram of a Simple Pop-up Menu

representations which involve diagrams and graphics library calls.

3.2.3 Hierarchy

An important concept absent from the simple flat FSM is hierarchy. This concept allows modeling of complex software systems using a structured hierarchical approach. To investigate how well MetaCASE tools model hierarchy of objects, I am adding this concept to the definition of our sample method. Hierarchy in this context refers to a hierarchical arrangement (generalization) of states and events in an FSM. In particular, any state of an FSM can consist of another FSM. This leads to levels of abstraction and explosion of states of one level into lower levels.

Diagrammatically, a nested state of an FSM can be represented by a link from that state to the lower level FSM. However, this may lead to consistency control problems which is discussed in the examination of the consistency control capabilities of each tool (within the object and document manager components).

3.2.4 Simple Constraints

Modeling FSM involves defining constraints that compliment the above concepts. In most MetaCASE tools, transitions are often modeled as simple relationships. Hence, there is an additional need for defining constraints such as the cardinality of the

participants in a relationship and unique identification measures. A good example is to ensure that the states and events participating in a transition of an FSM are the states and events of that FSM. Constraints of this type are simple to model and even implicit within the definition of primitive concepts. Tools that use visual meta-modeling techniques have more capabilities in defining implicit constraints. During the modeling of FSM, I examine some of the implicit or explicit simple constraints provided by the meta-modeling techniques.

3.2.5 Complex Constraints

A more complex constraint is *reachability* which refers to the existence of a path of non-zero transition from a state to another state. When the path consists of a single transition then the states are *directly reachable.* This constraint relates a state/event pair to a unique next state. In general, all the states of an FSM should be reachable from the initial state and they should be able to reach some final state via various transitions. These concepts are called *reachability from the initial state* and *reachability of the final state.* I examine the direct reachability and reachability from the initial state as examples of complex constraints. The reachability of final states is a similar constraint.

3.2.6 Observations

In the previous few sections, I have discussed the necessary definitions of my selected sample method (FSM). Observations section will contain my observed comments on the meta-modeling techniques of each tool during the modeling of FSM in the next chapter.

3.3 Categorization and Selection of Tools

Based on the architecture discussed in section 3.1, I now classify the MetaCASE tools and select representative ones for the study. I use the underlying data structure representation and meta-modeling techniques used by MetaCASE tools to classify the tools. This approach is sound and reasonable since a MetaCASE tool with certain desired behavior requires suitable meta-modeling techniques. Some of the problems caused by a mismatch in this area are: data redundancy, poor performance, increased complexity of software, and lack of functionality [81].

Most of the functionality provided by a MetaCASE is dependent on its meta-modeling techniques; fundamental limitations and capabilities of a tool can be traced back to the implementation of the meta-modeling technique and its characteristics. As an example, if a MetaCASE customized CASE tool does not allow modeling of the nested objects in a software system then it is likely that the implemented meta-modeling facility fails to model the hierarchy of the objects.

As discussed in section 3.1.1, there are three major types of meta-modeling techniques: ER-based, OO-based, and graph-based. The implementation of almost all the tools extend and modify these basic modeling techniques to facilitate modeling of more complicated software engineering artifacts. Examples include set-theoretical concepts, such as hierarchy, as well as diagrammatic objects and graphical management capabilities. In the following sections, I provide discussions of example tools from each category and select representative ones for the survey.

3.3.1 ER-based Tools

Some of the MetaCASE tools that use an ER-based meta-modeling technique, or an extension or variation of it, are Metaview [30], MetaPlex [14], Socrates [93], Totem [87], Toolbuilder [1], RAMATIC [5], and Customizer [19].

Among the ER-based tools Stilwell has reviewed the first four academic research tools [88]. Based on my research and Stilwell's results, all four of these tools are research tools with similar architectural concepts (e.g., object managers and transformation systems). Their Meta-modeling technique is a text-based meta language based on ER modeling with extensions to facilitate inheritance, hierarchy, and graphical associations. A single repository holds all the modeled information which is controlled by a data management system. These tools have focused on various issues. For example, Totem appears to have a well integrated architecture with facilities for software evolution. MetaPlex makes use of a knowledge-based object manager while Socrates allows easy modeling of software development processes. In general, they all provide the same level of functionality and suffer from similar problems. Examples of these problems are the representational problems (due to various implemented extensions to their meta-models) and methodology modeling difficulties (due to lack of good user interfaces).

The RAMATIC and Customizer tools have been reviewed by Martiin et al [56]. They are commercial tools with many similarities to the Toolbuilder system. Their most powerful aspect is the employed software developer user interfaces. RAMATIC is graphics-oriented. It uses generic routines, that can be associated with symbols. Its functionality is very limited. Customizer is not a real MetaCASE tool but only an extension on the Excelerator CASE tool that allows some degree of customization.

Much of the functionality of Excelerator is fixed and only the data-capture components are customizable. For example, various diagramming editors can be customized using Customizer and added to Excelerator.

As a representative of research tools, I review the Metaview system. This system was selected because it involves a project with a long history of research and a wide variety of publications. Its architecture is very clear and can be a good representative of the ER-based tools. Furthermore, the employed meta-modeling technique is most developed and comprehensive. I also examine the Toolbuilder system. This tool was selected since it provides the most functionality among the commercial ER-based tools. It is a recent tool that is used in numerous research and commercial projects, and it has never been reviewed before.

3.3.2 OO-based Tools

Examples of tools that use an OO-based or a variation of this meta-modeling technique are MetaEdit [84], QuickSpec [69], Phedias [95], and ConceptBase [43].

QucikSpec has been reviewed by Martiin et al [56]. It was originally developed to form the front-end of a CASE tool family called 'Meta Systems'. It is called MetaCASE because it has meta-modeling capabilities. Those capabilities, however, are limited to certain aspects of the Meta Systems that are customizable. The majority of the functionality of the Meta Systems is fixed. Phedias is a fairly recent attempt in MetaCASE technology but in many ways it is very similar to MetaEdit and RAMATIC. ConceptBase is basically a deductive object manager for MetaCASE tools. It amalgamates an OO-based meta-modeling technique with knowledge-based features to facilitate deductive reasoning on the data storage of MetaCASE tools.

Among the OO-based tools, I will examine MetaEdit which is the most complete and comprehensive example of OO-based MetaCASE tools. It is a popular commercial tool used by many organizations, and it has distinct methods engineering capabilities.

3.3.3 Graph-based Tools

The final category of MetaCASE tools, Graph-based, is a less-explored category. The meta-modeling techniques of the tools in this category are attempts to overcome the difficulties with the implementation of the OO-based and the ER-based techniques. Graph-based meta-modeling techniques are claimed to be expressive enough to model all the aspects of methodologies and yet provide clear querying and manipulating capabilities. Examples of tools in this category are a prototype tool by IBM called 4thought [79] and a proposed tool by Joint Research Center for Advanced Systems Engineering (JRCASE) called CASEMaker [81]. To investigate the claims about this category, I will examine both of the tools.

3.3.4 Other Tools and Components

There are MetaCASE tools and Components that I have not categorized or examined in my research. For example, ObjectMaker [65], Paradigm Plus [90], and Graphical Designer [36] are commercial tools with little or no available documentation and Hotdraw, Hardy, Refine and Goodstep [90] are merely components that can be used in construction of MetaCASE tools. Hence, this review does not include them. An extension to this research is the examination of the remaining tools and the related components based on the framework built in this dissertation.

	Metaview	Toolbuilder	MetaEdit	4thought	CASEMaker
Category	ER-based	ER-based	OO-based	Graph-based	Graph-based
Use	Research	Commercial	Research/ Commercial	Research	Research/ Commercial
History	Long	Short	Long	N/A	N/A
Number of Papers	High	Low	High	Average	Average
Number of Features	High	High	Very High	Average	Low
Issues Addressed	Important	Important	Important	Very Important	Very Important
Previous Reviews	Two	None	One	None	None

Table 3.1: MetaCASE Tool Selection Criteria

3.3.5 Concluding Remarks

Table 3.1 is a summary of my selection criteria of the representative tools. I selected representative tools from the three categories of MetaCASE tools as identified in this dissertation. Selected tools are from both the research and commercial communities. Most of them have a long history of usage with numerous publications and developed features. The only exceptions are the Graph-based tools (4thought and CASEMaker). I have selected these since they address some of the most fundamental issues in MetaCASE research. Most of the selected tools are not reviewed before and those with previous reviews are selected for comparison reasons.

My classification of MetaCASE tools differs from the existing three classifications in more than one way. One existing classification is based on the scale of the software system that the CASE tool is capable of supporting [67]. Another classification,

Karrer and Scacchi's approach, is based on the provided services or the type of the adopted software process. These two approaches are broad classifications of more than just MetaCASE tools and, therefore, not practical in MetaCASE studies. The last classification, Martiin et al's, is designed for MetaCASE tools. However, it is based on the properties and effectiveness of the tools and, as I discussed in section 2.4.2, has various shortcomings. A prominent one is the lack of architectural insights, which is a focus of my survey.

Chapter 4

The Survey

This chapter contains the review of the representative MetaCASE tools selected in section 3.3. Selected tools are: Metaview, Toolbuilder, MetaEdit, 4thought, and CASEMaker. The review follows the framework developed in section 3 by identifying and examining the typical components of the architecture of each tool. It also includes the modeling of the selected sample method (FSM) in the reviewed tools and my observational comments.

4.1 Metaview System

Metaview project is a joint effort between the universities of Alberta and Saskatchewan in Canada and dates back to McAllister's 1988 Ph.D. thesis [58]. A result of this project is the Metaview MetaCASE tool which is primarily a research tool. Currently, this research continues in areas such as incorporation of the 'methodology knowledge' into MetaCASE tools and better representation of 'aggregation' in the meta-modeling techniques employed by Metaview [85].

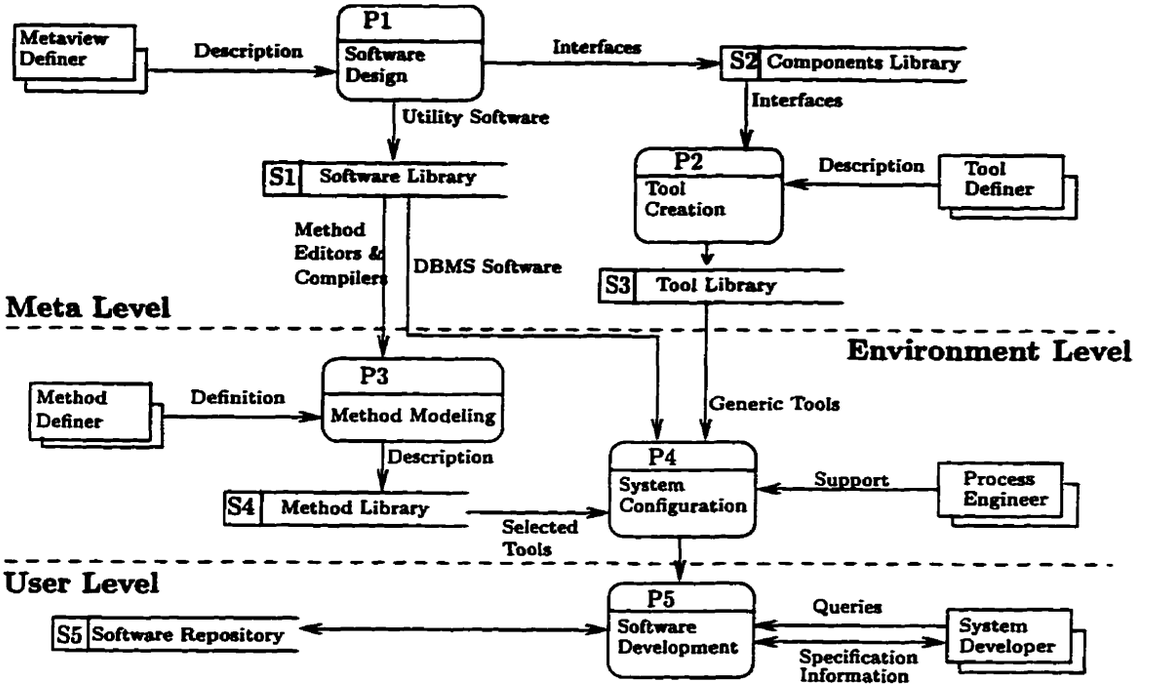


Figure 4.1: Metaview Architecture (Adapted From [30])

Figure 4.1 outlines the architecture of Metaview system using a data flow diagram. Based on data flow diagram conventions, boxes are labeled 'P' for processes, 'S' for data stores, and no labels for the CASE customizers and software developers interacting with the MetaCASE tool. The arrows show the flow of information between the boxes. This diagram divides the system into three levels of description (meta level, environment level, and user level) which were described in section 2.3.

In the meta level, there are two types of software developed by two types of developers. The Metaview Definer describes one type of software which forms the Metaview software library (S1) and the Tool Definer specifies the required tools to form the Metaview tools library (S3). The main difference between the two types of software is that the programs of the Metaview software library (S1) are not directly

visible to the System Developers. In the environment level, programs of software library (S1) are used by the Method Definer to form the methods library (S4). Furthermore, the Software Process Engineer makes use of all three libraries (S1, S3, and S4) to configure the generic components and produce the customized CASE tool. The System Developer in the user level uses this tool to produce software systems (S5). In the following sections, I will describe the components of Metaview system based on our architectural definition.

4.1.1 Data Descriptor, Access, and Storage Facilities

The meta-model defined for Metaview is EARA/GE (Entity, Aggregate, Relationship, Attribute, and Graphical Extensions). It is designed to describe various software development methodologies. The entities, relationships, and attributes are the familiar concepts from the ER modeling technique. Metaview extends these concepts to include specialization, aggregation, and some generalization elements. These extensions allow data abstraction and modeling of the hierarchical nature of the artifacts and components of software systems during the engineering process. Aggregation permits representation of heterogeneous collection of entities and relationships as a single entity. Furthermore a graphical extension to EARA model supports definition of graphical representation of software artifacts [30].

