MASTER IN COMPUTER SCIENCE

An Experiment with Marvel, a Software Development Environment

Ernst, Christoph; Goetzinger, Serge

Award date:
1992

Awarding institution:
University of Namur

FACULTES UNIVERSITAIRES
N.D. DE LA PAIX
NAMUR

INSTITUT D' INFORMATIQUE

An Experiment with Marvel, a Software Development Environment

par **Christoph ERNST**
Serge GOETZINGER

Promoteur

Professeur **Eric DUBOIS**

Mémoire présenté pour l'obtention du grade
de Licencié et Maître en Informatique

Année académique 1991 - 1992

rue Grandgagnage, 21, B-5000 NAMUR (Belgium)

Abstract

In this MS thesis we first describe the main concepts that underly "state of the art" software development environments. Then we give an in-depth study of a specific software development environment called Marvel. Finally, we use Marvel for developing an environment supporting a transformational process covering the whole software lifecycle and study the ability of Marvel for responding to this challenge.

Keywords: Software Development Environments, Software Process, Object Oriented Database, Rule Based Production Systems.

Résumé

Dans ce mémoire nous décrivons d'abord les concepts principaux qui sont à la base des ateliers logiciels. Ensuite, nous faisons une étude approfondie d'un atelier logiciel spécifique, à savoir Marvel. Finalement, nous utilisons Marvel pour générer un environnement supportant un processus transformationnel et couvrant l'entièreté du cycle de vie d'un logiciel afin d' étudier la capacité de Marvel de répondre à ce défi.

Mots-clé: Ateliers logiciels, Processus de développement, Base de données orientée objet, Systèmes de règles de production.

Acknowledgements

We gratefully acknowledge our supervisor Professor Eric Dubois for his constant support and faithful supervision throughout the development of this MS thesis.

A special thanks to the Marvel team of the Computer Science Department of Columbia University, especially Professor Gail Kaiser, Israel Ben-Shaul and George Heineman, for their advice and assistance during our stay in New York.

We wish to thank Philippe Du Bois of Namur University and Guy Vanden Bemden of University of Louvain-la-Neuve for their technical advice and helpful discussions during the elaboration of our environment.

We would also like to thank Pierre Flener for his careful reading and constructive remarks which helped us to produce this manuscript.

We would like to acknowledge our parents for their continuous support throughout our studies.

Christoph Ernst and Serge Goetzinger

TABLE OF CONTENTS

Table of Contents

PART II : IN DEPTH STUDY OF THE MARVEL SOFTWARE DEVELOPMENT ENVIRONMENT 21

INTRODUCTION

INTRODUCTION

With the introduction of third generation computers at the end of the 60s, it became possible to develop large software systems. However, such developments require:

1) the emergence of methods for specifying designing, coding and maintaining software pieces;

2) the possibility of coordinating teams of analysts and programmers.

The term *software engineering* was introduced at the beginning of the 70s for covering these aspects. Software engineering can be defined as "a disciplined utilization of a systematic, coherent set of principles, tools and techniques within a planned organizational framework, the objective of which is to achieve the on-schedule, cost effective development of quality software for a diverse range of non-trivial applications and environments " [Rat87]p3.

Quality software includes the following characteristics :

- **reliability** of a software system is the probability that it exhibits failure-free behavior over a specified time span.

- **efficiency** refers to the economical use of hardware resources .

- **usability** refers to the user friendliness of a software product.

- **security** refers to the ability of a software product to be safe against misuse.

- **reconfigurability** refers to the ability of software to be changed successfully in any required way and for any reason with minimal cost and effort.

- **reusability** refers to the extent to which the software can be reused in as many operational environments and applications as possible, within a given application area, with minimal alteration or further development.

To improve the quality of a software and to achieve higher team productivity, Software Development Environments have been proposed

In this MS thesis, we have specifically studied the Marvel Software Development Environment.

Until now, Marvel has only be used for programming-in-the-large tasks (i.e. the coding and maintenance phases) of large C applications (C/Marvel environment). Marvel offers assistance to the user for the carrying out of these tasks. Thus it can be characterized as active, as it may autonomously intervene during the coding and the maintenance phases, and is not restricted the static description of the objectbase (passive environment).

We try to develop a Marvel environment that covers the whole software lifecycle. This means that we investigate to what extent the Marvel kernel may be instantiated to development-in-the-large tasks rather than to programming-in-the-large tasks. We especially pay attention whether the dynamic features noticed in the programming-in-the-large environment (e.g. C/Marvel) are still operational in the development-in-the large environment.

In the first Part, we present the main goals of a Software Development Environment and describe a framework for an integrated environment covering the whole lifecycle. An analyze of four of them, namely ALMA, Adèle, ISTAR and Marvel, emphasizing their main features and deficiencies, is presented at the end of this part.

In the second part, we describe the architecture of the Marvel Kernel. The static part of a Marvel environment is defined in terms of object oriented classes definitions whereas the dynamic part is taken into account by production rules. Both definitions are written in a Marvel specific programming language. Marvel makes use of external tools to perform the activities encapsulated in a Marvel rule. These activities are defined in a specific communication language presented at the end of this part.

In the third part, we describe a development process that we try to instantiate with the Marvel kernel. We give a detailed description of the objectbase representing the main concepts of the development process and explain how it has been elaborated. Moreover, we analyze the process and decompose it into tasks that may be performed by Marvel rules. To specify the rules, we divide them into several categories. The result of our investigation of the different Marvel features is given at the end of this part.

PART I

SOFTWARE DEVELOPMENT ENVIRONMENTS

0. INTRODUCTION

0.1 Historical Evolution

To face the software crisis [Som89], computer scientists or more precisely, software engineers [Som89], have proposed new lifecycle-based methods for the development of large and complex software systems. First methods were proposed in the seventies.

The use of these methods in the industry was however weak. This had one main reason: they were executed manually (pencil and paper) and were thus very labor intensive. Moreover, the execution of those methods needed also the recording of a lot of information of different nature (text and graphics).

The only realistic solution to this problem was the automation of the different activities and techniques required by the methods by using software development tools. Editors, screen generators, data dictionaries and code generators are some examples of such tools. This led to the notion of **Software Development Environments** (SDE).

First generation environments presented, however, some deficiencies. They were isolated and only addressed specific phases of the software process, i.e. they only focused on a limited aspect of the whole software development process. Moreover, those tools enforced the user to adapt to the tool's way of thinking. Finally they did not assist the user of the environment **actively** [Vla82] in the software development process.

Second generation environments, from 1980 on, have partially remedied to first generation environments deficiencies. They cover the whole lifecycle by integrating complementary and compatible tools into a coherent set. This is implemented by articulating the tools around a common information database, called a **repository** [How81], [Ste87]. These environments are also called Computer Aided Software Engineering (**CASE**) environments as they enforce well-known software engineering methods. These CASE environments also make use of graphic manipulation facilities.

Third generation environments, which have appeared in the late 80's, aim at providing more user guidance during the software development process. These environments are **active** and **process-driven**. This means that they do not only have knowledge about the **static** structure (objects and their relationships) of a software lifecycle model, but they have also knowledge about the specific process (the lifecycle **dynamics**).

0.2 The Structure of this Part

This part is divided in two major sections. The first section (Section 1) gives a detailed description of second generation Software Development Environments. We begin by describing the goals of an SDE (Section 1.1), namely lifecycle coverage, assistance in the implementation of a method and automation in the software development.

In the next section (Section 1.2) we present a framework for an integrated SDE inspired from [Was89].

Moreover, we explain in more detail the concept of integration (Section 1.2.1) and describe the different types of integration that can be addressed. These are platform, presentation, data, control and process integration. Besides the integration mechanism we also present the vertical and horizontal software development tools that are presented in the framework. Finally (Section 1.2.2) we present the concept of repository and describe the different functionalities of CASE tools. These functionalities are grouped into creation and manipulation, documentation, verification and validation as well as project management functionalities.

In the second section (Section 2) we analyze four SDE, namely ALMA, ISTAR, Adèle2 and Marvel 3.0. The analysis consists in testing how far these SDEs verify the criteria of tool integration, lifecycle coverage, customizability, active user guidance and scale. First (Section 2.1) we give a detailed description of every criterion before analyzing the different tools (Section 2.2). We will not try to classify the analyzed SDEs but we will only try to show what are the main features and the main deficiencies of every environment.

1. SOFTWARE DEVELOPMENT ENVIRONMENTS

In this section we describe what a SDE is. This is done by presenting first the main goals of an SDE, that explain why should such an environment be used. Then we present a framework of an integrated SDE and explain the concepts of **integration** and **CASE functionality**.

1.1 The Goals of a SDE

The different goals of an SDE that are mentioned in the literature are the lifecycle coverage, the assistance in the implementation of a method and automation in the software development [Ost81], [Ste87], [Rat87], [Som89].

1.1.1 Lifecycle Coverage

"Software engineering environments are designed to support all stages of the software process from initial feasibility studies to operation and maintenance" ([Som89]p.381).

Thus an SDE should propose tools that support the requirements definition, the design, the implementation, the test and the maintenance phases. However, it is important to notice that tools supporting the implementation and test phases have quite a long tradition, whereas powerful tools that also support the requirements definition and the design process have appeared only recently.

1.1.2 Assistance in the Implementation of a Method

We first define the concept of "method" and then summarize the different tasks that can be assisted by the tools.

A method consists in (i) **models** for the specification of the different products of the software development phases, (ii) a **process** by which specifications are constructed and, (iii) a set of **tools** to aid design. Thus a method can be represented as follows :

- The **models** describe the relevant software development objects and their relationships. A model is a **static** entity.

- The **software process** itself is a **dynamic** entity and can be described as being the "set of activities carried out in order to effect the development or evolution of a software product" [Ost87].

- **Tools** can be divided in two classes [Vla82] : a) **active tools** that assist the software engineer directly in the application of a precise method, b) **passive tools** that diminish the non creative and boring aspects of her/his task. It is important to notice that the tools are inherent to the method that they help to implement.

The method itself tries to guarantee the accuracy, the feasibility, the return, the reliability, the quality and the evolution of the target information system to be developed. Thus the implementation of a method needs :

- the use of construction, transformation, normalization, derivation and control rules;

[Dat89]

- a rigorous specification of the results of each phase of the software development cycle;
- a consistency control between the different schemas;
- the support of different representations and different views according to the different users;
- a constant up-to date documentation base.

1.1.3 Automation in the Software Development

Automation can be applied at different levels in the software development process. For instance, automation can be applied to tasks during the development process such as schema transformation, code generation (e.g. a compiler) or error detection. Error detection should, however, not only detect the error, but also provide help for error correction.

1.2 A Framework for an Integrated SDE

The important concept in CASE environments is **tool integration**. The tools address different aspects of the development process and thus integration is intended to produce complete environments that support the entire software development lifecycle. This is implemented by integrating tools addressing different phases of the lifecycle and which cooperate and interface each with each other consistently and reliably thanks to a common database.

Figure 1.1 presents a framework for an integrated SDE which has been inspired from Wasserman [Was89]. It is important to notice that the requirement tools figure in the vertical tools class. The reason is that we consider that these tools have their proper formal languages to specify the requirements.

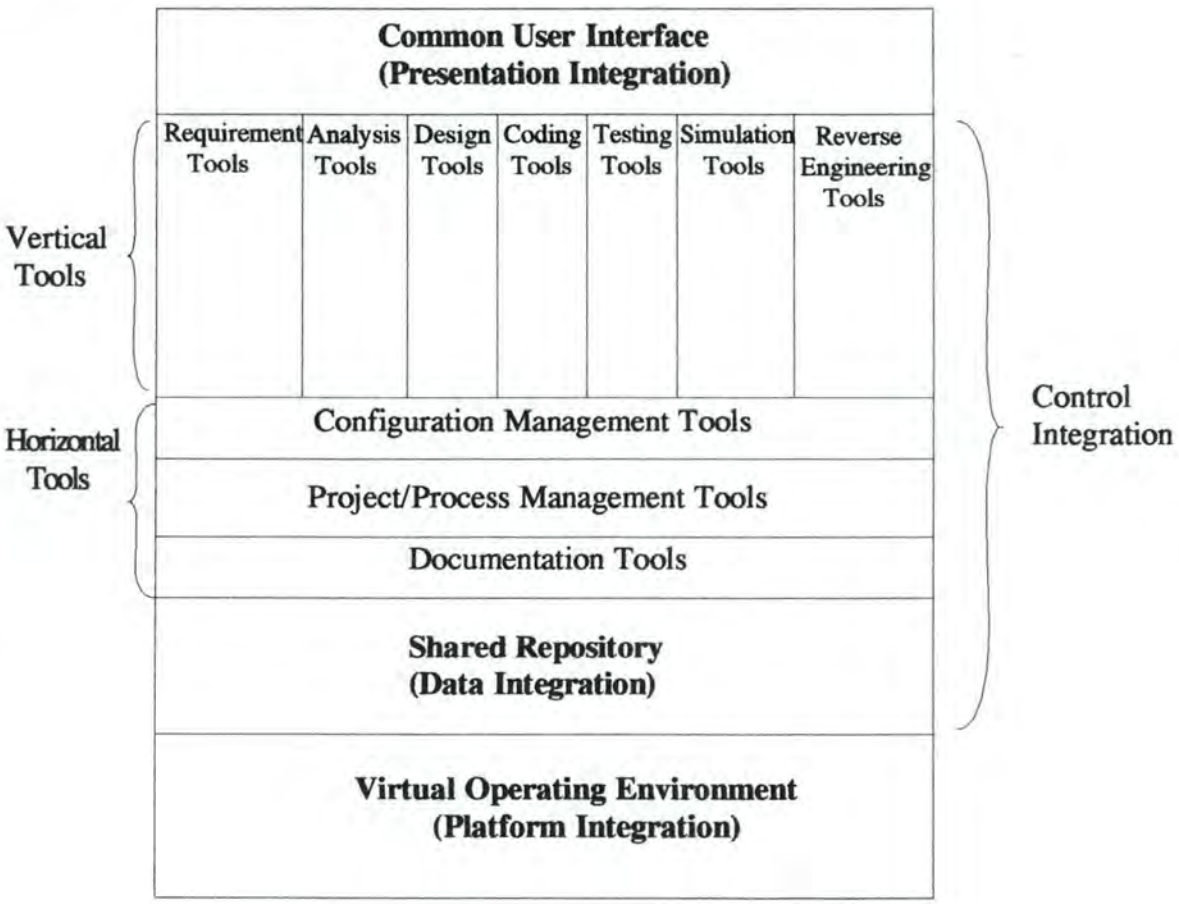

Figure 1.1: Framework for an Integrated SDE

1.2.1 Types of Tool Integration

There are five types of tool integration that must be addressed : platform integration, presentation integration, data integration, control integration and process integration.

1.2.1.1 Platform Integration

The fundamental issue for an integration is that the various tools must be inter-operable. Distributed processing makes possible to use network-based file systems and network communication to convert and transmit files from one execution environment to another and avoids that all software tools have to run on the same machine with the same operating system.

These distributed computing services give users a single consistent view of the computing environment and one can think of tools residing on top of a **virtual operating environment** which provides distributed services. A CASE environment can be seen as a set of applications that use the virtual operating environment.

Thus **platform integration** can be described as the set of system services that provide network and operating systems transparency to these applications. The advantage of this virtual operating model is that it takes heterogeneous hardware and software configurations into account [Was89].

1.2.1.2 Presentation Integration

Tools should not only work in the same operating environment, but they should also share a common **look and feel** [Cou91] from the user's perspective.

This common look and feel or **common user interface** should be designed to suit the needs and abilities of the individual users, it should be tailored to the user's means and not force the latter to adapt to a particular interface [Ost81].

Another important principle of the interface should be its **consistency** [Shn86],[Cou91]. This means that system commands and menus should have the same format, parameters should be passed to all commands in the same way and punctuation should be similar. Interface consistency across subsystems, which can be independently activated, is equally important.

Finally the user interface should have built-in **help** facilities accessible from the user's terminal and should provide different levels of help and advice [Cou91].

In order to share such a Common User Interface among tools, presentation standards based around a standard window manager and a set of "look and feel" guidelines are needed (e.g. OSF/Motif).

1.2.1.3 Data Integration

Tool integration requires both sharing of data among tools and managing the relationships among data objects produced by different tools. Traditionally, files and interprocess communication have been used for this purpose (first generation SDE, e.g. Unix). The current trend, however, is the use of a shared database (**repository**) that allows the different tools to communicate. This repository consists of a database storing the needed information (called objects) in a structured way. The tools present functionalities that allow to insert, modify, validate and check data and access data produced by other tools. The repository will further explained in Section 1.2.2.

1.2.1.4 Control Integration

Tools must be able to notify one another of events, i.e. they must cooperate to achieve a coordinated effect. As an example one can give an engineering design tool that is being used to create a design together with a technical publishing tool that is being used to print out the design document. Ideally, the document should be updated so that the designer does not have to do so. **Control Integration** refers to the ability of tools to perform such notification, as well as the ability to activate the tools under environment control [Was89].

1.2.1.5 Process Integration

Finally, the major benefits of CASE are achieved when it is used to support a well-defined **software engineering process** (Section 1.1.2). If the development process is well understood, the use of tools will become important. The latter can help to "define and track their software development activities so that they can manage the process more effectively and continually improve it " [Ost87].

Such process management tools must be integrated with the software development tools (Section 1.2.1.6) that are used at the different phases of software development. The process management tools will have to use the same mechanisms for presentation integration, data integration and control integration that are used by the other tools in the environment [Was89].

1.2.1.6 Software Development Tools

One can identify two classes of software development tools outside the integration mechanisms : **vertical tools** and **horizontal tools.**

Vertical tools are used for a specific phase of the software development process. They include requirement tools, analysis tools, design tools, coding tools, testing tools, simulation tools and reverse engineering tools.

Horizontal tools are used throughout the software development process and include tools for documentation, configuration management and project/process management.

1.2.2 CASE Tool Functionalities

An important element for providing an integrated SDE is the repository. The different development tools access the repository by providing different functionalities, namely creation and manipulation, documentation, validation and verification, and project management. The different functionalities are represented in Figure 1.2.

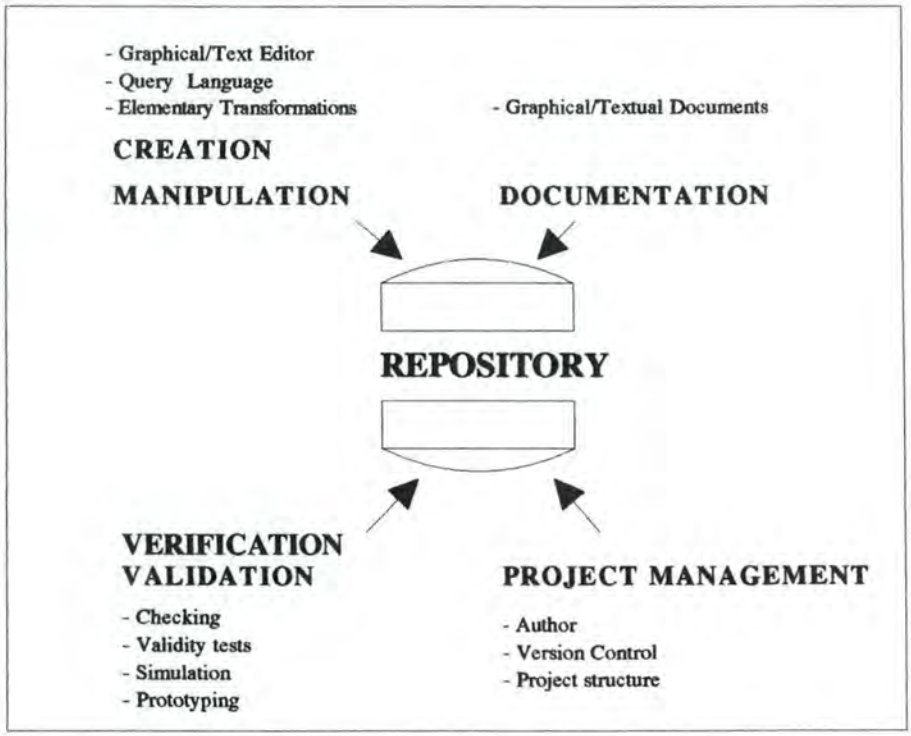

Figure 1.2: CASE Tool Functionalities

The Repository [How81], [Ste87]

During the development process of a software application, the software engineer has to deal with a lot of different types of information. This information can be code modules, test plans, design documents, requirement specification or any other software development product.

The repository or software engineering database allows to store and retrieve all these lifecycle products, called objects, their properties and their relationships. The named objects may be textual or internal representations of graphical objects.

The repository presents thus an integrated and unified medium for interfacing tools. The latter obtain their information from the repository and return their results to it without having to interface directly with other tools.

This integrated tools system eliminates the need for multiple copies of the same information. The Portable Common Tool Environment (**PCTE**) offers a concrete example of a repository.

The Creation/Manipulation Functionality

The access to the repository is ideally supported through a graphical editor. The specifications are entered in diagram form and their contents and forms are stored in the database.

If the dictionary is a database managed by a DataBase Management System (DBMS) package, the access is easy to implement. Sometimes the tool has a query language, like SQL, that allows to query the repository. In other cases, it is limited to predefined queries that can be accessed through a command menu.

Some transformations, like the transformation of an Entity-Relationship scheme to a collection of relation schemas or the normalization of the latter are performed by the tool.

The Documentation Functionality

The documentation function supports the integration of all information relative to the software project into the data dictionary. This means that the project documentation (from requirements specification to the final acceptance test plan) can be developed and maintained within the SDE. The project team has thus the possibility to access to a consistent, coherent and up-to-date documentation. The latter consists in graphical and textual descriptions such as diagrams or reports.

The Verification/Validation Functionality

The **verification function** analyzes if the specifications and programs stored in the repository are correct, i.e. it checks the **conformity**, **consistency** and **completeness** of the specification with respect to the conceptual model defining the dictionary's structure.

The **validation function** consists in the invocation of a help tool visualizing the schema by using prototyping or simulation techniques

The Project Management Functionality

The project management function should include version control and access control, i.e. the access to certain information of the repository is limited to a small number of persons (e.g. manager vs. programmer). Moreover, all the other project management tools, such as scheduling and planning tools, should have direct access to the repository.

2. ANALYSIS OF SDEs

In this section we analyze four different Software Development Environment's namely **ALMA** [Vla86], [Vla88], [Vla90], **ISTAR** [Ste86], [Dow87], **Adèle2** [Bel91], [Bel92], and **Marvel** [Kai88a], [Kai88b], [Kai90].

As it is impossible to analyze all of the existing SDEs, we just concentrate on those four, which we think to be representative of the state of the art. Of course, we are aware of the fact that no single environment can satisfy all the different needs and levels of expertise of the various users.

This section is organized as follows. First we present the different criteria that will be used for the analysis and explain the meaning of each of them. In a second step we apply these criteria on the mentioned SDEs to characterize their main features.

2.1 Analysis Criteria

The different criteria used to analyze the different environments are : tool integration, lifecycle coverage, active user guidance, customizability and scale. The first four criteria have been inspired by ([Vla90]pp.3-4) The last criterion, the criterion of scale, allows to distinguish an SDE that only supports programming tasks (in-the-large or in-the-in small) from an SDE that supports development tasks (in-the large or in-the small).

2.1.1 Tool Integration

In Section 1.2 we have presented a framework for an integrated SDE. The tool integration criterion allows to determine to what extent the analyzed SDE corresponds to that framework. The possible questions are: "What type of tool integration does it include ?" and "Is it possible to integrate new external tools to the environment ?"

2.1.2 Lifecycle Coverage

This criterion determines to what extent the analyzed SDE covers the different phases of the software development process. This means vertical coverage, from requirements definition to maintenance, as well as horizontal coverage, from elaboration activities to validation, documentation and project management activities.

2.1.3 Customizability

This criterion is used to find out if the environment is "frozen" to one particular method or if the SDE can be adapted to many different software engineering methodologies. Another question is to know if the SDE is organized around a kernel that can be tailored to the specific object types, relationships, agents and processes found in the user's own method.

2.1.4 Active User Guidance

This criterion is used to verify if the environment only knows about the **static** structure, i.e. objects and object relationships, of the software lifecycle model or if it also has knowledge about the **dynamic** features, i.e. the specific process to be followed for the software development. If the environment has knowledge about the software process, does it provide direct assistance, or guidance to the user in the different development tasks or not.

2.1.5 Scale

The scale criterion allows to test if the SDE supports programming tasks or if it supports development tasks. Programming environments only support the coding phase, that means tasks such as editing or compiling. They are characterized as large when the software application is a large one. Development SDE cover the whole lifecycle. Development-in-the large SDEs include configuration management and programming-in-the-many tasks, in other words project and team management.

2.2 Different SDEs

For each environment we try to find out how far the criteria are verified. The goal is not to set up a classification of the different SDEs, but to stress their main features and their main deficiencies.

2.2.1 ALMA

ALMA (Atelier Logiciel sur Machine Abstraite) [Vla86],[Vla87] is a SDE that aims at assisting in the development and maintenance of large software products.

Tool Integration :

ALMA allows to integrate autonomous external tools. This is implemented by using a common data structure, called the Project DataBase (PDB). The latter has an entity-relationship structure (conceptual model) that is built around three concepts :

- **Software objects** of different types. Examples are functions, modules, sources, test-data and programmers.

- **Relations** between these objects such as *is "refined into"*, *"is derived from"* or *"is a version of"*.

- **Properties** of objects or of relations. Examples are name, date or design decisions.

A very important characteristic of ALMA is that it allows the object properties to have **formal texts** as values. Thus any formal text can be attached to any software object or relation instance and will be manipulated with a knowledge of its syntax.

ALMA provides two kinds of tools. First, it provides high level tools for updating, querying, reporting and maintaining software objects and relations consistently in the

database. Secondly, the environment provides syntax directed tools, i.e. structural editors, for manipulating the formal texts attached to software objects and relations in the database. These tools are referred to as **generic** as the former tools are parameterized on the software lifecycle model, and the latter tools are parameterized on formalisms.

Lifecycle Coverage :

ALMA may cover the whole software development lifecycle. This is possible as the various software objects, relations and associated formal texts manipulated in the ALMA kernel may refer to all steps of a software lifecycle.(cf. the examples given above)

Customizability :

ALMA does also satisfy this criterion by providing a two level architecture. On a first level, ALMA presents a **meta-environment** which is parameterized as well on the software lifecycle models as on the languages in which formal texts associated with software objects or relations are written. The meta-environment's information (objects and their relationships) are stored in the Software Knowledge Base (SKB). On a second level, the meta-environment may be instantiated to an **instantiated environment** which is tailored to specific users and relies on a specific PDB schema.

Active User Guidance :

ALMA has, until now, not incorporated dynamic or behavioral aspects about a lifecycle model. However, in [Vla90] some extensions (not yet implemented) to the ALMA environment are proposed for the integration of lifecycle dynamics.

Scale :

As we have seen, ALMA may cover the whole software lifcycle and provides generic tools for carrying out the various tasks of the software development. Thus this SDE can be used for the development of large and complex software applications. However, the team work, as well as the planning task are not considered.

2.2.2 ISTAR

ISTAR is a software development environment that has been designed to support the **contractual approach**. The essence of the contractual approach is that every task in a software project is viewed as having the nature of a contract, i.e. a well defined package of work that can be performed independently by a **contractor** for a **client**. The key point here is that the contractor is free to fulfill the contract specification. In general, this freedom includes the ability to let subcontractors perform all or part of the work, and so on recursively. This will lead to the construction of a **contract hierarchy** (tree structure) that will correspond to the organization of the project.

Tool Integration :

ISTAR includes a comprehensive, extensible tool set covering every aspect of system development from requirements capture up to maintenance. Tools are grouped into **workbenches** that are coordinated sets of tools. ISTAR's tool development and integration facilities allow to create new tools and workbenches and to extend the workbenches with

new tools. Normally, the tools that are integrated into an ISTAR workbench are the tools of the underlying operating system.

Moreover, ISTAR does not have one large environment database but employs a large number of small databases, one for each contract, called the **contract database**. The workbenches used in a contract only operate on information of the corresponding contract database. Different contract databases can, however, exchange information.

Lifecycle Coverage :

The ISTAR environment may support every aspect of the software lifecycle. This results from the fact that ISTAR supports the **contractual approach**. Assume that the whole project is viewed as a contract between a client and a contractor. Then the latter can make subcontracts for each of the tasks that has to be carried out to develop the project. For example, every subcontract could represent one of the phases of the development process, namely the requirement, design, implementation or test phase and thereby cover the whole lifecycle. And for every contract, ISTAR provides the needed workbenches to carry out the contract specifications.

Customizability :

For different projects, the project organization is also different, and thus the **contract hierarchy** will vary. In other words, the contractual approach supported by ISTAR does not prescribe a particular software process, but is able to provide effective support for a variety of processes. Thus the ISTAR environment can be tailored to the needs of the concerned organization.

Active User Guidance :

The ISTAR environment has no knowledge about the **behavioral** features of the information stored in the local contract databases, but only knows their **static** structure.

Scale :

ISTAR is a fully integrated project support environment and seeks to provide an environment for managing the cooperation of large groups of people producing large and complex software systems. This means that ISTAR not only allows to support software development tasks, but also project and configuration management tasks.

2.2.3 Adèle 2

Tool Integration :

Adèle 2 verifies this criterion by having implemented a central repository called the Adèle DataBase (**Adèle-DB**). The latter is multi-version and based on an extended entity-relationship model. However, external tools (compiler, linker) do not work directly on the Adèle-DB objects, they only operate on files. Thus tools are integrated in Working Environments (**WE**) that fill the gap between the two needs (file view vs object view). A WE is a sub-database (**sub DB**) with a set of objects (directories and files), a set of tools and a set of methods and policies. The coordination between the WEs and the Adèle-DB is implemented by the activity manager (**AM**).

Lifecycle Coverage :

The Adèle SDE concentrates on the programming and maintenance of large software products.

Customizability :

The Adèle kernel is suitable for various projects and an adaptation can be implemented for each of them. The Adèle-DB is used to manage the software product whereas its development is supported by the WEs. Different projects will vary by the WE that will be used and thus it is the WE that allows to tailor the kernel according to the user's needs (files, tools, policies).

Active User Guidance :

The Adèle SDE allows to define the static and dynamic aspects of the software project under development. The static aspects are modeled by the extended Entity-Relationship model. The dynamic aspects (the process modeling) of the project's objects are defined using an Event-Condition-Action (**ECA**) formalism supported by a **trigger mechanism**.

Scale :

Adèle is a SDE that is used for the programming and the maintenance of large software products in a multi-version and a multi-user context.

2.2.4 Marvel 3.0

Marvel 3.0 is a Rule-Based Software Development environment kernel that combines object oriented techniques with techniques from rule based production systems.

Tool Integration :

Tool integration is supported by the implementation of a shared **objectbase**, that stores all the different components of the project under development, and by the use of Marvel **envelopes**. The latter implement an interface mechanism between Marvel and the external tools which allows to support these tools without modifying them or the Marvel kernel. This interface mechanism is necessary as the tools, also called Commercial Off-The-Shelf tools (**COTS**), do not operate on the objectbase itself but on the file system outside this objectbase. Thus, the main task of the Marvel envelopes is to mediate between the Objectbase Management System (**OMS**) and the external tools.

Lifecycle Coverage :

Marvel does not cover the whole software lifecycle, but is mainly restricted to the coding and maintenance phases. This is due to the fact that Marvel only interfaces with non interactive tools, the **COTS** tools, which are installed on the underlying operating system. Examples are the compiler, the linker and the editor tools. The main reason is that the envelopes which are used in the interface, are not general enough to integrate also interactive tools. Thus requirement tools or design tools which are highly interactive are not supported. This presents a severe restriction to the Marvel environment.

Customizability :

The customizability criterion is satisfied by the Marvel environment. Effectively, Marvel's kernel can be tailored to a particular project. This is supported in two phases.

First, a project **administrator,** specifies the project's data in terms of object oriented **classes**. S/he also specifies the model of the development process in terms of **rules** and writes the **envelopes** that interface to the external COTS tools. These descriptions are then loaded in the Marvel kernel, tailoring it as a **Marvel environment.** Secondly, an **end-user** uses this tailored environment for her/his own developing purposes.

Active User Guidance :

As we have already seen in the previous paragraph, Marvel combines the features of the object oriented paradigm with techniques form rule-based production systems (e.g. OPS5, Prolog). The former allow to specify the structural (static) aspects of the project by using object oriented class definitions. This specification is called the **data model**. The rules allows to model the behavioral (dynamic) aspects of the project and represents the **process model** which specifies the **rules** and the **envelopes** needed to carry out the software process.

The rules are applied on the **passive** objects, and specify the precondition enacting an activity and the multiple postconditions resulting from this activity. The Marvel environment allows thus to support **controlled automation** by applying **backward** and **forward chaining** among rules.

Scale :

Marvel may be used for programming-in-the-large tasks as the environment has problems to cover the whole software lifecycle. It should be used for compile/edit/link tasks that support a user in the coding phase. Moreover, Marvel 3.0 does not fully address the problem of team work and project management as it does not allow multiple concurrent accesses to the objectbase. However, this extension is envisioned for a future version.

3. CONCLUSION

In this part we have given an overview of the main characteristics of the current Software Development Environments. We presented a framework for an integrated SDE covering all the phases of the software development process. Moreover, we gave some concrete examples of current SDE by stressing their main features and deficiencies.

One of the most important features of SDEs like Marvel or Adèle are their facilities to model the lifecycle dynamics, i.e. the software process. These SDEs allow to guide the user, to a certain extend, actively through the software development process.

This aspect of active user guidance, which was the main deficiency of second generation environments, will be the key aspect of future SDEs.

PART II

IN DEPTH STUDY OF THE MARVEL SOFTWARE DEVELOPMENT ENVIRONMENT

0. INTRODUCTION

0.1 History

The Marvel project started in 1986 as a joint effort between Professor Gail Kaiser and Dr. Peter Feiler at Carnegie Mellon University. The prototype was built on top of a multi-user programming environment for C, called Smile and was completed in Fall 1986. In 1987, the project moved to Columbia University, where a first serious implementation work began and resulted in version 1.0. This version was always based on Smile. The second implementation (version 2.x) was already a completely standalone system with a persistent object manager replacing Smile. All versions were single-user environments. Marvel 3.0 was released in October 1991 and constitutes the first multi-user version of the project [Mar91a].

0.2 Goal of the Marvel Project

The long-term goal of the Marvel project is to develop a kernel for multi-user development environments that allow teams of programmers to cooperate in the development of a large-scale software project. The kernel provides concurrency control and object management primitives that will enable project administrators to build an environment that implements concurrent software process models describing a spectrum of interactions, ranging from cooperation among members of the same development team to isolation of teams who work on unrelated parts of the project.

0.3 Marvel 3.0

Marvel is a Rule-Based software Development Environment (RBDE) kernel that provides assistance in carrying out the software development process. Marvel is built on top of an object management system that abstracts the components of the project under development as objects and stores them in an objectbase. The software development process of the project is modeled in terms of rules, each of which encapsulates a development activity.

The Marvel kernel (Section 1) implements a **client/server** model (Section 1.1) that supports multiple concurrent users of the same objectbase. The end-user applies commands to access the objectbase. The latter is only accessible through the server's primitives. The server maintains a context for each client and performs the chaining resulting from each client's commands within that client's context.

In this paragraph we describe the generation of a Marvel environment. Figure 2.1 shows the different steps carried out to generate the environment. A project **administrator** writes a specification of the project data model and process model. Both specifications are written in the Marvel Strategy Language (Section 2), where a strategy represents a unit of modularity in Marvel (Section 2.2). The administrator loads these specifications into the kernel, creating a Marvel environment that supports both the data management and the process management requirements of the project.