Examples of the methodologies modeled in EARA/GE are: Structured Design [100], Higher-Order software [57], and recently the OMT methodology [75, 102]. The concepts of these methodologies are defined using the *Environment Definition Language* (EDL). This language allows textual modeling of many of the concepts present

in the methodologies. EDL does not provide modeling capabilities for defining constraints on the objects of the EARA model. These constraints enforce completeness and consistency checking on the objects. In Metaview, they are defined using the *Environment Constraint Language* (ECL).

Metaview is able to model a complex OO methodology (OMT), however, it has limitations as described by Stilwell [88]. Among the limitations is the impossibility of creating two relationships with the same participants. Furthermore, the aggregation mechanism, which is an important feature of the EARA model, fails to work well with the graphical extension.

Data storage facilities of Metaview are the various libraries of components, tools, softwares, and methods. The components and tools libraries (S2 and S3) support the user interface facilities. The software library (S1) provides the database management and utility functions to be used by the software process engineer. The methods library (S4) contains the descriptions of the software development methodologies. The software repository (S5), also referred to as specification database, contains the developed softwares. At the user level, the *database engine*, which is a Prolog-based database management system, manages the specification database.

4.1.2 User Interfaces

Metaview is an X-windows based multi-user system. Besides the Metaview Definer and the Tool Definer, there are three types of users, not necessarily distinct, with three different interfaces of Metaview.

At the environment level, the Method Definer interfaces with the tool by providing the EDL and ECL source codes and creating the graphical objects with a primitive

graphical editor. The EDL and ECL codes are compiled into tables which describe the definitions of the objects and predicates which enforce the constraints. Currently the compilers cannot handle the graphical definitions or the graphical constraints of the objects. Creating graphical tables and constraints is done manually.

Configuration of the CASE tool, to be used by software developers, is done by the Software Process Engineer who selects one or more methods and configures the generic tools. Currently, the only generic tool is the *Metaview Graphical Editor* (MGED). The database engine can also be customized and a *project daemon* may be created to handle consistency and completeness checkings. Configuration requires a large amount of knowledge about the methodologies, tools, and the system under development. Most of the work is done manually and there are no help systems available.

At the User level, the project daemon initializes and controls the server. It allows multi-user access and arranges the appropriate locking mechanisms. Software developers access and modify the specification database via an interface based on the MGED tool. This tool allows graphical editing of the objects of the specification database. Operations such as creation, deletion, and modification of objects, edges, and icons are supported using a variety of graphical editing facilities.

4.1.3 An Object and Document Manager

Tool components and Metaview tools libraries (S2 and S3) are the back-end libraries to the user interface facilities of Metaview system. These libraries are collections of C++ routines used in building the MGED graphical editor.

One of the capabilities of Metaview at the user level is to allow automatic transformation of objects between two supported representation formats: textual and graphical (*local transformation*). An open research issue is the automatic transformation of diagrams created using MGED to a Prolog form compatible with the Prolog-based specification database.

Metaview supports *global transformations* of documents between different formats, at different stages of development, or even between different methods. This is done using a language called the *Environment Transformation Language* (ETL) which I will discuss later.

Among the software in the software library is the project daemon which is a server program that provides a uniform interface between the tools and the database engine. This program is responsible for handling the concurrent accessing of the database from the tools. It also manages the specification documents and enforces consistency.

Metaview has an executable process model specification mechanism. This mechanism is based on an active database model. A special language, which is designed and hoped to be implemented in future, allows describing the process models, after which, automatic triggering rules act on the database and provide the necessary interaction with the CASE user [6].

4.1.4 A Query and Report Manager

Metaview stores the methodologies in the methods library. During the definition of these methodologies, there are no CASE customizer supporting systems and no querying is possible about the correctness or consistency of the model.

The software specifications are managed by a Prolog-based database that allows

simple queries about the states of the objects. These queries involve searches and tracing of the objects and information requests using the embedded metrics. Metrics are defined explicitly at the environment level during the definition of the methodologies. For example, metrics for DFDs and structure charts are embedded in the EDL and ETL of those methods [8]. They are often simple count functions that give the number of input and output data elements, processes, or terminators.

The reporting is done using screens that display the metrics for each object or group of objects. Furthermore, rules specified in ETL allow the Metaview system to calculate certain metrics and, based on the result, give some assistance to the CASE user. As an example, the depth of the decomposition in the DFDs can be analyzed based on the number of the data elements at each level. There are no other reporting facilities available at this point.

4.1.5 Transformation and Meta-programming Tools

Metaview supports semi-automatic transformation of formal documents from one format or methodology (like DFD of structured analysis) to another format or methodology (like structure chart of structured design). This form of global transformation is possible since both methodologies are modeled with the same language (EDL) and the transformation rules are expressed formally using the set-theoretic constructs of ETL [7].

At the environment level, the specific ETL code of a particular transformation is explicitly defined. Later, developers specify a software system based on one methodology and use those ETL definitions to automatically transform them into the first-cut of the software system in another methodology. Finally, developers must manually

improve, optimize, and complete the first-cut to a full and valid software system in the other methodology.

Metaview focuses on the 'upper CASE' and provides tools for the analysis and design stages of the software engineering process. It does not provide any means of transforming the design into source code. Therefore, meta-programming or other 'lower CASE' activities are absent.

4.1.6 Modeling FSM in Metaview

Based on the EARA/GE meta-modeling technique, Metaview provides languages (EDL and ECL) to model basic concepts of methodologies and the associated constraints. In this section, I will use these languages to model the selected sample method (FSM) which was discussed in section 3.2. Once methods are modeled in EDL, graphical concepts are associated with the objects of EDL using the GE capability and the executable databases are produced.

Primitive Concepts

The primitive concepts of FSM are the states and events. The EDL code on top of the next page illustrates these concepts defined as the entities of EARA model. In this model, the FSM_entity is a generic entity with the name and the id attributes. The subtypes of the FSM_entity are the state and the event entities The state entity can be an initial_state or a final_state. Subtypes inherit the structure of FSM_entity and add extra characteristics. As an example, event has an additional data attribute (event_data) which is of type text (sequence of string).

```
ENTITY_TYPE FSM_entity GENERIC
   ATTRIBUTES (id: string)
   ATTRIBUTES (name: string);

ENTITY_TYPE state IS_A FSM_entity
   BECOMES substate;

ENTITY_TYPE initial_state IS_A state;

ENTITY_TYPE final_state IS_A state;

ENTITY_TYPE event IS_A FSM_entity
   ATTRIBUTES (event_data: text);
```

Transitions

Transition is a relationship between states that take the current_state and the next_state roles. Another participant of a transition is the event entity. Transition is described by the following EDL code:

```
RELATIONSHIP_TYPE transition
   ROLES (current_state, event, next_state)
   PARTICIPANTS (state, event, state);
```

This model of FSM is not yet complete. Various constraints need to be enforced. These constraints will be discussed later in the section.

Hierarchy

The hierarchical concept of state is defined with the 'BECOMES' keyword of the state entity. It is further defined to be of type aggregate representing the explosion of a state into the state, event, and transition components. The state component in turn

may explode into further substates. The following EDL code describes this concept:

```
AGGREGATE_TYPE substate
   ATTRIBUTES (name: string)
   COMPONENTS (state, event, transition);
```

Simple Constraints

Our selected simple constraint is to ensure that the states and events involved in a transition of an FSM are the states and events of that FSM. This constraint is shown by the following ECL code:

```
CONSTRAINT (1) ownership (a: substate)
   ALL t FROM (transition @ a)  SATISFY
      t->current_state IN a
      t->event IN a
      t->next_state IN a;
```

The above constraint defines a to be a substate (FSM) and t to be a transition of a. It then ensures that the entities participating in the transitions of a are components belonging to a.

Complex Constraints

The direct reachability constraint ensures that given a current_state and an event, the system will transit to exactly one next_state. The ECL constraint number two on top of the next page is the definition of direct reachability constraint. Here t0 and t1 are transitions of a substate a. The body of the inner 'ALL' statement is a condition that finds all the transitions with their current_state and event equal to the current_state and event of a given transition (t0) and their next_state different than

the next_state of that transition. Then the constraint ensures that for every transition t0, the mentioned process produces no transitions (t1) that satisfy the condition.

```
CONSTRAINT (2) direct_reachability (a: substate)
   ALL t0 FROM {transition @ a}

      ALL t1 FROM {transition @ a :
         *->current_state == t0->current_state
         *->event == t0->event
         *->next_state != t0->next_state }   SATISFY
      t1 == <>      // there is no other next states

   OTHERWISE  message ("Transitions must have next states.");
```

A more interesting constraint is to ensure that the initial state can reach any other state. The following ECL code is the definition of this constraint:

```
CONSTRAINT (3) initial_reaching_others (a: substate)
   USES reached_states
   WITH all_states = [state @ a]

   ANY s0 FROM all_states SATISFY
      (AFTER reached_states :=
         [initial_state @ a];  // start with initial state

           add_direct_reachables(s0, a) SATISFY
           ALL s1 FROM all_states SATISFY s1 IN reached_states))

   OTHERWISE  message ("Initial state must reach every state.");
```

In the above model, I build a set of reached states starting from the initial state. To do this, I use a function called add_direct_reachables whose ECL definition I have omitted for clarity. This function recursively adds the direct reachable states of a state (specified by its first parameter) from a machine (specified by its second parameter)

using the previously defined direct_reachability constraint. Once the set of reached states is built, the above constraint ensures that every state of the machine is a reached state.

Observations

My concern during the use of EDL and ECL was the level of difficulty associated with the use of these languages. I have observed that defining the entities, aggregates, and relationships using EDL is a simple and straightforward process. However, those definitions are not complete without the constraints. Defining constraints, in particular complex constraints, using ECL is a cumbersome task and requires extensive knowledge of the language. Metaview separates the data definitions from the constraint definitions. As a result, definitions of even the simple constraints are difficult. This seems to be a major problem of Metaview given that there is no help system available during the methodology modeling process.

4.2 Toolbuilder System

Toolbuilder is a commercial MetaCASE tool built by IPSYS Software in UK. Currently, this tool is used by many researchers around the world in building experimental and industrial CASE tools. Current research in this area is to extend the tool in automating more of diagram editing and correctness checking capabilities [29].

Toolbuilder has a method specification capture component (METHS) that captures the underlying meta-model of the methodologies including the language of the diagrams and the input or output structures. These specifications are then transformed into parameters for generic tools and mechanisms which form the run-time

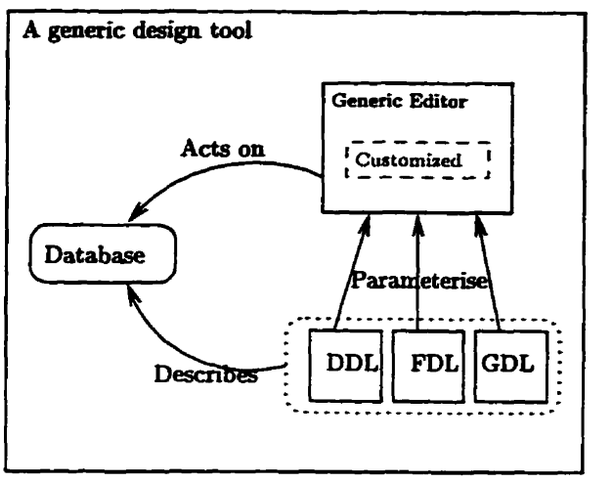


Figure 4.2: Components of a Generic Tool in Toolbuilder (From [1])

component. The run-time component interacts with the CASE user, capturing the specification of the software, according to the rules of the parameterized methodology [1].

It has been difficult to obtain technical details about Toolbuilder since it is a commercial tool with proprietary technology. Available documentation, prepared by academics using Toolbuilder, provides some technical detail [29]. Based on this documentation, Toolbuilder is a collection of integrated and generic tools and function libraries. It has two editors (diagram and text editors) that allow the building of the corresponding tools (diagram and text editing tools).

Figure 4.2 shows the constituent parts of a diagram editor (or design editor). In this figure, DDL describes the underlying data structure, FDL allows definition of the interface, and GDL permits the symbol and graphic associations. These descriptors parameterize the design editor and define its allowed behavior. In this architecture, the design editor remains generic. However, sometimes a specific behavior is desired. DDL, FDL, and GDL describe what is allowed and not how a desired action is to be

performed. To make the tool method-specific (capable of providing specific functions), Toolbuilder allows the CASE customizer to write customizer functions and link them to the design editor. Therefore, the design editor is parameterized by the descriptors and has a specific behavior defined by the linked functions. At a lower level the customizer functions can be written as C subroutines.

4.2.1 Data Descriptor, Access, and Storage Facilities

The meta-modeling technique defined for Toolbuilder is based on the realization that diagrams used in methodologies generally are in the form of a directed network of nodes and links. Nodes are modeled by entities and links by relationships. Diagrams also contain labels and values that are modeled by attributes and their values. The symbols and graphical styles are recognized as lexis of a graphical language. The legal uses of symbols are the syntax and the legal relationships among the nodes (or links) are the semantics. Similar principles apply to forms, structured text, and matrices. Hence a language is defined to capture and describe the methodologies.

Data description language (DDL) is used to express the schema of the data. A schema is the definition of entity types, attributes, and relationships among entities. The database or the data dictionary of Toolbuilder is produced by a compiler that compiles the schema defined by the DDL constructs. This database has two tiers: the 'file level' and the 'file content level'. There is also a distributed database management system which is proprietary to the IPSYS company. This system manages the database and provides the necessary locking and concurrency control.

The meta-model of Toolbuilder covers more than just the simple entities and relationships. Entities of this model have types that determine the allowed attributes.