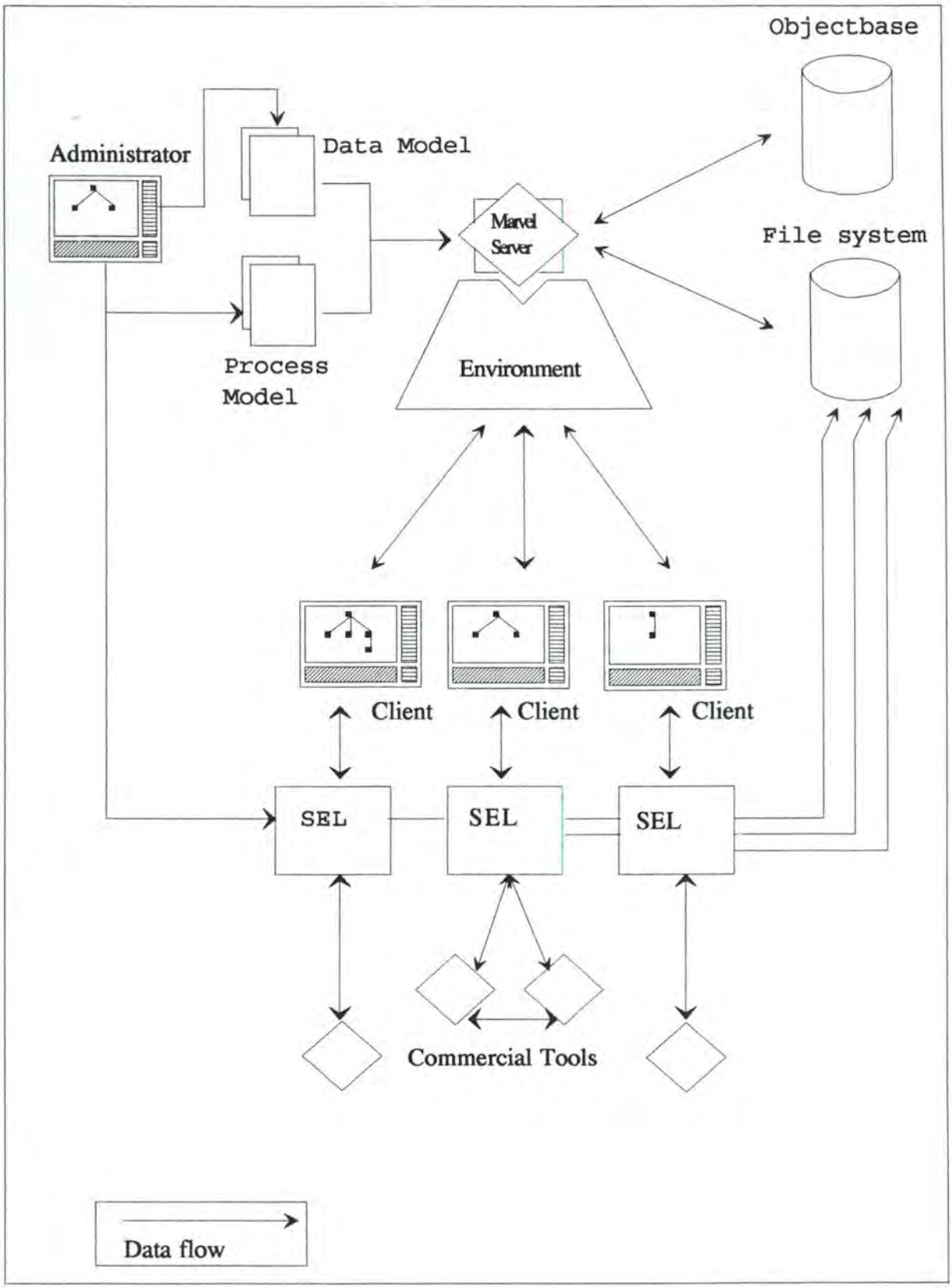

Figure 2.1: Generating a Marvel Environment [Mar91b]

The data model (Section 2.2.1) is specified in terms of classes each of which consists of a set of typed attributes that can be inherited from multiple superclasses. Attribute types include simple types, files, sets and directed links. Set attributes contain instances of other classes as their values, thus implementing **composite objects**, and giving the Marvel Object

Management System (OMS) a hierarchical traversal capability. Links are typed and point to any instances in the objectbase thus giving the Marvel OMS arbitrary graph traversal capability.

The process model (Section 2.2.2) is described in terms of rules (Section 2.2.2.1) that specify the behavior of the tailored Marvel environment in terms of what commands are available and what kind of assistance is provided. Marvel supports several models of assistance (Section 2.2.2.2), ranging from automation to consistency maintenance models. The set of rules that are loaded into a Marvel environment form a network of possible forward and backward chains (Section 2.1.2.2) . Marvel assists software developers by applying forward and/or backward chaining among the rules, automatically invoking the development activities modeled by these rules. Each rule contains a precondition, an activity and a list of mutually exclusive effects. The precondition must be satisfied to fire the rule. The activity is a general mechanism to invoke arbitrary external tools. Finally, one of the multiple effects is asserted to the Marvel objectbase according to the outcoming results of a tool.

To integrate Commercial-Off-The-Shelf (COTS) tools into the environment, Marvel uses an extended Unix shell language called SEL (Section 3). The tools need not to be modified as the envelopes (Section 3.2) build an interface between the objectbased Marvel system and the Unix file system.

An **end-user** loads a subset of the provided strategies (Section 2.2) into Marvel and creates by this way her/his personal environment. This end-user is not aware of the existence of rules, s/he only uses commands to execute her/his needs.

1. THE MARVEL KERNEL

Marvel is based on a client/server architecture. There is only one server for each objectbase (environment), although as many clients as wanted may be connected to this server [Dat90], [Tan89], [Mar92], [Mar91a], [Mar91b].

1.1 The Client/Server Architecture in Marvel

In this section we give a detailed description of the Marvel client/server architecture. First we present the components of the client and in a second step we describe the different components of the server. Figure 2.2 illustrates the different layers of the Marvel client/server architecture.

1.1.1 The Client's Components

In Marvel, a client has a one-to-one correspondence with a "process" (in terms of the Unix operating system). Every client executes a **session** running from its invocation to its exit. The session provides various client-specific information such as the controlling user's environment variables. A Marvel end-user may have multiple clients for the same server running under her/his user-id, either on the same or different machines.

The client's components are the User Interface Module, the Activity Manager and the Command Pre-Processor.

1.1.1.1 The User Interface Module (UI)

There are two User Interface modules between the end user and the environment, the Graphical User Interface (GUI) and the command Line User Interface (LUI). The latter is mainly used for batch processing using the Marvel built-in **execute** command. The important component of the GUI (an X windows system interface) is the display of the objectbase and its composite hierarchy.

The UI module (either the GUI or the LUI) receive the commands requested by the end-user and passes them to the command Pre-Processor interpreting (pre-processing) it. When the response from the server returns, the Client passes it back to the corresponding User Interface module.

1.1.1.2 The Activity Manager (AM)

The Activity Manager, called by the Inter Process Communication (IPC) module (Section 1.1.2.5), provides the interface between the envelope that executes a tool, and Marvel. It establishes all necessary communication between the tool, running as a child process, and the client by passing the received information back to the UI.

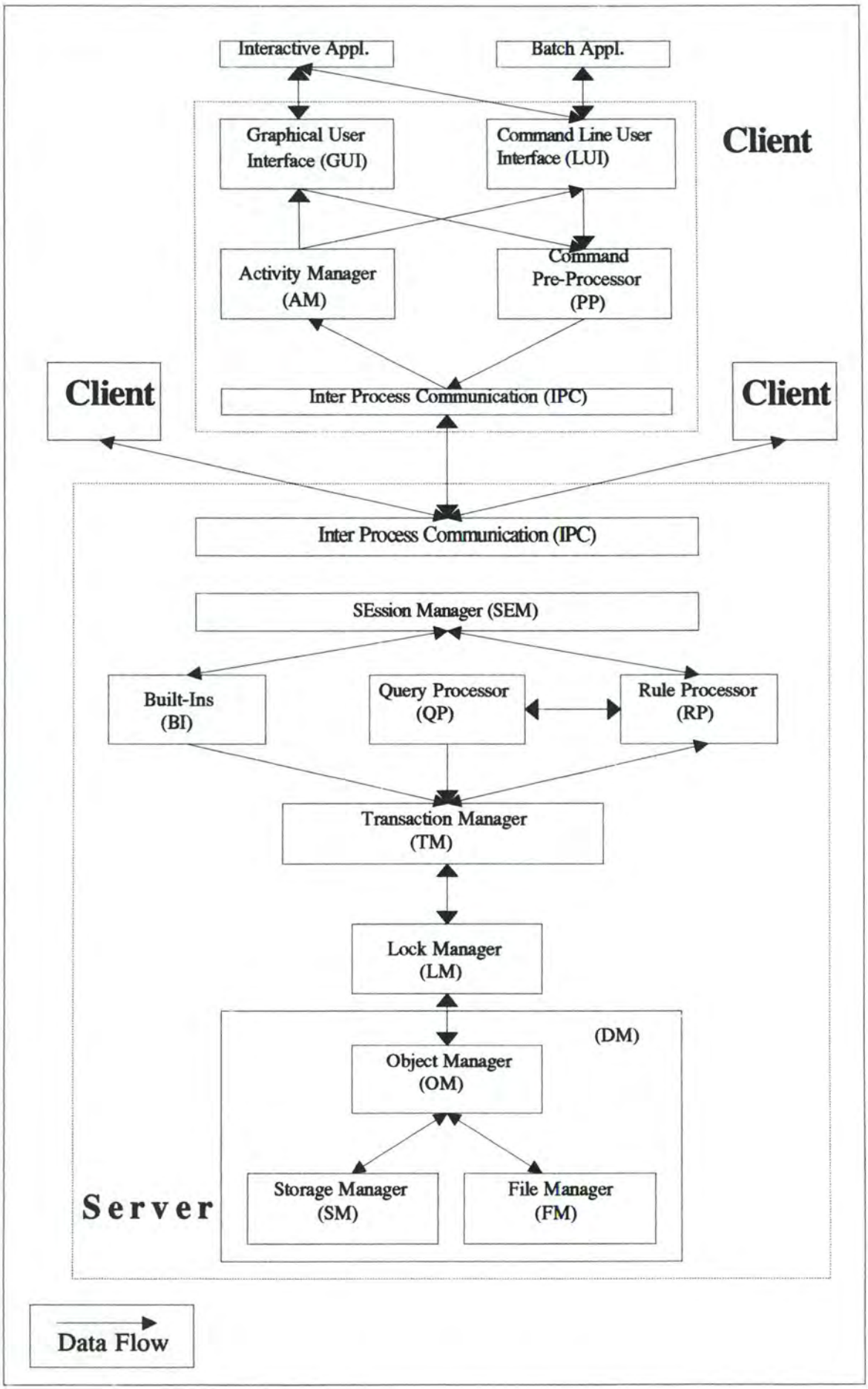

Figure 2.2: The Marvel Client /Server Architecture

1.1.1.3 The Command Pre-Processor (PP)

The command Pre-Processor module does pre-processing of built-in commands and other requests to the user before sending them to the server. The AM and the PP constitute the Command Interpreter (CI). The client module receives from the user interface a request and then pre-processes it. The CI determines whether the command can be executed locally, partially pre-processed or passed as it is to the server for execution. When the response from the server returns, the client passes the information back to the UI. If the response needs to trigger an activity, the AM is invoked to handle the execution of an activity.

1.1.1.4 Inter Process Communication (IPC)

The Inter Process Communication is implemented both in the server and the client. This layer provides the communication between the client application and the server. It is responsible for receiving and sending messages. Like the Object Manager (Section 1.1.2.1.A), the IPC must preserve the object abstraction when transferring objects between the client and the server via a sequential medium.

1.1.2 The Server's Components

The server is the central component of Marvel and all interactions of an end-user with Marvel through the client involve issuing requests to the server. Its main roles are:

1. Persistent objectbase management.
2. Rule and query processing.
3. Built-in command processing.
4. Transaction management and concurrency control.

The server's components are the Data Manager, the Transaction Manager, the Lock Manager and the Command execution Layer.

1.1.2.1 The Data Manager (DM)

The data management part of Marvel consists of an Object Manager (OM), a Storage Manager (SM) and a File Manager (FM). The DM module exchange information with the Transaction Manager (TM) module (Section 1.1.2.2)

A. The Object Manager (OM)

The in-memory Object Manager implements the object oriented database, called the Marvel objectbase. The OM handles all requests for creating, deleting and modifying objects. This module has however the possibility to call the Storage Manager (see below) for storing or fetching an object. Thanks to the OM module all the upper layer modules have only to deal with the object abstraction.

The objectbase is the repository or the dictionary of the Marvel system. It keeps track of all the software components and their properties, of all the useful project information. In the Marvel objectbase, each important project concept is represented as an object in the sense of object oriented languages [Kho90]. We have already seen that it is the server's object manager who implements and manages this object oriented database. An important characteristic of Marvel's objectbase is that it is active in the sense that updating data may trigger certain actions.

B. The Storage Manager (SM)

The Storage Manager module is responsible for managing objects on disk. It provides persistent storage and controls the flow of data from main to secondary storage. This module has no knowledge of the data model and therefore can be easily replaced. As said above, the SM provides services for, and communicates with, the OM. It is built on top of a General DataBase Management (**GDBM**) package, which is a package for efficient storage and retrieval of byte streams on disk. As this GDBM reorganizes the layout of disk storage periodically, objects are not stored continuously on disk. Object management in Marvel is divided into an in-memory manager and a Storage Manager. The objects seen as a whole form the project's objectbase.

C. The File Manager (FM)

The File Manager provides the interface between the OM and the **hidden** file system by providing system calls to access files on the file system. The underlying file system is the Unix file system which provides all of the data storage facilities for project information, i.e. objects. The file system is called hidden, in the sense that given the existence of the OM, the end-user may simply refer to environment objects such as modules, functions and design documents using some local name and s/he does not have to preoccupy with the problem of mapping that name to a database identifier. The mapping onto the Unix file system is explained in Section 2.2.1.1.E.

1.1.2.2 The Transaction Manager (TM)

The execution of a command in Marvel, either a built-in command or a rule, is modeled in terms of database operations. From the database point of view, the execution of a user command in Marvel consists of a set of database access units, each of which consists of a set of read and write operations. Database operations that must be executed together (atomically) as a unit are grouped into access units. The execution of an access unit is called a transaction. Transactions are created, managed and terminated by the Transaction Manager (TM) in order to encapsulate the execution of all the database operations involved in the command. This module also controls the concurrent access to the data and maintains its consistency by communicating with the Lock Manager (see below). The TM layer is called from the Rule Processor and the Query Processor modules (see below).

1.1.2.3 The Lock Manager (LM)

The Lock Manager module is a conflict-detection module and serves as a mediator between the TM and OM. The LM provides the handles for the Transaction Manager to enforce its policy. It provides also a concurrent access to the objects by using a lock

mechanism. When a transaction is working on an object, the LM locks it by denying the access to any other concurrent transaction. The lock is an access privilege to a single object that the lock manager can grant or withhold from a transaction. In Marvel, the LM reads a file, called the **compatibility matrix**, that specifies the lock modes and their compatibility.

1.1.2.4 The Command Execution Layer

The command execution layer consists of three parts: the Built-In commands module, the Query Processor module and the Rule Processor module.

A. The Built-Ins (BI)

The Built-In commands are a set of functions implemented in the kernel that provide a variety of services.

Built-in commands for the server can be divided in three categories:

1. Object manipulation commands (add, delete, move, copy...)
2. General services (change, print, screen refresh...)
3. Administrator commands (load, reset,...)

The administrator commands, however, are only available if the user is a recognized administrator.

B. The Query Processor (QP)

The Query Processor module translates the queries expressed in a query language into calls to the TM. The QP module is accessible only through the Rule Processor module (see below).

C. The Rule Processor (RP)

This module manages all requests (through the Session Manager module) from users to invoke rules and implements the chaining engine to enact a specific process. It assumes that the set of rules was loaded into Marvel using the **load** command. This invokes a separate program, called the **loader**, which translates the Marvel Strategy Language specifications into an intermediate representation and it is this intermediate representation that is loaded to the server.

D. The SEssion Manager (SEM)

The client executes a **session** running from its invocation to its exit. The SEM module of the server keeps a **context** for each session. This module either calls the Rule Processor for performing rule chaining with respect to the session or communicates with the Built-In command module to execute a built-in command.

1.1.2.5 Inter Process Communication (IPC)

This module has already been explained in Section 1.1.1.4.

2. THE MARVEL STRATEGY LANGUAGE (MSL)

The Marvel Strategy Language is a rule based processing language that provides means for defining objects, rules and tools [Mar91a].

To understand and illustrate the main concepts of the MSL language, we use a simple example for C programming. The small Marvel environment which will build in the following sections, should assist a C programmer in writing a C program. It should also allow her/him to run the program at the end of the work.

Remember that a C file may have some include files which it needs to compile. In our example, we add the *module* concept. A module is a component of a larger *program*, but constitutes already an executable unit. A module may contain other modules, as well as C files and include (header) files.

The important concept used in the MSL language is the one of **strategy** and will be explained in more details in the following sections.

2.1 A Strategy in Marvel

The complete process model for a software project is captured in a collection of interacting units in a way similar to the introduction of modules in a conventional programming language. The unit of modularity of MSL is called a strategy [Kai88a], [Bar88], [Kai90]. Objects, tools and rules for the same engineering project are combined in one or more MSL strategies. A particular strategy might provide only a partial view of the project which could be appropriate for a particular category of users or a particular phase of the project. The complete description is thus captured in a collection of strategies that cooperate together. As different programmers can develop modules to be combined in a coherent system, different administrators can develop strategies to be applied to the same project. Unlike modules, however, you do not need to merge (Section 2.3.3) the entire set of strategies to run the process. You may choose a subset of the provided strategies to tailor your own environment. This is useful to respond to a particular role such as programmer or manager for example, or to a particular phase of the software life cycle such as integration testing and maintenance.

The syntax of a generic strategy is illustrated in the following pattern:

```
strategy  <name>;
imports  <list of imported strategies>;
exports  <list of exported classes, tools and rules>;

objectbase

        classes...
        tools...
end_objectbase

rules...
```

As it can be seen in this pattern, each strategy has a name and consists of three parts.

1. An interface part to export/import strategies.
2. A specification part for class and tool definition.
3. A specification part for rule definition.

2.1.1 Data Model

The Marvel Data Model, i.e. the static features, is defined by using the object oriented approach. In this section we briefly define the concept of object and class as they are used in MSL and describe the different MSL attribute types.

2.1.1.1 An object oriented approach

Marvel integrates structural aspects of data modeling with behavioral aspects of the rule-based specification process.

A. Objects

An object is the basic component in the data model. It is a sequence of bytes in memory that has a defined class (or data type) and a set of attributes. Every object has a name, a unique object identifier, and a state, denoted by the values of its attributes.

The Object Management System (OMS) component in Marvel handles the persistency of objects, i.e. it retains its state across invocations of the environment.

B. Classes

A class is a purely static description of a set of possible objects, and specifies the attributes that each object has.

C. Attribute Types

In Marvel we distinguish four types of attributes: small, medium, large and link.

1. Small Attributes

Small attributes define the state of an object and can be chosen from a set of predefined types.

They consist of integer, real, boolean, string, enumerated and three special types: **user**, **clientid** and **time**. The different predefined typed are listed in the box below.

integer:	$-10^9 \ldots 10^9$
real:	$-10^{37} \ldots 10^{37}$
string:	An empty or not empty sequence of characters
boolean:	FALSE, TRUE
time:	timestamp
user:	A string which represents a User
enumerated:	A finite domain of possible values
clientid:	An integer (login number)

The **user** type corresponds to a Unix userid. MSL offers special operators to control this type, namely **CurrentUser** which returns the user-id of the owner of the client process that is currently served by the Marvel server, and **ResetUser**, which resets the user attribute to the default value.

The **clientid** type corresponds to the unique client identifier given by the system at login time. Its use is to provide access control to objects at the client level. The clientid can be accessed only by the **CurrentClient** and **ResetClient** operators, with an analogous semantics to the user operators described above.

The **time** type corresponds to an internal representation of the time. It can be controlled only by the **CurrentTime** operator which returns the system time.

2. Medium Attributes

Medium attributes map to files in the hidden file system. In order to provide **blackbox integration** (Section 3.1) of tools, Marvel provides an interface from the object-based OMS to the file-based tools. The OMS itself maps the request for files needed by the tool into corresponding file attributes. There are two kinds of file attributes: text and binary. Text files correspond to the source code (e.g. C source file) whereas the binary files correspond to the internal machine representation of the file (e.g. compiled C program). The two kinds of file attributes are shown in the box below.

text	: A mapping to a source file on the file system
binary:	A mapping to an object file on the file system

3. Large Attributes

The large attributes represent a relationship among objects, thus creating the composite-object hierarchy (a parent-child relationship). The composition hierarchy is an important concept in Marvel since it allows to abstract the project's components via composition. When the administrator is specifying the project data model, s/he has to identify how to divide the project into sub components that can be handled independently. Two types of large attributes exist: **single** and **set_of**. The former allows a child of that type to be created, while the latter allows an arbitrary number (i.e. a population) of objects to be created as children. The large attribute are illustrated in the box below.

```
INSTANCE (CLASS)
set_of INSTANCE
```

4. Link Attributes

Link attributes allow the data model to maintain arbitrary semantic relationship between two objects in the objectbase. One has to notice that there is a major difference between large attributes and link attributes. An object connected via a large attribute to its parent is considered to be part of it. Thus when deleting the parent object all its children will be deleted too. For link objects however, each object exists independently and the deletion of one does not affect the other but only removes the semantic relationship (links) between them.

There are two types of link attributes: **single** and **set_of** with analogous semantics as for the large attribute type, as can be seen in the box below.

```
link INSTANCE
set_of link INSTANCE
```

D. Meta Classes

Marvel provides two metaclass types that are implemented in the MSL language. The first is the ENTITY class type. Every data class in Marvel is defined to be a subclass of the ENTITY superclass.

The second metaclass type is the TOOL type which allows the definition of external activities that can be used by the rules. These classes, however, are only used for class definition.

E. Mapping of the Objectbase to the File System

Every MSL entity type is mapped to a Unix directory. Entity attributes are mapped according to their category. Attributes of large types are mapped to subdirectories of their corresponding entities, medium attributes are mapped to files, and all attributes of small types are stored in one file, called **attributes**.

F. Inheritance

Marvel provides an inheritance mechanism supporting the definition of a subclass/superclass relationship among classes. A subclass denotes specialization property, i.e. a subclass has additional properties besides the properties inherited from its superclass.

A multiple-inheritance mechanism is also provided in Marvel. Thus a class can inherit attributes from a set of classes. In the case where a class inherits an attribute that is defined in multiple superclasses, the first superclass (as defined in the list of superclasses) has the highest priority during merging (Section 2.3.3).

The inheritance mechanism does not work for tool classes that are defined as subclasses of the TOOL class.

G. Initialization

When defining classes, each attribute can be defined with a default value that will be assigned to objects at instantiation time. If no default value is provided by the administrator, the system provides its own default values. The only exception are enumerated types where the default value is designated by the administrator from the enumerated type set.

2.1.1.2 Example: Data Model of a C Program

In the following example, we have specified a data model for our C programming problem. It is important to notice that this model is only one possible one among others.

The root class in our data model is the PROGRAM. The program has a set of MODULEs, its proper source and object code, as well as a status. The status is an enumerated type which gives the situations of a program. The default value (e.g. *status: (Init, Exec,Error)=Init*) is the value asserted at the creation of each instantiation of this class.

FILE is a superclass and is not instantiated. It only inherits its attributes to the subclasses, i.e. CFILE and HFILE. Note that the *contents* and *object_code* attributes of the latter are specified with a ".c" and ".o" expression respectively. This provokes that each file gets such an extension. This extensions are required by a C compiler.

```
strategy data_model

# Interface with other strategies. This is the basic data model that all other strategies import
imports none;
exports all;

# Class definitions
objectbase

PROGRAM :: superclass ENTITY;

        source_code : text;
        includes : set_of MODULE;
```

```
        status : (Init, Exec, Error) = Init;
        execute : binary;
end

FILE :: superclass ENTITY;

        name : string;
        timestamp : time;
        status : (Init, NotCompiled, Compiled, Error) = Init;
        analyze_status : (NotAnalyzed, Analyzed) = NotAnalyzed;
end

CFILE :: superclass FILE;

        source_code : text =".c";
        object_code : binary =".o";
        inc : set_of link HFILES;
end


HFILE :: superclass FILE;

        source_code : text = ".h";
        object_code : binary ".o";
end

MODULE :: superclass ENTITY;

        modules : set_of MODULE;
        source_code : set_of CFILE;
        includes : set_of HFILE;
        obj_code : binary;
        status : (Init, NotBuilt, Built)=Init;
end
end_objectbase
```

2.1.2 Process Model

The **process model** is the second key component of MSL. It consists of a selection of rules that an administrator defines to describe the behavior prescribed in a development process. Each rule is defined in a strategy. The process model consists of one or several strategies that allows different views on the data model.

The process model covers two main aspects: the rule concept and the assistance to the user [Mar91a], [Kai90], [Kai88b].

2.1.2.1 The Rule Concept

A. Expert Systems Rules of Inference

Definition

A rule of inference consists of

1. a set of sentence patterns called **conditions** and
2. another set of sentence patterns called **conclusions.**

"Whenever we have sentences that match the conditions of the rule, then it is acceptable to infer sentences matching the conclusions" [Nil87].

B. Rule and Object Oriented Programming

The process model is seen as a set of rules to apply to the objects of the data model. Several approaches even allow multimethods, which means that the methods can be applied to the instances of the class as well as to instances of its subclasses. Thus, a rule is identified by a unique combination of its name and the types, order and number of its parameters. Extending the OO paradigm to the rule system means the possibility to add inheritance, dynamic binding and polymorphism to its behavior [Ben90].

- inheritance

Rules can be applied on the objects of the formal parameter class or any instantiation of its subclasses.

- dynamic binding

The dynamic (late) binding of parameters allows the chaining mechanism in Marvel (see Section 2.1.2.2.G).

- polymorphism

In Object Oriented Programming, polymorphism refers to the ability of an entity to refer at run-time to instances of various classes [Mey88]. In Marvel, rules may have the same name but different formal parameters. The overloading mechanism searches for the rule with the closest match with respect to the type of the actual parameter the rule was invoked with. (see rule overloading pp.42)

C. MSL Syntax of a Rule

```
Rule [ parameters ]:
  characteristic function:
  property list
  { activity }
   effect1;
   effect2;
  ...
```

Figure 2.3: MSL Syntax of a Rule

1. Identifier

A rule is identified by its name and the list of its formal parameters.

The name is interpreted by the Loader as a new user-level command and figures in the command list.

The formal parameters to a rule are specified by a list of *VARIABLE* : *CLASS* pairs. In Marvel, a variable is represented by a *?* followed by an identifier.

For example, *Compile [?c: CFILE]* would compile the C file to which is instantiated the *?c* variable.

2. Precondition

The precondition is a logical expression consisting of multiple clauses. Whenever it is satisfied, the adherent activity is initiated. The precondition is equivalent to the condition of an expert system rule. Zero or one precondition clause is allowed. A rule may be fired without precondition. However, in this case, the rule cannot be invoked during chaining (Section 2.2.2.2). For efficiency reasons, the precondition is implemented through the combination of the **characteristic function** and the **property list.**

a) Characteristic Function

The characteristic function binds a number of objects depending on the given expressions. These objects, also called derived parameters or bound variables, are used for the evaluation of the property list and for the correct execution of the activity. The general structure of the characteristic function is represented below.

```
(QUANTIFIER CLASS VARIABLE suchthat (EXPRESSION))
```

Quantifiers

Two quantifiers, EXISTS and FORALL, allow to define the scope of the bound variable.

Even if the quantifiers figure in the characteristic function, they are mainly evaluated in the property list. The only exception is when the binding returns an empty list and the variable has been existentially quantified. In this case, the property list is not evaluated an the precondition is not satisfied.

Class Variable

The class is any class defined in the data model. The variable is a unique identifier used to name the set of objects that satisfy the given expression.

Expression

A query may be made in a **navigational** or an **associative** manner. The navigational manner uses **large** and **link** attributes to make the bindings, while the associative manner filters all the objects of the given class by evaluating a logical expression based upon **small** attribute values.

Navigational	**Associative**
(ANCESTOR [VAR VAR])	(BVAR operator BVAR)
(MEMBER [BVAR VAR])	(BVAR operator OP)
(LINKTO [BVAR VAR])	

N.B. VAR specifies the whole object while BVAR specifies a special attribute of the object.

- Navigational Binding

Three types of operators are available for navigational bindings. Each of these operators can be inverted if the bound variable is used as the second argument in the expression.

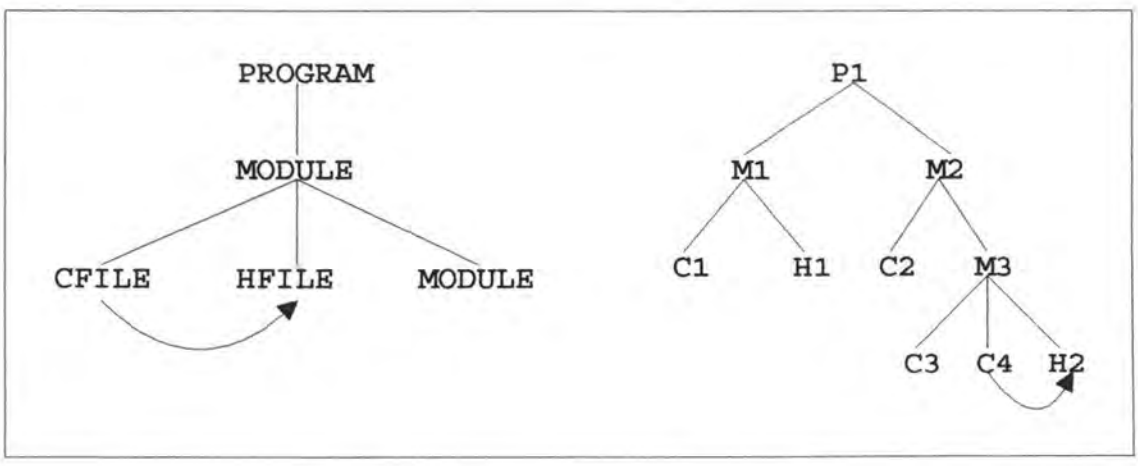

Figure 2.4: Composite Hierarchy

In the following examples, the expression between single quotes ' ' represents the actual parameter of an imaginary rule for which the results of its property list are evaluated. The example is based on the composite hierarchy shown in Figure 2.4.

1) Ancestor

The *ancestor* operator allows the rule to bind a variable to an object in the composite hierarchy that is an ancestor/descendant of the given object.

The following example will bind all **ancestors** of type MODULE from the CFILE 'C4' to the bound variable ?m.

```
(EXISTS MODULE ?m suchthat (ancestor [ ?m ?c ]))
```

The result for the given composite-hierarchy (Figure 2.4) would be:

$?m = \{M3, M2\}$

On the other hand, to find all the *descendants* of 'P1' that are of class HFILE and bind them into ?h, one may use

```
(FORALL HFILE ?h suchthat (ancestor [ ?p ?h ]))
```

where the result for the given composite-hierarchy would be:

$?h = \{H1, H2\}$

Note that the bound variable *?h* was used as second argument in the binding expression.

2) Member

The **member** operator is a more restricted version of the ancestor operator in that it binds objects based upon direct parent-child relationships.

For example, suppose you want to bind all *parents* of type MODULE from the CFILE 'C4' and bind them into ?m.

```
(EXISTS MODULE ?m suchthat (member [ ?m.source_code ?c ]))
```

As visible in Figure 2.4, 'C4' has one parent object 'M3', so $?h = \{M3\}$.

To bind to ?h a child of module 'M2' that is of type HFILE, one may use

```
(EXISTS HFILE ?h suchthat (member [ ?m.hfiles ?h ]))
```

where the result is an empty list : $?h = \{\}$

3) Linkto

Linkto is similar to **member** since it only probes one level deep: but it works on link attributes.

For example, suppose one would like to test if all the header files that a C file includes are analyzed. He/she can use the following binding expression to get the list of HFILEs into *?h.*

```
(FORALL HFILE ?h suchthat (linkto [ ?c.incs ?h]))
```

If *?c* would have been the CFILE ' C4' of the composite-hierarchy (see Figure 2.4), the result would have been *?h = {H2}*.

- Associative Binding

The associative operator allows the rule to choose objects from a specific class based upon a logical expression evaluated for each object. It ignores the composite object hierarchy and traverses all the objects of the given class and all its subclasses.

For example, you may want to bind ?c to all CFILEs already compiled:

```
(FORALL CFILE ?c suchthat (?c.status = Compiled))
```

b) Property List

The property list of a rule specifies the logical state of the objectbase that must be true to fire the rule on the given arguments. The property list evaluates the precondition on the bound variables, i.e. it compares each predicate with the set of objects collected in the bound variables. The output of a property list is either **false** or **true**.

In the following example, we show the difference between the characteristic function and the property list.

```
build_1 [ ?m: MODULE ]:
(And  (forall CFILE ?c  suchthat (member [?m.source_code  ?c])
        (forall HFILE ?h  suchthat (and (member [ ?m.includes  ?h ])
                                        (?h.status = Compiled)))
:
(?c.status = Compiled);
.....
```

```
build_2 [ ?m: MODULE ]:
(And (forall CFILE ?c  suchthat (member [ ?m.source_code  ?c])
        (forall HFILE ?h  suchthat (member [ ?m.includes  ?h ]))
:
(And (?c.status = Compiled)
        (?h.status = Compiled));
.....
```

Figure 2.5: Difference between the Characteristic Function and the Property List

In Figure 2.5, we find two rules which, at the first view, are identical as they both build an executable module if all CFILEs and all HFILEs are compiled.

Assume that there are several HFILEs where the condition is false (i.e. *?h.status =NotCompiled*), but for all CFILEs the condition is true (*?c.status= Compiled*). The property list of rule build_1 will return **true** as there are only objects into *?h* whose status is compiled. But for rule build_2, the property list returns **false** as all HFILEs are bound to *?h*, even those whose status is not compiled. The result is that the first rule is fired while the second is not.

3. Activity

An activity represents an integral software development task. In MSL, an activity is represented as follows:

{ tool_name tool_method input_arguments RETURN output_arguments }

An activity sends a message to a tool to execute one of its operations (methods). The distinction between tool name and method becomes necessary as one tool may have more than one operation. One has to note that an activity cannot invoke a Marvel built-in command.

Input arguments can be small attributes, literals, or medium attributes. Output arguments are either literals or medium attributes. Medium attributes are in fact Marvel's interface to the hidden file system, and they enable to pass files to the tool. If there are only medium attributes in the output section, the **return** keyword is optional.

Since Marvel is object-based and Unix tools are file-based, an interface mechanism is needed to communicate. This mechanism is called an **envelope** (Section 3).

The following example invokes a compiler of the source code *?c.source_code*. The resulting binary file is stored into *?c.object_code*.

```
{ COMPILER compile  ?c.source_code ?c.object_code }
```

4. Postcondition

The postcondition consists of one or a list of mutually exclusive expressions, called **effects**. Once the activity is completed, one of those effects is asserted, depending on an integer value, returned by the envelope, which is taken as an index, which means that (return_code = 0) provokes the assertion of the first effect, (return_code = 1) provokes the assertion of the second effect, and so on...

If the returned value is out of the scope of the effects list, an error message is produced in the user's screen and nothing is asserted.

Assertions can only be made on **small** and **link** attributes.

example: *(?c.status = Compiled)*

No assertions may be made on bound variables, except for **link/unlink** assertions. The main reason is that it is not yet possible to allow chaining (Section 2.2.2.2) on the bound variables.

5. Rule overloading

Due to polymorphism, several rules may be invoked on one object. To find the most appropriate one, Marvel performs a Breadth-First Search [BFS].