Attributes can have types that are printable, like strings and integers, or they could be entities by themselves. The latter types are known as the links or the relationships between entities. There are constraints on the types such as the cardinality of the entities and their attributes.

This meta-modeling technique supports hierarchy of the entities where subtypes inherit the attributes of their parent type. It is possible to define derived multi-valued relationships as collections of the values of other relationships. This is the familiar notion of the aggregation. Furthermore, recursion is also supported which is used to define derived relationships by a recursive function over other relationships. This meta-model is active in that triggers can be associated with events applying to attributes and relationships.

The graphical information about a diagram, like the various types of shapes, lines, texts, matrices, maps, and the relationship among them are described using the *graphical description language* (GDL). The *layout language* (LL) is used to describe how textual editing tools are to be customized. At run-time most of the functionality of the tools is provided by the customizer C functions which I discussed in the previous section.

Toolbuilder has been used to encode SSADM which is considered to be a complex methodology [29]. In addition, it has been used in many MetaCASE related research projects including teaching, reverse engineering and re-engineering [2]. My general impression about Toolbuilder is that this tool is very effective in capturing the data structures of the methodologies and creating graphical and textual editors. Toolbuilder has little capability beyond capturing the data structures of the methodologies and providing customized editors.

4.2.2 User Interfaces

The user interface of Toolbuilder is OSF/Motif-based which is used in the Unix environment. However, Toolbuilder has addressed portability across operating systems and hardware platforms. The customized tools are often single-user tools, however, Toolbuilder provides facilities for teamwork.

The interaction styles of the user interface (windows, menus, buttons, etc.) and its layouts are described by a *format description language* (FDL). FDL parameterizes the tools by associating specific actions with the creation and selection of an interaction object such as a menu or an icon. The interaction is through the design diagram and structure text editors which capture both the diagrammatic and textual aspects of the methodologies or the software systems.

During the specification of the methodologies, there is no help available for the CASE tool customizer. However with the specification, there are (not very explicit) facilities that would allow building of help systems, metrics, and rules to be used by software developers. In general, a conclusion in working with Toolbuilder by a long-time user (Ferguson) is the extreme complexity of extending the functionality of the tools beyond the default operations which, together with poor documentation, requires a lengthy learning curve [29].

4.2.3 An Object and Document Manager

Toolbuilder manages user interactions with its user interface management system. Initially a log-on facility is responsible for setting-up the environmental variables used by the tools, compilers, and the database. Once logged-on, the user interface manager controls the interaction between the tools, compilers, and the database. Compilers

are the back-end to the languages of the Toolbuilder (DDL, FDL, LL, and GDL).

Toolbuilder user's view of the underlying data is through the frame model. Frames are collections of graphical or textual objects with images on the screen and associated actions. The actions may be default functions, like cut-and-paste, or more specific functions like navigation. There is a root frame which is invoked by the run-time component. Frames determine the presentation of information to the user. FDL, LL, and GDL scripts are all generated within the frames for use by the run-time component.

Data consistency among different representations is the result of using a uniform meta-modeling technique in a single repository of data. However, recently high-level manipulation functions have been developed that may be used recklessly and, therefore, introduce inconsistencies. There is a need for a formal mechanism that would check the correctness of the models.

Toolbuilder has been used in building management process support systems. There are no documentation of this; however, the database of Toolbuilder has the capability of supporting automatic triggering rules. These rules can be used to define events and actions required in supporting a process that would be executable at run-time. This capability is implicit in the meta-model.

4.2.4 A Query and Report Manager

Toolbuilder does not provide a complete, clear, and high-level querying facility. However, the current stage of Toolbuilder's functionality is not clear and passing a judgment seems unfair. Based on the documentation available to us, there are functions developed for accessing the database (based on the attribute values) and visualizing

the data (using links and sequences). This is accomplished using queries defined by the CASE tool customizer to be used by software developers.

Toolbuilder allows generation of reports with structures defined using a specific sub-language. Once the report structure has been customized, the output generation aspect of Toolbuilder uses the design and structure text editors' off-line facilities to produce the appropriate output. Outputs generated in one development stage either become the inputs of later stages or are used directly by software developers. The customization of outputs has the following four phases: (i) controlling the source of the information (which is a crude form of integration), (ii) controlling the structure (which includes scripts for generating particular representations of data), (iii) controlling the logical appearance of data (at run-time), and (iv) controlling the physical appearance of the information (directly or using a publishing system).

4.2.5 Transformation and Meta-programming Tools

Toolbuilder uses integrated and shared data between its tools and allows transformations between textual and diagrammatic formats of data. This is like viewing the same data using different tools or in different environments. Furthermore, the report generating sub-language of Toolbuilder and the corresponding run-time components are used to define specific formats for the outputs. This may take place between different stages of the process and act as a transformation facility.

Toolbuilder provides 'lower CASE' capabilities including code generation, configuration management, and version control. These are based on Toolbuilder's open architecture which allows the integration of third party components like document processors and code generators. I have not been able to obtain any documentation

describing experiences in these areas.

4.2.6 Modeling FSM in Toolbuilder

Toolbuilder provides languages (DDL and GDL) for modeling concepts of methodologies. In this section, I will use these languages to model our sample method (FSM). Modeled methodologies have associated graphical objects in Toolbuilder and may further be formatted for manipulation and viewing purposes and eventually act as customized CASE tools.

Primitive Concepts

To model the primitive aspects of FSM, I will use DDL codes. This is similar to defining a database schema with predefined types (Node and Link) and keywords. The following DDL code presents a model of these primitives:

```
Node :: FSM_entity

FSM_entity =>  id : String, name : String;

Forward transition Is Link

Type state Is  FSM_entity
            (in_transition : In Set Of transition;
            out_transition : Out Set Of transition;
            substate : Owner Of Set Of state);

initial_state <= state;
final_state <= state;
```

In this model, FSM_entity is defined to be a generic predefined Node type with name and id attributes. State is an FSM_entity which inherits its structure and may be an

initial_state or a final_state. Transition is a relationship whose definition is forwarded for later. It is used in defining the fact that a state can be a source or a destination of many transitions. The substate type refers to the concept of ownership which I will discuss later.

Transitions

Relationships between entities can be modeled by the Link concept of DDL as follows:

```
Type transition Is Link
          (next_state : In state;
        current_state : Out state);

transition =>  id : String,
               event_data : Sequence of String;
```

In the above model, transitions are links from current_state to next_state. Events are modeled as attributes of transitions with sequence of string types. Although this is an oversimplification, it is sufficient for the purposes of this work.

Hierarchy

In our definition of FSM, any state entity may consist of another FSM. This can be defined by a DDL code that allows a state to contain other states identified by their name and a linktype called explosion:

```
state =>> explosion(name) : state;
```

This allows states to explode to more states. However, transitions occur within or between states and their substates. Therefore, we need constraints to ensure that states of a transition of an FSM are the states of the same FSM.

Simple Constraints

Unlike Metaview, Toolbuilder does not provide a separate language for defining constraints. Some constraints may be defined with DDL. In our problem, DDL allows definition of ownership, as seen previously, with the substate concept. Here 'owned' objects (substates) do not know of the 'owner' objects (parent states) and cannot access them. Therefore, a structured hierarchy based on the levels of states and their substates will restrict the scope. Transitions would only be able to access states within the same level (FSM). An exception is a transition between a state and its substates. This type of transitions are not restricted. They provide for defining inputs and outputs of an FSM which is omitted from our discussion.

Complex Constraints

In addition to DDL, GDL also allows enforcement of some restrictions such as the type and the number of the participants in a link. However, more complicated constraints like the reachability, which was discussed in section 3.2, are not modeled in any of the provided languages. In these cases, Toolbuilder allows definition of C functions that may be used to model the complex constraints. In order to access data and attribute values, Toolbuilder provides predefined functions, like **get_attribute()**, which form the interface between the CASE customizer defined C functions and the database. Once constraints are defined as C functions, they can be called directly using FDL formatted menu options or indirectly embedded within other methods and operations.

An example of a complex constraint is the direct reachability which says: given a **current_state** and an **event**, the system may transit to exactly one **next_state**. A more complex example is to ensure that the initial state can reach all other states. The

following pseudo C code illustrates a model of these constraints:

```
DIRECT_REACHABILITY (state: cs, Seq_Of_String: e){
     state: ns = NULL;     // to hold the next state
     // get an out_transition of current state and mark it
     String: t = get_out_transition (cs);
     while (t) {
          if(cs == t.current_state) { // given current state
             if(e == t.event_data) {  // and an event
                  if(ns == NULL)
                       // if next state is the first one
                        ns = t.next_state;
                  else if(ns != t.next_state)
                       // Give ERROR Message and Exit
             }
          }
          // get another unused out_transition and mark it
          t = get_next_out_transition (cs);
     }
     return (0);   // There is only one next state
}

INITIAL_REACHING_OTHERS (state: is) {
     Seq_Of_String: as = get_all_states(); // list of all states
     Seq_Of_String: rs = is;            // list of reached states
     as = as - is;        // take initial state out of all states
     while (1) {
          Seq_Of_String: t;           // to keep out_transitions
          Seq_Of_String: drs =0;      // directly reached states
          while (t = get_next_out_transitions (rs))
             drs = drs + t.next_state;
          if (drs) {
             rs = rs + drs;  // add them to reached states
             as = as - drs;  // take them out of all states
          }
          else break;  // there is no more reachable states
     }
     if (as = NULL)  return (0);  // all states are reached
     return (-1)      // there is a state that is not reached
}
```

My objective in coding these constraints is not to implement them but to experiment with the tools and present their capabilities and shortcomings. They are provided for illustration purposes only. The important aspect is the outcome of the coding process which has allowed me to draw conclusions about the reviewed tools.

Observations

In my experience, DDL is more expressive than EDL (of Metaview). Some of the constraints are modeled right within the DDL or enforced by the GDL codes. There is no separate language (like the ECL of Metaview) for constraints which makes modeling of constraints, that are closely related to the actual data, easier. However, DDL cannot model complicated constraints. In these cases, C functions are used which are easier to understand for programmers. However, using C functions creates the added complexity of managing the interface between the C code and the database. Toolbuilder provides predefined interface functions but is unable to eliminate the difficulty of understanding and maintaining those interfaces.

4.3 MetaEdit System

The MetaEdit family of tools, in particular MetaEdit+, is the result of a joint research project between Jyväskylä and Oulu universities in Finland. It is one of the most popular MetaCASE tools both from the research and the industrial point of view. MetaEdit+ is also called a *computer-aided method engineering* (CAME) tool. Although all MetaCASE tools provide means of methodology specifications, only MetaEdit+ claims to be a distinct CAME tool by providing additional flexible means of method management, integration, and reuse.

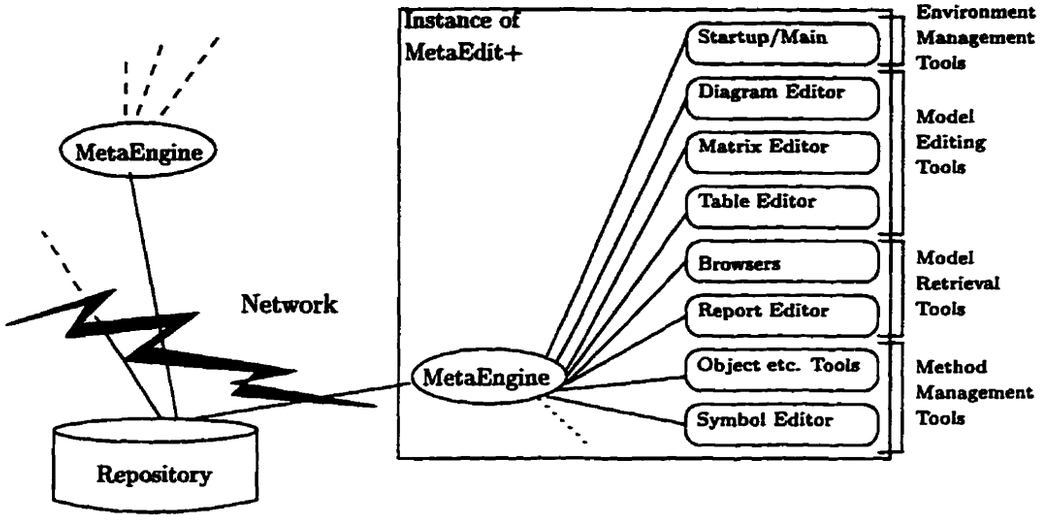


Figure 4.3: MetaEdit+ Architecture (Adapted From [60])

Considering the discussion about the three levels of MetaCASE modeling in section 2.3, we should look at the corresponding three levels in MetaEdit+ [84]. As a MetaCASE tool, which is used to build CASE tools, the three levels are similar to the model of figure 2.6. However, as a CAME tool, the three levels operate one level 'higher' than the other MetaCASE tools. As a result, the highest level is called the meta-meta-modeling level where the syntax and semantics of various meta-modeling techniques are defined. The subsequent levels of MetaEdit+ are the two highest levels of other MetaCASE tools. This allows definition of meta-modeling techniques and engineering of methodologies. I will refer to meta-meta-modeling as meta-modeling, since the concept of modeling is the same at any meta level.

The general architecture of MetaEdit+ is shown in figure 4.3. This architecture is based on conceptual modeling principles. Hence, the repository has an associated conceptual schema and the tools resemble external views of that schema. In addition, object-oriented design is used in development of MetaEdit+ which allows reuse and

interoperability between tools (as claimed by MetaEdit+ developers). The heart of MetaEdit+ is the *MetaEngine* which handles all operations on the underlying conceptual data. The tools of MetaEngine are divided into four groups: environment management tools (to manage features of the environment), model editing tools (to create, modify, and delete model instances or their parts), model retrieval tools (to retrieve design objects and their instances), and method management tools (for method specification, management, and retrieval).