" For each rule, a vector of BFS numbers that correspond to the number of parameters is generated. The vector represents, for each object, the distance, in BFS order (left to right at the same level), between the type (class) of the actual object parameter and the type of the corresponding formal parameter of the rule. If there is no ancestral relationship between a formal type and an actual type, or if the number of parameters is different, then the rule is disregarded." [Ben90]

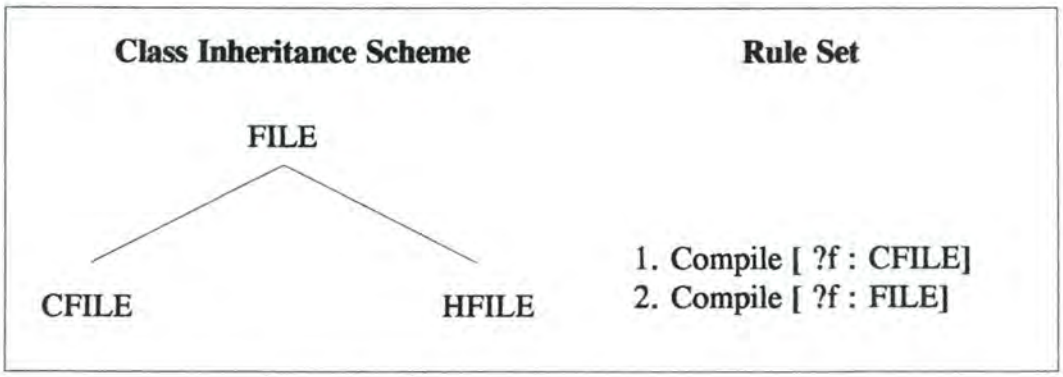

Figure 2.6: Rule Overloading Example

If the Compile rule is invoked with an object *Oc* of class CFILE, then the first rule will be selected. The BFS vectors would be [0] and [1] respectively. If, however, the compile rule is invoked with an object *Oh* of class HFILE, the second rule is selected since HFILE is a subclass of FILE and the second rule operates on objects of class FILE. The first rule is disregarded because there is no ancestral relationship between HFILE and CFILE (see Figure 2.6).

2.1.2.2 Assistance Model

"The assistance model is intended to perform two main tasks: to enact the process by means of automatic invocation of activities and to maintain the consistency of the objectbase by means of propagation " ([Mar91a] p.34). These tasks are carried out by chaining.

A. Definition of a Chain

"A chain in Marvel is a logical connection between two rules, specified by a match of a predicate in the effect(s) of one rule with the property list of another." [Mar91a]

B. Chaining Tables

When the strategies are loaded into the Marvel kernel, the latter creates two tables which contain the chaining possibilities between the rules. While running the process, the Marvel system may quickly scan these tables to provide the possible chainings in response to a user command.

ACTIVITY	PRECONDITION	POSTCONDITION
EDIT		NotCompiled NotAnalyzed
ANALYZE	NotAnalyzed Compiled	Analyzed NotAnalyzed
COMPILE	Analyzed NotCompiled	Compiled NotCompiled
BUILD	Compiled Init NotBuilt	Built NotBuilt
RUN	Built	
CLEAN	NotCompiled	NotBuilt

Figure 2.7: Activity Table

An activity table has an entry for each rule keyed by activity. Each entry of the activity table contains pointers to the precondition and postconditions in the predicate table. In Figure 2.7, you see the layout of an activity table. The values correspond to the process model of our example. This means for example that the EDIT rule has no precondition to be fired.

Predicate	Backward Pointer	Forward Pointer
NotAnalyzed		ANALYZE
Analyzed	ANALYZE	COMPILE
NotCompiled	COMPILE	COMPILE CLEAN
Compiled	COMPILE	BUILD
NotBuilt	BUILD	BUILD
Built	BUILD	

Figure 2.8: Predicate Table [Kai90]

The predicate table has an entry for each predicate or assertion that appears in a precondition or a postcondition. Each entry of the predicate table stores a list of pointers to all activities whose postconditions might satisfy it (backward chaining) and another list of pointers to all rules whose preconditions might be satisfied if this predicate becomes **true** (forward chaining) [Kai90].

Figure 2.8 shows the chaining possibilities of our process. For example, the **NotCompiled** predicate has a possible forward chain into the COMPILE rule.

C. Backward Chaining

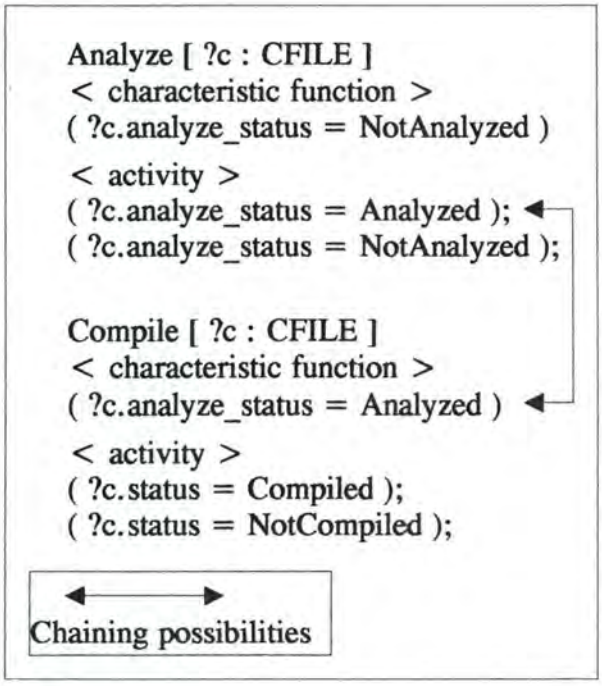

Figure 2.9: A Connection between Predicates [Mar91b]

When a user invokes a command corresponding to a rule, the Marvel kernel invokes the evaluator to check if the precondition is **true** (i.e. if the values of the predicate equal the actual state of the objectbase).

If the evaluator answers that the precondition is satisfied, then the Marvel system invokes a shell script to start the activity.

If the precondition is not satisfied, the evaluator returns a list of offending predicate/object pairs (**failed predicate**).

The Marvel system now tries to satisfy the failed predicate(s) by performing backward chaining. He follows the AND/OR tree mechanism of other backward chaining systems. However, since Marvel rules may have multiple effects, the only way to determine the real one would be to simulate execution. This has been judged unrealistic and so the execution of rules will have a permanent effect on the objectbase.

Backward chaining is recursive in that the system will try to satisfy a failed predicate in a backward chained rule by backward chaining.

Note that the execution of a rule may produce a different effect from the desired one and the precondition is not verified.

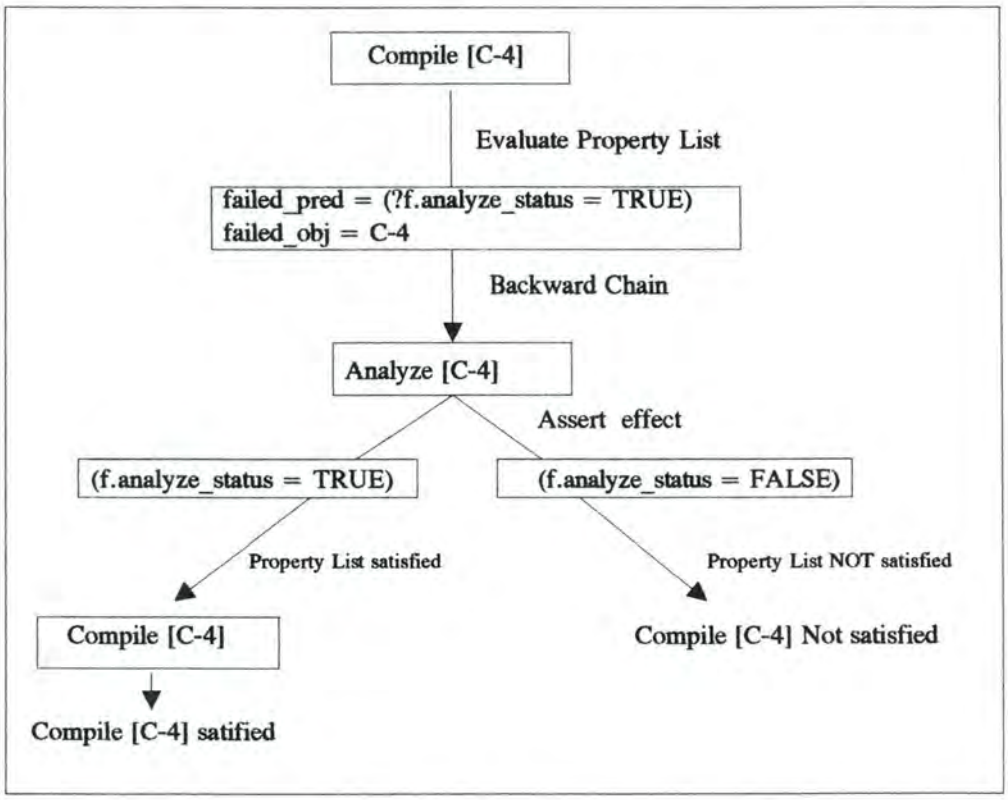

Figure 2.10: Backward Chaining to Satisfy a Property List [Mar91a]

Assume that *C4.analyze_status* = *NotAnalyzed*. If the Compile rule is invoked on *C4*, the precondition is not satisfied. However, the rule network allows a possible backward chaining to the **Analyze** rule (see Figure 10). So the system invokes the **Analyze** rule on *C4* in order to satisfy the precondition of the compile rule. If the *analyze* activity decides that it has been successfully analyzed, the first effect is asserted, i.e. *C4.analyze_status* = TRUE. As this match the precondition of the initiating rule compile, the backward chain succeeds. But as there are multiple effects in the analyze rule, the backward chain could also fail (i.e. if the second effect was asserted).

D. Forward Chaining

Forward chaining can be seen as an opportunistic approach, i.e. it performs the automatic invocation of a rule whenever the opportunity occurs. Once the assertions done, the Marvel system looks if there are rules whose precondition becomes satisfied. If so, he automatically fires that rule. However, the whole precondition has to be satisfied because there are no backward chainings during a forward chain.

The basic difference to common forward chaining systems is that the Marvel system fires all rules whose precondition are satisfied rather than only one among them. Like backward chaining, forward chaining is recursive in that a forward chained rule that was fired triggers other rules and so on.

Forward Chain	Charcteristic function bindings	Conditions	Assertions
Analyze [C4]	?h = {H2}	H2.status = Compiled C4.analyze_status = NotAnalyzed	C4.analyze_status = Analyzed
Compile [C4]	none	C4.analyze_status = Analyzed	C4.status = Compiled
Build [M3]	?c = {C3,C4} ?h = {H2}	C3.status = Compiled C4.status = Compiled H2.status = Compiled	M3.status = Built

Figure 2.11: A Forward Chaining Example

Remember the composite-hierarchy of Figure 2.4. Assume that the HFILE 'H2' and CFILE 'C3' are compiled and that (*C4.analyze_status = NotAnalyzed*). Thus the precondition for *Analyze[C4]* is satisfied. We invoke the analyze rule on 'C4'. As the condition is satisfied, the change on the objectbase is asserted (*C4.analyze_status = Analyzed*). This triggers a forward chain to *Compile[C4]*. Once again, the effect (*C4.status = Compiled*) is asserted and triggers a forward chain to *Build[M2]*. The characteristic function is evaluated and the sets for the bound variables *?c* and *?h* are created. The property list is evaluated to be true since (*C4.status = Compiled*) and (*H2.status = Compiled*). Therefore the *build* activity may be executed and an assertion may be done to complete the chain.

E. Chain Control

While loaded, Marvel creates an indirected graph representing the rule network. Sometimes however, the process should/must not allow several chaining possibilities. To control chaining, three keywords may be inserted in the property list or the effects (*no-forward, no-backward, no-chain)* [Mar91a].

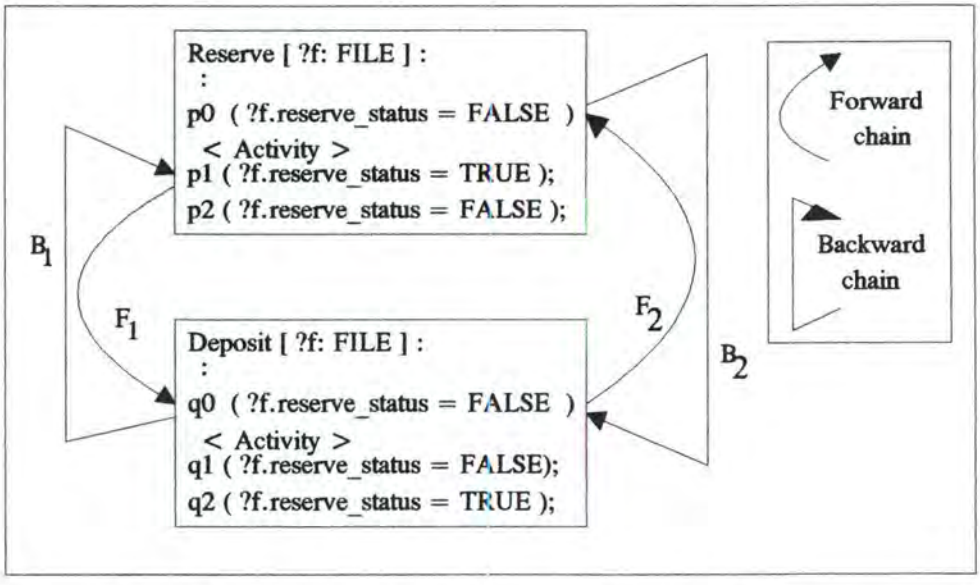

Figure 2.12: Default Rule Network Generation [Mar91a]

Without any constraints, these two rules will generate two forward and two backward chains, denoted by F_i and B_i respectively. For example, the **Reserve** rule has a forward chain F_1 (from predicate *p*1) to the **Deposit** rule (to predicate *q*0). This means that **Reserve** might forward chain to **Deposit**, meaning that it is not reserved anymore. This is clearly not desired. Furthermore, this can easily lead to an infinite loop, if the same effects are asserted over and over.

To avoid such a situation, the administrator disposes of the keywords mentioned above to add some constraints to the chaining mechanism. The following figure shows one possibility to modify the previous rules (**Reserve** and **Deposit**) in order to express what is desired and to avoid a possible infinite loop.

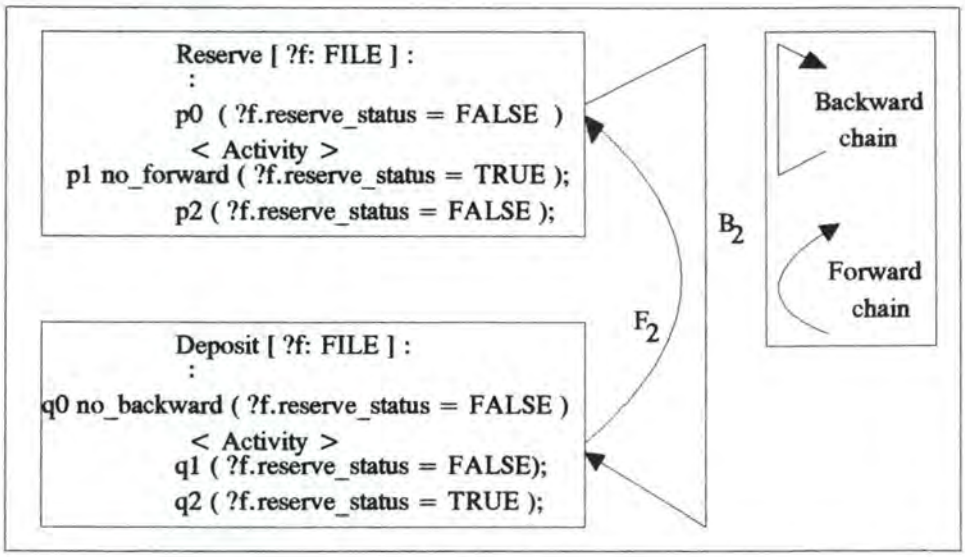

Figure 2.13: Control over Rule Network Generation [Mar91a]

Here, *p*1 is augmented by the *no_forward* directive, which prevents Marvel from forward chaining from the **Reserve** rule to the **Deposit** rule. In addition, the *q*0 predicate of the **Deposit** rule is augmented with the *no_backward* directive, which will prevent it from backward chaining to **Reserve**. B2 and F2 are left, with the semantics that a file cannot be reserved if it is already reserved without depositing it first, and one cannot deposit anything unless it was reserved first.

F. Consistency Chain

Chaining may be used for maintaining consistency of the objectbase. The consistency predicates are enclosed with square brackets. The main difference to an automation chain is that automation chains are aborted if there are problems while consistency chains have to be rolled back.

Several activities, however, cannot be undone (For instance, a mail which is sent or any other interaction with the end-user). So Marvel distinguishes between **activation** and **inference** rules, where the latter have no activity part. Inference rules may be used in consistency chains while activity rules may not.

As the objectbase is supposed to be in a coherent state before the firing of a rule, consistency chaining is only possible by forward chaining. Furthermore, consistency predicates are treated prior to automation predicates.

G. Binding of Parameters

During chaining, Marvel needs to bind actual parameters (objects) to the formal ones. To do this, Marvel uses a set of heuristics to search *near* an object to determine the objects to use during chaining. To this end, Marvel uses the composite-object hierarchy as well as the link attributes.

During a forward chain to rule r_i from a rule invoked with object O, we search for the object to bind to r_i formal parameter in the following order:

1. the object O itself
2. O's parent object
3. O's immediate children
4. the objects associated (through links) with O
5. O's proper ancestors

The heuristic cannot stop at the first candidate found, but must collect all the candidates together since it is possible to have multiple instantiations of the same rule with different objects as well as different rules with the same or different objects [Hei91.

2.1.2.3 Example: Process Model of a C Program

```
strategy process model

# The following rules are applied to the objects of the C data model that has been specified
# in Section 2.1.1.2 To make the data model accessible, the imports clause has to be
# added.

imports data_model;
exports all ;

objectbase

# The following classes are added to the data model (p.34). They are all of superclass
# TOOL and represent external tools. The (string =...) clause refers to the envelope name
# on the Unix file system.

EDITOR :: superclass TOOL;
        edit: string = edit;
end
```

```
COMPILER:: superclass TOOL;
        compile: string = compile;
        build: string = build;              # performs a link of the object codes
end

ANALYZER:: superclass TOOL;
        analyze: string = analyze;          # performs a lexical analysis
end

RUNNER:: superclass TOOL;
        run: string = run;
end

end_objectbase

rules

edit [ ?c:CFILE ]:
 :
{ EDITOR  edit ?c.contents }

(and [ ?c.status = NotCompiled ]
        (?c.analyze_status = NotAnalyzed));


analyze [ ?c:CFILE ]:

(forall HFILE ?h suchthat (linkto [ ?c.inc ?h ]))

(and (?h.status = Compiled)
        no_backward (?c.status = NotAnalyzed))

{ ANALYZE analyze ?c.contents }

(?c.analyze_status = Analyzed);
(?c.analyze_status = NotAnalyzed);


compile [ ?c:CFILE ]:
 :
(and (?c.status = NotCompiled)
        (?c.analyze_status = Analyzed))

{ COMPILER compile  ?c.source_code ?c.object_code }

(?c.status = Compiled);
(?c.status = NotCompiled);
```

```
build [ ?mo:MODULE ]:

(and (forall CFILE ?c suchthat (member [ ?mo.source ?c ]))
     (forall HFILE ?h suchthat (linkto [ ?c.inc ?c ]))):

(and (?c.status = Compiled)
     (?h.status = Compiled)
     (or (?mo.status = Init)
         (?mo.status = NotBuilt)))

{ COMPILER build ?c.object_code ?h.object_code ?mo.obj_code

no_forward (?mo.status = Built);
no_chain (?mo.status = NotBuilt);


run [ ?mo:MODULE ]:
 :
(?mo.status = Built)

{ RUNNER  run ?mo.obj_code }
;

hide clean [ ?mo:MODULE ]:

(exists CFILE ?c suchthat (member [ ?mo.source_code ?c]))

[ ?c.status = NotCompiled ]
{}
(?mo.status = NotBuilt);
```

In the example, we have defined six rules that allow the user to handle a C file. (A complete environment contains more than this, but missing rules may be specified in a similar way.) Once the user has completed the editing of a C file, Marvel performs a forward chaining to the **analyze** rule. If all include files (HFILE) are compiled, the C file is analyzed. If the C file is successfully analyzed, then there is another forward chain to the **compile** rule. To run a test program, in our example a MODULE, the user has to build it first. Note that the *no_forward* keyword (see built rule) avoids an automatic forward chain to the **run** rule. In the same manner, the *no_chain* keyword in the build rule avoids an infinite loop on this rule. There are also possible backward chains in this process (from **compile** to **analyze**). On the other hand, we do not want to force the user to edit a file (*no_backward* in the characteristic function of the **analyze** rule). The *clean* rule is an inference rule for consistency maintenance (the precondition clause is in square brackets). In fact, if the user edits a file, the status of the concerned module has to be changed to *NotBuilt*. The hide keyword avoids that the rule figures in the menu provided to the user.

2.1.3 Merging of Strategies

We have already seen that the unit of modularity in MSL is called a **strategy**. Strategies can be mixed and matched to provide behavior suitable for the various users of the environment, thus providing only a limited access to the objectbase. This mixing is called **merging** of Marvel strategies [Kai90]. As strategies combine classes, tools and rules, merging of strategies implies merging of the latter. This is only possible if the different strategies do not overlap, i.e. if they do not contain one of the following features:

. classes with the same name,
. tools with the same name,
. rules with the same name and the same activity.

When overlapping strategies are merged, the Marvel consistency checker verifies that the conflicting items are unifiable. Different possibilities of unification of strategies are:

1. Two classes can be unified if their attributes are disjoint or if attributes having the same name also have the same type.

2. Tools can be unified if envelopes with the same signature - operation name and types of formal parameters - have an identical body.

3. Rules with the same name are associated with different activities invoking different tools. The MSL loader resolves a conflict by renaming one of the rules (with interactive help from the user). Rules with the same activity but different names are taken to be different rules that happen to invoke the same activity and are thus not merged. Rules with the same name and the same activity are merged if they are **consistent.**

The checking of rule **consistency** is hard since one would like to combine only those preconditions and postconditions if the resulting combination does not lead to a logical contradiction. In practice, however, the checking is only done for obvious contradictions such as **P(a)** and **¬P(a)**.

If there are no obvious contradictions, then the precondition of the merged rule is the logical conjunction of the merged preconditions and the set of postconditions is the logical disjunction of all the postconditions of the merged rule.

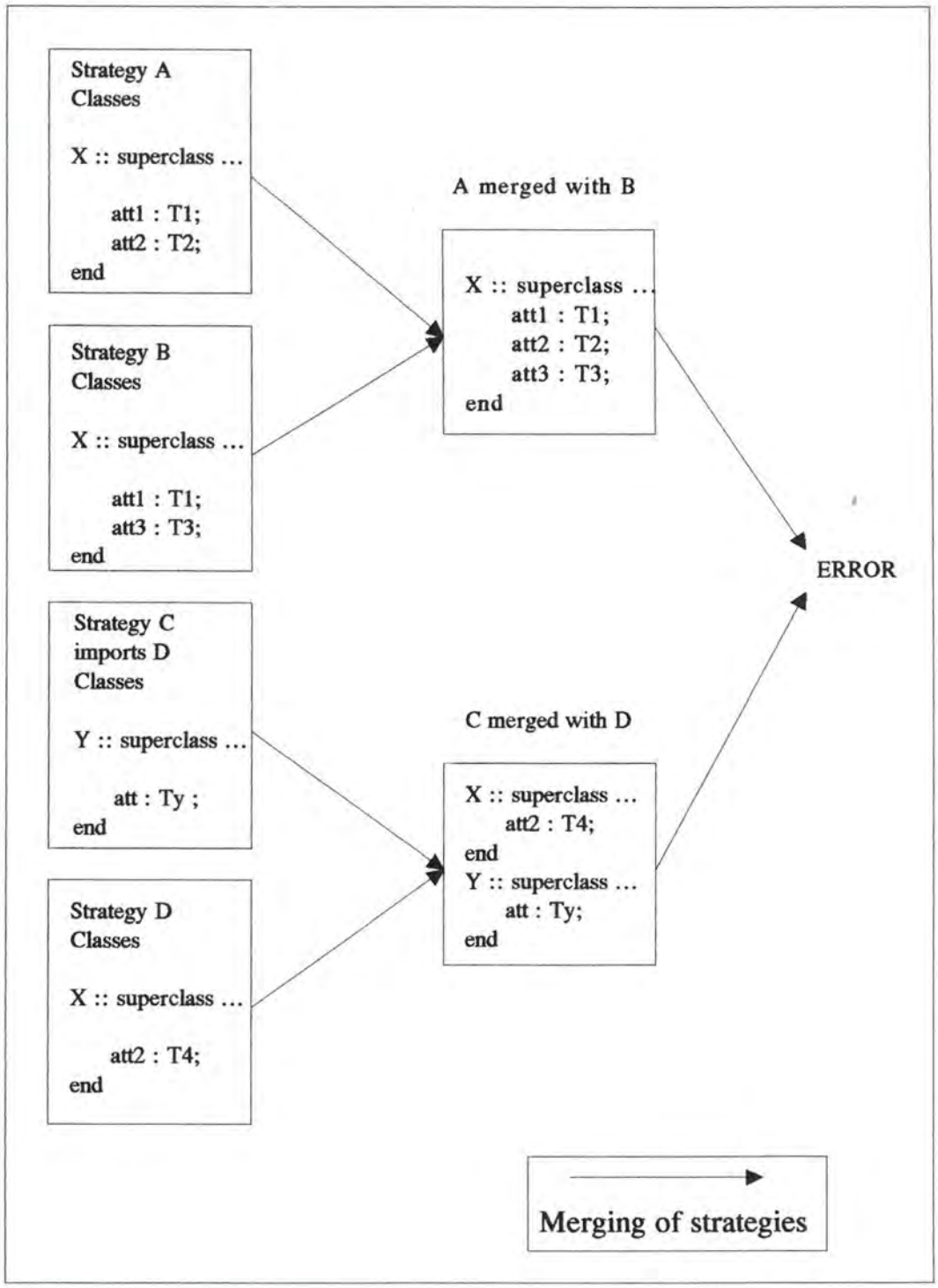

Figure 2.14: Merging of Strategies [Bar88]

Consider the four strategies A, B, C and D shown in Figure 14. The end-user loads A, B, C in that order. A and B are overlapping, since both define class X. They both define attribute *att1* but with the same type T1, so class X is unified. If only A and B were loaded, then every instance of X would have three attributes, *att1*, *att2*, *att3*. Since C imports D, Marvel automatically loads D as well and merges them first, before merging with the composite of A and B. There are no conflicts between C and D. When A-B (this means that the strategy A is merged with the strategy B) is merged with C-D, however, an error is detected because *att2* from A-B and *att2* from C-D have different types, T2 and T4.

3. SEL: A TOOL INTEGRATION LANGUAGE

The **Shell Envelope Language** (SEL) is an extended Unix shell script language to write Marvel *envelopes.* This concept has first been introduced in ISTAR [Ste86]. Since tools are expensive to develop in terms of both time and cost, an envelope serves as an interface between an existing external tool and the Marvel environment. In the next paragraphs we explain the **black box policy** that is behind the envelope concept and we give an overview of its implementation [Mar91], [Gis91].

3.1 The Black Box Policy

To support the integration of external tools or Commercial Off-The Shelf (COTS) tools without modification, the Marvel kernel views each activity as a *black box.* This is achieved by using envelopes that represent the activity's implementation and abstract the details of the interfaces of tools. The support of the black box abstraction requires a decoupling of the envelopes and the Marvel kernel. This is implemented by:

1. allowing the administrator to declare the types of incoming and outgoing data of the envelope in a clean controlled manner (explicitly). Thus static type checking [Mey88] can ensure that the activity's declared interface within a rule is consistent with that defined within its envelope;

2. without having a knowledge about the implementation of Marvel's objectbase and of the working data model;

3. returning an arbitrary number of typed arguments;

All these requirements have been implemented in the Shell Envelope Language that allows Marvel to provide a powerful black box interface.

3.2 The Shell Envelope Language (SEL)

It is in the Marvel shell envelope that the activity's interface is declared.

The SEL framework of an envelope is represented in the following pattern:

```
# [optional]
# < mandatory >

ENVELOPE [name ];

SHELL < shell that is used (ksh, sh or csh) >

INPUT

        < list of the input parameters of the form:  type :  name;  >  OR  < none ; >


OUTPUT
```

```
    < list of the output parameters of the form:  type :  name;  > OR < none; >

BEGIN

    < shell script >


END
```

The framework begins with a line containing the keyword **ENVELOPE** and an optional name for the envelope. Next comes a line specifying which shell is to be used to run the script; the choices available are **sh**, **ksh** and **csh**.

After this preamble comes the parameter specification. The input parameters come first, followed by the output parameters. Each parameter declaration contains the type and a name of the parameter. Note that the parameter type is mentioned first, contrary to the way that declarations are specified in MSL. Both the input and output sections are required, and if one is to be empty the keyword *none* must be used.

After the output section follows the shell script between **BEGIN** ... **END**. The shell script of an envelope has usually four conceptual parts:

1. The initialization procedure on the passed objects.
2. The tool execution.
3. The "cleaning up" part.
4. The return of the envelope outputs and the status code.

The status code returned to Marvel is an integer value greater than 0, and must match an effect in the *effects* section (postcondition) to be asserted in a rule.

The status code and all ouput parameters are returned by using the **RETURN** statement. The shell's own *exit* command only allows to return an integer value.

Here follows now a complete example of a *compile* envelope [Mar91a] (see Box p.55).

The input parameters are the C source file (*cfile*), the different header files (*hfiles)* and the name (i.e. the location) of the resulting object code file (*obj_file*).

As the output section is empty, the keyword *none* has been used. The initialization procedure of the shell script consists in the creation of an temporary directory ($tmp_dir) that contains soft links to all of the included HFILEs that the C source file needs. Here follows the invocation of the Unix C compiler (*cc*) with the given parameters. The third part removes the temporary file ($tmp_dir) after running the compiler tool. The fourth and final part collects and returns the envelope outputs and status code. In this case only a status code is being returned (0 or 1). The envelope uses the **RETURN** statement and the return status must be written as a string, also called a literal, i.e. the double quotes are required.

```
ENVELOPE compile;
SHELL sh;
INPUT
        text: cfile;
        set of HFILE: hfiles
        binary: obj_file;

OUTPUT none;

BEGIN

        tmp_dir=/tmp/compile$$
        mkdir $tmp_dir

        include_dir=""""
        if [ "x$hfiles" != "x" ]
        then
                ln -s $hfiles $tmp_dir
                include_dir="-I$tmp_dir"
        fi

        cc -$CCFLAG -c $include_dir $cfile -o $obj_file -ll -lc-lm -lX11
        cc_status=$?

        if [ "x$tmp_dir" != "x" ]
        then
                rm -r $tmp_dir
        fi

        if [ $cc_status -eq 0 ]
        then
                echo compile successful
                RETURN "0";
        else
                echo compile failed
                RETURN "1"
        fi
END
```

PART III

AN EXPERIMENT WITH THE MARVEL SOFTWARE DEVELOPMENT ENVIRONMENT

0. INTRODUCTION

In this third part, we consider the instantiation of Marvel for software development purposes. More specifically, we consider its use for the support of the software development process proposed by Professor Dubois [Dub91].

This experiment allows us to investigate the Marvel features in depth and to examine to what extend the environment can be adapted to our goals.

We first describe the software development method based on the transformational approach. Next, we give a detailed description of our environment. At the end of this part, we evaluate our work and see to what extent it is possible to implement the method with the Marvel kernel. We also reveal some missing features of the current Marvel version.

1. SOFTWARE DEVELOPMENT PROCESS

In this section we will present the software development process presented in [Dub91].

This software development process is based on a **formal transformational** approach [Som89]. This involves developing a formal specification of the software and transforming this specification using correctness-preserving transformations to a program (code).

Figure 3.1 (inspired from [Dub91]) shows the different transformations that are required in the transformational approach.

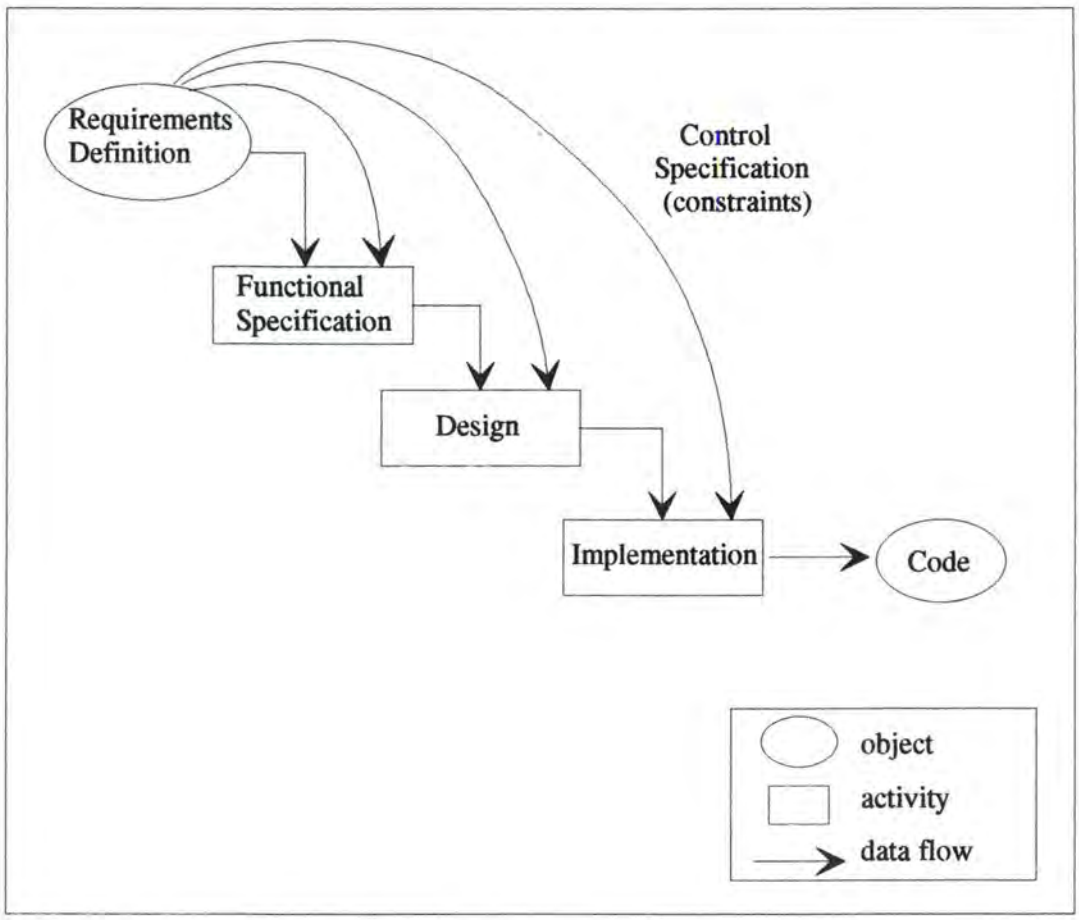

Figure 3.1: Transformational Approach

First, the functionalities sketched in the **requirements definition** are the basis of the **functional specification**. Once this is finished, the software engineer can begin with the **design** of the solution. Finally, the design is **implemented** and provides the final **code**. Every transformation has to be consistent with the non functional requirements (e.g. space, performance, security) defined in the requirements definition.

In the following sections we will present in more detail the different phases of the transformational approach.

1.1 The Requirements

The requirements phase is not part of the transformational approach, i.e. the requirements definition (services, constraints and goals) is the result of interviews with the information system user. The requirements are expressed in a natural language supplemented by diagrams (like an ERA model) and forms (decision tables) that are understandable by the users and the development staff.