4.3.1 Data Descriptor, Access, and Storage Facilities

The meta-model used in MetaEdit+ is called GOPRR: Graph, Objects, Properties, Relationships, and Roles. This data modeling technique is used at the conceptual level for specification of Methodologies as they are observed, interpreted, and recorded. The objects of GOPRR refer to the independent and identifiable design objects (similar to the entities of an ER diagram). These objects often have properties (attributes) such as the name of a process in a DFD. Relationships are associations between objects that appear as lines between shapes in diagrams. Roles define specific ways that objects participate in relationships.

The concept of Graph was added to the OPRR model for several reasons. It denotes an aggregate concept which is the containment of a set of objects and their relationships. This allows representation of complex objects using decomposition into contained objects. In addition, it allows different 'representational' graphs (like matrix or diagram) of the same 'conceptual' graph (like a DFD). Furthermore, it permits the use of multiple methods at various levels of the project while maintaining links among them.

Other capabilities provided by the Graph concept are the inclusion of generalization and specialization constructs as well as polymorphism. This last extension allows viewing of an object in one method as an object and in another method as a relationship or property. GOPRR allows method integration in such a way that objects, relationships, and roles can be reused and properties can be shared. My concern with such a flexibility is model integrity, consistency, and completeness. MetaEdit+ claims that, to some extent, a model checking system is present.

MetaEdit+ provides support for OO analysis and design methodologies, such as OMT and Booch, as well as structured analysis and design methodologies such as Yourdon's SA/SD. Developers of MetaEdit+ claim that their system provides one of the largest selection of method supports. An advantage of this tool is the capability to change the description of the methodology even while it is in use. These changes are reflected on the models immediately. The major focus of this tool is on capturing the methodology-related data in an integratable form. More work must be done on ensuring the correctness of the captured data and providing detailed analysis and simulation tools.

All the captured data is stored in the repository of MetaEdit+. Captured data include the method specifications represented as GOPRR concepts and other information bases needed to operate the tools. Among the information bases are the object representation symbols (such as spatial coordinates and shapes), MetaCASE user description information (such as user status and permissions), and output specifications and reports. There is a data management system that allows data sharing and concurrency. This is achieved using a locking mechanism which admittedly reduces the performance.

4.3.2 User Interfaces

MetaEdit+ is a multi-user and multi-platform system with a client-server architecture. Each client has an instance of MetaEdit+ with copies of all the available tools and the controlling MetaEngine. The main launcher allows only the permitted capabilities based on the MetaCASE user status information located in the repository. As an example, a browser allows hierarchical access to the models and the meta-models stored in the repository based on the user permission. Further interfaces with the system is through the graphical or textual tools.

Specification of the methodologies is through the conceptual object type tools, such as the relationship tool, and their textual representations. This is often done by filling in a form that describes the conceptual object. A symbol editor is used to specify and design the graphical objects and their behavior while linking them to the conceptual object type. There is no help available during the specification of the methodology [88]. However, facilities are provided for defining help, metrics, and reporting systems for the CASE users. Another open problem in MetaEdit+ is to guide the CASE user in selecting the right methodology for a particular project based on the type of the project. Ongoing research in this area is acknowledged by MetaEdit+ developers [60].

CASE users interact with the tool using model editing tools such as diagram, matrix, or table editors. These tools are invoked from a *WorkSpace* which holds information about the MetaCASE user and can be configured to the user's likings. Added facilities by the CASE customizer can provide metrics that are used to produce reports. It can also act as help systems that may be triggered automatically by the CASE users.

4.3.3 An Object and Document Manager

The MetaEngine is the direct back-end of the tools used by developers to access the repository. Implementation of the conceptual data model and the operational signatures are embedded in the MetaEngine. Accordingly, tools request services from the MetaEngine but operate solely based on their own specific paradigm. In the client-server architecture, MetaEngine deals with all the communication issues with the server and various tools. Hence, data sharing is made possible through data integration.

As discussed earlier, this architecture allows transformation of objects from one format to another. This is possible among the three supported formats: diagrammatic, matrix, and tabular. In addition, objects can be viewed in one method as an object and in another method as a relationship or property. Although such flexibility is highly desirable, it may cause problems. A prominent one is to ensure consistency and integrity within and between methods. This problem is the topic of one of the future research directions of MetaEdit+.

Another MetaEdit+ research direction is the building of a flexible process support which may coordinate various activities and manage deliverables [55].

4.3.4 A Query and Report Manager

MetaEdit+ provides tools for editing specification of methodologies which is stored in the repository. No support is available for making queries about the correctness of the methodology. However, consistency and completeness of diagrams may be verified during the description process.

Querying the states of the developer specified data is possible using the customized

querying and reporting facilities. The CASE tool customizer can use the report editor and the associated procedural query and data manipulation language to generate the capability for producing textual descriptions of the models. This capability can be used by software developers in querying the models of the repository. These queries may be simple software system consistency checks or various data state requests.

MetaEdit+ allows creation of reports that provide simple textual descriptions of the model of the software systems. As described above, reports are generated using the report editor and the associated languages.

4.3.5 Transformation and Meta-programming Tools

Data objects in MetaEdit+ can be viewed in different tools. This acts as a local transformation facility. In essence, there is only one copy of the objects that is viewed in different environments and formats. An example is a textual-to-graphical transformation. There are no additional transformation languages that could be used to transform models between methodologies or tools.

Transformation between different phases of the software engineering process is possible using a combination of the specification of methodologies and the report generating facilities. With the help of the report editing tool, functions can be defined to take data outputs of one phase (like object descriptions) and produce data inputs for the next phase (like language-dependent class definitions).

Although the focus of MetaEdit+ is to act as an 'upper CASE', there are semi-automatic facilities that could be used as meta-programming tools. MetaEdit+ provides predefined SQL and Smalltalk code generation capabilities.

4.3.6 Modeling FSM in MetaEdit

Based on the GOPRR meta-modeling technique, MetaEdit+ provides a graphical interface to model the methodology concepts and constraints. CASE customizers use object modeling techniques to define the graphical representation of the methodologies which is transformed into the textual code of that model. The GOPRR of MetaEdit+ is similar to the previous two MetaCASE tools in two ways. Firstly, it allows definition of graphical objects associated with the methodology concepts (which I do not examine here). Secondly, it uses generalization and aggregation to build a hierarchy of objects (which I will examine).

Primitive Concepts

Figure 4.4 presents the GOPRR representation of our sample method (FSM). This model of FSM consists of a generic object called FSM_object which is only an abstract type and can never be instantiated. This generic object has a property called name which is represented by the oval in the figure. FSM_object is specialized into another object called state which inherits the name property and adds an identifying property called id. Identifying properties are represented by double ovals. The state object can be instantiated or further specialized to be an initial_state or a final_state.

Transitions

Events are the properties of transition relationships defined as a relation between two states. The states participating in this relationship take the from or the to roles. These roles belong to the current_state or the next_state objects. Transition relationship has two properties called the id and the event_data.

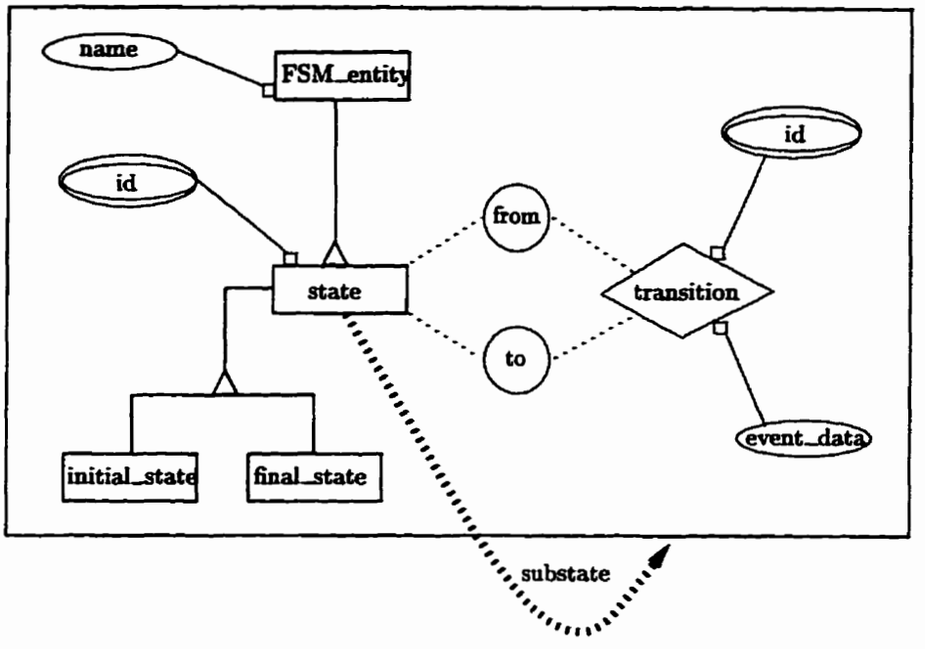


Figure 4.4: GOPRR Representation of FSM in MetaEdit+

Hierarchy

In figure 4.4, the 'Graph' of GOPRR for FSM is represented by the large rectangle which contains the definition of the flat FSM. To represent hierarchy of states and their substates, a particular type of relationship is used which relates the **state** object to the 'Graph' of FSM. This relationship is shown by a thick dotted arrow called **substate**.

Once the graphical representation of the model of FSM is defined by the CASE customizer, the corresponding textual code is automatically generated. A sample fragment of the textual model of FSM is shown on top of the next page. This code is a simple description of the model and lists all the Graphs, Objects, Properties, Roles, and Relationships.

```
Graph:          substate      Aggregation relationship
Object:         state         Can explode to a substate
Property:       id            Identifier
Role:           to            Cardinality: 1;
Relationship:   transition    Represented by a line
```

Simple Constraints

Many of the constraints, such as the cardinality of participants in a relationship or the type of the roles, are modeled within the GOPRR graphical representation. Consider the constraint ensuring that the states of a transition of an FSM are the states of that FSM. This constraint is implicit in our GOPRR model of FSM since the states taking the from or the to roles in a transition relationship are the states of that particular FSM (or 'Graph', as called in MetaEdit).

Complex Constraints

Consider the more interesting direct reachability constraint which ensures that given a current state and an event, the system will always transit to exactly one next state. This constraint is ensured by having unique identifiers and enforcing cardinality constraints in the roles of the states. Hence, the state taking the to role (next_state) can have a cardinality of exactly one. This is shown in the cardinality part of the definition of the to in the textual format of the FSM model. MetaEdit+ does not provide any separate constraint definition facilities.

Observations

The GOPRR graphical modeling of FSM is advantageous since it is easy to model and simple to understand. However in modeling constraints of a complicated nature,

GOPRR modeling faces difficulties. As an example, consider the constraint ensuring that the initial state can reach any other state. Based on my examination of GOPRR meta-modeling technique, it is not easily possible to model this constraint. This appears to be a major shortcoming of MetaEdit+.

4.4 4thought System

The 4thought prototype is a result of the Advanced Software Design Technology (ASDT) project of the Center for Advanced Studies (CAS) in IBM Canada [78]. This project was abandoned in 1995 for budget reasons. Since 4thought addresses some of the fundamental MetaCASE issues, I have decided to examine it. The underlying theory for the development of 4thought is the Theory-Model Paradigm [50, 77]. It is an attempt to provide confidence about the correctness of a complex design. Of course the real solution to the problem of correctness is the 'proof' paradigm which requires great effort in using formal methods and, consequently, is not practical.

In the Theory-Model Paradigm, there are theories (methodologies) that prescribe categories of design concepts and rules. A design is defined to be a collection of facts that form a model for the theory. Once the model falls within a category and satisfies the rules, it is valid. This approach is easier and more practical than the difficult formal proof paradigm. Yet it is still partially formal since methodologies are described using a formal specification language such as Z [22]. The Theory-Model Paradigm provides the theoretical basis for the development of the 4thought system.

It is important to notice that 4thought approaches MetaCASE from a different viewpoint. Most MetaCASE tools focus on providing means of capturing the data model of methodologies using various meta-modeling techniques. Then they provide

ways of customizing easy to use CASE tools. In this process, ensuring the correctness of the captured methodologies and even the produced software system is secondary. With 4thought, the focus is towards the correct capturing of the methodologies and verifying that a design meets the constraints of the methodology with which it was developed.

4thought is only a prototype and lacks a complete architectural design. Based on the available documentation, it is a collection of stand-alone components. These components are integrated to provide the functionality that supports the Theory-Model Paradigm. First, methodologies are formalized using various formalization techniques like Z. Then they are modeled, both graphically and logically, using meta-modeling techniques like GraphLog [17]. Next, executables are generated by transforming the meta-models into Prolog programs. Finally, software systems are specified and checked using the executable methodologies.

One of the components that is used in 4thought is a Prolog database server that contains the information captured using meta-modeling techniques. Visualization of data in the form of graphs is through the visualization and querying capabilities of the GraphLog system. Textual views of the data are also available, and are integrated with the graphical information.

4.4.1 Data Descriptor, Access, and Storage Facilities

Based on the Theory-Model Paradigm, methodologies are first formalized using a formal language. This step is particularly important for the informal methodologies. The key objective in doing this is to ensure that there is a precise meaning for the concepts found in the methodologies. Since most of the fundamental concepts in software

engineering can be described using elements of set theory and predicate logic, a formal language, such as Z, has been used for this part. Examples of modeled methodologies are JSD, OMT, and OOA [101]. Developers of 4thought argue that only a subset of the predicate calculus is sufficient to represent the elements of methodologies. Hence, a visual logic programming language (GraphLog) has also been used to graphically express the concepts of methodologies.