1.2 The Functional Specification Phase

In the functional specification phase the needed functionalities and data are **specified** in a precise, complete and consistent manner with regards to the requirements definition. This is achieved by using formal languages that have the following advantages:

"

- a concise description of the functionalities
- a non ambiguous understanding of the specifications
- a more rigorous development of the software product
- possible detection of contradictions " [Dub91]p.II.7

Furthermore, it is important that the functional specification focus only on a WHAT description (functionalities and data) of the complete Information System (IS) and does not include any HOW description (solution that describes how to build the IS). Thus every functionality is described according to the schema of Figure 3.2.

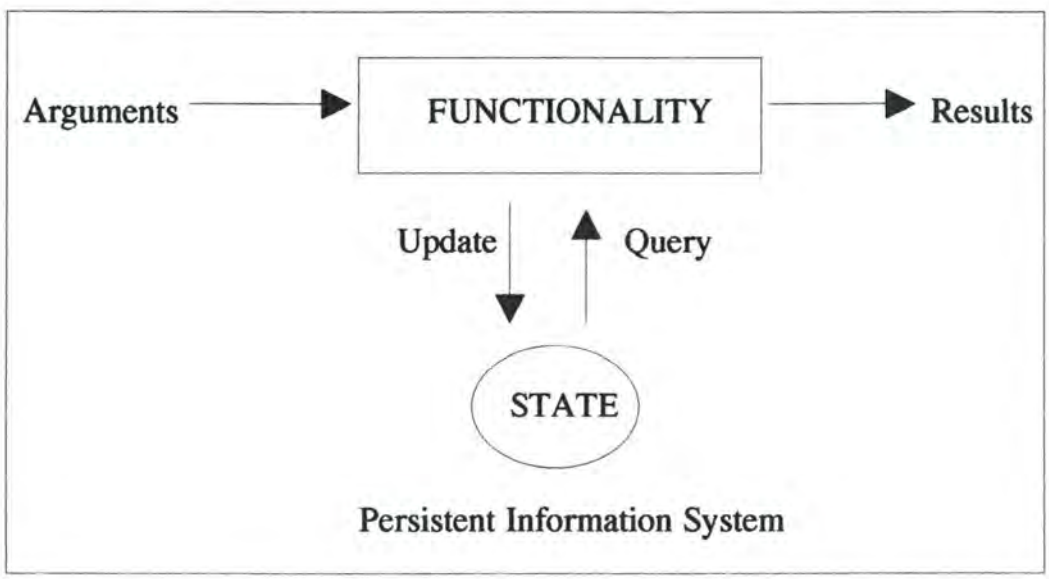

Figure 3.2: Specification of a Functionality

The **arguments** and the **results** describe the data manipulated by the functionality, and forms the **interface** part of the functionality specification. The functionality is formally specified by using a precondition clause and a postcondition clause, that form the **definition rules** of the functionality. The former is an assertion that characterizes the properties that must be satisfied by the arguments of the functionality, and the latter is an assertion that makes explicit the properties of the results by specifying the relationship between the

arguments and the results. The functionality may access (query or up-date) the persistent information system (the **STATE**). The STATE is expressed in terms of elementary functions on a ERA schema formally described. Thus two kinds of functionalities can be identified, those that have an effect on the STATE, and those that have no effect.

The formal language that is used to specify the definition rules is based on a first-order predicate logic extended with a certain number of higher level concepts.

1.3 The Design Phase

During the design phase the software engineers introduce the HOW details, i.e. the description of the solution that defines how to build the target IS. The proposed solution has to be correct with regards to the functional specification and has to be consistent with the "non-functional" requirements (e.g. performance, resources and security) informally expressed in the requirements definition. The result of the design phase is a design architecture that can be described as a set of modules that offer a certain number of services. The **design architecture** is performed in two steps, i.e. first a global design is established and then a detailed design.

1.3.1 The Global Design

In the global design a **hierarchical logical architecture** is established. This is a solution that can be executed on an **abstract machine** [Som89] and the derived **modules** are work units for the software engineer.

The set of modules is organized in different abstract levels. For every level, it is possible to describe a module by making abstraction of the details introduced in the modules of lower levels. A module offers a set of services to other modules and "uses" [Par72] services offered. The "uses" relationship defines a Directed Acyclic Graph (DAG).

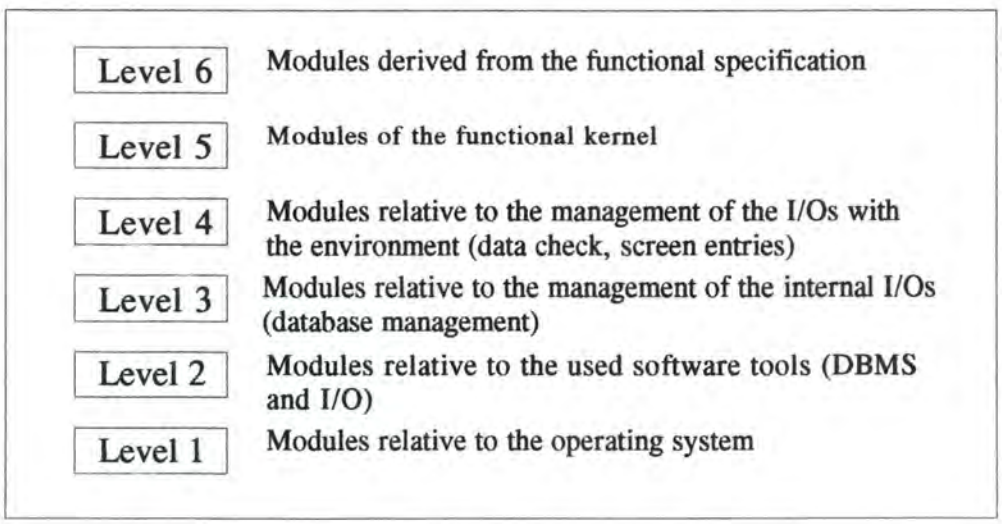

Figure 3.3: Logical Architecture of an IS

The logical architecture (Figure 3.3) consists in a graph that represents the hierarchy of the modules and their relationships. The hierarchy may be elaborated top-down or bottom-up way [Som89]. Once the graph has been constructed, the modules of level 6 to 3

are specified in term of an "Abstract Data Type" formally described using an equational logic dialect. Each module is documented to keep trace of the design decisions underlying its identification. The modules of level 1 and 2 are not specified in the global design. These modules are predefined and may be used for the construction of the hierarchy.

1.3.2. Detailed Design

In this design step, the higher level modules (levels 6 to 3) are described by their corresponding algorithms described in a pseudo-code language.

For the modules of level 5 and 6 an abstract description of the algorithm that implements the services offered is conceived. Once these modules are defined, the corresponding screen modules (level 4) are described. One module of level 3 is an abstract definition of the STATE, called the Possible Access Schema (**PAS**). The PAS is a representation that is correct and consistent with the ERA model. The other modules of level 3 represents the accesses to the STATE and are defined in an algorithmic description language. Once all access modules are defined, the Necessary Access Schema (**NAS**) is derived. The NAS is a subpart of the PAS, and represents all the "necessary" (really used) accesses to the STATE. For more details of the PAS and the NAS see [Hai86].

1.4 Implementation and Maintenance

During these phases a physical architecture is chosen. The physical architecture is a solution that is correct, efficient and executable on a real machine. The derived modules are work units for the implementation on the real machine.

Based on this architecture, an integration plan is conceived. Then, the physical architecture is implemented as a set of programs or program units that are written in a programming language. Once the module (unit) coding is finished, every unit is verified if it meets its specification. If all the units meet their specification, they are integrated according to the plans conceived before.

The maintenance refers to the change of the requirements and the tracebility of these changes in the implementation.

2. THE MARVEL ENVIRONMENT

2.1 A Methodology to build Marvel Environments

The team of Professor Kaiser proposes the following methodology [Mar91a] to define a process in MSL. We use this methodology for the definition of our environment.

1. Define the data model, with the process model in mind.

2. Define a set of rules with some chains in mind. In particular, design chains that describe automation activities that you want to chain, and eliminate some other chains using the predicate directives.

3. Define carefully consistency chains in order to make the required propagations. This implies writing new rules and possibly modifying rules already defined in step 2. Note that inference rules are usually used to propagate consistency. There is a possibility to hide such rules to the user by using the *hide* keyword.

4. Load the set of rules into Marvel using the *load* command, and examine the rule network using the chaining-graph utility.

5. Build a prototype to test your rules on the data and see if they perform properly.

6. If you have to change the rules but not the data model, simply change the rules and goto step 4. If the data model needs to be changed, remove the prototype objectbase, reload, and rebuild a new objectbase, and continue testing. Note that currently there is no utility in Marvel to evolve the schema.

7. Once you are satisfied with the behavior, build the **real** objectbase and allow clients to use the environment.

This methodology comes out of their experience with the Marvel kernel. This means that during the construction of the data model, the administrator should already think about the corresponding process model s/he needs to execute. This is particularly important in order to avoid big changes to the data model later. This request is illustrated in the following example.

Assume that the administrator would like to separate the source file from its object code, i.e. s/he wants to create two object instances of two different classes. Later on, if s/he wants to compile this file, s/he realizes that it is not possible with the current architecture, as it is not possible to have write privileges on bound variables. In fact, the bound variable found in a query has a read only access. So, the administrator has to change his objects by putting the source and the object code in the same class. This is the main reason to think already about the process while defining the classes. Furthermore, this methodology should be applied as it is not possible to evolve the objectbase.

The chaining graph is useful to detect all the undesirable, conflicting and missing chainings. This may help the administrator to adapt the rules and if necessary add or change attributes of the data model to remedy to the situation.

2.2 The Protagonists

Every large software development project is made of a team of software engineers. Generally, one engineer is responsible for the coordination of the work. In our environment, we call this person the **manager**. The remaining **software engineers** have no special status. Note that the number of team members should not be too important (8 - 10 at maximum). Otherwise, the Marvel system performance decreases rapidly.

a) The Manager

The manager's task is to coordinate the teamwork. Without his intervention, no programmer may begin a development task in the environment. The manager is also in charge to add and/or delete objects from the objectbase. We assume that s/he is the only one to perform such activities as s/he has a global overview of the whole objectbase. This will help to guarantee consistency maintenance of the objectbase. Thus, if a programmer needs to add an object to the objectbase, s/he has to send a request to the manager.

The manager has the same privileges as the programmer, i.e. s/he can carry out all the tasks that a programmer can do. However, the former cannot participate in the process development. But if s/he wants to be part of the development team, the manager must follow the same procedure as a programmer, i.e. s/he must assign himself to the corresponding modules.

b) The Software Engineers

A software engineer implements the work assigned to him by the manager. He has no access to the management commands (rules). An engineer performs two kinds of work. The most important one is the conception work, i.e. creation (specification, coding...) of the different modules. On the other hand, s/he may also review the results of the other team members.

2.3 The Data Model

In this section, we describe the construction of the data model. Figure 3.4 (see below) presents the different classes needed for the construction of the Marvel objectbase. Two types of classes can be identified: those belonging to the project under development and those belonging to the team carrying out the development process. The project itself can again be divided in three categories of classes: those needed for the class definition of the formal specification phase, those needed for the class definition of the logical architecture phase and those needed for the physical architecture phase. In our environment, we consider that the logical architecture refers to the design phase, and the physical architecture refers to the implementation and test phases. It is important to notice that all the classes that are just needed for inheritance purpose are not represented (see Section 2.3.1)

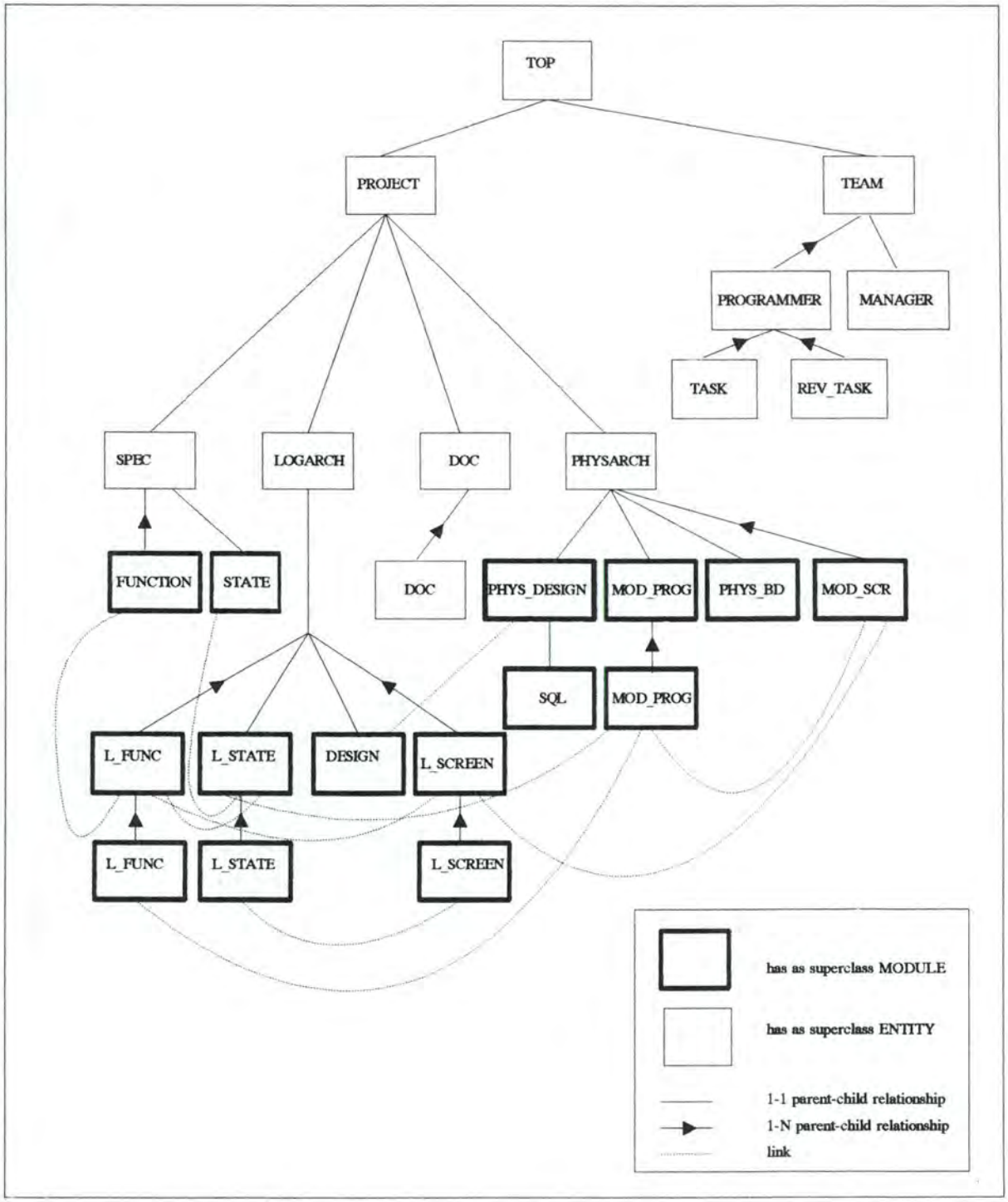

Figure 3.4: Composite Class Hierarchy of the Data Model

In the following sections, we explain the construction of the data model, i.e. the composite-class hierarchy, in more detail. The actual programming code of the different concepts is joined in boxes during the description.

2.3.1 The Superclasses

In every data model of a Marvel environment, there are some class definitions that are not directly related to the project under investigation. They are either designated for inheritance purpose or simply to organize the data model.

In our data model, we specify a top level class called TOP (see Box 1). Remember that a data model needs one and only one class that represents the root of the composite-class hierarchy (see Figure 4).

```
TOP :: superclass ENTITY;
       projects : PROJECT;
       team : TEAM;
end
```

Box 1: TOP Class

We have introduced this special class to show that the software development process is composed of two main concepts. First, there is a team of software engineers that performs the different tasks of the software development. The complexity of tasks as well as the coordination among the team make team management useful (Section 2.3.3). The second aspect of the software development is the organizational structure of the project (Section 2.3.2). The two attributes of class TOP (see Box 1) figure out this duality.

Other classes have been added for inheritance purposes. This allows to benefit from the MSL overloading mechanism, i.e. the administrator does not have to specify a rule for each subclass but only one for the superclass. It also facilitates the coding work of the administrator. One of these inheritance classes is MODULE (see Box 2). This class does not figure in the composite-class hierarchy as no object is instantiated to this class.

```
MODULE :: superclass ENTITY;
       name : string = "noname";
       engineer : string = "none";
       reviewer : string = "none";
       pro_eng : link TASK;
       rev_eng : link REV_TASK;
       feedback : text;
       status : (Initialized,Assigned,Active,Done,Reviewd,Maint,Ready)=Initialized;
end
```

Box 2: MODULE Class

The MODULE superclass regroups all the attributes that are related to the assignment of a development task to a software engineer. The *engineer* and *reviewer* attributes get the name from the engineer in charge with this object. *Pro_eng* and *rev_eng* establish temporal links to the tasks of a software engineer. (The classes TASK and REV_TASK are explained in the team management Section 2.3.3). At the first view, the *engineer* and the *pro_eng* attributes as well as the *reviewer* and the *rev_eng* attributes seem to be redundant. However, we need link attributes and string attributes as the links are removed after the completion of the task, while the string attributes are persistent. This allows later on to identify the engineers who worked on that object.

The string attributes get default values (e.g. *engineer* = "none") as there are some rules that may fail without them. In fact, the Marvel system's default of the string type is not recognized by the shell envelopes and triggers a "mutation" (switching) among the parameter's values.

The *feedback* attribute contains the comments that a software engineer may make about a module.

Finally, the *status* attribute represents in a chronological order the different phases in the life of a module. However, a MODULE object does not need to pass through every state of the given enumerated type set. For instance, the module needs not to be maintained, but it has to pass to its final state (i.e. *status* = *Ready*).

The other superclass of the data model is CONTENTS (see Box 3). This class has only one attribute, namely *contents* of type text. The class is added to the data model to simplify different rules (e.g. **print_out** rule, **touch** rule,... see description below). The *contents* attribute is not joined to the MODULE class as there are some classes (those representing programming code) with no *contents*, but rather a *source* attribute. The latter need special extensions according to the external tools and thus cannot inherit a general attribute.

```
CONTENTS :: superclass ENTITY;
        contents : text;
end
```

Box 3: CONTENTS Class

2.3.2 The Project

A project includes all specification and programming steps that are necessary to transform the requirements analysis provided from outside into an executable software piece. The intermediate steps are organized along the software development process described in Section 1.

In that process, there are three phases that need to be modeled in the data model: the formal specification of the requirements, the logical architecture and the physical architecture including coding. In fact, the test and maintenance phase have no further impact on the data model description.

The PROJECT class (see Box 4) encapsulates the whole work of the programming team.

The *specification*, *log_architecture* and *phys_architecture* attributes represent the three different phases of the software development process. The types of those attributes are declared in the correspondent sections below.

The *status* attribute gives information about the status of the project. In fact, you may have an object hierarchy without working on that project (for instance the prototype

objectbase shown in Section 2.5). A project cannot be activated unless the informal requirements were provided (and accepted), and thus the PROJECT *status* is set to *Active*.

```
PROJECT :: superclass ENTITY;
        name : string;
        status : (Active, NotActive) = NotActive ;
        specification : SPEC;
        log_architecture : LOGARCH;
        phys_architecture : PHYSARCH;
        documentation : DOC;
end
```

Box 4: PROJECT Class

The DOC class (see Box 5), representing the documentation of the project, are joined at this level of the composite-hierarchy to insist that documentation should be provided for each development phase.

```
DOC :: superclass CONTENTS;
        name : string;
        module : string;
        author : string;
        docs : set_of DOC;
end;
```

Box 5: DOC Class

The DOC class is not a subclass of the MODULE class, and thus cannot be assigned to a software engineer. In fact, every engineer may edit a document object at any time. A document has a *name*, a *module* name for which it is made and the *author* of the documentation. The *docs* attribute gives a recursive definition of this class. This allows the team manager to construct a specific hierarchy of documents. (e.g. s/he may construct a hierarchy where all children are sections of a chapter)

2.3.2.1. The Specification Phase

All the elements of the specification phase are regrouped in the SPEC class (see Box 6 below). The latter are added not only for structural purpose (readability), but it also helps to drive the software process, i.e. the logical architecture phase cannot be started until the functional specification phase is over.

```
SPEC :: superclass ENTITY;
        spec_name     : string;
        status        : (Initialized, Active, Done) = Initialized;
        functions     : set_of FUNCTION;
        db_primitives : STATE;

end
```

Box 6: SPEC Class

The *status* attribute is used to specify the state of the functional specification phase. *Initialized* means that the specification phase has not yet been started. The specification is *Active* as long as the functionalities have not all been specified. When the status is *Done*, the specification phase is over. This means also that the team may begin with the design of the logical architecture.

The *functions* attribute represents the different functionalities to specify. There are as many instantiations of the FUNCTION class (see Box 7) as there are functionalities in the requirements. For example, in a hospital management problem, admission, transfer and exit of patients would be represented by objects of class FUNCTION. The *log_repres* attribute represents the relationship to a module in the logical architecture. The definition of the L_FUNC class will be explained in the logical architecture phase (Section 2.3.1.2).

```
FUNCTION :: superclass MODULE,CONTENTS;
        log_repres : link L_FUNC;
end
```

Box 7: FUNCTION Class

The *db_primitives* attribute of class SPEC (see Box 6 above) represents the specification of the functionalities that access the persistent data system. All these specifications are stored in one object of the STATE class (see Box 8). The primitives that are declared in an object of STATE class may be used in the specifications of the functionalities (objects of class FUNCTION). As both classes (FUNCTION and STATE) are children of the class SPEC in the composite-class hierarchy, all the primitives which are used to specify the functionalities (in FUNCTION objects) have to be specified in the object of class STATE. However, the latter may not be automated. We assume that the engineers who are working on a functional specification send a request to the engineer in charge with the state specification as soon as the necessity occurs to add a new access primitive to the persistent data. But this request is not formalized in the process. The *split_into* attribute represents a relationship to the L_STATE class of the logical architecture.

```
STATE :: superclass MODULE,CONTENTS;
      split_into : set_of link L_STATE;
end
```

Box 8: STATE Class

2.3.2.2. The Logical Architecture Phase

The logical architecture phase represents the design phase (see Section 1.3) of our environment.

All the elements of this phase are regrouped in the LOGARCH class (see Box 9). With an analogous semantics as for the specification class, LOGARCH serves not only for readability but is also used to drive the software process.

```
LOGARCH :: superclass ENTITY;
      status : (Initialized, Active, Done)= Initialized;
      pas : DESIGN;
      state : set_of  L_STATE;
      functions : set_of  L_FUNC;
      interface : set_of  L_SCREEN;
end
```

Box 9: LOGARCH Class

The *status* attribute is used to specify the state of the logical architecture phase. *Initialized* is the default value asserted at the instantiation of the object. The *status* attribute may change its value to *Active* if and only if the specification phase is completed. When all the descriptions of the modules are realized, the status becomes *Done*. This means that the project team may start the physical architecture phase.

The *pas* attribute represents the PAS. As said before, the PAS is a transformation of the ERA model. The DESIGN class (see Box 10 below) represents this schema in the data model. An instantiation of this class is a module of level 3 (*level* attribute) according to the logical architecture in Section 1.3.1. The *san* attribute of this class points to the NAS schema in the physical architecture.

```
DESIGN :: superclass MODULE,CONTENTS;
      san : link PHYS_DESIGN;
      level : integer=3;
end
```

Box 10: DESIGN Class

The *state* attribute (see Box 9) represents the algorithms of the data access primitives. The latter are constructed by implementing the specifications (modules of level 3), with respect to the PAS. The L_STATE class represents those algorithm descriptions. The L_STATE class is specified recursively (*sub_state* attribute, see Box 11 below). This allows the project manager to add intermediate objects to represent some semantics among the primitives. For example, chronological or functional coherence may be used to improve the structure of the objectbase. The *coded-in* attribute links these objects to objects containing the actual programming code (e.g. SQL).

```
L_STATE :: superclass MODULE,CONTENTS;
        sub_state : set_of  L_STATE;
        coded_in : set_of link MOD_PROG;
        level : integer=3;
end;
```

Box 11: L_STATE Class

The *functions* attribute (see Box 9) represents the algorithms, i.e. modules of level 6/5, (in pseudo-code) associated with functionalities. Therefore, an L_FUNC class (see Box 12) was added to the data model. These algorithms may use data access primitives and need some interface screens as well. The *uses_state* and *uses_screen* attributes show respectively these possibilities. The L_FUNC class is specified recursively (*sub_functions* attribute, see Box 12) as a functionality may apply other functionalities. As for the L_STATE class, L_FUNC has an attribute *coded_in* that points to the programming code of the physical architecture. The *level* attribute's value (6 or 5) is specified by the software engineer.

```
L_FUNC :: superclass MODULE,CONTENTS;
        uses_state : set_of link L_STATE;
        uses_screen : set_of link L_SCREEN;
        coded_in : set_of link MOD_PROG
        sub_functions : set_of  L-FUNC;
        level : integer;
end;
```

Box 12: L_FUNC Class

The *interface* attribute (see Box 9) represents the screen definitions. While writing the algorithms for the functionalities, the programmer detects the necessity to communicate with the end-user. The screens (e.g. a menu) are specified in the L_SCREEN (see Box 13 below) class. The L_SCREEN class is specified recursively (*sub_screens* attribute see Box 13) analogous to L_FUNC. The communication among the programmers is not formalized so that there are informal requests to the programmer in charge with the layout of the screens. The *coded_in* attribute of class L_SCREEN points to the programming code

of the screen. Finally, the *uses_state* attribute points to the L_STATE modules used by the screen.

```
L_SCREEN :: superclass MODULE;
        level : integer=4;
        coded_in : link MOD_SCR;
        sub_screens : set_of L_SCREEN;
        uses_state : set_of link L_STATE;
end
```

Box 13: L_SCREEN Class

2.3.2.3. The Physical Architecture Phase

The physical architecture phase corresponds to the implementation phase (see Section 1.4).

The PHYSARCH class (see Box 14) contains all elements defined during the physical architecture phase. Once again, this class does not only serve for the readability of the object hierarchy, but also intervenes in the process.

```
PHYSARCH :: superclass ENTITY;
        phy_name : string;
        status : (Initialized, Active, Done)= Initialized;
        nas : PHYS_DESIGN;
        db : PHYS_DB;
        code : MOD_PROG;
        screen_code : set_of  MOD_SCR;
end
```

Box 14: PHYSARCH Class

The *status* attribute has an analogous meaning as for the LOGARCH class. The status becomes *Active* if and only if the logical architecture is *Done*.

The *nas* attribute represents the NAS. The PHYS_DESIGN class (see Box 15) covers this description.

The *db_language* attribute represents the NAS schema that is adapted to a concrete database language. The SQL class (see Box 16) represents this schema. The latter is the only one of the three design schemes that has to be changed if you switch to another database language. The attribute *contents* contains the SQL schema.

```
PHYS_DESIGN :: superclass MODULE;
      db_language : SQL;
end;
```

Box 15: PHYS_DESIGN Class

```
SQL :: superclass MODULE;
      contents : text = ".sql";
end;
```

Box 16: SQL Class

The *db* attribute (see Box 14 above) represents the actual database (i.e. the physical tables) of the project. This consists of a simple coding of the results stored in the SQL class object. A PHYS_DB class is created for this purpose.

```
PHYS_DB :: superclass MODULE;
      contents : text = ".tbl";
end;
```

Box 17: PHYS_DB Class

The *code* attribute (see Box 14 above) represents all the programming units (database access primitives and functionalities), and the *screen_code* represents all the screen modules. The *code* attribute is of class MOD_PROG (see Box 18 below). The *source* and *obj_code* attributes represent respectively the source and object code of the module, where the latter is of type *binary.* The *comp_res* attribute gets the errors that may occur while compiling or building an object of class MOD_PROG. The *comp_status* indicates the status of compilation.

The MOD_PROG class is recursively defined (*subroutines* attribute). An object may contain either a procedure code or a test module code. For example, to built a test program, all the children in the composite-object hierarchy must be *Compiled;* this status represents an executable unit. The *screens* attribute points to screen code modules that the programming code may include.

```
MOD_PROG :: superclass MODULE;
      source : text = ".src";
      obj_code : binary = ".obj";
      comp_res : text;
      comp_status : (Initialized, NotCompiled, Compiled, NotBuilt,  Built) =Initialized;
      subroutines : set_of MOD_PROG;
      screens : set_of link MOD_SCR;
end;
```

Box 18: MOD_PROG Class

The *screen_code* attribute of the PHYSARCH class (see Box 14) represents the interface modules of the project. The attributes of the class MOD_SCR (see Box 19) have a similar meaning as in the MOD_PROG class (see Box 18). However, these objects are not executable and thus the *Built*, *NotBuilt* values do not figure in the enumerated type of the *comp_status* attribute.

```
MOD_SCR :: superclass MODULE;
      source : text = ".dec";
      obj_code : binary;
      comp_res : text;
      comp_status : ( Initialized, NotCompiled, Compiled) = Initialized;
end;
```

Box 19: MOD_SCR Class

2.3.2.4. The Maintenance Phase

The maintenance phase of the development process needs no additional classes in the data model. This phase is entirely performed with the existing objects. In fact, maintenance is primarily a processing problem (rules) and is thus resolved in the process model (see Section 2.4.2.4.D).

2.3.3 The Team Management

Besides the project, the team management is the second key component of the data model. Every software engineer is represented by her/his own object in the objectbase. The team management helps for task coordination and for task assignation control. The TEAM class (Box 20 below) regroups the members of a programming team. This class is only added for organizational purposes of the objectbase. The *chief* attribute designates the manager of the team. The *members* attribute refers to the other software engineers working on the project.

```
TEAM :: superclass ENTITY;
        chief : MANAGER;
        members : set_of  ENGINEER;
end;
```

Box 20: TEAM Class

To represent the software engineers, the ENGINEER class (see Box 21 below) is specified. The software engineer has two attributes for identification. The *login* attribute of type *user* serves to check the identity of the current user. The *name* attribute has the same purpose, but at another level. The *name* attribute is only used in the environment whereas the *login* attribute is compared to a Unix user-id. For example, it is possible to have different engineers for one and the same user account. This structure allows to perform identity control even when the team uses only one account, i.e. the system may allow/deny access to objects according to the *name* value, even if there is only one account with several engineers.

```
ENGINEER :: superclass ENTITY;
        login : user;
        name : string;
        jobs : set_of TASK;
        rev_jobs : set_of REV_TASK;
end;
```

Box 21: ENGINEER Class

The MANAGER class (see Box 22 below) has the same attributes as the ENGINEER class. The former class is used to point out the particular role played by the manager (the manager disposes of more rules than an engineer).

```
MANAGER :: superclass ENGINEER;
end;
```

Box 22: MANAGER Class

Every software engineer (including the manager) may perform two types of tasks: write or review a module, where module refers to all the objects of class MODULE. To represent the difference between them, we add two classes to the data model. The *jobs* attribute points to the TASK class (see Box 23). This class is dedicated to the write work. The TASK has a *status* attribute that indicates whether or not a module is currently linked to this TASK. Moreover, the *beg_date* and *end_date* attributes represents the schedule of that TASK. The *comments* attribute contains specific details about the work to be done.

```
TASK :: superclass ENTITY;
        status : (Assigned, NotAssigned) = NotAssigned;
        beg_date : integer;
        end_date : integer;
        comments : text;
end;
```

Box 23: TASK Class

The REV_TASK class (see Box 24 below) is dedicated to review jobs. This class has only a *status* attribute with the same semantics as for TASK.

```
REV_TASK :: superclass ENTITY;
        status : (Assigned, NotAssigned) = NotAssigned;
end;
```

Box 24: REV_TASK Class

Beside the signification of the tasks (write and review), the two class definitions have another purpose. As we assume that the number of instantiations per class (TASK, REV_TASK) is restricted for each programmer, the two classes may avoid a deadlock of the process. Assume that there are only objects of class TASK but no objects of class REV_TASK. It may occur that all these objects of class TASK would be *Assigned*, without any review task assigned yet. As the programmers need to review the module before they may release the links between the objects, the process would be blocked. To avoid such a possibility, we introduce these two different classes. A side effect is that every programmer has to perform both kinds of task, so that the work may be distributed more equally.

2.4 The Process Model

The process model is presented in two steps. The first step describes the general context in which the process is developed. In a second step, we give a detailed description of the different rules.

2.4.1 About the Process

In Marvel there are two possibilities to drive a process. The process can either be **backward-driven** or **forward-driven**. In the former case one starts from a goal and attempts to perform a specific task. In the latter case one goes the other way round, from a specific task to a goal. For our process we use the forward-driven approach as the backward-driven approach is not adapted. By using the backward approach, the process would start with the coding phase and go back to the physical architecture phase and then to the logical architecture phase. This does not correspond, however, to the desired progression of the process development. It is important to notice that the forward-driven approach does not exclusively include forward chainings between rules, but may include some backward chainings as well.

The number of chainings we use to drive the process is limited. The main reason is that we want to control the automation of the process execution. Thus we explicitly prohibit chaining possibilities that otherwise would have been done (e.g. the **modify** and the **control** rules). This also explains the important number of chaining directives (**no_backward**, **no_forward** and **no_chain**) in the rule definitions (MSL code).

This limited number of chainings is a direct consequence of the way we implement the team management. As consecutive tasks (modify a function, control a function) on a same module are performed by different team members, we cannot allow a chaining between them.

The team management allows every team member to know exactly which tasks s/he has to carry out. Furthermore, only the assigned team member may access the module. This has as a consequence that the Marvel multi-user management, a lock/unlock mechanism, is not used. The only exception is, however, the maintenance phase. At this level, there is no team member assigned anymore, so that everyone may access any module. In this case the Marvel lock/unlock mechanism avoids concurrent access to a same module.

2.4.2 Specification of the Process

2.4.2.1 Evolution of the Process

The evolution of the process describes the sequence of tasks that have to be done to perform a software development process.

A. The Specification Phase

In Figure 3.5 we see the sequence of rule enactions corresponding to the specification phase.

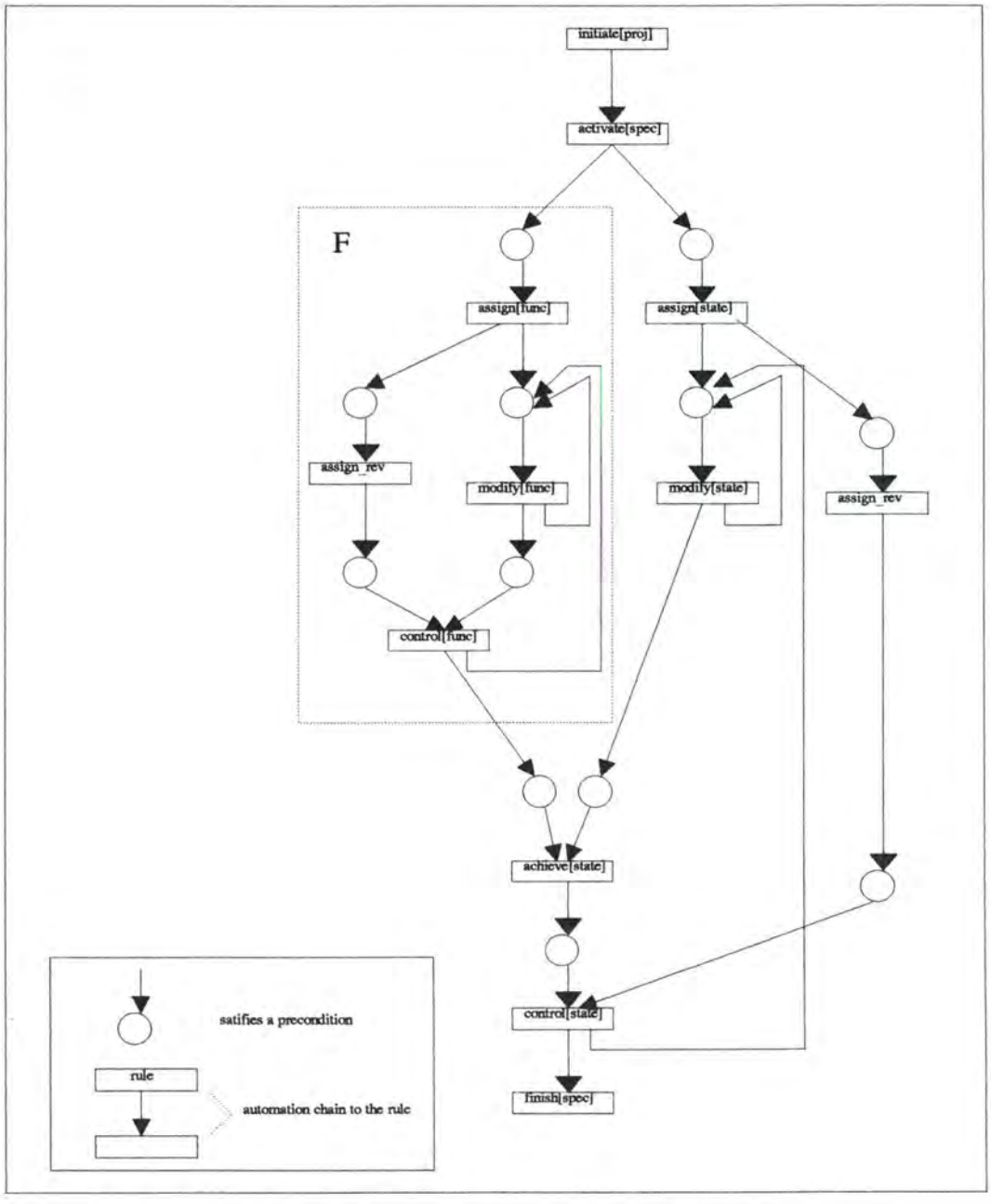

Figure 3.5: Rule Enaction in the Specification Phase

The software process always starts with the specification phase. However, prior to this, the manager has to initiate the project, which automatically activates the specification phase. From this point on, the manager may assign the different team members to their tasks. In the following, we call the software engineers accordingly to the task they perform, i.e. **programmer** (write task) or **reviewer**. Once a module (state or function) is assigned, the corresponding programmer can modify it. A module can be modified in more than one attempt. Once the modification(s) is (are) done, the reviewer controls the module and, if necessary, gives feedback to the programmer in charge. If there are more than one functional modules, the preceding operations (see box F within Figure 5) are repeated in parallel as often as there are function modules. The state module may only be achieved if all the function modules have been completed before. A successful review of the state module ends the specification phase and activates the logical architecture phase.

B. The Logical Architecture Phase

In Figure 3.6 we see the succession of rule enactions corresponding to the logical architecture phase.

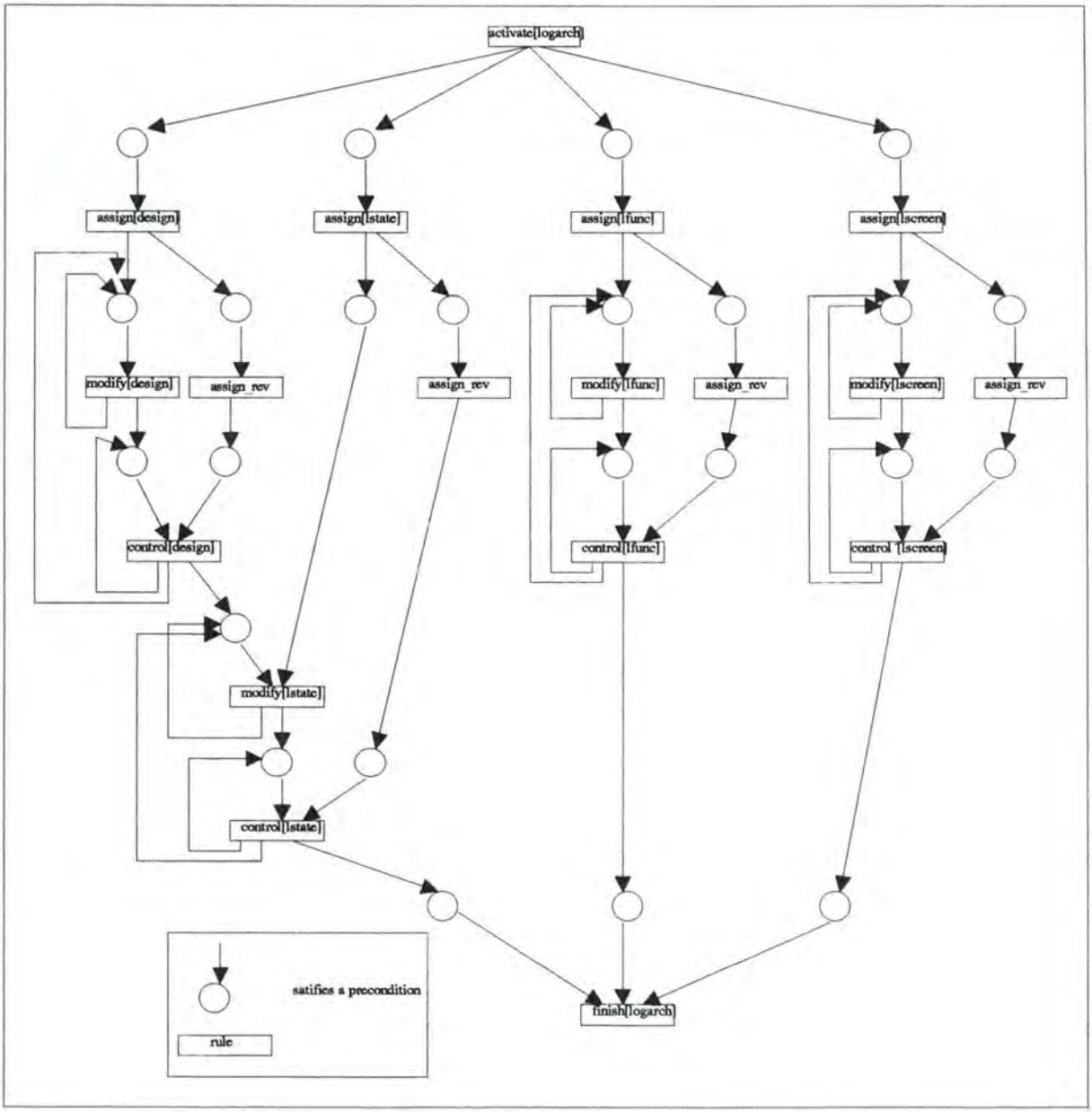

Figure 3.6: Rule Enaction in the Logical Architecture Phase

Once the logical architecture phase is activated, the project manager assigns the team members to their tasks. From there on, the team member can modify her/his corresponding module(s), namely *design*, *lfunc* and *lscreen*. These modules can then be controlled by their assigned reviewer. The *lstate* module can, however, only be modified if the corresponding design module (PAS) has already been completed. If there are more than one *lfunc*, *lstate* and/or *lscreen* modules, the preceding steps have to be repeated for each of them. Once all the modules have been successfully reviewed, and thus accepted, the logical architecture phase is done, and the physical architecture phase is automatically activated.

C. The Physical Architecture Phase

In Figure 3.7 we see the succession of rule enactions corresponding to the physical architecture phase.

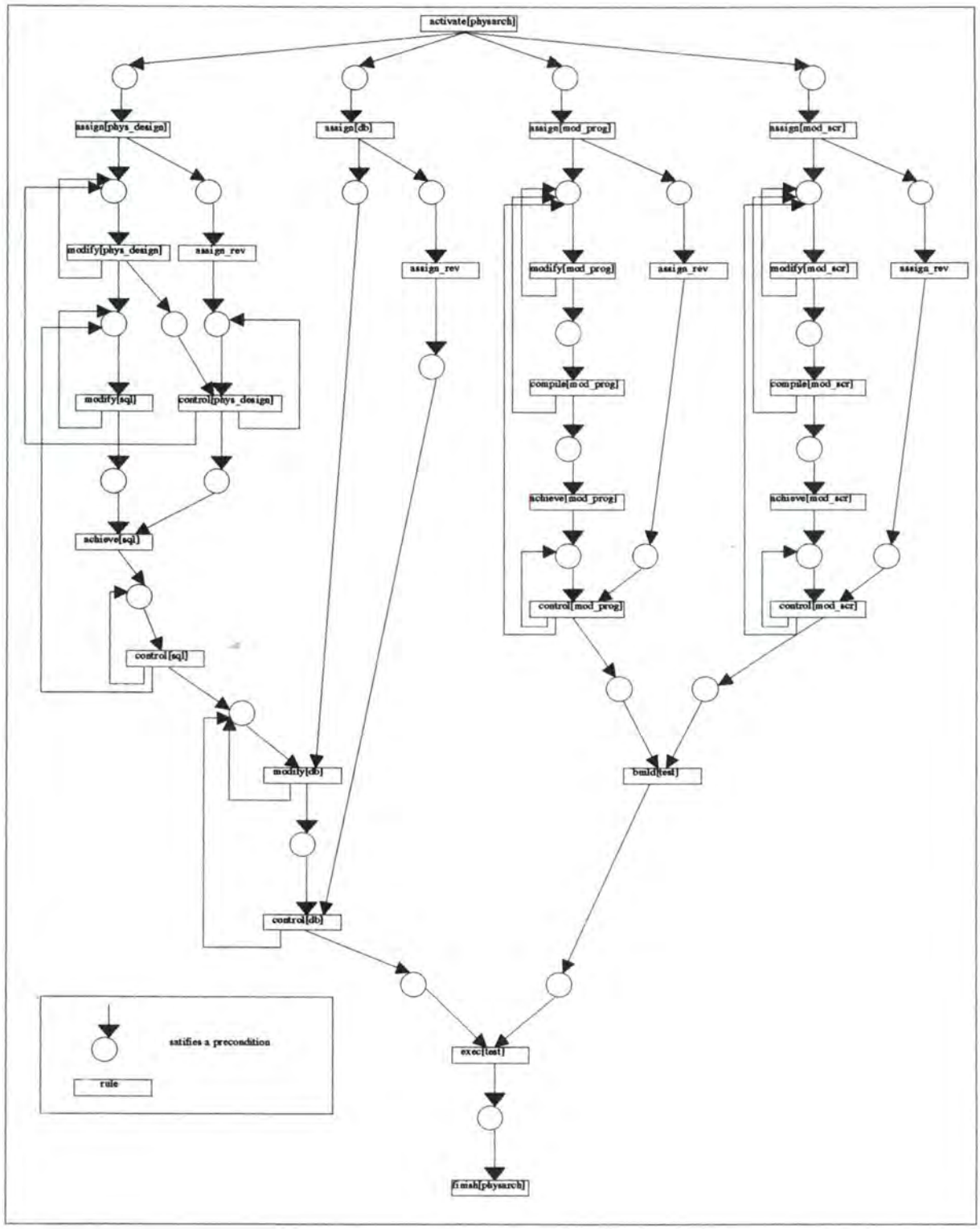

Figure 7: Rule Enaction in the Physical Architecture Phase

Once the physical architecture phase is activated, the project manager can start assigning the team members to the modules. One team member modifies (draws) the NAS

(physical design) with regards to the PAS defined before. The same team member also transforms the NAS to an SQL conform schema. However, the SQL schema may only be reviewed if the NAS has been approved before. And once the SQL schema has been successfully reviewed, the database may be physically constructed. Parallel to these steps, the coding modules are modified by their corresponding programmers. When a set of modules are successfully reviewed, the functionality they define can already be executed. Once all the functionalities are done, the physical architecture phase is finished.

D. The Maintenance Phase

The main purpose of the maintenance phase in our Marvel environment is to provide assistance during the correction and evolution of the objectbase. The initiative is taken by the software engineers. If they decide to change an object, they can use the environment to "mark" all the other objects affected by that change. It is up to the software engineers to correct these objects and thus to obtain a "coherent" environment again.

The consistency maintenance of the objectbase is assured by several inference rules and the chaining possibilities among them as shown in Figure 3.8.

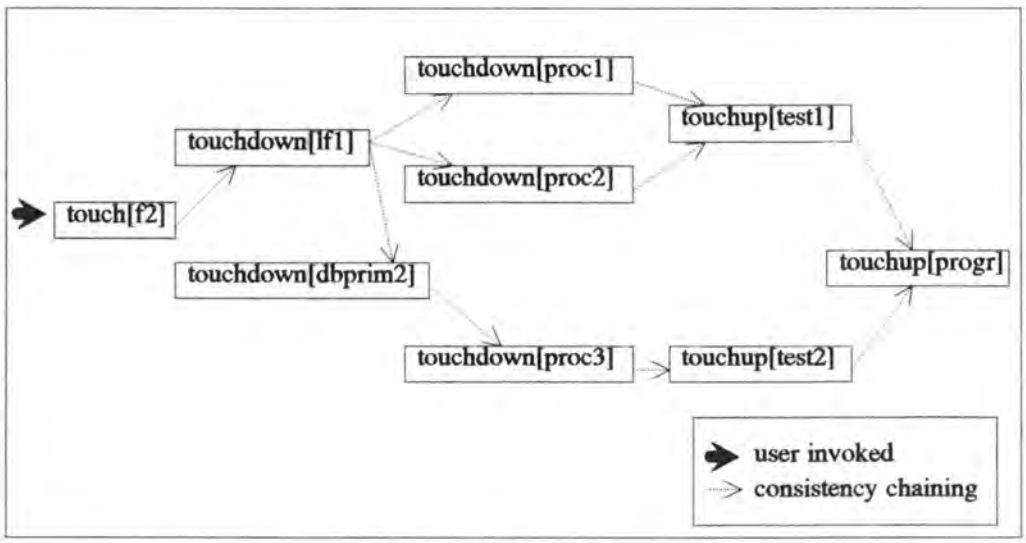

Figure 3.8: Chaining Graph of Inference Rules

If a software engineer wants to change the target object *f2*, s/he can first use the **touch** rule to change the object's status to *Maint*. This will invoke a consistency chain as shown in Figure 3.10. The **touch** rule forward chains to the **touchdown** rule, which is recursively applied to all those objects affected by the changing of the object *f2*. The affected objects also have their status set to *Maint* to indicate that they are in the maintenance phase. From this point on, the programmer may start changing the target object.

2.4.2.2 Specification of the Rules

In the following sections, we give a detailed description of the rules used in our environment. All the rules are described according to the following pattern.

The **Marvel code** part represents the syntax of the rule and is represented in a CodeBox.

The **context** part specifies the goal of the rule and explains the possible assumptions that are made. If there are comments to the different instantiations of a rule, they are specified here.

The **precondition** part explains the contents of the characteristic function and gives details about the property list.

The **activity invocation** part, explains the actions a programmer must carry out during the activity and explains the role played by the envelope.

The **effects** part gives comments about the postcondition and possible chainings that are related to the changes to the objectbase.

Sometimes, one or even more of these parts are not explicitly mentioned. This means that there are no special remarks about that part for the current rule. For example, an inference rule has no activity invocation.

There are some rules available only for managerial purposes (Section 2.4.2.2.A). Other rules are specified for programming purposes (Section 2.4.2.2.B). The third category are those rules added for process control (Section 2.4.2.2.C) and the last category is used to keep the objectbase in a consistent state (Section 2.4.2.2.D).

A. Management Rules

The first category of rules is dedicated to the manager of the programming team. These rules should help the manager to distribute the work among the team members. There are two categories of managing rules. The first category concerns assigning tasks, i.e. the establishment of *links* between the modules and the tasks of a programming engineer. The second category of rules is used to show the evolution of the development process.

The manager disposes of two assign rules. The first is to assign a development task and the other to assign a review task. The two rules are necessary as there are two types of tasks in the data model.

a) Assign Rule

context :

The **assign** rule produces a link between a programmer's task and a module. Module may be all those instantiations of classes that are subclasses of the MODULE class, i.e. all those objects that can be assigned.

Moreover, the manager has to specify a schedule for the task. The begin and end dates, which s/he has to introduce are not validated by the Marvel environment. That means that the manager may assert any value to these attributes. The begin dates may be overlapping so as to indicate that a programmer may perform more than one task between these dates.

```
assign [ ?func:FUNCTION,  ?ta:TASK ]:

(and (exists SPEC  ?spec  suchthat (member [ ?spec.functions  ?func ])
      (exists ENGINEER ?pg suchthat (member [ ?pg.jobs ?ta ]))):

(and  no_chain (?spec.status = Active)
       no_chain (?func.status = Initialized)
       no_chain  (?ta.status = Assigned))

{ INTERACTIVE  schedule ?ta.comments ?pg.name  return ?beg ?end ?name }

(and (?func.status = Assigned)
      (?ta.status = Assigned)
      (?func.engineer = ?pg.name)
      (?func.name = ?name)
      (?ta.beg_date = ?beg)
      (?ta.end_date = ?end)
      (linkto [ ?func.pro_eng  ?ta ]));
```

CodeBox 1: Assign Rule

There is no test if the person executing the **assign** rule is really the manager in the objectbase. In fact, we assume that the assign strategy containing the rules figures only in the set of strategies dedicated to the manager.

The **assign** rule has two formal parameters, one is a subclass of MODULE and one of type TASK. In CodeBox 1, we see an example where the **assign** rule is applied on an object of class FUNCTION (subclass of MODULE) that is defined in the specification phase. The first parameter of the rule designates the class that is used in the characteristic function. If the parameter is of class DESIGN, the SPEC class is replaced by the LOGARCH class in the characteristic function, according to the composite-class hierarchy shown in Figure 3.4.

precondition :

The characteristic function searches for the ancestor object which is an instantiation of one of the following classes: SPEC, LOGARCH or PHYSARCH. It also determines the programmer who owns the task object that is given as the second parameter.

If the searched objects have been found, and this will be the case in a consistent objectbase, the property list evaluates the following precondition clauses:

- if the development phase (i.e. specification, logical architecture, physical architecture) to which the module belongs to is the current one, i.e. (*ph.status* = *Active*).

- if the task to which one wants to connect the module is not yet assigned.
- if the module itself is not yet assigned (*status = Initialized*)

activity invocation :

If this precondition is satisfied, the rule executes the *schedule* envelope that starts an interactive session in the client window where the manager may introduce the different values, namely the module's name, the begin and end date of the task as well as a comment about this task. This comment may be as long as s/he wants. All these data will be sent to the addressed programmer via mail.

effects :

At the end of the assigning work, the different attributes are updated according to the values read during the interactive session are asserted to the correspondent attributes. Finally, the two objects (*?func* and *?ta*) are linked to each other. This link is only temporary and will stay only as long as the module has not successfully passed the review task. The link allows the corresponding programmer to access the module.

The **assign** rule performs a forward chain to the **send** rule (see Section C below) if the (*?mo.status=Assigned*) predicate is satisfied. In fact, other chainings are possible but have to be prohibited. For instance, assign could forward chain to the **modify** rule, but this activity is performed by another team member.

b) Assign_rev Rule

```
assign_rev [ ?func:FUNCTION,  ?ta:TASK ]:

(and  (exists ENGINEER ?pg suchthat (member [ ?pg.rev_jobs ?ta ])):

(and  (or no_chain (?func.status = Assigned)
          no_chain (?func.status =  Active)
          no_chain  (?func.status = Done))
      (?pg.name < > ?func.engineer)
      (?ta.status = NotAssigned))

{}

(and  no_chain (?ta.status = Assigned)
      no_chain (?func.reviewer = ?pg.name)
      (linkto [ ?func.rev_eng  ?ta ]));
```

CodeBox 2: Assign _rev Rule

context :

The **assign_rev** rule creates a link between a reviewer's task and a module. However, contrary to the **assign** rule, the manager must not (and may not anyway) specify a schedule. We estimate that a mail to the reviewer at the time when her/his task may begin is more useful than a predefined schedule.

The **assign_rev** rule may only be invoked after the **assign** rule (see Figures 5,6 and 7), but prior to the **control** rule (see below). By this way, the manager may decide later on which engineer will be assigned to review a module. This allows a more flexible process, i.e. the manager may distribute tasks according to the actual availability of the team members.

The **assign_rev** rule has two formal parameters, one is a subclass of MODULE and one of class REV_TASK. In CodeBox 2, we see an example where the rule is instantiated with a parameter of class FUNCTION.

precondition :

The characteristic function searches for the engineer object that owns the task that the manager has selected. The property list evaluates the following precondition clauses:

- if there is already a programmer in charge with this module.
- if the programmer and the chosen reviewer are two different persons.
- if the review task to which you want to connect the module is not yet assigned.

effects :

As for the **assign** rule, the two parameter objects are linked together. The *reviewer* attribute gets the engineer's name and the *status* of the review task is set to *Assigned* while the module's *status* remains unchanged.

The **assign_rev** rule performs no chaining. Actually, this rule does not offer any interesting chaining possibilities.

c) Overview Rule

```
overview [ ?spec:SPEC]:
(forall MODULE ?mo suchthat (ancestor [ ?spec  ?mo ])):

{ LISTER list  ?mo.m_name  ?mo.status  ?mo.engineer  ?mo.reviewer }
;
```

CodeBox 3: Overview Rule

context :

To improve the management task, the manager should have a permanent overview of the project's evolution. Particularly, the knowledge of the evolution of the different tasks may help her/him to intervene as soon as possible. The **overview** rule gives a list of all existing objects corresponding to a development phase.

The **overview** rule is only available to the manager, even if there is no explicit test to verify it. This rule figures only in the rule set of the manager.

The overview rule is applied on the different development phases, i.e. SPEC, LOGARCH and PHYSARCH. We give an example in the CodeBox 3.

precondition :

The characteristic function searches for all descendants of the selected development phase object. However, the property list is empty which allows the manager to follow the project evolution at any time.

activity invocation :

All the objects found are listed and ordered by class. The *list* envelope prints a list in the client window showing the module name, the programmer, the reviewer and the current status of the object. The default values asserted to these attributes at instantiation time guarantee that the information provided is significant. For instance, if no programmer is assigned, the correspondent value in the list is *none*.

effects :

Besides the provided list there is no change on an object of the objectbase. Thus, no chaining possibilities are provided.

B. Programming Rules

The second category of rules are dedicated to all the team members. These rules are needed to execute the software development process. For better readability, we classify them in two categories. The first category regroups all the rules that invoke an edit activity. The second category, which we call coding rules, contains all the rules (e.g. compile, build) necessary for the coding steps in the software development.

1. Edit Rules

In this section, we explain the rules that invoke an edit tool (Emacs editor). Each of the rule names reflect a special context under which the editor tool is invoked.

a) Modify Rule

```
modify [ ?mo : MODULE ]:

( and ( exists TASK ?ta suchthat ( linkto [ ?mo.pro_eng ?ta ]))
      ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ])):

( and ( ?pg.login = CurrentUser )
      ( or no_chain ( ?mo.status = Assigned )
           no_chain ( ?mo.status = Active )
           no_chain ( ?mo.status = Reviewed ))

{ EDITOR edit ?mo.contents }

( ?mo.status = Active );
( ?mo.status = Done );
```

CodeBox 4: Modify Rule

context :

The **modify** rule edits an object file attribute. This rule can only be invoked by the programmer of the object and not by the reviewer. Modify should be understood either as changing an existing file or creating a new file.

The **modify** rule may be invoked on each object of the MODULE class. However, the preconditions may be different. For example, the programmer has to wait until the NAS is completed before defining a database object of class PHYS_DESIGN, whereas the modification of a FUNCTION object can be started immediately after the assignation. In CodeBox 4, we give the general description of the **modify** rule. The changes performed for several subclasses are visible in the rule enaction graphs of the development phases (see Figures 5,6 and 7).

precondition :

The characteristic function searches for the programmer who is linked to the module via one of her/his tasks. The property list then evaluates the following precondition clauses:

- if the programmer's login found in the query is identical with the current programmer's login of the Unix system (in other words, if s/he is the assigned engineer).
-if the module status has one of the following values: the module's file attribute has not been edited before (*status = Assigned*), it was edited but not completed (status=Active), or it has been reviewed but there are changes to make (*status=reviewed*).

activity invocation :

The activity runs the *edit* envelope. This envelope loads an editor. When the programmer has left the editor, s/he is asked, in the client window, if the editing work is completed or not.

effects :

Depending on the programmer's choice at the end of an editing session, the module's status becomes *Active* or *Done*, where *Done* means that the module is ready to be reviewed.

The **modify** rule performs a forward chaining to the **send** rule if the (*?mo_status=Done*) predicate is satisfied. At the end of the modification either the assigned reviewer is informed of the completion of the task or, if no reviewer is assigned, the manager will be notified to assign one. A possible chain to the **control** rule was prohibited as the review task is performed by another engineer.

b) Control Rule

```
control [ ?mo:MODULE ]:
( and ( exists TASK ?ta1 suchthat ( linkto [ ?mo.pro_eng  ?ta1 ]))
      ( exists REV_TASK ?ta2 suchthat ( linkto [ ?mo.rev_eng  ?ta2 ]))
      ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs  ?ta2 ])));

( and  no_chain ( ?mo.status = Done )
       ( ?pg.login = CurrentUser ))

{ EDITOR review ?mo.source ?mo.feedback ?mo.engineer }

no_forward ( ?mo.status = Reviewd );
( and ( ?mo.status = Ready )
       no_chain ( ?ta1.status = NotAssigned )
       no_chain ( ?ta2.status = NotAssigned )
      ( unlink [ ?mo.pro_eng  ?ta1 ])
      ( unlink [ ?mo.rev_eng  ?ta2 ]));
```

CodeBox 5: Control Rule

context :

The **control** rule is used by a programmer to review a module. The rule may only be invoked if a review task has been assigned before. The reviewer gets the module's file attribute in a read-only mode. If there are some comments to make, they are saved in a feedback file that is sent to the programmer of the module.

precondition :

The characteristic function searches for the two task objects (*?ta1*, *?ta2*) (see CodeBox 5) linked to the module. As the tasks will be unlinked from the module when the reviewer has no criticisms to make, these objects must be known. Furthermore, the reviewer is also needed in order to compare her/his name (*?pg.login*) with the *CurrentUser* (reviewer) attribute. The property list evaluates the following preconditions:

- if the reviewer who is linked to the module is the current user (Unix system login);
-if the editing session (specification) of the module was accomplished by the programmer.

activity invocation :

The activity runs the *review* envelope. This envelopes pumps up an Emacs editor which is divided in two parts. In the upper part, the module's file attribute is listed in read-only modus. In the lower part, the reviewer may enter her/his comments that will be saved to the *feedback* attribute (*?mo.feedback*). If the latter is not changed, then the review task is supposed to be completed without any critics. Otherwise, the feedback is automatically sent to the programmer.

effects :

Depending on the reviewer's decision, the module is either reviewed, if there are critical remarks, or ready, if the module is approved by the reviewer. In the latter case, the links between the module at one side and the programmer and the reviewer at the other are unlinked, so that the module is no longer accessible neither through the **modify** rule, nor through the **control** rule. Of course, the status attributes of the TASK and REV_TASK objects are reset to *NotAssigned*, which means that the corresponding reviewer and programmer may be reassigned to other modules.

The **control** rule has a possible forward chaining to the **finish** rule (see below) if the (*?mo.status* = *Ready*) predicate is satisfied. However, this chaining is only performed if the other descendants of the development phase have all been completed, i.e. their corresponding status is *Ready* before and the module that has been successfully reviewed was the last module to be defined in the corresponding phase.

c) Doc Rule

```
doc [ ?doc:DOC ]:
  :
{ EDITOR edit_doc ?doc.contents  return ?name ?module }

( and ( ?doc.name = ?name )
       ( ?doc.author = CurrentUser )
       ( ?doc.module = ?module ));
```

CodeBox 6: Doc Rule

context :

Every software development process should be sufficiently documented. The **doc** rule edits a documentation object, or more precisely the object's file attribute contents. The structure of the documentation objects is left to the manager (see data model Figure 3.4). There is no special restriction to edit a document object and the engineer is free to document her/his work. The documentation may be linked to the modules it refers to, but this link is not required. We assume that the engineer introduces an existing module name in the *module* attribute, but there is no test if this name really corresponds to an existing object in the objectbase.

activity invocation :

The activity runs the *edit_doc* envelope. This envelope calls the tool to edit the corresponding module. The programmer has to introduce a document name and the name of the referred module(s). Both values are read interactively in the client window.

effects :

The values are asserted to the correspondent attributes. The *author* attribute is asserted with the current user's name by using the *CurrentUser* primitive (see CodeBox 6).

d) Touch Rule

```
touch [ ?co:CONTENTS ]:
:
no_chain ( ?co.status = Ready )

{ EDITOR touch  ?co.contents }
[ ?co.status = Maint ]
```

CodeBox 7: Touch Rule

context :

The **touch** rule is used for file editing purpose. This rule is however only used in the maintenance phase. In fact, the **touch** rule "breaks" the existing object "network" (links and parent-child relationships) by trying to change a target module after the development phase has been completed.

The **touch** rule does not check if the engineer is the person who worked on the module before. Thus any engineer may use it if the module status is *ready*.

If an engineer changes the target module, all the modules affected by this change may become inconsistent. Thus, all the affected modules will have their status changed to *Maint* in case the target module has been changed.

It is important that all necessary links, i.e. those indicating the propagation of the modules, have been installed (using the **propagate** rule below) before firing the **touch** rule. Otherwise, the consistency of the objectbase is no longer guaranteed.

The rule is applied on objects of class CONTENTS (see CodeBox 7).

precondition :

The **touch** rule has no characteristic function. The property list tests if the module has been completed before (*?co.status=Ready*).

activity invocation :

The activity invokes a **touch** envelope that writes a warning in the client window. The warning should alert the engineer that possible modifications on the file will invoke consistency chainings. The engineer may then either leave the rule without modifying the target module or s/he may change it, which automatically invokes the propagation chain.

effects :

If the engineer does not change the target module, the rule has no effect. Otherwise, the status is changed to *Maint* indicating that the module is in maintenance phase.

There is a consistency chain if the [*?co_status = Maint*] predicate is satisfied (see CodeBox 7). This chain will change recursively the statuses of the affected modules to *Maint*. This is performed by several inference rules (see consistency rules below).

e) Maint Rule

```
maint [ ?co:CONTENTS ]:
:
no_chain ( ?co.status = Maint )
{ EDITOR change ?co.contents }
no_chain ( ?co.status = Ready); (1)        # for text files only
# [ ?co.comp_status = NotCompiled ] (2);  for source files only
```

CodeBox 8: Maint rule

context :

The **maint** rule is the counterpart of the **touch** rule. This rule is analogous to the **modify** rule, but only modifies the module if the object's status is *Maint*. With this rule the programmer may "repair the damage" provoked by the **touch** rule.

This rule does not check the programmer's identity.

precondition :

The property list checks if the status is set to *Maint*.

activity invocation :

The activity runs the *change* envelope which invokes the editor tool to allow the engineer to change the corresponding file.

effects :

No effect is asserted when the programmer has not yet completed her/his work. Otherwise, the status is switched to the corresponding value, i.e.(*?co.status = Ready*) or [*?co.comp_status = NotCompiled*] (see CodeBox 8).

If the second effect [*?co.comp_status = NotCompiled*] is asserted, then all ancestor objects of class MOD_PROG are no longer executable and thus a consistency chain is invoked that resets all ancestor objects to *NotBuild*. For the first effect (*?co.status = Ready*), no chain is useful and is thus prohibited.

If the **maint** rule is invoked with an object of class CONTENTS, then the first (1) effect may be asserted. For an object of class MOD_PROG or MOD_SCR the second (2) effect may be asserted.

2. Coding Rules

The coding rules are needed principally during the physical architecture phase. These rules perform activities such as compiling a source file or running an executable program.

a) Compile Rule

```
compile [ ?mod:MOD_PROG ]:
( forall MOD_PROG ?mod1 suchthat ( member [ ?mod.subroutines  ?mod ] )):
( and  no_backward ( ?mod1.comp_status = Compiled )
                   ( ?mod.status = NotCompiled ))

{ COMPILER  compile  ?mod.source  ?mod.obj_code
                     ?mod.comp_res  ?mod1.source }

( ?mod.comp_status = Compiled );
( ?mod.comp_status = NotCompiled);
```

CodeBox 9: Compile Rule

context :

The **compile** rule invokes a compiler on the *source* file attribute. All the descendant (or children) modules of the parameter object are supposed to be procedures of the latter. In other words, the parameter module represents a program, and the descendants are the procedures used by this program.

The compile rule is invoked on objects of class MOD_PROG or MOD_SCREEN (see example in CodeBox 9).

precondition :

The characteristic function binds all the children objects of a module. The property list checks if they are compiled.

activity invocation :

The activity runs a compiler on the object's source code. If the compilation succeeds, the object's binary file attribute *obj_code* (object code) will contain the object code. If the compilation fails, the errors are stored in the *comp_res* attribute and may be consulted by the **viewErr** rule (see below).

b) Build Rule

```
build [ ?mo:MOD_PROG ]:

( and ( forall MOD_PROG ?mod1 suchthat ( member [ ?mod.subroutines  ?mod1 ]))
      ( forall MOD_SCR ?scr suchthat ( linkto [ ?mod1.screens  ?scr ]))
       ( forall MOD_SCR ?scr1 suchthat ( linkto [ ?mod.screens  ?scr1 ]))):

( and ( ?mod1.comp_status = Compiled )
      ( ?scr1.comp_status = Compiled )
      ( ?scr.comp_status = Compiled )
      ( or ( ?mod.comp_status = NotBuilt )
           ( ?mod.comp_status = Compiled )))

{ COMPILER build  ?mod.source  ?mod.obj_code ?mod1.obj_code
                  ?scr1.obj_code  ?scr.obj_code  ?mod.comp_res  }
( ?mod.comp_status = Built);
( ?mod.comp_status = NotBuilt);
```

CodeBox 10: Build Rule

context :

The **build** rule creates an executable program. This rule is applied on MOD_PROG (see CodeBox 10).

precondition :

The characteristic function searches for all the children of the parameter object (*?mo*). Furthermore, all the screens objects (*?scr*) that are linked either to those children or to the parameter object itself are bound too. The property list checks if all the bound objects are compiled.

activity invocation :

The build envelope compiles the source code (*?mod.source*) of the current object and links it together with the object codes (*?mod.obj_code*) of the other modules. If this building fails, the results are written in the *comp_res* file attribute which may be consulted with the **viewErr** rule.

c) ViewErr Rule

```
viewErr [ ?mod:MOD_PROG ]:
:
{ VIEWER  view_err ?scr.comp_res }
;
```

CodeBox 11: ViewErr Rule

context :

The **viewErr** rule allows to consult the errors that may occur during a compilation or a building activity. In fact, the **compile** rule inserts all compilation errors in a compilation results file (*comp_res*, see CodeBox 11). The rule lists the contents of this file in the client window.

activity invocation :

The *view_err* envelope prints the contents of the *comp_res* attribute on the default printer.

d) Exec rule

```
exec [ ?mod:MOD_PROG ]:
:
no_forward ( ?mod.comp_status = Built )
{ RUNNER exec  ?mod.obj_code }
;
```

CodeBox 12: Exec Rule

context :

The **exec** rule runs an executable program. This rule is only used during the coding phase of the software development process. This rule is applied on objects of class MOD_PROG (see CodeBox 13).

precondition :

The property list checks if there is an executable program, i.e. if the program has been built.

activity :

The activity runs the *exec* envelope. This calls a command tool window where the user may run the executable program. To return to the Marvel environment, the programmer has to delete this window by typing *exit*.

e) Print_out Rule

context :

The **print_out** rule allows the printing of the text files on the default printer.

precondition :

The property list checks if the module's file has already been edited, i.e. if there is a file to print.

```
print_out [ ?co:CONTENTS ]:
:
( or  no_chain ( ?co.status = Active )
      no_chain ( ?co.status = Done )
      no_chain ( ?co.status = Reviewd )
      no_chain ( ?co.status = Ready ))
{ PRINTER print  ?co.contents  }
;
```

CodeBox 13: Print_out Rule

activity invocation :

The file of the module is sent to the default printer. This printer has possibly to be changed in the print envelope.