The meta-modeling technique used by 4thought is based on the graphical representation of the concepts of methodologies. As the first step, ER diagrams are used to model the primitive concepts. This meta-model provides a basis for visualizing concepts as graphs where entities are the nodes and the relationships are the arcs of the graph. This allows association of a semantic network with the meta-model. However, ER models are not sufficient to describe methodologies. Therefore, GraphLog has been used to provide more semantic power. It allows representation of hierarchy using the nesting of nodes within nodes. This is referred to as the *Hygraph* structures that are formal and rich semantic networks [17].

A methodology includes derived concepts that are expressed in terms of the primitive ones. These concepts may include logical constraints and rules of the methodology. In addition, operations for viewing or updating the captured data while preserving the integrity constraints are also required. GraphLog, as the meta-modeling technique, allows definition of concepts, constraints, queries, and update transactions. Since GraphLog is a logic programming language, it allows definition of the logical constraints and rules that may be used in verifying the correctness, consistency, and completeness of the specified methodologies and software systems.

The Prolog-based database of 4thought holds information about the methodology

and the specified software systems. The entities of the ER diagrams are represented as Prolog predicates in the database. Relationships, integrity constraints, queries and the update transaction operations of the GraphLog are the Prolog rules and facts. A database management system controls and manages transactions with the database. However, 4thought is a prototype tool and advanced database functionalities are not expected.

4.4.2 User Interfaces

The user interface of 4thought is mainly through the GraphLog system where methodologies and software systems can be graphically specified, visualized, and edited. I am not aware of the details of this user interface, but we expect it to be a standard data visualization and editing facility. Besides GraphLog, the other facilities of 4thought are prototypes involving little or no user interface issues.

4.4.3 An Object and Document Manager

Initially, 4thought allows the MetaCASE users to connect to the Prolog database and create a workspace that contains a meta-model of a methodology and snapshots of the specified software systems. Further Prolog programs can be invoked as helping or guiding facilities. Among these programs, is a viewing facility which creates snapshots of data about a specific design. This facility needs to provide a faster refreshing capability and more up-to-date views. There is also a Prolog program that allows correctness verification of a specified software system against the rules of the methodology that it was built with. In addition, Prolog programs exist that help in searching the database of a particular software system. Prolog programs are compiled with the

standard Prolog compiler creating the run-time components of the 4thought system.

4.4.4 A Query and Report Manager

The GraphLog system allows inquiries about the modeled methodologies, specified software systems, and their consistency and correctness. These queries may involve searches through the database and computations about the states of the data. The GraphLog system has higher expressive powers than SQL, in particular, when expressing graph traversal operations without using recursion. This, in general, can be considered to be one of the advantageous aspects of graph-based meta-modeling techniques.

There are no explicit reporting facilities in the 4thought system.

4.4.5 Transformation and Meta-programming Tools

There is a GraphLog-to-Prolog translator which takes the GraphLog representation of the methodology and the software system produces Prolog facts and rules for the use by the Prolog-based database system. This transformation can be considered to be a formal transformation, since the language of GraphLog is logic which can be mathematically transformed into logical statements readable by the Prolog compiler. The GraphLog-to-Prolog translator has performance difficulties which is most problematic for large-scale software systems. This area is an active research area of the deductive database community.

The 4thought system provides no 'lower CASE' facilities.

4.4.6 Modeling FSM in 4thought

The 4thought modeling of FSM differs from the previously seen ER-based or OO-based tools. It is a three level process consisting of a simple ER model, a Z formalization, and a GraphLog definition of FSM. The ER model has been fully discussed during the modeling of FSM in the ER-based tools. In this section, I will formalize selected aspects of FSM using Z and GraphLog. From GraphLog modules, 4thought creates the executable Prolog programs.

Primitive Concepts

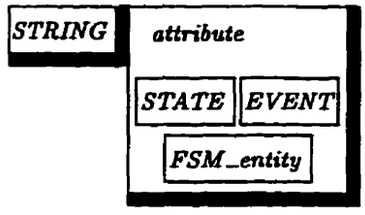
The following Z code represents the basic entities of FSM:

$$[FSM\_entity, STRING]$$
$$STATE, EVENT : \mathbb{P}\ FSM\_entity$$
$$\langle STATE, EVENT \rangle \text{ partition } FSM\_entity$$

BasicEntities

$$\begin{aligned}
&entity : \mathbb{P}\ FSM_entity \\
&states : \mathbb{P}\ STATE \\
&events : \mathbb{P}\ EVENT \\
&id, name, event_data : \mathbb{P}\ STRING \\
&initial_state : STATE \\
&final_state : \mathbb{P}\ STATE \\
&attribute : STRING \leftrightarrow FSM_entity \\
&attribute : STRING \leftrightarrow STATE \\
&attribute : STRING \leftrightarrow EVENT \\
\hline
&initial_state \in states \\
&final_state \subset states \\
&\forall a : \text{dom } attribute \bullet a \in id \lor a \in name \lor a \in event_data \\
&\forall b : \text{ran } attribute \bullet a \in entity \lor a \in states \lor a \in events
\end{aligned}$$

In this model, we first define a generic *FSM_entity* with an *id* attribute and then build the basic entities from *FSM_entity*. These entities are *STATE* and *EVENT* entities. The *STRING* entity is defined for assigning a type to the attributes. Entities defined in Z can be represented as nodes of a graph in the GraphLog model. Relations, such as the attribute relation, are represented as Hygraph patterns or box graphs. Such a GraphLog model is a query which filters the database and only allows passing of the proper attribute relations. The following figure represents the attribute relation in the GraphLog model:



Transitions

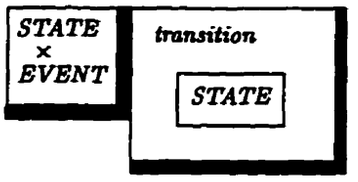
The next step in modeling FSM is to define the transition relation which is a relation from a pair of current state and event onto a next state. This relation is a function which represents the deterministic aspect of FSM. Therefore, for every state/event pair there is exactly one next state. The following Z schema presents this definition:

BasicFSM

$$\begin{array}{l} \textit{BasicEntities} \\ \textit{transition} : (\textit{STATE} \times \textit{EVENT}) \rightarrow \textit{STATE} \\ \hline \textit{transition} \in (\textit{states} \times \textit{events}) \rightarrow \textit{states} \end{array}$$

Transitions can be modeled in GraphLog using a similar Hygraph pattern. For this, $STATE \times EVENT$ is defined as a set which is related to the *STATE* set using the

transition relation. The following Hygraph pattern represents the transition relation in GraphLog:



Hierarchy

Now we can formalize the added concept of depth or hierarchy of states. This concept is simply another relation which is shown in the following Z schema:

DepthFSM

$$
\begin{array}{l}
BasicFSM \\
substate : \mathbb{P}\, BasicFSM \\
decomposition : STATE \leftrightarrow BasicFSM \\
\hline
decomposition \in states \leftrightarrow substate
\end{array}
$$

The GraphLog model of the above relation is also another Hygraph pattern which I omit its representation due to the similarities with the previous Hygraph patterns.

Simple Constraints

In the 4thought model of FSM, some of the constraints are implicit within the definitions. For example, our definition of transition of a given FSM involves only states and events of that FSM. Furthermore, the effect of a transition involves only states and events of that FSM.

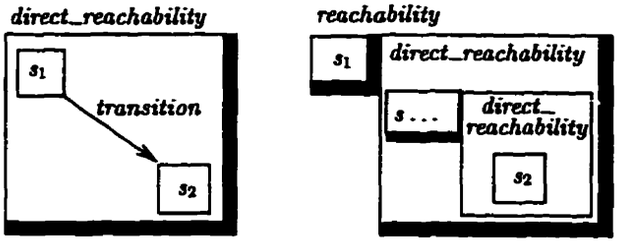
Complex Constraints

Besides the simple implicit constraints, the modeled FSM needs to satisfy more complicated constraints. One of these constraints is reachability which may be defined by the following Z schema:

Reachability

$$
\begin{array}{l}
DepthFSM \\
direct\_reachability : STATE \leftrightarrow STATE \\
reachability : STATE \leftrightarrow STATE \\
\hline
direct\_reachability \in states \leftrightarrow states \\
reachability \in states \leftrightarrow states \\
\forall s_1, s_2 : states \bullet s_1 \mapsto s_2 \in direct\_reachability \Leftrightarrow \\
\quad (\exists e : events \bullet (s_1 \mapsto e) \mapsto s_2 \in transition) \\
reachability = direct\_reachability^{+}
\end{array}
$$

In this schema, the state s_2 is directly reachable from s_1 if there is a transition from s_1 to s_2. In general, a state is reachable from another, if it is directly reachable by some path of nonzero length. To model these concepts in GraphLog, we need to define two Hygraph patterns for *direct_reachability* and *reachability* relations which is done similar to the previous patterns. Based on the information provided by the Z schema of the reachability constraint, we can now model the specific 'further' GraphLog definitions of the reachability constraint. These definitions are shown by the following GraphLog models:



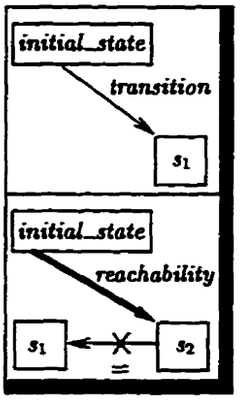
To encode the reachability of every state from the initial state, we now only need to add the following Z schema:

FSM

Reachability

$$reachability(\!| \{initial\_state\} |\!) \cup \{initial\_state\} = states$$

The GraphLog definition of this constraint is presented next which says: if a state s_2 is not equal to a state s_1 (which is reachable from the *initial_state*), then s_2 is reachable form the *initial_state*.



Observations

Although modeling methodologies in Z provides formal semantics, it is not practical since the formalized methodologies are not used directly by the 4thought system. CASE customizers are required to do this for a better understanding of the methodologies. Z is a difficult mathematical formalism which is often disliked by developers. Based on my experiences, if CASE customizers take the time to learn Z, and if Z formalization of methodologies can be transformed directly into the executable Prolog

program, then the use of Z can be justified. The Z modeling is simple for some aspects of methodologies. Other aspects, including the informal guidelines, are more difficult to model in Z. This difficulty can be outweighed by the formality of the model which is useful for model checking.

My experiences in using GraphLog show that this modeling technique is more suitable for software engineering purposes than most of the other techniques that we have examined so far. It allows construction of simple database filters and enforcement of various constraints. However, GraphLog uses Hygraph patterns or box graphs which appear to be counterintuitive. It has a limited capability and does not provide a complete usable database schema. Thus, developers need to use some form of a complimentary ER-based database schema which, in turn, adds to the complexity of using 4thought.

4.5 CASEMaker System

CASEMaker is the most recent MetaCASE project undertaken by the Joint Research Center for Advanced Systems Engineering (JRCASE) and Macquarie University in Sydney, Australia. This project is only at a proposal level which can hardly be reviewed in comparison with other more developed tools. However, I believe it addresses many fundamental issues about MetaCASE technology and is worth examining. In addition, this study can help us understand the current state-of-practice in building MetaCASE tools.

Motivation for the development of CASEMaker is claimed to be the demand for 'better' MetaCASE tools. The word 'better' refers to the capability of providing more than just the data-capture which is about the only major functionality offered by

tools such as Toolbuilder and MetaEdit. CASEMaker claims to be able to offer more in areas such as design simulation, metrics, transformations, and guidance services. However, as noted previously, this project is only at a proposal level and results do not yet exist.

The proposed architecture of CASEMaker has two parts: components (customizable building blocks of the CASE tool) and assembly-customization tools (mechanisms for the assembly and customization of the components). There are also three sections recognized in CASE tools: the MetaCASE user interface, the design support facility, and the database. In each of the three sections, there are components libraries and assembly-customization tools.

Details of the architecture are yet unknown but the existing documentation shows numerous issues being addressed in the design of CASEMaker. The user interface of CASEMaker is an aspect whose requirements are identified and its concepts are designed [54]. The user interface is referred to as *MetaDesigner* consisting of a group of generic GUI classes. These classes can be integrated and configured (into the builder) so that specific CASE tools can be built.

An extensive discussion of specific components of CASEMaker remains as a future research to when a prototype is available. At this point, I examine the fairly complete meta-modeling technique (*hypernodes*) used in CASEMaker [80].

Similar to GraphLog structures, hypernodes are sets of nodes and edges of a graph, where a node can be a graph itself. As a result, hypernodes generalize nodes as oppose to the edges (which is the case in hypergraphs). The hypernode model was originally proposed in 1990 and described as a graph-based deductive data model with a Datalog-based query language [51]. Later it was formalized using set theory into the

hypernode query language (HNQL) which is described to be a high-level procedural query language [80]. Based on these descriptions, HNQL provides functions for the creation, deletion, and manipulation of hypernodes as well as declarative querying.

The hypernode model was finally extended by Louis Scott in his Ph.D. dissertation to include scripting capabilities by the addition of concepts such as the assignment statements and while loops [80]. He argues that the hypernode model can provide support for modeling a variety of concepts in methodologies. These concepts include data structure representations, such as the OO data, and dynamic behavior representations, such as the Petri-net data. Scott has developed a prototype hypernode-based CASE tool to simulate software behavior. I believe efforts to show the effectiveness of the implemented meta-modeling techniques are the practical first steps towards building 'better' tools.

I have provided some detail to introduce CASEMaker but any further detail without having a prototype would seem to shift us away from our objective which is to review the existing tools. In general, this tool is a hybrid between a formal research work (like 4thought) and a commercial tool (like Toolbuilder). Its capabilities remain to be seen.

4.5.1 Modeling FSM in CASEMaker

In this section, I model the selected aspects of FSM using the meta-modeling technique provided by CASEMaker. This technique is based on hypernode graphs and the associated query language. Models built with these graphs are complimented by textual codes that are used for defining constraints and typechecks. Hypernode models are then associated with the graphical objects and executed to create the underlying

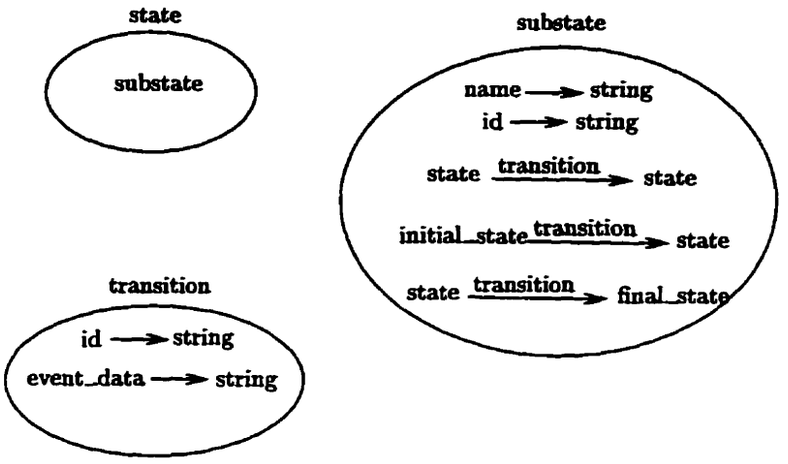


Figure 4.5: Hypernode Representation of FSM in CASEMaker

data model of various CASE tools.

Primitive Concepts

Figure 4.5 presents the hypernodes definition of the schema for basic entities of FSM. Hypernode entities are represented as ellipses with labels. Attributes of these entities are shown by the attribute name followed by an arrow which is pointing to the attribute type. For example, name and id are the attributes of the substate. In all the previous tools, I used the FSM_entity and the inherited state to represent the concept of inheritance. Hypernode also provides this capability but with slight differences which I will discuss later in this section.

Transitions

Relationships between entities are captured graphically by joining the participants with an arrow labeled with the relationship name. For example, transition is a relationship between states which is represented as an edge within the substate hypernode

shown in figure 4.5. Among the states participating in a transition, there is an initial_state with transitions only going out and a final_state with transitions only coming in. Transitions have id and event_data attributes of string types represented in the transition hypernode graph.

Hierarchy

To represent the added hierarchical aspects of FSM, the type definition of state is nested within itself via substate. In this definition, substate is a relationship between states and their substates.

Simple Constraints

Building a graphical hypernode schema, as above, allows automatic generation of the corresponding database. However, this model is not yet complete. There are various constraints that need to be modeled. Hypernode provides a textual notation for defining constraints which are referred to as typecheckings of a schema. As an example, consider the following textual code:

```
Typechecking substate:
        id: (1, 1);
        state: (0, unspecified);
        initial_state: (1, 1), outdegree wrt state (0, unspecified);
        initial_state MUTEX state;
```

The above constraints ensure that the substate hypernode has one value for the id attribute and an arbitrary number of states. It also has one initial_state with arbitrary number of outgoing arrows with respect to other states and, finally, the initial_state is represented differently from other states. These constraints deal with cardinality

of hypernodes, edge relationships, and representation issues.

Similar to MetaEdit, some of the other constraints are implicit within the definition of hypernodes. As an example, consider ensuring that the states involved in a transition (current and next states) are the states of that FSM. In the hypernode model of FSM, transitions occur within a substate and, therefore, states involved in a transition are states of that substate. As a result, the ownership constraint is implicitly defined.

Complex Constraints

A more complex constraint is the direct reachability constraint which ensures that given a current state and an event, the system will always transit to exactly one next state. This is inherent in the definition of the transition in hypernodes which does not permit non-deterministic sequences of a transition-to-transition nature. To encode the direct reachability constraint, we also need to define cardinality constraints on the roles of states. This brings us to the concept of schema/instance in hypernodes.

The hypernode modeling allows definition of multiple levels of abstraction which is referred to as a schema/instance relationship. Often various instances of one hypernode schema can be defined. Arrows directed from the instance hypernodes to the schema hypernode are used to represent schema/instance relationships. A function maps the instance onto a schema and ensures structural similarity (or inheritance) while allowing various cardinality and link-type constraints.

An example constraint that could be enforced is to ensure that the id attributes are unique within the instances of the schema. This is done by constraining the cardinality of the outdegree of the id in the state instance to be exactly one with

respect to the state schema during the definition of the typechecking codes.

Another constraint is to define roles for the states participating in transitions and to define cardinality constraints based on the defined roles. This is very similar to the OO modeling concepts of MetaEdit. These definitions would allow encoding of the direct reachability constraint. I have omitted the details of this process due to the complexity of defining multiple levels of abstraction in CASEMaker.

More complex constraints, like the reachability of every state from the initial_state, do not appear to be any easier to model in hypernodes.

Observations

Based on my observations, the simplicity of modeling the entities and some of the simple constraints is an important feature of hypernodes. The hypernode meta-modeling provides the most expressive powers during the modeling process but it also adds to the modeling complexity. Hence, more complex constraints are not easily modeled. In general, usability of this modeling technique is dependent on the user interface of CASEMaker which remains to be developed.

Chapter 5

Discussion

CASE tools are not widely used in industry. My investigation in chapter 2 revealed that the high cost of CASE adoption and use is a major underlying reason. In section 2.3, I showed that MetaCASE offers alternative low cost solutions. However, as discussed in chapter 4, most MetaCASE tools have problems with respect to their capability, ease of use, and functionality. I believe current MetaCASE tools are not built with developers in mind and do not offer the functions that developers want. Arthur Ryman, who started the 4thought project, believes design tools are not widely used since they are not programmer-friendly [76].

In this chapter, I examine the MetaCASE problems by summarizing and providing comparative discussions about the studied tools. I use the typical component subdivisions of section 3 and compare and analyze the advantages and disadvantages of each tool in section 5.1. In addition, an evaluation of the tools is provided in section 5.2 based on my experiences during the modeling of FSM.

5.1 Comparison on a Component-basis

The component-based analysis of this section is conducted to help us identify the similarities and differences between various tools. It consists of a summary followed by the open problems of each component.

5.1.1 Data Descriptor, Access, and Storage Facilities

All the reviewed tools provide a mix of textual and visual languages for modeling data, graphical objects, and constraints. The ER-based tools (Metaview and Toolbuilder) offer textual languages while the OO-based tools (MetaEdit) provide visual capabilities. The Graph-based tools (4thought and CASEMaker) use a mix of both languages in their set-theoretic or logic-based modeling techniques. With the exception of CASEMaker, all these languages extend the core capabilities of their implemented meta-modeling techniques to allow representation of inheritance, hierarchy, and graphical concepts. Developers of the reviewed meta-modeling techniques claim that many software engineering artifacts can be modeled using the provided languages. They claim that complex methodologies, like SSADM and OMT, can be modeled with the provided techniques.

The databases of the existing and developed tools (Metaview, Toolbuilder, and MetaEdit) offer file locking mechanisms which permit concurrent access of the data. Furthermore, commercial tools (Toolbuilder and MetaEdit) provide facilities for software evolution. These topics remain to be examined in our future research.

Open Problems

Examination of the existing tools shows that their implementation of the meta-modeling techniques is generally limited to data-capture. The employed meta-modeling techniques allow modeling of methodology prescribed data structures and some of the associated constraints. However, modeling of most of the process rules and guidelines of methodologies is absent. Process rules and guidelines were discussed in the RPG model of section 2.2.1. Without modeling the process rules and guidelines, developers do not find CASE tools to be viable or even beneficial. This leads to failures in CASE adoption. Capability of modeling the process rules and guidelines is, therefore, an important desired feature.

A necessary meta-modeling requirement is to model the three aspects of software systems (data structures, dynamic behavior, and functions). Besides the data structures, other aspects of software systems are not fully addressed by the implementation of meta-modeling techniques of the tools that we examined. These tools do not have explicit capabilities for modeling dynamic behavior and functions. Modeling of the three aspects must be done consistently and with direct links between the artifacts of different aspects. It is only then that further analysis, such as simulation and animation, can be conducted on the modeled software systems.

The above problem leads to the lack of simulation or animation facilities in Meta-CASE tools. Except CASEMaker, other tools have not addressed this capability which would prove useful in ensuring the correctness of the captured specification of the software systems. This capability may also encourage developers in using the customized CASE tools. Simulation and animation capability is an important feature that must be included as a core function of any MetaCASE tool.

5.1.2 User Interfaces

The CASE customization user interface of the reviewed tools is either a simple text editor, a form-based system, or a graphical diagram editor depending on the language of the meta-modeling technique. Commercial tools (Toolbuilder and MetaEdit) provide the easiest user interfaces while research tools (Metaview) are not concerned with UI issues. Most tools have capabilities for building help systems during customization to help the software developers.

In the evaluated MetaCASE tools, the user interfaces of the customized CASE tools are distinctly easier to use. These tools would be beneficial if we assume that CASE customizers are different than software developers. CASE customizers must be experts while CASE users need not to know the MetaCASE technology. However, these tools are not beneficial if our goal is to allow software developers to build their own CASE tools. In this case, the user interfaces must be more consistent and suitable to software developers knowledge and expertise.

Open Problems

An important shortcoming in methodology specification and CASE tool component configuration is the lack of help systems. There is no responsive system that may help the CASE tool customizer in modeling the methodologies and configuring the components of the CASE tools. Tasks are often cumbersome and require MetaCASE tool experts.

Although the user interfaces of the customized CASE tools are much easier to use, they lack directive guidance systems. Such systems may help CASE users in selecting a methodology for a particular development project. They would also help

in navigating between different customized CASE tools. In order to achieve a high degree of developer acceptance, MetaCASE tools need to focus on these user interface issues.

5.1.3 An Object and Document Manager

All the reviewed tools provide facilities that act as engines between the user interface and the database. These facilities establish links between various objects and documents. Links ensure consistency and completeness in the modeled software system. The consistency and completeness checking is at the syntax level and is based on the modeled constraints of the underlying methodology. Consistency checking capabilities of the tools, therefore, depend on how well the employed meta-modeling technique captures the methodology related information.

In terms of management process support, all the reviewed tools are capable of some functionality. They have an active database or facilitate definition of triggers that may be used in defining events and actions, necessary for supporting management process controls. Of the examined tools, Metaview and MetaEdit acknowledge ongoing research in this area. This topic remains to be fully examined in our future research.

Open Problems

Examined implementation of the meta-modeling techniques suffer from various inconsistencies due to the extensions on their core capabilities. Metaview has difficulties in using its two extended capabilities: aggregation and graphical extensions. Toolbuilder must manage the CASE customizer defined C functions and its interface with

the database schema. Although MetaEdit does not have direct problems, it is unable to provide complex modeling capabilities (as seen in the modeling of complex constraints of FSM). The 4thought system has the burden of managing three different techniques: an ER-based database schema, a Z model, and the GraphLog filters and queries. CASEMaker is an exception but its capabilities remain to be seen. In short, since the implementation of the meta-modeling techniques of the existing tools lack a clear and predefined structure, management of the objects and documents remains as an open problem area.

In addition, an open architectural design of MetaCASE tools and their customized CASE tools is necessary to permit the integration and use of other existing tools within the customized CASE tool. This is an important developer-desired capability which adds to the complexity of managing the database objects and documents.

5.1.4 A Query and Report Manager

Among the studied tools, 4thought and CASEMaker offer the most formal querying facilities. This is due to the querying capabilities of their implemented meta-modeling techniques (GraphLog and hypernodes respectively). Other tools only allow a limited degree of querying about the captured software systems.

The reporting facilities offered by Toolbuilder and MetaEdit seem to be most comprehensive, but other tools also provide the basic capabilities. Often there are different report generation control levels but the final format is mostly text-based. The text describes the modeled methodologies or the specified software systems.

Open Problems

A high level of functionality with respect to data querying is desired. Most of the reviewed tools focus on the software systems. They allow some form of model checking of the captured software systems but do not provide any formal and extensive capabilities with respect to the captured methodologies.

An exception is the 4thought system. It uses Z formalizations but the modeled information is not used automatically within the tool. I think a worthwhile future research in this area is to build a variation of Z that could be used automatically by the tool to create the executable databases. Such a variation must be sufficiently expressive for building database schemas. It should also allow various automatic and formal model checkings of the captured methodologies.

5.1.5 Transformation and Meta-programming Tools

In all of the reviewed tools, a single repository is used to store the captured information about the software systems. This allows viewing of the same data in different formats. All of the tools offer capabilities, often as a side effect of their querying and reporting facilities, for defining transformations between phases of the software process. However, only Metaview provides a semi-automatic mechanism, through ETL language constructs, that may be used to define mathematical transformations.

Although some of the commercial tools support the later phases of the software process, many of the reviewed tools provide only the 'upper CASE' functionalities.

Open Problems

Research is necessary to examine the use of formal and automatic transformation capabilities similar to the ETL facility of Metaview. This would be between the various phases of the software process or various customized CASE tools. As an example, in 'lower CASE', this mechanism would allow meta-programming and code generation. In general, an open issue in almost all the reviewed MetaCASE tools is to provide customization capabilities of integrated CASE tools that support the entire software process.

5.2 Comparison Based on FSM Modeling

Table 5.1 summarizes my hands-on experiences during the modeling of FSM in the selected tools. I have rated the meta-modeling techniques of each tool based on how they model primitive concepts, inheritance, hierarchy, and various constraints. This rating varies from 'very simple' to 'average' to 'very difficult'. I have also categorized the tools based on their modeling notation which is either textual or visual. Furthermore, table 5.1 includes a categorization of the tools within the range of 'very low' to 'very high' expressive powers.