C. General Rules

In the general rules section, we specify all those rules that are not directly related to the construction of a software product, but are necessary for a correct execution of the process. These rules allow to pass from one development phase to another. In a consistent objectbase, all these rules are invoked automatically by chaining and need thus not to be fired manually.

a) Initiate Rule

```
initiate [ ?p:PROJECT ]:
:
( ?p.status = NotActive)
{ INTERACTIVE ini  }
( ?p.status = Active );
```

CodeBox 14: Initiate Rule

context :

The **initiate** rule is the first rule to be fired when a team starts with the development of a software. The rule activates the project, so that the assignment work may be started. This is the only rule that is fired manually by a manager. It is only available in the set of rules dedicated to the team manager. There is no further test about the identity of the latter.

precondition :

No characteristic function is necessary. The property list evaluates whether the project status is *Active* or *NotActive* (see CodeBox 14).

activity invocation :

If the project was not yet activated, the *initiate* envelope is run and the manager is asked if the requirements are met, which allows the project team to start with the specification phase.

effects :

A positive response changes the status to *Active*. The **initiate** rule performs an automation chain to the **activate** rule.

b) Activate Rule

```
hide activate [ ?log: LOGARCH]:
( and ( exists PROJECT ?proj suchthat ( member [?proj.log_architecture  ?log]
        ( exists SPEC ?spec suchthat ( member [ ?proj.specification ?spec ])):
( ?spec.status = Done )
{}
 no_chain ( ?log.status = Active )
```

CodeBox 15: Activate Rule

The above rule is used to activate the logical architecture phase. The physical architecture is activated in a similar way. The specification phase, however, has no preceding phase and is thus activated automatically after project initiation.

context :

The **activate** rule allows to start with a new software development phase. The rule is either invoked after the **initiate** rule or the **finish** rule. Unless this rule is not fired, the programmer cannot assign engineers to modules under this development phase.

As the engineers need not to fire this rule manually, it is hidden to the latter, i.e. it does not figure among the commands in the rule menu. The activate rule is applied on SPEC, PHYS_ARCH or LOG_ARCH (see example in CodeBox 15).

precondition :

The characteristic function searches for the preceding development phase of the process (e.g. specification precedes logical architecture). The property list tests if the phase has been finished.

effects :

The development phase is activated and the development process may continue.

c) Achieve Rule

```
achieve [ ?st:STATE ]:
( and ( exists SPEC ?spec suchthat ( member [ ?spec.db_primitives ?st ]))
      ( forall FUNCTION ?func suchthat ( member [ ?spec.functions  ?func ]))):

( and no_chain ( ?func.status = Ready)
     no_backward ( ?st.status = Active ));
{}
( ?st.status = Done )
```

CodeBox 16: Achieve Rule

context :

The **achieve** rule is used in different contexts. The main goal is to complete a task that has not been completed (or could not have been completed) by another rule.

For instance, a programmer may not complete an object of class STATE unless all the functionalities (of class FUNCTION) are completed. To finally terminate the modification of the state, i.e. to set the status to *Done*, the **achieve** rule is fired (for more details about the **achieve** rule see the code in the Appendix C).

precondition :

As the situations in which the **achieve** rule may be used are very different, the precondition is specific for each of them. Remember, however, that the objects on which the precondition is verified are queried by the characteristic function. In CodeBox 16, we give one of the possible **achieve** rules.

effects :

The status attribute of the object is updated.

The **achieve** rule waits until all the functions are completed. Only then it allows to finish the modification of the state (*?st*) which means that the module may be reviewed.

d) Finish Rule

```
hide finish [ ?spec:SPEC ]:
( forall MODULE ?mo suchthat ( ancestor [ ?spec  ?mo ] )):
( ?mo.status = Ready )
{}
( ?spec.status = Done );
```

CodeBox 17: Finish Rule

context :

The **finish** rule completes a development phase. In fact, to respect the order in which the development phases are performed, it is essential that a phase has to be completed before the process can go on to the following phase. The **finish** rule definition for the other development phases is implemented by replacing the SPEC class by LOGARCH or PHYSARCH (see CodeBox 17).

precondition :

The characteristic function searches all descendants of the phase object. The property list then checks if all the found objects are *Ready*, which means that the phase is completed.

effects :

The status of the correspondent phase object is set to *Done*. The **finish** rule invokes a forward chain to activate the next development phase.

e) Send Rule

```
send [ ?mo:MODULE ]:
:
( and no_backward ( ?mo.status = Done )
      no_backward ( ?mo.reviewer  <> "none"))
{ MAILER mail_rev  ?mo.reviewer  ?mo.name }
;
```

CodeBox 18: Send Rule

context :

The **send** rule organizes an automatic mailing mechanism among the team members. The contents of the mails are standardized. The rules are invoked automatically if a team member completes her/his task on an object such that this object may be reviewed by another team member. For instance, the programming engineer gets a schedule via mail as soon as the assignment is done.

precondition :

The **send** rule is applied on the module that has been completed (*?mo.status = one*). If the *reviewer* attribute is not *none* (the default value), then there is already a reviewer assigned (see CodeBox 18). For more code details, see the *mail* strategy in the Appendix C.

activity invocation :

The activity runs the *mail_rev* envelope. This envelope simply sends the standard text to the reviewer indicating also the name of the module.

D. Consistency rules

The objectbase has to reflect the process development status at any time. Some rules are defined only for consistency maintenance. Most of them are not visible to the user. To keep the database consistent and powerful, the programmers have to use some rules manually.

a) Propagate Rule

```
propagate [ ?mo:MODULE,  ?mo1:MODULE ]:
  :
{}
( linkto [ ?mo.attr ?mo1 ] );
```

CodeBox 19: Propagate Rule

context:

The **propagate** rule produces a link between two subclasses of the MODULE class. This link shows the propagation of the module through the different development phases.

The rule must be used by the engineer. There is no chaining related with this rule, neither is there a help facility. In fact, the Marvel system cannot anticipate these links. But to perform properly, the engineer should use the **propagate** rule as soon as possible to visualize the propagation.

The rule has the same effect as the *link* built-in command. The use of the **propagate** rule is however easier as the correspondent attribute is already defined, contrary to the link built-in command.

The link established by the rule is vital if the process should work correctly. This link has to be persistent. As the link may be removed by the *unlink* built-in command, we assume that the latter is only used for correction activities.

The parameters used for these rules are of subclasses of MODULE. The *attr* attribute is replaced with the current link attribute of that subclass (see data model in Section 2.3).

effects :

A link is established between the two modules.

b) Inference Rules for Consistency Maintenance

To keep the objectbase consistent at any time, the process model contains some inference rules that are fired after the execution of an engineer's activity. All these rules are hidden to the user. Once the consistency chain is invoked, it cannot be stopped by the

engineer anymore. In the case that one of the rules fails, for any reason whatever, the whole chain is **rolled back.**

We add some consistency rules that "propagate" some changes in order to reflect the current actual state of the objectbase at any time. The consistency chains are invoked either by the **touch** or by the **maint** rule.

The consistency chain follows the "roads" constructed by the **propagate** rule and thus the latter has to be used correctly. In the following, we explain one of those rules, namely **touchdown**, in more details. All the other rules are built according to the same schema.

a) Touchdown Rule

```
hide touchdown [ ?lfunc:L_FUNC ]:
( exists FUNCTION ?func suchthat ( linkto [ ?func.log_repres  ?lfunc ])):
[ ?func.status = Maint ]
{}
[ ?lfunc.status = Maint ];
```

CodeBox 20: Touchdown Rule

context :

The rule changes the status of the L_FUNC object (*?lfunc*) that is linked by the FUNC object (*?func*) if the latter has been touched (i.e. *status* = *Maint*). Note that both the property list and the effect clause are between square brackets (see CodeBox 20).

precondition :

The characteristic function searches the "previous" object. This object may be either connected via a link or a parent-child relationship. The property list then evaluates if the status of the "previous" has been changed by the **touch** rule.

effects :

If the precondition is satisfied, the status of the current object is also changed. The rule verifies if there are eventually other chainings to perform.

In Figure 3.9, we show the possible consistency chainings among the rules. The doted line is the representation in the Marvel print graph option for consistency chainings.

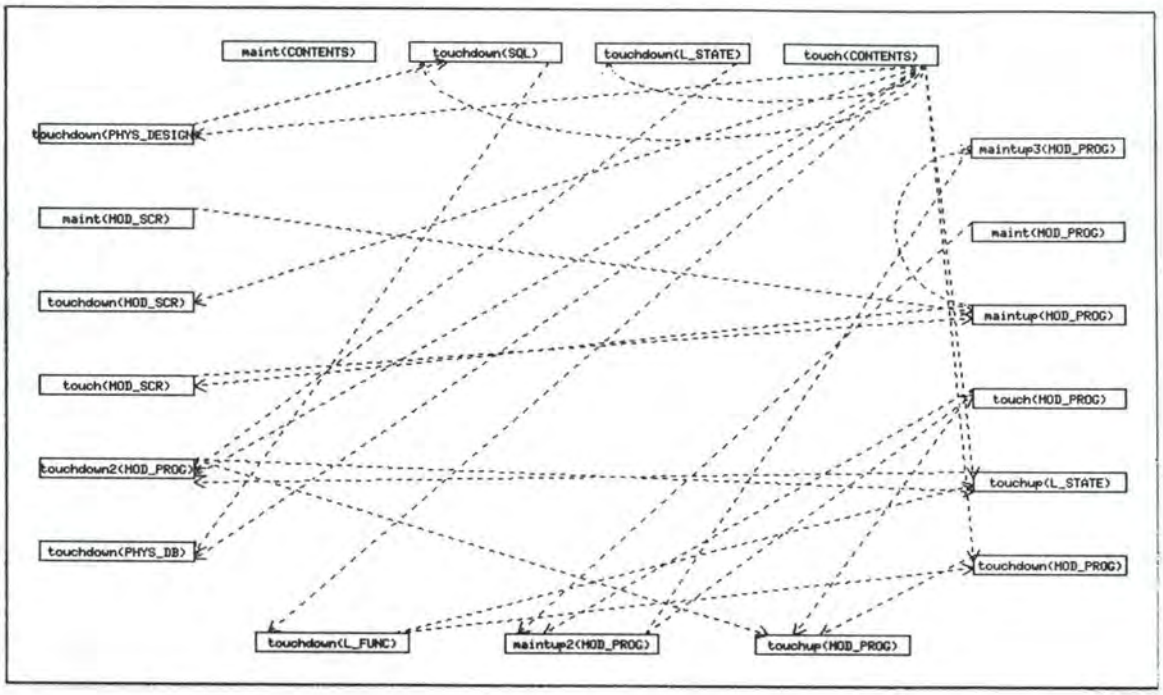

Figure 3.9: Consistency-Chain Graph.

The consistency chain graph represents the consistency chainings of the maintenance phase. A complete example of a possible chain is given in Section 2.4.2.1.D.

2.5 Initialization of the Objectbase

Before a team may start the actual development work, the objectbase has to be "filled-up" in order to represent the development process. The objectbase may be created progressively, but it is preferable to have a minimum of objects of each class in the beginning to get a better overview of its extent. We have built such a "prototype objectbase" for our environment (see Figure 3.10). The names of the objects represent their contents in an abstract manner. These names may be changed, if desired, by the manager using the *rename* built-in command. Remember, however, that each assignable module gets a name during the execution of the **assign** rule.

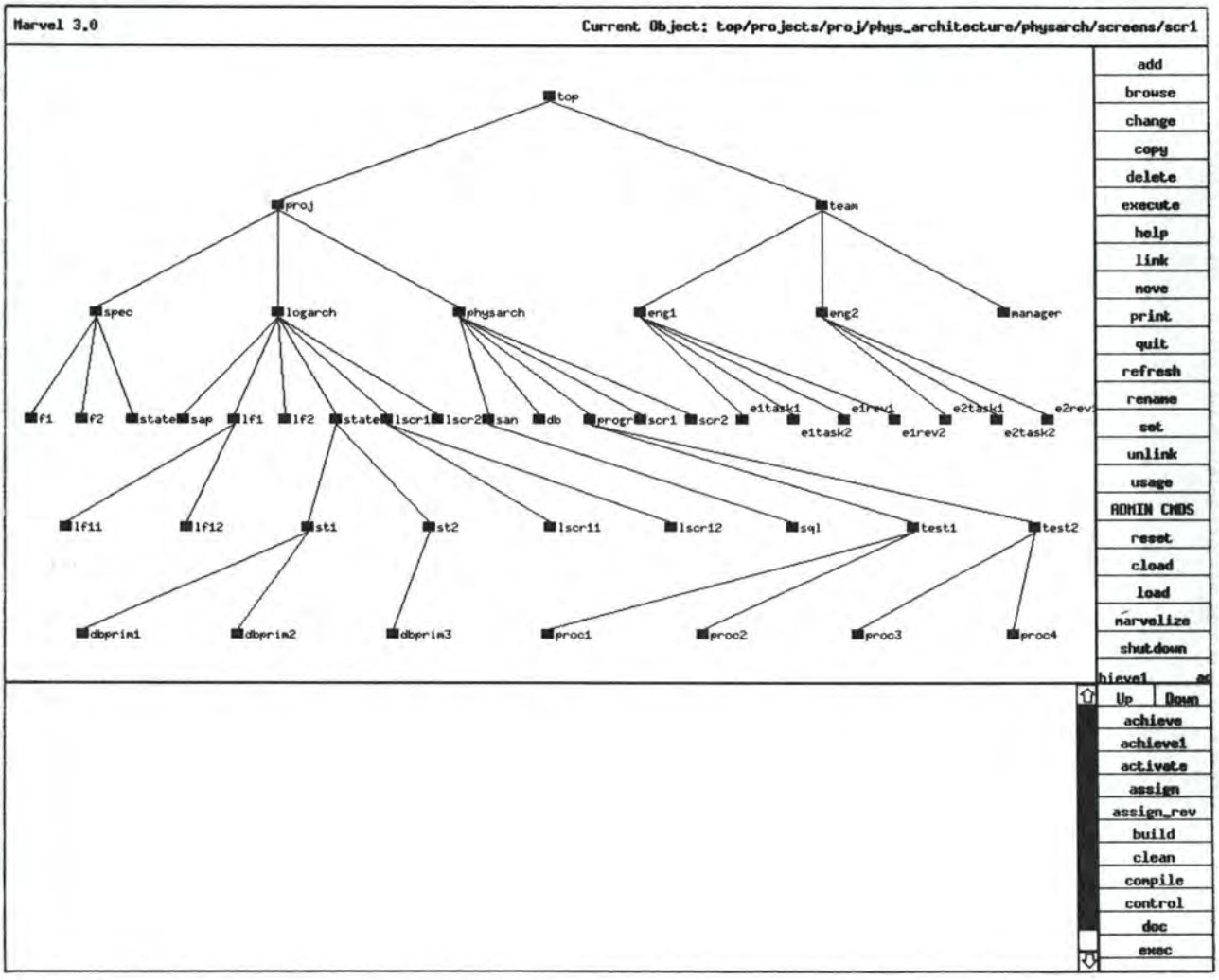

Figure 3.10 : Example of "Prototype Objectbase"

3. MARVEL EVALUATION

In this section we present four categories of problems. By conceptual problems (Section 3.1) we understand features that we found but that are not operational in the current version. The missing features section (Section 3.2) brings out some features that might improve the "power" of the Marvel kernel. Furthermore, we give some constructive remarks about the Marvel Graphical User Interface (GUI) (Section 3.3). Finally, we mention the programming errors (Section 3.4) that we discovered during the development of our Marvel environment.

3.1 Conceptual Problems

The conceptual problems we detected are the Unix file system restrictions, the limitation of multi-parameter rules and the MSL syntax.

3.1.1 Unix File System Restrictions

The composite-hierarchy is mapped to the Unix file system and every object is represented in a unique Unix directory. The object hierarchy presented in Figure 3.11 is a possible one. This means that every engineer (*eng1* and *eng2*) may have several tasks with identical names (*task1* and *task2*) as the respective directories are unique (*.../team/members/**eng1**/jobs/task1* and *.../team/members/**eng2**/jobs/task1*). This denomination causes, however, troubles during rule invocation.

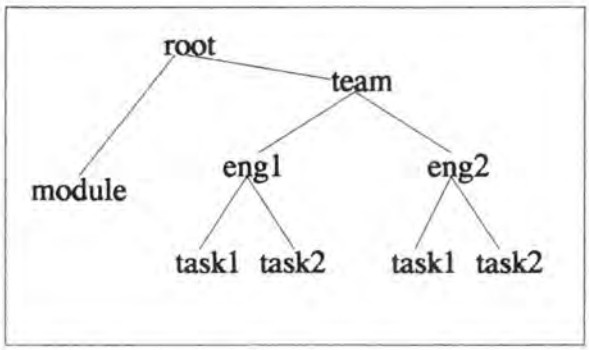

Figure 3.11: Composite-Object Hierarchy

For instance, if the manager wants to assign the *module* to *task1* of *eng1*, the Marvel system produces an error message saying that *task1* is not unique and it does not invoke the **assign** rule. This behavior is difficult to understand, as the end-user explicitly selects the two object parameters of the rule. Consequently, the semantics of the composite-object hierarchy cannot be fully represented by the Unix file system. This has as a consequence that objects of the same class must have different names.

3.1.2 Chaining on Multi-Parameter Rules

The chaining mechanism is the important feature of Marvel, but it is not yet fully operational. Although the definition of multi-parameter rules is allowed, the chaining mechanism, as it is currently implemented, is not totally applicable to those rules. A multi-

parameter rule does not allow other rules to forward chain into it. This is due to the way the Marvel system determines the parameter object at execution time, i.e. dynamic binding of parameter (see Part 2). A chaining is always performed based on a predicate that allows to find the correspondent parameter, whereas the other parameter cannot be found with this approach. For example, the **assign** rule in our environment has a possible forward chaining to the **assign_rev** rule. Both are multi-parameter rules, but while it is possible to start a forward chain from the **assign** rule, it is impossible to forward chain into the **assign_rev** rule as the Marvel system cannot determinate the second parameter of that rule (of class REV_TASK).

3.1.3 The MSL Syntax

To be more readable, the MSL syntax should include some additional keywords, especially to distinguish the different parts (characteristic function, property list...) of a rule. Another problem is the difficulty to visualize chaining possibilities. In fact, MSL does not explicitly show the chaining possibilities of the rule. Additional keywords could improve the situation that could avoid the use of the *print-graph* option.

3.2 Missing Features

In the current version there are no privileges on built-in commands, neither is there a possibility to use built-in commands in rules. We describe the usefulness of these features and give some ideas how they could be implemented.

3.2.1 Privileges on Built-In Commands

There is no mechanism to provide access privileges either to objects, or to built-in commands. Without access restrictions, every team member can add or delete objects, which may cause severe consistency problems (e.g. too much objects, additional links that destroy the consistency defined in the data model). In our environment, we assume that the manager is the only person who may add/delete objects of the objectbase. There is however no possibility to control this assumption. A hard-coded solution would remedy to this problem. Different solutions are imaginable. A first possibility would consist in a restrictive access to the built-in commands. This means that the administrator has to define for every user a set of the accessible (or not accessible) built-in commands analogous to the subset of rules. A second possibility would be to define for every object class the access privileges, i.e. add, delete or change an object, with regards to the read/write/execute privileges of the corresponding Unix file. In this case it would be the owner of the account who would define the accesses.

As the first possibility is independent of the Unix file system, the access privileges are easier to be defined by the administrator, but perhaps more difficult to implement in the Marvel kernel. In our environment, the manager would have access privileges to all the built-in commands, whereas the programmers would have restricted accesses. On the other side, the second possibility is more flexible and easier to implement. In our environment, the latter solution would not be convenient, as the hierarchy of a team (manager and software engineers) could not be clearly defined by the administrator. This means that in a "multi-user" environment like ours, it is important that the project manager defines the objectbase, and thus the access privileges.

3.2.3 Built-In Commands and Rules

In the current Marvel version, the only built-in commands that can be used in a rule specification are the *link* and *unlink* commands. In some cases, other built-in commands could be useful. You could imagine adding an object within a rule with the built-in *add* command. We can illustrate this by giving an example of our environment. In the latter, a *func* object has to be linked to an *lfunc* object. In the current version of our environment, we only assume that this is realized by a team member. With the possibility to use built-in commands in rules, the process could create the object (*lfunc*) and the corresponding link automatically. This would improve the consistency of the objectbase which would no longer be based on assumptions.

3.3 About the Interface

In this section we briefly describe the Marvel window and give suggestions how it may be improved.

3.3.1 The Marvel Window

The Marvel GUI is shown in Figure 3.12. It is composed of one window, called the Marvel Window, divided in several sub windows. It is important to notice that this GUI is complemented by a **client window**. The latter is a command tool window where the client session has been started and that is used for user communication purposes.

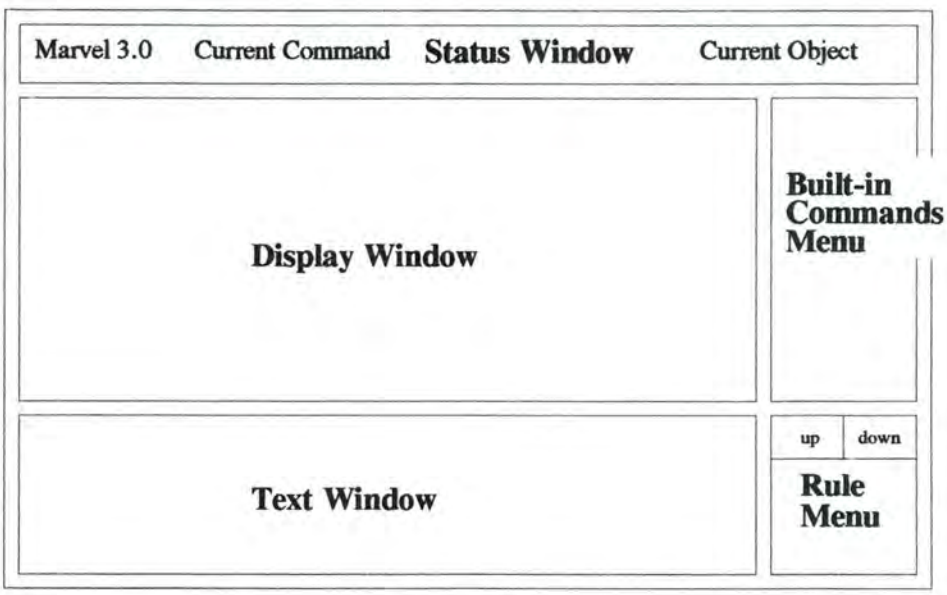

Figure 3.12: The Marvel Window

The **Status Window** displays the current **Marvel version**, the **Current Object** and the **Current Command.** The Marvel version value is permanently present (in our case 3.0), whereas the Current Command is only shown if a command is selected, but not yet executed (e.g. the user is selecting the parameter object(s)). The Current Object value gives the access path (in terms of the Unix file system) to the current object. For instance,

if the *test1* object of Figure 3.10 is the current object, its value would be *top/projects/proj/phys_architecture/physarch/code/progr/subroutines/test1*.

The **Display Window** shows the objects present in the composite-object hierarchy and is also used to display the chaining graph built with the *print-graph* command.

The **Text Window** is used for communication purposes. On one side, the user specifies the parameter object(s) for a selected rule in this window. On the other side, the Marvel system provides information about objects or rules in response to a user request, or the system displays performed chainings and possible error messages.

The **Built-In Commands Menu** contains all the predefined commands of the Marvel system. These commands are either used to make modifications on the composite-object hierarchy (e.g. add, delete, move) or they are used to change the display layout of the latter (e.g. zoom, display options). If a built-in command has several options, the latter are accessible by clicking with the mouse on the corresponding command menu box.

The administrator has an additional command menu, which is situated in the lower part of the built-in Command Menu (see Figure 3.10) and help the administrator in the environment construction.

Finally, the **Rule Menu** contains all or a subset of the rules specified by the project administrator to model the dynamic behavior of the environment. If all the rules do not fit in the Rule Menu box, the *Up/Down* buttons allow to scroll through them.

3.3.2 Towards an Unique Communication Window

In the current Marvel version, there are two different windows where communication with the user is performed. In the Text Window (see Figure 3.12), the user gets all communications concerning the composite-object hierarchy and enacts the rules. In the Client Window, all the communications related to activity invocations (envelopes) are performed.

A better and more coherent solution would be to allow all types of user communication in one Window, namely the Text Window. The necessity for a unique window is illustrated in Figure 3.13.

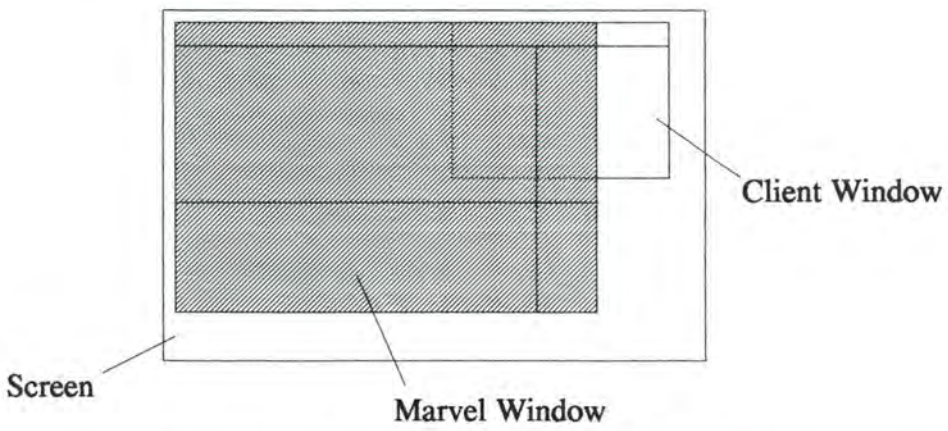

Figure 3.13: Overlapping of the Client Window with the Marvel Window

In fact, the Marvel Window is not resizable and most of the time it hides the Client Window. This is confusing for a user if the Text Window is blocked in an activity invocation and is waiting for a response in the Client Window, that is not visible at this time. In a unique window, such a situation could not disturb a user.

If this unique communication window is difficult to implement, the user must a least have the possibility to resize the Marvel Window in order to allow him to have both communication windows visible on the screen.

3.3.3 Problems with the GUI

We are conscious that the current version of the Marvel GUI is still at a prototype level and thus presents some weaknesses which we will emphasize in the following paragraphs.

The composite-object hierarchy may not have more than 10 levels. Once a user defines the 11th, the Marvel system shuts down (core dump) and the current objectbase is unrecoverable. This problem could be avoided if it would be documented.

Another problem in the Status Window, is the overlapping of the Current Command value by the Current Object value once the latter becomes too long. Of course this problem is of less importance than the one cited before, but nevertheless it causes a readability problem and should be taken into consideration for future versions.

Thanks to the Display Window, the end-user has a correct representation of the tree structure of the objectbase, which could be summarized as *What you define is what you see*. Once the objectbase becomes too large, the user may use the *zoom* command to guarantee the readability of the objectbase structure.

The *print-graph* command whose output (chaining graph) is displayed in the Display Window, constitutes a powerful tool for the administrator. However, there are some technical problems with that command. First, this command needs at least 5 rules to print a chaining graph. Although it may not seem very useful to print a chaining graph with less then 4 rules, there is no reason to prohibit it. Furthermore, if there are too many rules loaded simultaneously in the Marvel system, the chaining graph becomes unreadable (Figure 3.14 shows the chaining graph of our environment). Finally, the chaining graph is not permanently the same for a same environment. In fact, the graph in Figure 3.14 is not similar to the graph presented in Figure 3.15 although they were both constructed with exactly the same rule set.

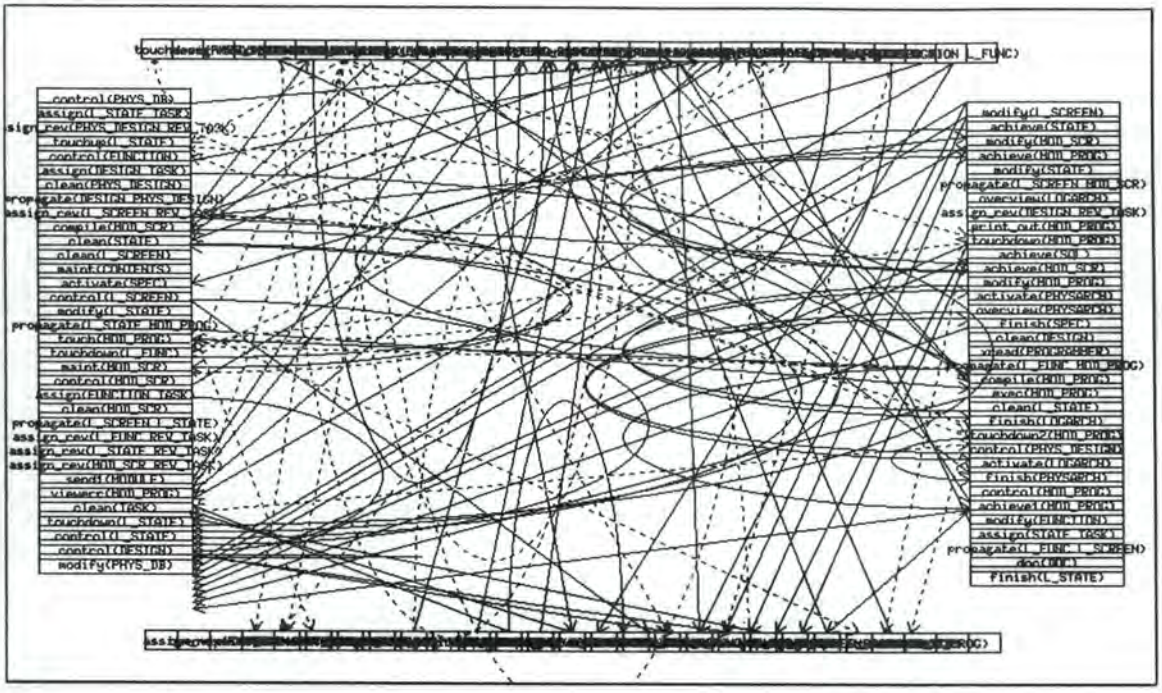

Figure 3.14: Chaining Graph

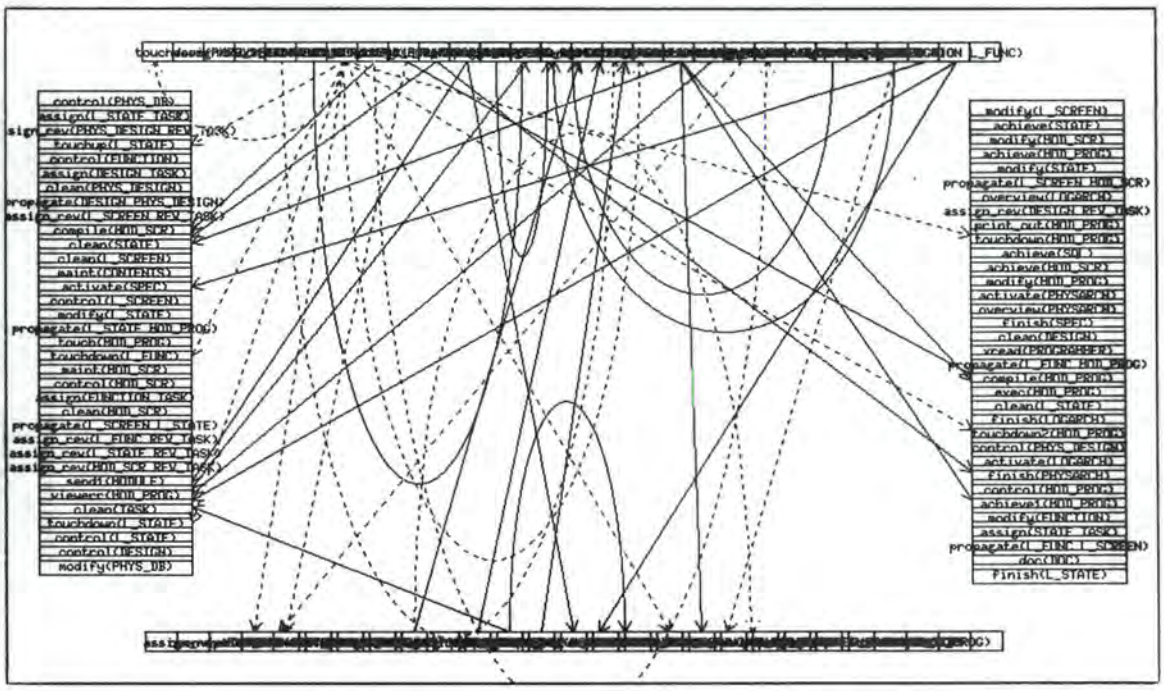

Figure 3.15: Chaining Graph

The Text Window allows to make available a "history" of the previous command executions which may be consulted by scrolling through this window. The space of the Text Window buffer seems, however, too small. This "problem" was detected when the administrator asked to print the definition of a rule in the text window. It happens that this rule had many instantiations, but it was not possible to consult all of them. An extension of the buffer size may probably remedy to this situation.

Furthermore, the end-user may select the parameter object(s) of a rule by typing it (them) in the Text Window. The Marvel system supports an easier and more ergonomic possibility, namely to click on them with the mouse. This may avoid typing errors. However, once the rule is selected there is no *undo* command to cancel its invocation. In this case, the only way to simulate the *undo* command is to fire the rule without arguments, which produces an error message in the Text Window.

The produced error messages are too general. They do not precisely express the reason why an error occurred and sometimes the same error message is used for different types of errors. This is very confusing for the user. For instance, if the user gives more than one object for a single-parameter rule or if s/he applies a rule on a wrong object class, the error message will be the same in both cases. Another problem has been detected for error messages produced by a failed precondition. Remember that the precondition is composed of a characteristic function and a property list. Nevertheless, the produced error messages are inconsistent, i.e. a failed characteristic function produces no error messages, whereas a failed property list does, but it does not mention the failed predicate.

3.4 Programming errors

The programming errors we detected deal with the lacking inheritance of link attributes, the denomination of attributes and rules, and the MSL loader.

3.4.1 Inheritance Attribute

The Marvel system provides for an object class inheritance mechanism. When we tried to inherit the MODULE superclass attributes to all its subclasses, we run into problems as the link attributes were not inherited. However, there is no plausible reason why link attributes should not be inherited. Thus we assume that this is a programming error (bug) that should be corrected in future Marvel versions. We solved the problem by explicitly specifying the corresponding link attributes in every subclass of the MODULE superclass.

3.4.2 Attribute and Rule Names

Every Marvel class definition contains a set of attributes. According to the Marvel manuals, every identifier may be used to denote the attributes. However, for the attributes of type string, the "name" identifier causes problems. As long as the server is running, the value asserted to this "name" attribute causes no problem. Once the client session is finished and the server killed, the "name" attribute loses its current value and is replaced with the default value at the next interactive session. As the loader does not notify an error during the compilation of the data model, this is a severe problem.

An analogous problem was detected with the rule denomination. The rule name must be different from the names of the built-in commands. For instance, we called a rule *print_out* instead of *print*. We tried the latter denomination, but the system did not invoke the correspondent rule without giving an adequate error message. Even if this is a restriction, it is understandable for semantic reasons. On another side, we called a rule *review* according to the task it should perform. But this name was refused by the Marvel system and caused an error message although the name does not figure among the built-in

command names. If these restrictions are known, they should be explicitly mentioned in the Marvel manuals.

3.4.3 The MSL Loader

The MSL loader is used by the administrator to translate the MSL specifications (i.e. the data and the process models) into an intermediate representation. The resulting error messages of the loader are, however, not expressive enough to help the administrator in the error correction. Other errors are even not mentioned by the loader. In the sequel, we give some concrete examples of this problem.

In our data model we used single quotes ' ' instead of double quotes " " (see Box 25) and the loader gave the following error message : ('.' '.'). This is obviously not a very expressive error message to help the administrator to localize the syntax error.

```
TASK :: superclass...
..
end;

MODULE :: superclass...
        contents : text ='.src'     # use of single quotes is incorrect
        pro_eng : link task         # lower case is an error
end.
```

Box 25 : Example of Loader Problems

In an other case, we made a type error by defining an attribute type as (pro_eng: *link* **task**) instead of (pro_eng: *link* **TASK**). The loader, however, completed the parsing of the data model without any error messages, although the *link* keyword has always to be followed by a class type identifier, i.e. TASK in uppercase letters according to the definition of the class (see Box 25). When we wanted to start the Marvel server, the system crashed ("segmentation fault").

When the administrator tries to load a strategy within the Marvel Text Window by using the built-in *load* command containing some syntax errors, the resulting error message is : **"rc is"** followed by a number. To find the corresponding error, the administrator has to search outside the Marvel Window to identify the error.

In future versions of Marvel, it will be necessary to "upgrade" the loader to a powerful compiler.

CONCLUSION

CONCLUSION

In this work, we reported an experiment using Marvel to support a specific development process covering the whole software lifecycle. Through this experiment, we have learned about the benefits of using Marvel with respects to another Software Development Environment (SDE) but we have also detected some weaknesses. More specifically our contribution was reported in three parts:

In Part I, we presented the main concepts characterizing second generation SDE's. Then, we defined a set of criteria and analyzed to what extent they are verified by four representative SDE's.

In Part II, we made a synthesis of the main papers published on Marvel and illustrated the main concepts by progressively building a small Marvel environment.

In Part III, we described the Marvel environment that we have implemented with the Marvel kernel for covering a whole lifecycle developing process. We isolated the main concepts of the development process in order to conceive the correspondent data model and identified the sequence of tasks to implement them with Marvel rules. This in depth study allowed us to evaluate the Marvel system at the end of this part.

The Marvel environment allows us to support some kind of transformational approach up to a certain degree. Moreover, it includes some team management aspects. It is important to emphasize that the primary goal of the experiment was to investigate to what extend Marvel is able to meet the whole lifecycle. Thus the resulting environment should rather be considered as a tool for learning a development process than to be used for real applications. This is due to the following reasons.

First, the quality of a Marvel environment is largely dependent of the various tools it may invoke. In our environment, we only use a normal editor to edit the various specifications and algorithms. Thereby, we have not integrated feedback due to corrections in specification.

Second, the notion of active user guidance in our environment is limited. In fact, the environment assures the chronological sequence of tasks by prohibiting the premature firing of a rule that enacts this task, rather than it indicates the different steps to follow by the user. A development task is always initiated by a user and not by the Marvel system. As there are no automated activities to perform between the development tasks in the specification and the logical architecture phases (no adequate tools are provided), the process consists in a list of user initiated task without active guidance in between.

Third, the necessity to restrain the number of chainings decreases the "active" aspect of the environment. Remember that we had to reduce the number of chainings as Marvel does not yet provide the possibility to chain between different users.

Finally, the present environment is based on a large number of assumptions. A first category concerns the user-friendliness of the product. All actions are supposed to be performed by a software engineer who does not attempt to crash the system. These assumptions could be avoided by improving the user interface of our environment. Another category is due to missing features of the Marvel kernel noticed before (Section 3.2 of Part III).

In this work, we only reported the way we followed for instantiating Marvel to our own purpose but we feel necessary the existence of a methodology bringing active guidance to the Marvel administrator.

To conclude, we would like to say that the criticisms made in Part III are not intended to devaluate the Marvel kernel, but rather to indicate the features that should be considered in future research.

We feel that our instantiation of the environment could be improved with the features presented at the end of Part III. Furthermore,

- its active user guidance has to be reconsidered which means to provide powerful tools for the specification and design phases.

- another interesting attempt could be the integration of the C/Marvel environment into our environment in order to improve the modeling of the coding phase. This experiment could even be undertaken without further modifications of the Marvel kernel.

REFERENCES

REFERENCES

[Bar88], BARGHOUTI, N.S., KAISER, G.E., *Implementation of a Knowledge Based Programming Environment*, Proc. 21st Annual Hawaii Int'l Conf. on Systems Sciences, IEEE Computer Society Press, Los Alamitos, Calif., pp. 54-63, 1988.

[Bel92], BELKHATIR, N., MELO, W.L., ESTUBLIER,J., NACER, A.N., *Supporting Software Maintenance Evolution Process in the Adele System*, Proc. of the 30st Annual ACM Southeast Conference, Raleigh, NC, April 8-10, 1992.

[Bel91], BELKHATIR, N., ESTUBLIER, J., MELO, W.L., *Adele 2 : A Support to Large Software Development Process*, Proc. of 1st International Conference on the Software Process, IEEE Computer Society Press, Redondo Beach, CA, October 21-22, 1991

[Ben90], BEN-SHAUL, I., *An Object Oriented Framework for Rule-Based Development Environments (Position Paper)*, Department of Computer Science, Columbia University New York, July 1990.

[Cou91], COUTAZ, J., BASS, L., Developing Software for the User Interface, SEI Series in Software, Addison Wesley Publishing Company, 1991.

[Dat90], DATE, C.J., *An introduction to Database Systems*, Volume I, 5th ed., The Systems Programming Serie, Addison Wesley, 1990.

[Dow87], DOWSON, M., *Integrated Project Support with IStar*, IEEE Software, 4(6), September 1987, pp.6-15.

[Dub91], DUBOIS, E., *Méthodologie de Développement de Logiciels (MDL)*, Notes de cours de 2ème Licence et Maîtrise en Informatique, Année Académique 1991-1992.

[Gis91], GISI, M.A., *Extending A Tool Integration Language (Experience Report)*, Department of Computer Science, Columbia University New York, April 1991.

[Hai86], HAINAUT, J.-L., *Conception Assistée des Applications Informatiques* , Tome 2, Conception de la base de données, Masson, 1986.

[Hei91], HEINEMAN, G.T., KAISER, G.E., BARGHOUTI, N.S., BEN-SHAUL, I., *Rule Chaining in MARVEL : Dynamic Binding of Parameters (Technical Report)*, Department of Computer Science , Columbia University New York, May 1991.

[How82], HOWDEN, E.W., *Contemporary Software Development Environments*, Communications of the ACM, 25(5), May 1982, pp.318-329.

[Kai90], KAISER, G.E., BARGHOUTI, N.S.,SOKOLSKY, M.H., *Preliminary Experience with Process Modeling in the MARVEL SOFTWARE DEVELOPMENT Environment Kernel*, Proc. 23rd Ann. Hawaii Int'l Conf. Systems Sciences, IEEE Computer Society Press, Los Alamitos , Calif., 1990, pp. 131-140.

[Kai88a], KAISER, G.E., BARGHOUTI, N.S., FEILER, P.H, SCHWANKE, R.W., *Database Support for Knowledge-Based Engineering Environments*, IEEE Expert, Vol.3, No.2, Summer 1989, pp. 18-32.

[Kai88b], KAISER, G.E., FEILER, P.H., POPOVITCH, S.S., *Intelligent Assistance for Software Development and Maintenance*, IEEE Software ,Vol.5, No.3, May 1988, pp. 40-49.

[Kho90], KHOSHAFIAN, S., ABONOUS, R., *Object Orientation Concepts, Languages, Databases*, User Interfaces, Wiley, 1990.

[Mar92], *MARVEL 3.0 Implementor's Manual* , Department of Computer Science, Columbia University New York, January 23, 1992.

[Mar91a], *MARVEL 3.0 Administrator's Manual* , Department of Computer Science, Columbia University New York, October 9, 1991.

[Mar91b], *MARVEL 3.0 User's Manual*, Department of Computer Science, Columbia University New York, October 25, 1991.

[Mey88], MEYER, B. *Object-Oriented Software Construction*, Prentice Hall International, Series in Computer Science, 1988.

[Nil87], NILSSON, N.J., *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann Publishers, Inc. Los Altos, CA, 1987.

[Ost87], OSTERWEIL, L., *Software Processes Are Software Too*, in Proc. 9th Int. Conf. Software Engineering, Monterey, CA, March 1987, pp.2-13.

[Ost81], OSTERWEIL, L., *Software Environment Research: Directions for the Next Five Years*, Computer 14(4), April 1981, pp.35-43.

[Par72], PARNAS, D., *On the criteria to be used in decomposing systems into modules*, Communication of the ACM, 15 (2), 1972, pp.1053-1058.

[Rat87], RATCLIFF, B., *Software Engineering: Principles and Methods*, Blackwell Scientific Publications, 1987.

[Shn86], SHNEIDERMAN, B., Designing the User Interface, Reading , Mass.: Addison Wesley, 1986.

[Som89], SOMMERVILLE, I., *Software Engineering* , 3rd ed., Addison-Wesley, 1989.

[Sten87], STENNING, V., *On the Role of an Environment*, in Proc. 9th Int. Conf. Software Engineering, Monterey, CA, 1987, pp.30-34.

[Sten86], STENNING, V., *An Introduction to ISTAR*, in SOMMERVILLE, I, ed., *Software Engineering Environments*, Peter Peregrinus Ltd., London, 1986, pp.1-22.

[Tan89], TANENBAUM, A., *Computer Networks* , 2nd ed. Prentice Hall International Inc., Englewood Cliffs, N.J, 1989.

[Vla90], van LAMSWEERDE, A., *A Distributed Rule Base Architecture for Making Project Databases Active*, CS Department, University of Louvain-La-Neuve, April 15, 1990.

[Vla88], van LAMSWEERDE, A., DELCOURT, B., DELOR, E., SCHAYES, M.C., CHAMPAGNE, R., *Generic Lifecycle Support in the ALMA Environment*, IEEE Transactions on Software Engineering, vol. SE-14, no. 6, June 1988, pp. 720-741.

[Vla87], van LAMSWEERDE, A., BUYSE, M., DELCOURT, B., DELOR, E., ERVIER, M., SCHAYES, M.C., BOUQUELLE, J.P., CHAMPAGNE, R., NISOLE, P., SELDESLACHTS, J., *The Kernel of a Generic Software Development Environment* , Proceedings 2nd ACM Software Engineering Symposium on Pratical Software Development Environments. ACM Sigplan Notices 22(1), January 1987, pp. 208-217.

[Vla82], van LAMSWEERDE, A., *Les outils d'aide au développement de logiciels: un aperçu des tendances actuelles*, in Proc. J11A 82, Paris, June 1982.

[Was89], WASSERMAN, A.I., *Tool Integration in Software Engineering Environments*, in Fred Long (Ed.), Software Engineering Environments, International Workshop on Environments, Chinon, France, LNCS 467, Springer Verlag, September 1989, pp.137 149.

APPENDIX

APPENDIX A : MSL Reference Manual

This is the full definition of the MSL language. It is written in the forms of tokens (terminals) and productions. The parser and semantic analyzer are implemented in yacc, an LALR shift-reduce parser generator, and the lexical analyzer is implemented in lex, a lexical-analysis generator, which recognizes regular expressions. Familiarity with yacc and lex will help to understand this appendix but is not required. Familiarity with context-free grammars is required.

1. The Tokens

1.1 Basic patterns

```
DIGIT       [ 0-9 ]
LETTER      [ a-z A-Z ]
BOTH        { LETTER }| { DIGIT }
LETTERS     { LETTER }+
SPACES      [ \t ]
IDSTRING    { LETTER } ({ LETTERS }|{ DIGIT }+ | \_ )*
SUFFIX      ({ BOTH } | \, | \.) *
COMMENT     \ #.*
QUOTEID     " [ 0-9 a-z A-Z ]* "
QUOTESTR    " \ " ([ ^ \ " ] | \\\")*\"
COMMENT     "#.*
```

1.2 Keywords

CurrentClient	clientid	imports	no_forward	startegy
CurrentTime	consistency	insert	not	string
ResetClient	end	integer	objectbase	suchthat
ancestor	end_objectbase	link	or	superclass
and	exists	linkto	real	text
automation	exports	member	remove	time
binary	false	nil	return	true
boolean	forall	no_backward	rules	unlink
built_in_overload	hide	no_chain	set_of	user

1.3 Special Tokens

( ) { } [ ]

```
=       -- EQ_OP_tok     (EQ)
< >     -- EQ_OP_tok     (NEQ)
< =     -- EXP_OP_tok    (LEQ)
> =     -- EXP_OP_tok    (GEQ)
>       -- EXP_OP_tok    (GT)
<       -- EXP_OP_tok    (LT)
```

```
::      -- D_COLON
+=      -- MATH_OP_tok  (PLUS_EQ)
-=      -- MATH_OP_tok  (MINUS_EQ)
(*      -- OTHER_LEFT_KW    (NO_CHAIN)
*)      -- OTHER_RIGHT_KW  (NO_CHAIN)
```

1.4 Numbers and Identifiers

This section outlines some of the very low-level details that lex understands. It uses the standard notations of regular expressions. A character in quotes is a literal character, | represents options, + indicates one or more of the specified item, * represents 0 or more of the specified item. The ? represents an optional item. Items in (...) are groupings, while items in {...} indicate a user-defined character class (see 1.1).

```
"-"{DIGIT}+ | {DIGIT}+                          -- IVAL

"-"?{DIGIT}*"."({DIGIT}+) |
"-"?{DIGIT}*"."({DIGIT}+) (E|e)"-"?{DIGIT}+ -- RVAL

"?"{IDSTRING}                                   -- VARIABLE
"?"{IDSTRING}"."{IDSTRING}                      -- BVAR
"?"{IDSTRING}":"{IDSTRING}                      -- PARAM

{IDTSRING}("/"{IDSTRING})+                      -- PATH
{IDSTRING}                                      -- ID
{QUOTEID}                                       -- QUOTE_ID
{QUOTESTR}                                      -- QUOTE_STR
"{SUFFIX}"                                      -- FILE_NAME
```

2. The Productions

This section outlines all the productions of the grammar. Multiple entries denote alternative derivations of a non-terminal.

start	STRATEGY_KW ID imp_exp objbase rule_section oversection
imp_exp	IMPORTS_KW imp_name_list ; EXPORTS_KW exp_name_list ;
imp_name_list	**nothing**
	ID
	imp_name_list , ID
exp_name_list	**nothing**
	ID
	exp_nama_list , ID
objbase	**nothing**
	OBJECTBASE_KW classes ENDOBJECTBASE_KW
classes	class
	classes class

class	ID D_COLON superclasses attributes END_KW
superclasses	SUPERCLASS_KW ;
	SUPERCLASS_KW super_name_list ;
super_name_list	ID
	supername_list , ID
attributes	attrib
	attributes attrib
attrib	ID : noninitiable_type ;
	ID : autoinitiable_type ;
	ID : initiable_type ;
	ID : initiable_type EQ_OP_TOK init_val ;
autoinitiable_type	USER_KW
	TIME_KW
	CLIENTID_KW
initiable_type	STRING_KW
	INITEGER_KW
	REAL_KW
	BOOLEAN_KW
	file_type
	enumerated_type
noninitiable_type	ID
	SETOF_KW ID
	LINK_KW ID
	SETOF_KW LINK_KW ID
enumerated_type	(et_name_list)
file_type	TEXT_KW
	BINARY_KW
et_name_list	ID
	et_name_list , ID
init_val	ID
	FILE_NAME
	PATH
	QUOTE_STR_KW
	QUOT_ID_KW
	BOOL_VAL_TOK
	IVAL
	RVAL
rule_section	**nothing**
	RULES_KW rules
rules	rule
	rules rule
rule	ID [parameters] : bindings : precond activity mult_posts
	HIDE_KW ID [parameters]:bindings:precond activity multposts
parameters	**nothing**
	PARAM
	parameters , PARAM
bindings	**nothing**
	binding
	(BOOL_OP_TOK binding_list binding)

binding	(QUANTIFIER_TOK ID VARIABLE SUCHTHAT_KW binding_cond)
binding_expr_list	binding_cond
	binding_expr_list binding_cond
binding_cond	(set_expr)
	(expression)
	(multiple_bind_cond)
multiple_bind_cond	BOOL_OP_TOK binding_expr_list binding_cond
	NOT_TOK binding_cond
binding_list	binding
	binding_list binding
activity	{ }
	{ action }
action	ID ID act_var_list
outputs	**nothing**
	RETURN_KW OUT_VAR_LIST
out_var_item	VARIABLE
out_var_list	**nothing**
	OUT_VAR_LIST OUTBVAR_ITEM act_var_item
	QUOTE_ID
	QUOTE_STR_ID
	VARIABLE
act_var_list	**nothing**
	act_var_list act_var_item
mult_posts	;
	mult_post_list
mult_post_list	post ;
	mult_post_list post ;
post	allowed_post_cond
	(BOOL_OP_TOK post_list allowed_post_cond)
	post_list allowed_post_cond
allowed_post_cond	consistency_cond
	automation_cond
	both_cond
	other_cond_post
precond	**nothing**
	which_cond
which_cond	allowed_pre_cond
allowed_pre_cond	both_cond
	automation_cond
	other_cond_pre
	consistency_cond
allowed_list	which_cond
	allowed_list which_cond
expr_cond	BVAR MATH_OP_TOK operand
operand	BVAR
	QUOTE_ID
	QUOTE_STR
	RVAL
	IVAL

consistency_cond	[solo_cond]
	CONSISTENCY_KW (solo_cond)
automation_cond	(solo_cond)
	AUTOMATION_KW (solo_cond)
other_cond_pre	OTHER_LEFT_KW solo_cond OTHER_RIGHT_KW
	NO_BACKWARD_KW (solo_cond)
other_cond_post	OTHER_LEFT_KW solo_post OTHER_RIGHT_KW
	NO_FORWARD_KW (solo_post)
	NO_BACKWARD_KW (solo_post)
	NO_BACKWARD_KW [solo_post]
	NO_CHAIN_KW (solo_post)
solo_cond	expressions
	expr_cond
solo_post	POST_EXPR
	POST_LINK_EXPR
	POST_UNLINK_EXPR
multiple_cond	BOOL_OP_TOK allowed_list which_cond
	NOT_KW which_cond
expression	BVAR exp_op expression_tail
	BVAR EQ_OP_TOK BOOL_VAL_TOK
	BVAR exp_op IVAL
	BVAR exp_op RVAL
post_expr	BVAR EQ_OP_TOK BOOL_VAL_TOK
	BVAR EQ_OP_TOK EXPRESSION_TAIL
expression_tail	BVAR
	ID
	QUOTE_ID
	QUOTE_STR
set_expr	MEMBER_KW [BVAR VARIABLE]
	ANCESTOR_KW [VARIABLE VARIABLE]
	LINK_TO_KW [BVAR VARIABLE]
	LINK_TO_KW [BVAR NIL_TOK]
exp_op	EQ_OP_TOK
	EX_OP_TOK

APPENDIX B : Data Model

strategy datamodel

```
imports none;
exports all;

objectbase

TOP :: superclass ENTITY;
   top_name : string;
   projects : PROJECT;
   team : TEAM;
end

PROJECT :: superclass ENTITY;
   proj_name : string;
   status : ( NotActive , Active )= NotActive ;
   specification : SPEC;
   log_architecture : LOGARCH;
   phys_architecture : PHYSARCH;
   documentation : DOC;
end

######################## TEAM  MANAGEMENT ########################

TEAM :: superclass ENTITY;
   team_name : string;
   chef : MANAGER;
   members : set_of ENGINEER;
end

MANAGER :: superclass ENTITY;
   login : user;
   eng_name : string;
   jobs : set_of TASK;
   rev_jobs : set_of REV_TASK;
end

ENGINEER :: superclass ENTITY;
   login : user;
   eng_name : string;
   jobs : set_of TASK;
   rev_jobs : set_of REV_TASK;
end

TASK :: superclass ENTITY;
   status : ( Assigned, NotAssigned )= NotAssigned;
   beg_date : integer;
   end_date : integer;
   comments : text;
end

REV_TASK :: superclass ENTITY;
   status : ( Assigned, NotAssigned )= NotAssigned;
end

############################ SUPERCLASSES ########################
```

```
MODULE :: superclass ENTITY;
   m_name : string = "noname";
   engineer : string = "none";
   reviewer : string = "none";
   status : (Initialized, Assigned, Active, Done, Reviewd, Maint, Ready )= Initialized;
   feedback : text;
   pro_eng : link TASK;
   rev_eng : link REV_TASK;
   doc : link DOC;
end

CONTENTS :: superclass MODULE;
  contents : text;
end

############################ SPECIFICATION ######################

SPEC :: superclass ENTITY;
   spec_name : string;
   db_primitives : STATE;
   config_changes : C_STATE;
   functions : set_of FUNCTION;
   status : ( Initialized, Active, Done ) = Initialized;
end

FUNCTION :: superclass MODULE,CONTENTS;
   log_repres : link L_FUNC;
   pro_eng : link TASK;
   rev_eng : link REV_TASK;
   doc : link DOC;
end;

STATE :: superclass MODULE,CONTENTS;
   split_into :set_of link  L_STATE;
   pro_eng : link TASK;
   rev_eng : link REV_TASK;
   doc : link DOC;
end

######################## LOGICAL ARCHITECTURE ####################

LOGARCH :: superclass ENTITY;
   log_name : string;
   sap : DESIGN;
   interface : set_of L_SCREEN;
   functions : set_of L_FUNC;
   state : set_of L_STATE;
   status : ( Initialized, Active, Done )= Initialized;
   contents : text;
end

L_STATE :: superclass MODULE,CONTENTS;
   level : integer=3;
   sub_cat : set_of L_STATE;
   coded_in : set_of link MOD_PROG;
   pro_eng : link TASK;
   rev_eng : link REV_TASK;
   doc : link DOC;
end
```

```
L_FUNC :: superclass MODULE,CONTENTS;
   level : integer;
   sub_lfunc : set_of L_FUNC;
   uses_state : set_of link L_STATE;
   uses_scr : set_of link L_SCREEN;
   coded_in : set_of link MOD_PROG;
   pro_eng : link TASK;
   rev_eng : link REV_TASK;
   doc : link DOC;
end

L_SCREEN :: superclass MODULE,CONTENTS;
    level : integer=4;
    sub_lscreen : set_of L_SCREEN;
    uses_lstate : set_of link L_STATE;
    contents : text = ".scr";
    code : link MOD_SCR;
    feedback : text;
    pro_eng : link TASK;
    rev_eng : link REV_TASK;
    doc : link DOC;
end

DESIGN :: superclass MODULE,CONTENTS.
    level : integer=3;
    contents : text = ".dsg";
    pro_eng : link TASK;
    rev_eng : link REV_TASK;
    doc : link DOC;
    sap : link PHYS_DESIGN;
end

###################### PHYSICAL ARCHITECTURE ######################

PHYSARCH :: superclass ENTITY;
   san : PHYS_DESIGN;
   screens : set_of MOD_SCR;
   db : PHYS_DB;
   code : MOD_PROG;
   phy_name : string;
   status : ( Initialized, Active, Done )= Initialized;
end

MOD_PROG :: superclass MODULE;
  source : text;
  obj_code : binary;
  comp_res : text;
  comp_status : (Initialized , NotCompiled , Compiled , NotBuilt, Built)= Initialized;
  subroutines : set_of MOD_PROG;
  screens : set_of link MOD_SCR;
  pro_eng : link TASK;
  rev_eng : link REV_TASK;
  doc : link DOC;
end

PHYS_DESIGN :: superclass MODULE,CONTENTS;
  contents : text = ".dsg";
  pro_eng : link TASK;
  rev_eng : link REV_TASK;
  doc : link DOC;
```

```
  db_language : SQL;
end

SQL :: superclass MODULE,CONTENTS;
  doc : link DOC;
end

DOC :: superclass  CONTENTS;
  doc_name :string;
  module : string;
  author : string;
  docs : set_of DOC;
end

MOD_SCR :: superclass MODULE;
   source : text;
   obj_code : binary;
   comp_res : text;
   comp_status : (Initialized , NotCompiled , Compiled) = Initialized;
   level : integer;
   pro_eng : link TASK;
   rev_eng : link REV_TASK;
   doc : link DOC;
end

PHYS_DB :: superclass MODULE,CONTENTS;
   pro_eng : link TASK;
   rev_eng : link REV_TASK;
   doc : link DOC;
end
end_objectbase
```

APPENDIX C : Process Model

strategy achieve

```
imports datamodel;
exports all;

rules

achieve [ ?st:STATE ]:
  ( and ( exists SPEC ?spec suchthat ( member [ ?spec.db_primitives ?st ]))
        ( forall FUNCTION ?func suchthat ( member [ ?spec.functions ?func ]))):
  ( and no_chain ( ?func.status = Ready )
        no_backward ( ?st.status = Active ))
  {}
  [ ?st.status = Done ];

achieve [ ?sql:SQL ]:
  ( exists PHYS_DESIGN ?dsg suchthat ( member [ ?dsg.db_language ?sql ])):
  ( and no_chain ( ?dsg.status = Ready )
        no_backward ( ?sql.status = Active ))
  {}
  ( ?sql.status = Done );

achieve [ ?mod:MOD_PROG ]:
:
( and no_backward ( ?mod.status = Active )
      no_backward ( ?mod.comp_status = Compiled ))
{}
no_backward ( ?mod.status = Done );

achieve [ ?scr:MOD_SCR ]:
:
( and no_backward ( ?scr.status = Active )
      no_backward ( ?scr.comp_status = Compiled ))
{}
no_backward ( ?scr.status = Done );

achieve1 [ ?mod:MOD_PROG ]:
:
( and no_chain ( ?mod.status = Maint )
      (or ( ?mod.comp_status = Compiled )
          ( ?mod.comp_status = Built )))
{}
no_chain ( ?mod.status = Ready );


achieve1 [ ?scr:MOD_SCR ]:
:
( and  no_chain ( ?scr.status = Maint )
       ( ?scr.comp_status = Compiled ))
{}
( ?scr.status = Ready );

#####################################################################
```

strategy activate

```
imports datamod1;
exports all;
```

```
objectbase

INTERACTIVE :: superclass TOOL;
   ini : string = ini;
end

end_objectbase

rules

initiate [?p:PROJECT]:
 :
 ( ?p.status = NotActive )
{ INTERACTIVE ini }
 ( ?p.status = Active );

hide activate [ ?spec:SPEC ]:
 ( exists PROJECT ?proj suchthat ( member [ ?proj.specification ?spec ])):
 ( and no_backward ( ?proj.status = Active )
              ( ?spec.status = Initialized ))
 {}
 no_chain (?spec.status = Active );

hide activate [ ?log:LOGARCH ]:
 ( and ( exists PROJECT ?proj suchthat ( member [ ?proj.log_architecture ?log ]))
     ( exists SPEC ?spec suchthat ( member [ ?proj.specification ?spec ]))):
 ( ?spec.status = Done )
 {}
 no_chain (?log.status = Active );

hide activate [ ?phy:PHYSARCH ]:
 ( and ( exists PROJECT ?proj suchthat ( member [ ?proj.phys_architecture ?phy ]))
     ( exists LOGARCH ?log suchthat ( member [ ?proj.log_architecture ?log ]))):
 ( ?log.status = Done )
 {}
 no_chain (?phy.status = Active );

######################################################################
```

strategy assign

```
imports datamodel;
exports all;

objectbase

INTERACTIVE :: superclass TOOL;
 schedule : string = schedule;
end

end_objectbase

rules

assign [ ?st:STATE , ?ta:TASK ]:
 ( and ( exists SPEC ?spec suchthat ( member [ ?spec.db_primitives ?st ]))
     ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
 ( and no_chain ( ?ta.status = NotAssigned )
     no_chain ( ?spec.status = Active )
     no_chain ( ?st.status = Initialized))
```

```
 { INTERACTIVE schedule ?ta.comments ?pg.eng_name  return  ?beg ?end ?name }
 ( and ( ?st.status = Assigned )
       ( ?ta.status = Assigned )
       ( ?st.engineer = ?pg.eng_name )
       ( ?st.m_name = ?name )
       ( ?ta.beg_date = ?beg )
       ( ?ta.end_date = ?end )
       ( linkto [ ?st.pro_eng ?ta ]));

assign [ ?func:FUNCTION , ?ta:TASK ]:
 ( and ( exists SPEC ?spec suchthat ( member [ ?spec.functions  ?func ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
 ( and ( ?ta.status = NotAssigned )
       ( ?spec.status = Active )
       ( ?func.status = Initialized))
 { INTERACTIVE schedule  ?ta.comments ?pg.eng_name  return  ?beg ?end ?name }
 ( and ( ?func.status = Assigned )
       ( ?ta.status = Assigned )
       ( ?func.engineer = ?pg.eng_name )
       ( ?func.m_name = ?name )
       ( ?ta.beg_date = ?beg )
       ( ?ta.end_date = ?end )
       ( linkto [ ?func.pro_eng ?ta ]));

assign [ ?dsg:DESIGN , ?ta:TASK ]:
 ( and ( exists LOGARCH ?log  suchthat ( member [ ?log.sap ?dsg ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
 ( and no_chain ( ?ta.status = NotAssigned )
       no_chain ( ?log.status = Active )
       no_chain ( ?dsg.status = Initialized ))
 { INTERACTIVE schedule  ?ta.comments ?pg.eng_name return  ?beg ?end ?name }
 ( and ( ?dsg.status = Assigned )
       ( ?ta.status = Assigned )
       ( ?dsg.engineer = ?pg.eng_name )
       ( ?dsg.m_name = ?name )
       ( ?ta.beg_date = ?beg )
       ( ?ta.end_date = ?end )
       ( linkto [ ?dsg.pro_eng ?ta ]));

assign [ ?lst:L_STATE , ?ta:TASK ]:
 ( and ( exists LOGARCH ?log  suchthat ( ancestor [ ?log ?lst ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
 ( and no_chain ( ?ta.status = NotAssigned )
       no_chain( ?log.status = Active )
       no_chain ( ?lst.status = Initialized))
 { INTERACTIVE schedule ?ta.comments ?pg.eng_name return  ?beg ?end ?name }
 ( and ( ?lst.status = Assigned )
       ( ?ta.status = Assigned )
       ( ?lst.engineer = ?pg.eng_name )
       ( ?lst.m_name  = ?name )
       ( ?ta.beg_date = ?beg )
       ( ?ta.end_date = ?end )
       ( linkto [ ?lst.pro_eng ?ta ]));

assign [ ?lfunc:L_FUNC , ?ta:TASK ]:
 ( and ( exists LOGARCH ?log  suchthat ( member [ ?log.functions ?lfunc ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
 ( and no_chain ( ?ta.status = NotAssigned )
       no_chain ( ?log.status = Active )
       no_chain ( ?lfunc.status = Initialized))
 { INTERACTIVE schedule  ?ta.comments ?pg.eng_name return  ?beg ?end ?name }
```

```
 ( and ( ?lfunc.status = Assigned )
       ( ?ta.status = Assigned )
       ( ?lfunc.engineer = ?pg.eng_name )
       ( ?lfunc.m_name = ?name )
       ( ?ta.beg_date = ?beg )
       ( ?ta.end_date = ?end )
       ( linkto [ ?lfunc.pro_eng ?ta ]));

assign [ ?lscr:L_SCREEN , ?ta:TASK ]:
 ( and ( exists LOGARCH ?log  suchthat ( member [ ?log.interface ?lscr ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
 ( and no_chain ( ?ta.status = NotAssigned )
       no_chain ( ?log.status = Active )
       no_chain ( ?lscr.status = Initialized))
 { INTERACTIVE schedule ?ta.comments ?pg.eng_name return  ?beg ?end ?name }
 ( and ( ?lscr.status = Assigned )
       ( ?ta.status = Assigned )
       ( ?lscr.engineer = ?pg.eng_name )
       ( ?lscr.m_name = ?name )
       ( ?ta.beg_date = ?beg )
       ( ?ta.end_date = ?end )
       ( linkto [ ?lscr.pro_eng ?ta ]));

assign [ ?dsg:PHYS_DESIGN , ?ta:TASK ]:
 ( and ( exists PHYSARCH ?phy  suchthat ( member [ ?phy.san ?dsg ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))
       ( exists SQL ?sql suchthat ( member [ ?dsg.db_language ?sql ]))):
 ( and no_chain ( ?ta.status = NotAssigned )
       no_chain ( ?phy.status = Active )
       no_chain ( ?dsg.status = Initialized ))
 { INTERACTIVE schedule ?ta.comments ?pg.eng_name return  ?beg ?end ?name}
 ( and ( ?dsg.status = Assigned )
       ( ?ta.status = Assigned )
       ( ?dsg.engineer = ?pg.eng_name )
       ( ?sql.engineer = ?pg.eng_name )
       ( ?dsg.m_name = ?name )
       ( ?ta.beg_date = ?beg )
       ( ?ta.end_date = ?end )
       ( linkto [ ?dsg.pro_eng ?ta ]));

assign [ ?mod:MOD_PROG , ?ta:TASK ]:
 ( and ( exists PHYSARCH ?phy  suchthat ( ancestor [ ?phy ?mod ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
 ( and no_chain ( ?ta.status = NotAssigned )
       no_chain ( ?phy.status = Active )
       no_chain ( ?mod.status = Initialized ))
 { INTERACTIVE schedule ?ta.comments ?pg.eng_name return  ?beg ?end ?name }
 ( and ( ?mod.status = Assigned )
       ( ?ta.status = Assigned )
       ( ?mod.engineer = ?pg.eng_name )
       ( ?mod.m_name = ?name )
       ( ?ta.beg_date = ?beg )
       ( ?ta.end_date = ?end )
       ( linkto [ ?mod.pro_eng ?ta ]));

assign [ ?db:PHYS_DB , ?ta:TASK ]:
 ( and ( exists PHYSARCH ?phy  suchthat ( member [ ?phy.db ?db ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
 ( and no_chain ( ?ta.status = NotAssigned )
       no_chain ( ?phy.status = Active )
       no_chain ( ?db.status = Initialized ))
```

```
 { INTERACTIVE schedule ?ta.comments ?pg.eng_name  return  ?beg ?end ?name }
 ( and ( ?db.status = Assigned )
       ( ?ta.status = Assigned )
       ( ?db.engineer = ?pg.eng_name )
       ( ?db.m_name = ?name )
       ( ?ta.beg_date = ?beg )
       ( ?ta.end_date = ?end )
       ( linkto [ ?db.pro_eng ?ta ]));

assign [ ?scr:MOD_SCR , ?ta:TASK ]:
 ( and ( exists PHYSARCH ?phy  suchthat ( member [ ?phy.screens ?scr ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
 ( and no_chain ( ?ta.status = NotAssigned )
       no_chain ( ?phy.status = Active )
       no_chain ( ?scr.status = Initialized ))
 { INTERACTIVE schedule ?ta.comments ?pg.eng_name  return  ?beg ?end ?name }
 ( and ( ?scr.status = Assigned )
       ( ?ta.status = Assigned )
       ( ?scr.engineer = ?pg.eng_name )
       ( ?scr.m_name = ?name )
       ( ?ta.beg_date = ?beg )
       ( ?ta.end_date = ?end )
       ( linkto [ ?scr.pro_eng ?ta ]));

################################################################
```