As seen in the table, modeling primitive concepts, inheritance, and hierarchy is a very simple task in Metaview and MetaEdit. Metaview uses a textual language for defining data without the concern for constraints which are modeled using a different set-theoretic language. MetaEdit uses a simple and visual object-oriented notation for defining data with the standard capabilities for defining simple constraints. Due to the intuitive nature of the employed meta-modeling techniques in the above tools,

	Metaview	Toolbuilder	MetaEdit	4thought	CASEMaker
Primitive Concepts	Very Simple	Average	Very Simple	Average	Average
Inheritance	Very Simple	Simple	Very Simple	Simple	Average
Hierarchy	Very Simple	Simple	Very Simple	Simple	Simple
Simple Constraints	Difficult	Simple	Very Simple	Average	Average
Complex Constraints	Difficult	Simple	Not Possible	Difficult	Very Difficult
Notation	Textual	Textual	Visual	Textual/ Visual	Textual/ Visual
Expressive Power	Average	High	Average	High	Very High

Table 5.1: Comparison of meta-modeling techniques based on how they model FSM.

defining data is very simple. This simplicity has drawbacks in the expressiveness of the used notation. Therefore, complex constraints are difficult or impossible to model in these tools.

Other tools provide an implementation of the meta-modeling techniques which is not intuitive and definitely not easy to learn. Toolbuilder uses predefined concepts, such as Nodes and Links, which are hard to manipulate. 4thought and CASEMaker use set-theoretic or logic-based languages that are very difficult to learn. However, once learned, these languages are easy to use. It is important to notice that this added learning difficulty has been offset by the high or very high level of the expressive powers of these meta-modeling techniques.

Constraints, on the other hand, are generally difficult to model in most of the tools. This is often due to the difficulty of modeling logical and mathematical constraints.

In the case of Metaview, the separation of data from constraints has made definitions of the constraints more difficult. As for other tools, they provide an implicit way of modeling the simple constraints which eases this task. However, they have difficulty in modeling the more complex constraints. This problem is most visible in the case of MetaEdit. Toolbuilder provides a solution to this problem by allowing the use of C functions in modeling complex constraints. This adds to the complexity of maintaining a consistent database.

Based on my observations, CASEMaker's hypernode meta-modeling technique provides the most expressive capabilities among the studied tools. It uses a visual notation that matches the concepts of software engineering. It also provides most of the ER and OO concepts within its Graph-based technique. However, this added expressive capability has caused usability difficulties (as seen during the modeling of the complex constraints of FSM).

5.3 Results of the Discussion

The following list summarizes the important open issues of the reviewed MetaCASE tools based on the component-based analysis and my observations:

- Implemented meta-modeling techniques have unclear and inconsistent structures.
- Information capture is limited to the data structures with little or no means of specifying the methodology prescribed process or guidelines.
- A higher degree of formality is necessary in capturing methodologies that would allow automatic model checkings.

- Model simulation/animation facilities are needed to perform analysis tasks.
- A better support of the entire software process and methodology navigation is necessary.
- An open architecture is desired to permit the integration and use of other existing development or evolution tools.
- An extensive help system is necessary during CASE customization.
- Customized CASE tools must provide guidance and recommendations on the selection of project specific methods.

Chapter 6

Conclusions

As stated in section 1.2, this dissertation examines MetaCASE tools from an architectural point of view and identifies their capabilities and shortcomings. I claimed that meta-modeling techniques implemented in the existing MetaCASE tools are limited to data-capture and are difficult to use. I further claimed that the existing MetaCASE tools provide little or no analysis facilities, CASE customizer help, or CASE user guidance systems. Demonstration of my claims has been done in the following way.

In chapter 2, I extensively described the background research in software engineering environments. In section 2.2.1, I looked at models of methodologies (RPG) and CASE tools (SMP) and for the first time identified their relationships. This led to an organized study of CASE tools and their underlying methodologies and allowed me to identify the parts of methodologies that are not supported by CASE tools (e.g., guidelines). In addition, I examined various research reports and opinions about CASE and MetaCASE and investigated the truth in many of their claims. In particular, in section 2.3 I analyzed how well MetaCASE answers the questions and

problems surrounding the adoption and use of CASE tools. I identified similarities and differences between the notations and styles of different researchers. Finally, I looked at the existing reviews on software engineering environments and established the need for a more comprehensive survey in section 2.4.

In chapter 3, I developed a framework of study to structure and examine the existing MetaCASE tools. This framework involves an architectural definition of MetaCASE tools as outlined in section 3.1. This original approach allows division of a MetaCASE tool into core components necessary to provide the MetaCASE functionality. I identified the most fundamental component to be the data modeling technique as discussed and grouped in section 3.1.1. To experiment with different MetaCASE tools and explore their capabilities, I defined a variation of finite state machines with added hierarchy in section 3.2. Another original aspect of my framework is the categorization of the MetaCASE tools (as described in section 3.3) which is based on their underlying data modeling techniques.

In chapter 4, I examined the current state of research and practice. I considered the important and representative MetaCASE tools in chapter 4. My review was based he presented component-based framework. It identified the existing or missing components of the reviewed tools. In addition, I modeled the sample method using the meta-modeling techniques of those tools. This gave me hands-on experience with the modeling techniques and an in-depth understanding of fundamental abilities and limitations of each tool.

Finally, in chapter 5, I summarized the results of my survey from the component-basis and the observational aspects. On a component-basis, I identified some of the open problems in the existing MetaCASE tools. From an observational aspect, I was

able to confirm some of my findings and provide a comparative and tabular evaluation of the examined tools. In section 5.3, I listed the results of my examinations as a summary of the MetaCASE open issues. Among the issues are the claims of my thesis which was therefore fully demonstrated.

6.1 Contributions

This dissertation has contributed to the MetaCASE research in many ways. For the first time, I have provided a much needed organization of this field, comparison of various terminologies, and clarification of the confusing views. I have also provided an extensive bibliography of over 100 MetaCASE related publications which may act as a resource for further MetaCASE research. In this dissertation I have provided a glossary of MetaCASE terminologies and described the various MetaCASE tools with a uniform terminology. Therefore, newcomers of the field now have sufficient information to understand the MetaCASE technology without being mystified.

For the first time, I have provided an architectural definition of MetaCASE tools. It allows an organized study of different components and identification of their functionality. Based on this definition, the essential core components and the alternative implementations of MetaCASE tools can be recognized. As an example, I was able to identify the most essential core component as being the employed meta-modeling technique.

Based on the employed meta-modeling techniques, I have provided a novel categorization method to organize and structure the existing MetaCASE tools. This categorization method is designed for MetaCASE tools. Categories are indicative of the nature of the tool and its possible component-based architecture. Therefore,

it is superior in comparison to the existing methods which are outdated, arbitrarily selected, or not designed for MetaCASE tools.

A major contribution of this research is the introduction of a framework of study consisting of the architectural definition, the categorization method, and the definition of a sample experimental method (FSM). This framework can be adopted for in-depth architectural analysis of MetaCASE tools. To establish the practicality of this framework, I have provided a detailed survey of the existing MetaCASE tools.

In the survey, I have included a model of the FSM methodology in each tool. These models can be used as starting points in customizing CASE tools that are based on the FSM methodology. This would be of interest to MetaCASE users with plans to customize CASE tools that allow dynamic behavioral modeling of software systems. As I discussed before, behavioral modeling has key importance in software simulation and analysis. In addition, I have evaluated the reviewed MetaCASE tools based on how well they model the FSM method. This evaluation can be used in selecting and adopting MetaCASE tools by organizations.

The survey, on its own, provides valuable information about the components of the reviewed tools. I have identified various related and differing key components of the tools and modeled the sample method using their underlying meta-modeling techniques. I have also provided comparative evaluations of the examined tools. This information may be used by MetaCASE researchers in understanding and building MetaCASE tools. It may also be of valuable use to organizations considering to adopt and use MetaCASE tools.

Perhaps the most important contribution of this dissertation is the identified open

problems of MetaCASE tools. Open problems are based both on the needs of MetaCASE users and the unavailability of desired features. This knowledge can be used by MetaCASE developers in building better and functional tools. It is hoped to make a bridge between MetaCASE user needs and tool functionality. There are various approaches to building these bridges and addressing the open problems which I will mention briefly in the next section.

6.2 Limitations and Future Research

In my research I was constrained by three major factors: available documentation, time, and budget.

My examination of MetaCASE tools was based on documentation only. Due to time and budget constraints, I was not able to purchase, learn, and use the reviewed tools. My findings during the component-based study are based on the available literature. They have not been verified against the actual tools. The modeling of FSM in various tools was also done based on the documentation and for illustration purposes only. They were not tested by the tools that employed the meta-modeling techniques which I used in the modelings. Further research in this area is necessary for experimenting with the actual tools and testing the correctness of my FSM models.

In my studies, I was able to gather a large amount of MetaCASE related publications which is evident from the bibliography of this dissertation. However, some of the commercial tools lack readily available documentation mainly due to the proprietary technology they employed. My examination consisted of only five MetaCASE tools and within the scope of this research achieved adequate results. An extension to this research may cover more tools but no surprising results are expected.

The scope of this dissertation is mainly architecture based. I examined the components of representative MetaCASE tools. However, details of the inter-connection of the components remain to be examined. Further research in this area may reveal more implementation details about MetaCASE tools and together with performance analysis experiments provide a way of identifying performance bottlenecks. Therefore, a sophisticated experimental system development study, which is beyond the scope of my research, could be undertaken to address further issues such as tool usability, effectiveness, and performance.

As one of the contributions of my dissertation I identified a list of MetaCASE open issues in section 5.3. Here, I provide a list of some of the important avenues for future research and possible routes:

- To extend the functionality of MetaCASE from purely data-capture tools, more comprehensive implementation of the meta-modeling techniques are needed to allow capture of process and guidelines information from methodologies. Process information may be captured using model constructs allowing the definition of orders for tasks and documents and various transformation capabilities. Guidelines may be automated using knowledge bases and cognitive reasoning tools [52].

- To allow automatic model checkings, more formal meta-modeling techniques need to be used. One possible way is to develop an easy to understand version of a formal language, like Z, which is automatically used by the tool. This is an active current research area that may be applied to MetaCASE [3].

- To analyze the captured data, model simulation/animation facilities are necessary. A good approach is to select or develop a modeling technique with the

intentions of simulation/animation in mind. Such a technique will have the data structures as well as the functions and the dynamic behavior modeling capabilities. To animate the software system, messages from the simulated models will trigger the animator component and that may create animated processes. As a result, captured methodologies and modeled software systems can be analyzed and their performances may be evaluated [37].

- To address the user interface problems of MetaCASE tools, research is needed to examine the human computer interaction issues relating to MetaCASE. A possibility for a usable and familiar interface is to use a web browser with Java applets to visualize, manipulate, and even animate data. Some of the issues to consider are performance, data consistency, and screen updates. Another approach is to use a more specialized, distributed, interactive, and graphical interface such as Clock [35].

MetaCASE research in Software Technology Laboratory is at its early stages. Results of this dissertation may allow the formulation of a list of requirements necessary to build a MetaCASE tool. It is our objective to address many of the issues identified as open problems of MetaCASE tools. The most essential part of a MetaCASE tool, as identified in this study, is the meta-modeling technique which we plan to design and prototype in the near future.

Bibliography

[1] A. Alderson. MetaCASE technology. In *Proceedings of European Symposium on SDE and CASE Technology*, Lecture Notes in Computer Science, pages 81–91. Springer-Verlog, June 1991.

[2] G. Allen and A.R. Jackson. MetaCASE technology in the support of information systems teaching. In *Proceedings of First International Conference on MetaCASE.* Sunderland, UK, 1995.

[3] P. Allen, M. Rawson, and B. Ryan. A tool to support OOA and Z: An integrated approach. In *Proceedings of First International Conference on MetaCASE.* University of Sunderland, UK, 1995.

[4] M.J. Aslett. *A Knowledge Based Approach to Software Development: ESPRIT Project ASPIS.* North-Holland, Amsterdam, 1991.

[5] P. Bergsten, J. Bubenko, R. Dahl, M. Gustafsson, and L.A. Johansson. RAMATIC - A CASE shell for implementation of specific CASE tools. Technical Report T6.1, TEMPORA Project, SISU, Stockholm, 1989.

[6] G. Boloix, P.G. Sorenson, and J.P. Tremblay. Process modeling using a Metasystem approach to software specification. Technical Report TR 92-11, Department of Computer Science, University Of Alberta, Canada, 1992.

[7] G. Boloix, P.G. Sorenson, and J.P. Tremblay. Transformations using a Metasystem approach to software development. *Software Engineering Journal*, 7:425–437, 1992.

[8] G. Boloix, P.G. Sorenson, and J.P. Tremblay. Software metrics using a metasystem approach to software specification. *Systems Software*, 20:273–294, 1993.

[9] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, Redwood City, 1993.

[10] T. Bruckhaus, N.H. Madhavji, I. Janssen, and J. Henshaw. The impact of tools on software productivity. *IEEE Software*, 13:29–38, September 1996.

[11] J.Jr. Bubenko. Selecting a strategy for computer-aided software engineering (CASE). Syslab Report 59, University of Stockholm, Sweden, June 1988.

[12] Y. Chan and S. Huff. Assessing information systems strategy. Working Paper 93-05, Queen's University, February 1993.

[13] R.N. Charette. *Software Engineering Environments Concepts and Technology*. McGraw-Hill, 1221 Avenue of Americas, New York, 1986.

[14] M. Chen and J.F.Jr. Nunamaker. MetaPlex: An integrated environment for organization and information systems development. In *Proceedings of 10th International Conference on Information Systems*, pages 141–151. ACM Press, New York, 1989.

[15] P. Chen. The Entity-Relationship model towards a unified view of data. *ACM Transactions on Database Systems*, 1, March 1976.

[16] J. Chikofsky. Software technology people can really use. *IEEE Software*, pages 8–10, March 1988.

[17] M.P. Consens. Creating and filtering structural data visualizations using Hygraph patterns. Ph.D. Thesis, University of Toronto, Canada, 1994.

[18] M. Crozier, D. Glass, J. Hughes, W. Johnston, and I. McChesny. Critical analysis of tools for CASE. *Information and Software Technology*, 31:486–496, November 1989.

[19] Customizer reference guide. Index Technology Co., Cambridge, USA, 1987.

[20] M. DeBellis. The knowledge-based software assistant program. Notes Form Lectures in Stanford University. Anderson Consulting, Palo Alto, USA.

[21] T. DeMarco. *Structured Analysis and System Specification.* Prentice-Hall, New Jersey, 1978.

[22] A. Diller. *Z: An Introduction to Formal Methods.* John Wiley and Sons, Chichester, England, 1990.

[23] M.B. Dixon, J.F. Coxhead, and E.A. Dodman. MetaCASE and Audit: Automated generic quality assessment. In *Proceedings of First International Conference on MetaCASE.* Sunderland, UK, 1995.

[24] A.L du Plessis. A method for CASE tool evaluation. *Information and Management*, 25:93–102, 1993.

[25] A. Endres. Preface. In *Proceedings of European Symposium on SDE and CASE Technology*, Lecture Notes in Computer Science. Springer-Verlog, June 1991.

[26] A.V. Lamsweerde et al. Generic lifecycle support in the ALMA environment. Tr, Honeywell Systems and Research Center, Minneapolis USA, 1989.

[27] Excelerator reference guide. Index Technology Co., Cambridge, USA, 1987.

[28] S.I. Feldman. MAKE - a program for maintaining computer programs. *Software Practice Experiences*, 9:255–265, 1979. Standard Reference.

[29] R.I. Ferguson. The beginner's guide to IPSYS TBK. University of Sounderland Occasional Paper 93/3, 1993. http://osiris.sunderland.ac.uk /rif/metacase/metacase.tools.html.

[30] P. Findeisen. The EARA model for Metaview: A reference. University Of Alberta Research Homepage Article, 1994. http://web.cs.ualberta.ca /šofteng/Metaview/project.html.

[31] P. Findeisen. The Metaview system. University Of Alberta Research Homepage Article, 1994. http://web.cs.ualberta.ca /šofteng/Metaview/project.html.

[32] G. Forte. Tools fair: Out of the lab, Onto the shelf. *IEEE Software*, pages 70–79, May 1992.

[33] D. Garlan and M. Shaw. An introduction to software architecture. SEI TR 21, Carnegie-Mellon University, Pittsburgh, USA, 1993.

[34] A. Gill. *Introduction to the Theory of Finite-State Machines.* McGraw-Hill, USA, 1962.

[35] T.C.N. Graham, C.A. Morton, and T. Urnes. ClockWorks: Visual programming of component-based software architectures. *Journal of Visual Languages and Computing*, pages 175–196, July 1996.

[36] Graphical Designer reference guide. Advanced Software Technologies, Colorado, USA, 1996. http://davinci2.csn.net/~jefscot/index.html.

[37] D.R.C. Hill. *Object-Oriented Analysis and Simulation.* Addison- Wesley, USA, 1996.

[38] A. Isazadeh. Behavioral views for software requirements engineering. Ph.D. Thesis, Department of Computing and Information Science, Queen's University, Canada, 1996.

[39] S. Isoda. SoftDA - A computer aided software engineering system. In *Proceedings of Fall Joint Computer Conference*, pages 142–151, 1987.

[40] S. Isoda, S. Yamamato, H. Kuroki, and A. Oka. Evaluation and introduction of the structured methodology and a CASE tool. *Journal of Systems and Software*, 28:49–58, 1995.

[41] M.A. Jackson. *System Development.* Prentice-Hall, New Jersey, 1983.

[42] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering, A Use Case Driven Approach.* Addison Wesley, Reading, Massachusetts, 1992.

[43] M. Jarke. ConceptBase: A deductive object manager for meta databases. Reference Guide, 1996. http://www-i5.informatik.rwth-aachen.de/CBdoc /cbflyer.html.

[44] J.M.Brant. HotDraw. M.Sc. Thesis, University of Illinois, USA, 1992.

[45] A. Karrer and W. Scacchi. Meta environments for software production. University of Southern California Homepage Article, 1994. http://www.usc.edu /edu/dept/ATRIUM/index.html.

[46] S. King, P. Layxell, and S. Williams. CASE 2000: The future of CASE technology. *Software Engineering Journal*, 9:138–139, 1994.

[47] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[48] P. Laamanen. Automation of software product metrics: A proposal for a metamodel based metrics engine. In *Proceedings of First International Conference on MetaCASE*. Sunderland, UK, 1995.

[49] D.A. Lamb. *Software Engineering, Planning for Change*. Prentice-Hall, New Jersey, 1988.

[50] D.A. Lamb, A. Malton, and X. Zhang. Applying the Theory-Model Paradigm. Internal Technical Report 1996IR-01, Queen's University, February 1996.

[51] M. Levene and A. Poulovassilis. The Hypernode model and its associated query language. In *Proceedings of the 5th Jerusalem Conference on Information Technology*, pages 520–530. IEEE Computer Society Press, October 1990.

[52] M. Lioyd. Knowledge based CASE tools: Improving performance using domain specific knowledge. *Software Engineering Journal*, 9, July 1994.

[53] K. Lyytinen and P. Kerola. MetaPHOR: Meta-modeling, principles, hypertext, objects, and repositories. TR 7, Department of Computer Science, University Of Jyväskylä, Finland, December 1994.

[54] G. Maokai and L. Scott. Developing the user interface for the MetaCASE toolset. Research report, Macquarie University Joint Research Center for Advanced Systems Engineering, Sydney, Australia, 1996.

[55] P. Martiin. Towards flexible process support with a CASE Shell. In *Proceedings of Advanced Information Systems Engineering CAiSE'94*, Lecture Notes in Computer Science, pages 14–27. Springer-Verlog, June 1994.

[56] P. Martiin, M. Rossi, V. Tahvanainen, and K. Lyytinen. A comparative review of CASE shells: A preliminary framework and research outcomes. *Information and Management*, 25:11–31, 1993.

[57] J. Martin and C. McClure. *Design Techniques for Analysis and Programmers.* Prentice-Hall, New Jersey, 1985.

[58] A. McAllister. Modeling concepts for specification environments. Ph.D. Thesis, University of Saskatchewan, Canada, 1993.

[59] C. McClure. *CASE is Software Automation.* Prentice-Hall, New Jersey, 1989.

[60] MetaEdit+ refernce guide. Metacase Consulting, Jyväskylä, Finland, 1996.

[61] S. Misra. CASE system characteristics: Evaluative framework. *Information and Technology*, 32:415–422, July 1990.

[62] I. Mitchell and C. Hardy. MetaCASE editorial. In *Proceedings of First International Conference on MetaCASE.* Sunderland, UK, 1995.

[63] R.J. Norman and M. Chen. Working together to integrate CASE. *IEEE Software*, pages 13–16, March 1992.

[64] K.S. Oakes, D. Smith, and E. Morris. Guide to CASE adaption. SEI TR 15, Carnegie-Mellon University, Pittsburgh, USA, 1992.

[65] ObjectMaker reference guide. Mark V Systems, Encino, California, 1996. http://www.markv.com/.

[66] J.K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1 Jacob Way, Reading, Massachusetts, 1994.

[67] D.E. Perry and G.E. Kaiser. Models of software development environments. *IEEE Transactions on Software Engineering*, 17, March 1991.

[68] R.S. Pressman. *Software Engineering, A Practitioners Approach.* McGraw-Hill, 1221 Avenue of Americas, New York, 2 edition, 1987.

[69] QuickSpec reference guide. Meta Systems Ltd., Ann Arbor, Michigan, 1989.

[70] J. Rader, A.W. Brown, and E. Morris. An investigation into the state of the practice of CASE tool integration. SEI TR 15, Carnegie-Mellon University, Pittsburgh, USA, 1993.

[71] Rational news. Company Homepage, 1995. http://www.rational.com/htdocs /news/pr146.html.

[72] A. Reeves, M. Marashi, and D. Dudgen. A software design framework or how to support real designers. *Software Engineering Journal*, pages 141–155, July 1995.

[73] T. Rose and M. Jarke. A decision-based configuration process model. In *Proceedings of 12th International Conference on Software Engineering*, pages 316–325. Nice, France, 1990.

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

[75] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Deign.* Prentice-Hall, New Jersey, 1991.

[76] A. Ryman. Why programmers don't use software design tools (and how to build tools they will). York University Seminar Series, Toronto, Canada, 1997. http://www.cs.yorku.ca/General/seminars/abstracts/ryman.html.

[77] A.G. Ryman. The Theory-Model Paradigm in software design. Technical Report 74.048, IBM Co., October 1989.

[78] A.G. Ryman. Foundations of 4thought. In *Proceedings of CAS Conference*, pages 133–155, Toronto, November 1992. IBM CANADA.

[79] A.G. Ryman. Constructing software design theories and models. In *Proceedings of ICSE'93 Workshop: Studies of Software Design*, Lecture Notes in Computer Science, pages 103–114. Springer-Verlog, May 1993.

[80] L. Scott. Hypernode model support for software design CASE. Research Report 96/7, Macquarie University Joint Research Center for Advanced Systems Engineering, Sydney, Australia, July 1996.

[81] L. Scott. MetaCASE concept document. Research Report 96/8, Macquarie University Joint Research Center for Advanced Systems Engineering, Sydney, Australia, August 1996.

[82] SEEWG: A software engineering environment for the Navy. NAVMAT Software Engineering Environment Group Report, March 1982.

[83] S. Shlaer and S. Mellor. *Object Lifecycles Modeling the World in States.* Prentice-Hall, New Jersey, 1992.

[84] K. Smolander, K. Lyytinen, V. Tahvanainen, and P. Martiin. MetaEdit- A flexible graphical environment for methodology modeling. In *Proceedings of Advanced Information Systems Engineering CAiSE'91*, Lecture Notes in Computer Science, pages 168–193. Springer-Verlog, May 1991.

[85] P.G. Sorenson and J.P. Tremblay. Using a Metasystem approach to support and study the design process. In *Proceedings of ICSE'93 Workshop: Studies of Software Design*, Lecture Notes in Computer Science, pages 88–102. Springer-Verlog, May 1993.

[86] P.G. Sorenson, J.P. Tremblay, and A.J. McAllister. The Metaview system for many specification environments. *IEEE Software*, pages 30–38, March 1988.

[87] A.V. Staa and D.D. Cowan. An overview of the Totem software engineering Meta-Environment. Technical Report PUC-Rio.infMCC 35/95, Pontificia Universidade Catolica do Rio de Janeiro, Brazil, November 1995.

[88] M.G. Stilwell, J.P. Tremblay, P.G. Sorenson, and P. Findeison. A survey of customizable Metasystems. University of Saskachewan communications, 1996.

[89] S.C. Stobart, J.B. Thompson, and P. Smith. The use, problems, benefits, and future directions of CASE in the UK. *Information and Software Technology*, 33:629–636, 1991.

[90] Sunderland homepage article: MetaCASE. University of Sunderland, UK, 1996. http://osiris.sunderland.ac.uk/rif/metacase/metacase.tools.html.

[91] W. Teielman and L. Masinter. Interlisp programming environment. *IEEE Computer*, 14:25–33, April 1981. Standard Reference.

[92] I. Thomas. PCTE interfaces: Supporting tools in software engineering environments. *IEEE Software*, 6:15–23, November 1989.

[93] T.F. Verhoef and A.M.T. Hofstede. Structuring modeling knowledge for CASE Shells. In *Proceedings of Third International Conference on CAiSE*, pages 502–524. Springer-Verlag, May 1991.

[94] I. Vessey, S. L. Jarvenpar, and N. Tractinsky. Evaluation of vendor products: CASE tools as methodology companionships. *Communications of ACM*, 35:90–105, April 1992.

[95] X. Wang and P. Loucopoulos. The development of Phedias: a CASE Shell. In *Proceedings of CASE'95*, pages 122–131. IEEE Computer Society Press, July 1995.

[96] A.I. Wasserman. Tool integration in software engineering environments. In *Proceedings of Software Engineering Environments Conference*, Lecture Notes in Computer Science, pages 137–149. Springer-Verlog, 1989.

[97] J.L. Whitten, L.D. Bentley, and V.M. Barlow. *Systems Analysis and Design Methods.* Irwin, Burr Ridge, Illinois, 1994.

[98] Y. Yamamoto. An approach to generation of software lifecycle support systems. Ph.D. Thesis, University of Michigan, USA, 1981.

[99] E. Yourdon. *Modern Structured Analysis.* Yourdon Press, Englewood Cliffs, New Jersey, 1989.

[100] E. Yourdon and L.L. Constantine. *Structured Design.* Yourdon Press, Englewood Cliffs, New Jersey, 1979.

[101] X. Zhang. A Theory-Model formalization of Shlaer-Mellor Object-Oriented Analysis. In *Proceedings of CAS Conference*, pages 324–333, Toronto, November 1994. IBM CANADA.

[102] Y. Zhuang. Object-Oriented modeling in Metaview. MSc. Thesis, University of Alberta, Canada, 1994.

Glossary

This is the glossary of some of the terms used in this dissertation. For a more detailed discussion please see section 2.3 and figure 2.6.

MetaCASE Developer:	The definer and builder of the MetaCASE tool generic components and mechanisms
CASE Customizer:	The definer and customizer of the specific CASE tools generated by a MetaCASE
Information Engineer:	Refers to the CASE Customizer
Customized CASE:	The product of CASE customization process in the Meta-CASE tools
Software System:	The product of software development process
CASE User:	The software engineer who is using CASE tools to develop software systems
Software Developer:	Refers to the CASE User
Customer:	Is the final user or purchaser of the developed software systems
MetaCASE User:	Refers to the CASE Customizer and the CASE User of MetaCASE tools