strategy assign_rev

```
imports datamodel;
exports all;

rules

assign_rev [ ?dsg:PHYS_DESIGN , ?ta:REV_TASK ]:
 ( and ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta ]))
       ( exists SQL ?sql suchthat ( member [ ?dsg.db_language ?sql ]))):
 ( and ( or no_chain ( ?dsg.status = Assigned )
            no_chain ( ?dsg.status = Active )
            no_chain ( ?dsg.status = Done ))
       ( ?pg.eng_name < > ?dsg.engineer )
       ( ?ta.status = NotAssigned ))
 { }
 ( and no_chain ( ?ta.status = Assigned )
       no_chain ( ?dsg.reviewer = ?pg.eng_name )
       no_chain ( ?sql.reviewer = ?pg.eng_name )
             ( linkto [ ?dsg.rev_eng ?ta ]));

assign_rev [ ?mod:MOD_PROG , ?ta:REV_TASK ]:
 ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta ])):
 ( and ( or no_chain ( ?mod.status = Assigned )
            no_chain ( ?mod.status = Active )
            no_chain ( ?mod.status = Done ))
       ( ?pg.eng_name < > ?mod.engineer )
       ( ?ta.status = NotAssigned ))
{ }
 ( and no_chain ( ?ta.status = Assigned )
       no_chain ( ?mod.reviewer = ?pg.eng_name )
             ( linkto [ ?mod.rev_eng ?ta ]));

assign_rev [ ?db:PHYS_DB , ?ta:REV_TASK ]:
```

```
 ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta ])):
 ( and ( or no_chain ( ?db.status = Assigned )
           no_chain ( ?db.status = Active )
           no_chain ( ?db.status = Done ))
       ( ?pg.eng_name < > ?db.engineer )
       ( ?ta.status = NotAssigned ))
 { }
 ( and no_chain ( ?ta.status = Assigned )
       no_chain ( ?db.reviewer = ?pg.eng_name )
             ( linkto [ ?db.rev_eng ?ta ]));

assign_rev [ ?scr:MOD_SCR , ?ta:REV_TASK ]:
 ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta ])):
 ( and ( or no_chain ( ?scr.status = Assigned )
           no_chain ( ?scr.status = Active )
           no_chain ( ?scr.status = Done ))
       ( ?pg.eng_name < > ?scr.engineer )
       ( ?ta.status = NotAssigned ))
 { }
 ( and no_chain ( ?ta.status = Assigned )
       no_chain ( ?scr.reviewer = ?pg.eng_name )
             ( linkto [ ?scr.rev_eng ?ta ]));

assign_rev [ ?dsg:DESIGN , ?ta:REV_TASK ]:
 ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta ])):
 ( and ( or no_chain ( ?dsg.status = Assigned )
           no_chain ( ?dsg.status = Active )
           no_chain ( ?dsg.status = Done ))
       ( ?pg.eng_name < > ?dsg.engineer )
       ( ?ta.status = NotAssigned ))
 { }
 ( and no_chain ( ?ta.status = Assigned )
       no_chain ( ?dsg.reviewer = ?pg.eng_name )
             ( linkto [ ?dsg.rev_eng ?ta ]));

assign_rev [ ?lst:L_STATE , ?ta:REV_TASK ]:
 ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta ])):
 ( and ( or no_chain ( ?lst.status = Assigned )
           no_chain ( ?lst.status = Active )
           no_chain ( ?lst.status = Done ))
       ( ?pg.eng_name < > ?lst.engineer )
       ( ?ta.status = NotAssigned ))
{ }
 ( and no_chain ( ?ta.status = Assigned )
       no_chain ( ?lst.reviewer = ?eng.name )
             ( linkto [ ?lst.rev_eng ?ta ]));

assign_rev [ ?lfunc:L_FUNC , ?ta:REV_TASK ]:
 ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta ])):
 ( and ( or no_chain ( ?lfunc.status = Assigned )
           no_chain ( ?lfunc.status = Active )
           no_chain ( ?lfunc.status = Done ))
       ( ?pg.eng_name < > ?lfunc.engineer )
       ( ?ta.status = NotAssigned ))
 { }
 ( and no_chain ( ?ta.status = Assigned )
       no_chain ( ?lfunc.reviewer = ?pg.eng_name )
             ( linkto [ ?lfunc.rev_eng ?ta ]));
assign_rev [ ?lscr:L_SCREEN , ?ta:REV_TASK ]:
 ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta ])):
 ( and ( or no_chain ( ?lscr.status = Assigned )
```

```
          no_chain ( ?lscr.status = Active )
          no_chain ( ?lscr.status = Done ))
      ( ?pg.eng_name < > ?lscr.engineer )
      ( ?ta.status = NotAssigned ))
{ }
 ( and no_chain ( ?ta.status = Assigned )
      no_chain ( ?lscr.reviewer = ?pg.eng_name )
            ( linkto [ ?lscr.rev_eng ?ta ]));

assign_rev [ ?st:STATE , ?ta:REV_TASK ]:
 ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta ])):
 ( and ( or no_chain ( ?st.status = Assigned )
          no_chain ( ?st.status = Active )
          no_chain ( ?st.status = Done ))
      ( ?pg.eng_name < > ?st.engineer )
      ( ?ta.status = NotAssigned ))
 { }
 ( and no_chain ( ?ta.status = Assigned )
      no_chain ( ?st.reviewer = ?pg.eng_name )
      ( linkto [ ?st.rev_eng ?ta ]));

assign_rev [ ?func:FUNCTION , ?ta:REV_TASK ]:
 ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta ])):
 ( and ( or no_chain ( ?func.status = Assigned )
          no_chain ( ?func.status = Active )
          no_chain ( ?func.status = Done ))
      ( ?pg.eng_name < > ?func.engineer )
      ( ?ta.status = NotAssigned ))
{ }
( and no_chain ( ?ta.status = Assigned )
      no_chain ( ?func.reviewer = ?pg.eng_name )
      ( linkto [ ?func.rev_eng ?ta ]));

#####################################################################
```

strategy compiler

```
imports datamodel;
exports all;

objectbase

COMPILER :: superclass TOOL;
   compile : string = compile;
   scrcompile : string = scrcompile;
   build : string = build;
end

VIEWER :: superclass TOOL;
   viewerr : string = viewerr;
end

RUNNER :: superclass TOOL;
   exec : string = exec;
end

end_objectbase

rules

compile [ ?mod:MOD_PROG ]:
```

```
 ( forall MOD_PROG ?mod1 suchthat ( member [?mod.subroutines  ?mod1])):
 ( and no_backward ( ?mod1.comp_status = Compiled )
              ( ?mod.comp_status = NotCompiled))
 { COMPILER  compile ?mod.source ?mod.obj_code ?mod.comp_res ?mod1.source }
 ( ?mod.comp_status = Compiled );
  no_forward ( ?mod.comp_status = NotCompiled );

compile [ ?scr:MOD_SCR ]:
  :
  ( ?scr.comp_status = NotCompiled )
  { COMPILER scrcompile ?scr.source ?scr.obj_code ?scr.comp_res }
  ( ?scr.comp_status = Compiled );
  no_forward ( ?scr.comp_status = NotCompiled );

build [ ?mod:MOD_PROG ]:
 ( and ( forall MOD_PROG ?mod1 suchthat ( member [ ?mod.subroutines  ?mod1]))
      ( forall MOD_SCR ?scr suchthat ( linkto [ ?mod1.screens ?scr ]))
      ( forall MOD_SCR ?scr1 suchthat ( linkto [ ?mod.screens ?scr1 ]))):
 ( and ( or ( ?mod1.comp_status = Compiled )
          ( ?mod1.comp_status = Built ))
      ( ?scr.comp_status = Compiled )
      ( ?scr1.comp_status = Compiled )
      ( or ( ?mod.comp_status = NotBuilt )
          ( ?mod.comp_status = Compiled )))
 { COMPILER build ?mod.source ?mod.obj_code ?mod1.obj_code ?scr.obj_code ?scr1.obj_code
?mod.comp_res}
 ( ?mod.comp_status = Built );
 ( ?mod.comp_status = NotBuilt );

viewerr [ ?mod:MOD_PROG ]:
 :
 { VIEWER viewerr ?mod.comp_res }
;

viewerr [ ?scr:MOD_SCR ]:
 :
 { VIEWER viewerr ?scr.comp_res }
;

exec [ ?mod:MOD_PROG ]:
:
 no_forward ( ?mod.comp_status = Built )
 { RUNNER exec ?mod.obj_code }
 ;

#####################################################################
```

strategy control

```
imports datamodel;
exports all;

objectbase

EDITOR :: superclass TOOL;
   review : string = review;
end

end_objectbase

rules
```

```
control [ ?st:STATE ]:
 ( and ( exists TASK ?ta1 suchthat ( linkto [ ?st.pro_eng ?ta1 ] ))
       ( exists REV_TASK ?ta2 suchthat ( linkto [ ?st.rev_eng ?ta2 ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta2 ]))):
 ( and no_chain ( ?st.status = Done )
       ( ?pg.login = CurrentUser ))
 { EDITOR  review  ?st.contents ?st.feedback ?st.engineer }
 no_forward ( ?st.status = Reviewd );
 ( and ( ?st.status = Ready )
       no_chain ( ?ta1.status = NotAssigned )
       no_chain ( ?ta2.status = NotAssigned )
       ( unlink [ ?st.pro_eng ?ta1 ])
       ( unlink [ ?st.rev_eng ?ta2 ]));

control [ ?func:FUNCTION ]:
 ( and ( exists TASK ?ta1 suchthat ( linkto [ ?func.pro_eng ?ta1 ] ))
       ( exists REV_TASK ?ta2 suchthat ( linkto [ ?func.rev_eng ?ta2 ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta2 ]))):
  ( and no_chain ( ?func.status = Done )
       ( ?pg.login = CurrentUser ))
 { EDITOR  review ?func.contents ?func.feedback ?func.engineer }
 no_forward ( ?func.status = Reviewd );
 ( and ( ?func.status = Ready )
       no_chain( ?ta1.status = NotAssigned )
       no_chain ( ?ta2.status = NotAssigned )
       ( unlink [ ?func.pro_eng ?ta1 ])
       ( unlink [ ?func.rev_eng ?ta2 ]));

control [ ?dsg:DESIGN ]:
 ( and ( exists TASK ?ta1 suchthat ( linkto [ ?dsg.pro_eng ?ta1 ] ))
       ( exists REV_TASK ?ta2 suchthat ( linkto [ ?dsg.rev_eng ?ta2 ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta2 ]))):
  ( and no_chain ( ?dsg.status = Done )
       ( ?pg.login = CurrentUser ))
 { EDITOR  review  ?dsg.contents ?dsg.feedback ?dsg.engineer }
 no_forward ( ?dsg.status = Reviewd );
 ( and no_forward ( ?dsg.status = Ready )
       no_chain ( ?ta1.status = NotAssigned )
       no_chain ( ?ta2.status = NotAssigned )
       ( unlink [ ?dsg.pro_eng ?ta1 ])
       ( unlink [ ?dsg.rev_eng ?ta2 ]));

control [ ?lst:L_STATE ]:
 ( and ( exists TASK ?ta1 suchthat ( linkto [ ?lst.pro_eng ?ta1 ] ))
       ( exists REV_TASK ?ta2 suchthat ( linkto [ ?lst.rev_eng ?ta2 ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta2 ]))):
  ( and no_chain ( ?lst.status = Done )
       ( ?pg.login = CurrentUser ))
 { EDITOR  review  ?lst.contents ?lst.feedback ?lst.engineer }
 no_forward ( ?lst.status = Reviewd );
 ( and ( ?lst.status = Ready )
       no_chain ( ?ta1.status = NotAssigned )
       no_chain ( ?ta2.status = NotAssigned )
       ( unlink [ ?lst.pro_eng ?ta1 ])
       ( unlink [ ?lst.rev_eng ?ta2 ]));

control [ ?lfunc:L_FUNC ]:
 ( and ( exists TASK ?ta1 suchthat ( linkto [ ?lfunc.pro_eng ?ta1 ] ))
       ( exists REV_TASK ?ta2 suchthat ( linkto [ ?lfunc.rev_eng ?ta2 ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta2 ]))):
```

```
  ( and no_chain ( ?lfunc.status = Done )
       ( ?pg.login = CurrentUser ))
  { EDITOR  review  ?lfunc.contents ?lfunc.feedback ?lfunc.engineer }
  no_forward ( ?lfunc.status = Reviewd );
  ( and ( ?lfunc.status = Ready )
       no_chain ( ?ta1.status = NotAssigned )
       no_chain ( ?ta2.status = NotAssigned )
       ( unlink [ ?lfunc.pro_eng ?ta1 ])
       ( unlink [ ?lfunc.rev_eng ?ta2 ]));

control [ ?lscr:L_SCREEN ]:
  ( and ( exists TASK ?ta1 suchthat ( linkto [ ?lscr.pro_eng ?ta1 ] ))
       ( exists REV_TASK ?ta2 suchthat ( linkto [ ?lscr.rev_eng ?ta2 ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta2 ]))):
  ( and no_chain ( ?lscr.status = Done )
       ( ?pg.login = CurrentUser ))
  { EDITOR  review  ?lscr.contents ?lscr.feedback ?lscr.engineer }
  no_forward ( ?lscr.status = Reviewd );
  ( and ( ?lscr.status = Ready )
       no_chain ( ?ta1.status = NotAssigned )
       no_chain ( ?ta2.status = NotAssigned )
       ( unlink [ ?lscr.pro_eng ?ta1 ])
       ( unlink [ ?lscr.rev_eng ?ta2 ]));

control [ ?dsg:PHYS_DESIGN ]:
  ( and ( exists TASK ?ta1 suchthat ( linkto [ ?dsg.pro_eng ?ta1 ] ))
       ( exists REV_TASK ?ta2 suchthat ( linkto [ ?dsg.rev_eng ?ta2 ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta2 ]))):
  ( and no_chain ( ?dsg.status = Done )
       ( ?pg.login = CurrentUser ))
  { EDITOR  review  ?dsg.contents ?dsg.feedback ?dsg.engineer }
  no_forward ( ?dsg.status = Reviewd );
  ( ?dsg.status = Ready );

control [ ?sql:SQL ]:
  ( and ( exists PHYS_DESIGN ?dsg suchthat ( member [ ?dsg.db_language ?sql ]))
       ( exists TASK ?ta1 suchthat ( linkto [ ?dsg.pro_eng ?ta1 ] ))
       ( exists REV_TASK ?ta2 suchthat ( linkto [ ?dsg.rev_eng ?ta2 ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta2 ]))):
  ( and no_chain ( ?dsg.status = Ready )
       no_chain ( ?sql.status = Done )
       ( ?pg.login = CurrentUser ))
  { EDITOR  review  ?sql.contents ?sql.feedback ?dsg.engineer }
  no_forward ( ?sql.status = Reviewd );
  ( and ( ?sql.status = Ready )
       no_chain ( ?ta1.status = NotAssigned )
       no_chain ( ?ta2.status = NotAssigned )
       ( unlink [ ?dsg.pro_eng ?ta1 ])
       ( unlink [ ?dsg.rev_eng ?ta2 ]));

control [ ?db:PHYS_DB ]:
  ( and ( exists TASK ?ta1 suchthat ( linkto [ ?db.pro_eng ?ta1 ] ))
       ( exists REV_TASK ?ta2 suchthat ( linkto [ ?db.rev_eng ?ta2 ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta2 ]))):
  ( and no_chain ( ?db.status = Done )
       ( ?pg.login = CurrentUser ))
  { EDITOR  review  ?db.contents ?db.feedback ?db.engineer }
  no_forward ( ?db.status = Reviewd );
  ( and  ( ?db.status = Ready )
       no_chain ( ?ta1.status = NotAssigned )
       no_chain ( ?ta2.status = NotAssigned )
```

```
      ( unlink [ ?db.pro_eng ?ta1 ])
      ( unlink [ ?db.rev_eng ?ta2 ]));

control [ ?mod:MOD_PROG ]:
 ( and ( exists TASK ?ta1 suchthat ( linkto [ ?mod.pro_eng ?ta1 ] ))
       ( exists REV_TASK ?ta2 suchthat ( linkto [ ?mod.rev_eng ?ta2 ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta2 ]))):
  ( and no_chain ( ?mod.status = Done )
        ( ?pg.login = CurrentUser ))
 { EDITOR  review  ?mod.source ?mod.feedback ?mod.engineer }
 no_forward ( ?mod.status = Reviewd );
( and
      ( ?mod.status = Ready )
      no_chain ( ?ta1.status = NotAssigned )
      no_chain ( ?ta2.status = NotAssigned )
      ( unlink [ ?mod.pro_eng ?ta1 ])
      ( unlink [ ?mod.rev_eng ?ta2 ]));

control [ ?scr:MOD_SCR ]:
 ( and ( exists TASK ?ta1 suchthat ( linkto [ ?scr.pro_eng ?ta1 ] ))
       ( exists REV_TASK ?ta2 suchthat ( linkto [ ?scr.rev_eng ?ta2 ]))
       ( exists ENGINEER ?pg suchthat ( member [ ?pg.rev_jobs ?ta2 ]))):
  ( and no_chain ( ?scr.status = Done )
        ( ?pg.login = CurrentUser ))
 { EDITOR  review  ?scr.source ?scr.feedback ?scr.engineer }
 no_forward ( ?scr.status = Reviewd );
 ( and
      ( ?scr.status = Ready )
      no_chain ( ?ta1.status = NotAssigned )
      no_chain ( ?ta2.status = NotAssigned )
      ( unlink [ ?scr.pro_eng ?ta1 ])
      ( unlink [ ?scr.rev_eng ?ta2 ]));

#####################################################################
```

strategy doc

```
imports datamodel;
exports all;

objectbase

EDITOR :: superclass TOOL;
   edit_doc : string = edit_doc;
end

end_objectbase

rules

doc [ ?doc:DOC ]:
 :
 { EDITOR edit_doc ?doc.contents return ?name ?module }
( and ( ?doc.doc_name = ?name )
    ( ?doc.author = CurrentUser )
    ( ?doc.module = ?module ));

#####################################################################
```

strategy finish

```
imports datamodel;
exports all;

rules

finish [ ?spec:SPEC]:
 ( and ( forall FUNCTION ?func suchthat ( member [ ?spec.functions ?func ]))
     ( forall STATE ?st suchthat ( member [ ?spec.db_primitives ?st ]))):
 ( and no_backward ( ?func.status = Ready )
     no_backward ( ?st.status = Ready )
     no_backward ( ?cst.status = Ready ))
 {}
 ( ?spec.status = Done );

finish [ ?log:LOGARCH ]:
 ( and ( forall L_FUNC ?lfunc suchthat ( member [ ?log.functions ?lfunc ]))
     ( forall L_STATE ?lst suchthat ( member [ ?log.state ?lst ]))
     ( forall DESIGN ?dsg  suchthat ( member [ ?log.sap ?dsg ]))
     ( forall L_SCREEN ?lscr suchthat ( member [ ?log.interface ?lscr ]))):
 ( and no_backward ( ?lfunc.status = Ready )
     no_backward ( ?lst.status = Ready )
     no_backward ( ?dsg.status = Ready )
     no_backward ( ?lscr.status = Ready ))
 {}
 ( ?log.status = Done );

finish [ ?phy:PHYSARCH ]:
 ( and ( forall PHYS_DESIGN ?dsg suchthat ( member [ ?phy.san ?dsg ]))
     ( forall PHYS_DB ?db suchthat ( member [ ?phy.db ?db ]))
     ( forall MOD_PROG ?mod suchthat ( ancestor [ ?phy ?mod ]))
     ( forall MOD_SCR ?scr suchthat ( member [ ?phy.screens ?scr]))
     ( forall SQL ?sql suchthat ( ancestor [ ?phy ?sql ]))):
 ( and no_backward ( ?dsg.status = Ready )
     no_backward ( ?db.status  = Ready )
     no_backward ( ?mod.status = Ready )
     no_backward ( ?scr.status = Ready )
     no_backward ( ?sql.status = Ready ))
{}
 ( ?phy.status = Done );

finish [ ?lst:L_STATE ]:
 ( forall L_STATE ?lst1 suchthat ( member [ ?lst.sub_cat ?lst1 ])):
 ( and no_backward ( ?lst1.status = Ready )
           ( ?lst.status = Initialized ))
 {}
 no_forward ( ?lst.status = Ready );

#####################################################################
```

strategy mail

```
imports datamodel;
exports none;

objectbase

MAILER :: superclass TOOL;
   mail : string = mail;
```

```
   mail1: string = mail1;
   mail2: string = mail2;
end

end_objectbase

rules

send [ ?mo:MODULE ]:
  ( exists TASK ?ta suchthat ( linkto [ ?mo.pro_eng ?ta ] )):
  no_backward ( ?mo.status = Assigned )
  { MAILER mail ?mo.m_name ?mo.engineer  ?ta.comments ?ta.beg_date ?ta.end_date }
;

send1 [ ?mo:MODULE ]:
  ( and ( exists MDL ?mdl suchthat ( ancestor [ ?mdl ?mo ]))
       ( exists MANAGER ?man suchthat ( ancestor [?mdl ?man ] ))):
  ( and   no_backward ( ?mo.status = Done )
         no_backward ( ?mo.reviewer = "none" ))
  { MAILER mail1 ?mo.m_name  ?man.eng_name }
;

send2 [ ?mo:MODULE ]:
  :
  ( and no_backward ( ?mo.status = Done )
      no_backward ( ?mo.reviewer < > "none"))
  { MAILER mail2 ?mo.reviewer ?mo.m_name }
;

##################################################################
```

strategy maint

```
imports datamodel;
exports all;

objectbase

EDITOR :: superclass TOOL;
  change : string = change;
end

end_objectbase

rules

maint [ ?co:CONTENTS ]:
:
no_chain ( ?co.status = Maint )
{ EDITOR change ?co.contents }
no_chain ( ?co.status = Ready );

maint [ ?mod:MOD_PROG ]:
:
no_chain ( ?mod.status = Maint )
{ EDITOR change ?mod.source }
[ ?mod.comp_status = NotCompiled ];

maint [ ?scr:MOD_SCR ]:
:
no_chain ( ?scr.status = Maint )
```

```
{ EDITOR change ?scr.source }
[ ?scr.comp_status = NotCompiled ];

maintup [ ?mod:MOD_PROG ]:
( exists MOD_SCR ?scr suchthat ( linkto [ ?mod.screens  ?scr])):
 [ ?scr.comp_status = NotCompiled ]
{}
[ ?mod.comp_status = NotBuilt ];

maintup2 [ ?mod:MOD_PROG ]:
( exists MOD_PROG ?mod1 suchthat ( member [ ?mod.subroutines ?mod1 ])):
[ ?mod1.comp_status = NotCompiled ]
{}
[ ?mod.comp_status = NotBuilt];

maintup3 [ ?mod:MOD_PROG ]:
( exists MOD_PROG ?mod1 suchthat ( member [ ?mod.subroutines ?mod1 ])):
[ ?mod1.comp_status = NotBuilt ]
{}
[ ?mod.comp_status = NotBuilt];

#####################################################################
```

strategy modify

```
imports datamodel;
exports all;

objectbase

EDITOR :: superclass TOOL;
   edit : string = edit;
   graph : string = graph;
   graph_2 : string = graph2;
   console : string = console;
   edit_code : string = edit_code;
end

end_objectbase

rules

modify [ ?st:STATE ]:
 ( and ( exists TASK ?ta suchthat ( linkto [ ?st.pro_eng ?ta ]))
      ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
  ( and ( ?pg.login = CurrentUser )
       ( or no_chain ( ?st.status = Assigned )
          no_chain ( ?st.status = Active )
          no_chain ( ?st.status = Reviewd )))
 { EDITOR edit ?st.contents }
 no_forward ( ?st.status = Active );
 ( and ( ?st.status = Initialized)
     ( ?st.status = Active ));

modify [ ?func:FUNCTION ]:
 ( and ( exists TASK ?ta suchthat ( linkto [ ?func.pro_eng ?ta ]))
      ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
 ( and ( ?pg.login = CurrentUser )
       ( or no_chain ( ?func.status = Assigned )
          no_chain ( ?func.status = Active )
          no_chain ( ?func.status = Reviewd )))
```

```
  { EDITOR edit ?func.contents }
  ( ?func.status = Active );
  [ ?func.status = Done ];

modify [ ?dsg:DESIGN ]:
  ( and ( exists TASK ?ta suchthat ( linkto [ ?dsg.pro_eng ?ta ]))
        ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
   ( and ( ?pg.login = CurrentUser )
         ( or no_chain ( ?dsg.status = Assigned )
              no_chain ( ?dsg.status = Active )
              no_chain ( ?dsg.status = Reviewd )))
  { EDITOR graph ?dsg.contents }
  ( ?dsg.status = Active );
  ( ?dsg.status = Done);

modify [ ?lst:L_STATE ]:
  ( and ( exists TASK ?ta suchthat ( linkto [ ?lst.pro_eng ?ta ]))
        ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))
        ( exists LOGARCH ?log suchthat ( ancestor [ ?log ?lst ]))
        ( exists DESIGN ?dsg suchthat ( member [ ?log.sap ?dsg ]))):
   ( and ( ?pg.login = CurrentUser )
         no_backward ( ?dsg.status = Ready )
         ( or no_chain ( ?lst.status = Assigned )
              no_chain ( ?lst.status = Active )
              no_chain ( ?lst.status = Reviewd )))
  { EDITOR edit ?lst.contents }
  ( ?lst.status = Active );
  ( ?lst.status = Done);

modify [ ?lfunc:L_FUNC ]:
  ( and ( exists TASK ?ta suchthat ( linkto [ ?lfunc.pro_eng ?ta ]))
        ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
   ( and ( ?pg.login = CurrentUser )
         ( or no_chain ( ?lfunc.status = Assigned )
              no_chain ( ?lfunc.status = Active )
              no_chain ( ?lfunc.status = Reviewd )))
  { EDITOR edit ?lfunc.contents }
  ( ?lfunc.status = Active );
  ( ?lfunc.status = Done);

modify [ ?lscr:L_SCREEN ]:
  ( and ( exists TASK ?ta suchthat ( linkto [ ?lscr.pro_eng ?ta ]))
        ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
   ( and ( ?pg.login = CurrentUser )
         ( or no_chain ( ?lscr.status = Assigned )
              no_chain ( ?lscr.status = Active )
              no_chain ( ?lscr.status = Reviewd )))
  { EDITOR edit ?lscr.contents }
  ( ?lscr.status = Active );
  ( ?lscr.status = Done);

modify [ ?dsg:PHYS_DESIGN ]:
  ( and ( exists TASK ?ta suchthat ( linkto [ ?dsg.pro_eng ?ta ]))
        ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))
        ( exists DESIGN ?dsg1 suchthat ( linkto [ ?dsg1.sap ?dsg ]))):
   ( and ( ?pg.login = CurrentUser )
         ( or no_chain ( ?dsg.status = Assigned )
              no_chain ( ?dsg.status = Active )
              no_chain ( ?dsg.status = Reviewd )))
  { EDITOR graph_2 ?dsg.contents ?dsg1.contents }
  ( ?dsg.status = Active );
```

```
  ( ?dsg.status = Done);

modify [ ?db:PHYS_DB ]:
  ( and ( exists TASK ?ta suchthat ( linkto [ ?db.pro_eng ?ta ]))
        ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))
        ( exists PHYSARCH ?phy suchthat ( member [ ?phy.db ?db ]))
        ( exists SQL ?sql suchthat ( ancestor [ ?phy  ?sql ]))):
   ( and ( ?pg.login = CurrentUser )
         ( ?sql.status = Ready )))
  { EDITOR console ?db.contents }
  ( ?db.status = Active );
  ( ?db.status = Done);

modify [ ?sql:SQL ]:
  ( and ( exists PHYS_DESIGN ?dsg suchthat ( member [ ?dsg.db_language ?sql ]))
        ( exists TASK ?ta suchthat ( linkto [ ?dsg.pro_eng ?ta ]))
        ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
   ( and ( ?pg.login = CurrentUser )
         ( or no_chain ( ?sql.status = Initialized )
              no_chain ( ?sql.status = Active )
              no_chain ( ?sql.status = Reviewd ))
         ( or no_chain ( ?dsg.status = Ready )
              no_chain ( ?dsg.status = Done )
              no_chain ( ?dsg.status = Reviewd )
              no_chain ( ?dsg.status = Active )))
  { EDITOR graph_2 ?sql.contents ?dsg.contents}
  no_forward ( ?sql.status = Active );
  ( and ( ?sql.status = Initialized )
        ( ?sql.status = Active ));

modify [ ?mod:MOD_PROG ]:
  ( and ( exists TASK ?ta suchthat ( linkto [ ?mod.pro_eng ?ta ]))
        ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
   ( and ( ?pg.login = CurrentUser )
         ( or no_chain ( ?mod.status = Assigned )
              no_chain ( ?mod.status = Active )
              no_chain ( ?mod.status = Reviewd )
              no_chain ( ?mod.comp_status = NotCompiled)))
  { EDITOR edit_code ?mod.source }
  ( and ( ?mod.status = Active )
        no_forward ( ?mod.comp_status = Initialized ));
  ( and ( ?mod.status = Active )
        ( ?mod.comp_status = Initialized ) # dummy for chaining
        no_backward ( ?mod.comp_status = NotCompiled ));

modify [ ?scr:MOD_SCR ]:
  ( and ( exists TASK ?ta suchthat ( linkto [ ?scr.pro_eng ?ta ]))
        ( exists ENGINEER ?pg suchthat ( member [ ?pg.jobs ?ta ]))):
   ( and ( ?pg.login = CurrentUser )
         ( or no_chain ( ?scr.status = Assigned )
              no_chain ( ?scr.status = Active )
              no_chain ( ?scr.status = Reviewd )
              no_chain ( ?scr.comp_status = NotCompiled)))
  { EDITOR edit_code ?scr.source }
  ( and ( ?scr.status = Active )
        no_forward ( ?scr.comp_status = NotCompiled ));
  ( and ( ?scr.status = Active )
        ( ?scr.comp_status = Initialized ) # dummy for chaining
        ( ?scr.comp_status = NotCompiled ));
```

strategy overview

```
imports datamodel;
exports all;

objectbase

LISTER :: superclass TOOL;
   list : string = list;
   list1: string = list1;
   list2: string = list2;
end

end_objectbase

rules

overview [ ?spec:SPEC ]:
  ( and ( forall FUNCTION ?func suchthat ( member [ ?spec.functions ?func ]))
        ( forall STATE ?st suchthat ( member [ ?spec.db_primitives  ?st ]))):
  { LISTER list ?func.m_name ?func.status ?func.engineer ?func.reviewer
              ?st.m_name ?st.status ?st.engineer ?st.reviewer)
;

overview [ ?log:LOGARCH ]:
  ( and ( forall L_FUNC ?lfunc suchthat ( member [ ?log.functions ?lfunc ]))
        ( forall L_STATE ?lst suchthat ( ancestor [ ?log ?lst ]))
        ( forall L_SCREEN ?lscr suchthat ( member [ ?log.interface ?lscr ]))
        ( forall DESIGN ?dsg suchthat ( member [ ?log.sap ?dsg ]))):
  { LISTER list1 ?lfunc.m_name ?lfunc.status ?lfunc.engineer ?lfunc.reviewer
              ?lst.m_name ?lst.status ?lst.engineer ?lst.reviewer
              ?lscr.m_name ?lscr.status ?lscr.engineer ?lscr.reviewer
              ?dsg.m_name ?dsg.status ?dsg.engineer ?dsg.reviewer}
;

overview [ ?phy:PHYSARCH ]:
  ( and ( forall PHYS_DESIGN ?dsg suchthat ( member [ ?phy.san ?dsg ]))
        ( forall SQL ?sql suchthat ( ancestor [ ?phy ?sql ]))
        ( forall MOD_PROG ?mod suchthat ( ancestor [ ?phy ?mod ]))
        ( forall MOD_SCR ?scr suchthat ( member [ ?phy.screens ?scr ]))
        ( forall PHYS_DB ?db suchthat ( member [ ?phy.db ?db ]))):
  { LISTER list2 ?dsg.m_name ?dsg.status ?dsg.engineer ?dsg.reviewer
              ?sql.m_name ?sql.status ?dsg.engineer ?dsg.reviewer
              ?db.m_name ?db.status ?db.engineer ?db.reviewer
              ?scr.m_name ?scr.status ?scr.engineer ?scr.reviewer
              ?mod.m_name ?mod.status ?mod.engineer ?mod.reviewer}
 ;
#################################################################
```

strategy print

```
imports datamodel;
exports all;

objectbase

PRINTER :: superclass TOOL;
   print : string = print;
end
```

```
end_objectbase

rules

print_out [ ?co:CONTENTS ]:
:
 ( or no_chain ( ?co.status = Active )
     no_chain ( ?co.status = Done )
     no_chain ( ?co.status = Ready )
     no_chain ( ?co.status = Reviewd ))
 { PRINTER print ?co.contents }
 ;

print_out [ ?mod:MOD_PROG ]:
:
 ( or no_chain ( ?mod.status = Active )
     no_chain ( ?mod.status = Done )
     no_chain ( ?mod.status = Ready )
     no_chain ( ?mod.status = Reviewd ))
 { PRINTER print ?mod.source }
 ;

print_out [ ?scr:MOD_SCR ]:
:
 ( or no_chain ( ?scr.status = Active )
     no_chain ( ?scr.status = Done )
     no_chain ( ?scr.status = Ready )
     no_chain ( ?scr.status = Reviewd ))
 { PRINTER print ?scr.source }
;

#################################################################
```

strategy propagate

```
imports datamodel;
exports all;

rules

propagate [ ?st:STATE , ?lst:L_STATE ]:
:
{}
( linkto [ ?st.split_into ?lst ]);

propagate [ ?func:FUNCTION , ?lfunc:L_FUNC ]:
:
{}
( linkto [ ?func.log_repres ?lfunc ]);

propagate [ ?lfunc:L_FUNC , ?lst:L_STATE ]:
:
{}
( linkto [ ?lfunc.uses_state ?lst ]);

propagate [ ?lfunc:L_FUNC , ?lscr:L_SCREEN ]:
:
{}
( linkto [ ?lfunc.uses_scr ?lscr] );

propagate [ ?lfunc:L_FUNC , ?mod:MOD_PROG ]:
```

```
:
{}
( linkto [ ?lfunc.coded_in ?mod ]);

propagate [ ?lscr:L_SCREEN , ?scr:MOD_SCR ]:
:
{}
( linkto [ ?lscr.code ?scr] );

propagate [ ?lscr:L_SCREEN , ?lst.L_STATE ]
:
{}
( linkto [ ?lscr.uses_lstate  ?lst ]

propagate [ ?lst:L_STATE , ?mod:MOD_PROG ]:
:
{}
( linkto [ ?lst.coded_in  ?mod] );

propagate [ ?dsg:DESIGN , ?phdsg:PHYS_DESIGN ]:
:
{}
( linkto [ ?dsg.sap  ?phdsg] );

propagate [ ?mod:MOD_PROG , ?scr:MOD_SCR ]:
:
{}
( linkto [ ?mod.screens  ?scr] );

propagate [ ?mo:MODULE , ?doc:DOC ]:
:
{}
( linkto [ ?mo.doc ?doc ] );

#####################################################################
```

strategy touch

```
imports datamodel;
exports all;

objectbase

EDITOR :: superclass TOOL;
   touch : string = touch;
   change : string = change;
end

end_objectbase

rules

touch [ ?co:CONTENTS ]:
no_chain ( ?co.status = Ready )
{ EDITOR touch ?co.contents }
[ ?co.status = Maint ];

touch [ ?mod:MOD_PROG ]:
:
no_chain ( ?mod.status = Ready )
{ EDITOR touch ?mod.source }
```

```
( and [ ?mod.status = Maint ]
    [ ?mod.comp_status = NotCompiled ]);

touch [ ?scr:MOD_SCR ]:
:
no_chain ( ?scr.status = Ready )
{ EDITOR touch ?scr.source }
( and [ ?scr.status = Maint ]
    no_backward [ ?scr.comp_status = NotCompiled ]);

hide touchdown [ ?lfunc:L_FUNC ]:
( exists FUNCTION ?func suchthat ( linkto [ ?func.log_repres ?lfunc ])):
[ ?func.status = Maint ]
{}
[ ?lfunc.status = Maint ];

hide touchdown [ ?lst:L_STATE ]:
( exists STATE ?st suchthat ( linkto [ ?st.split_into  ?lst])):
 [ ?st.status = Maint ]
{}
[ ?lst.status = Maint ];

hide touchdown [ ?mod:MOD_PROG ]:
( exists L_FUNC ?lfunc suchthat ( linkto [ ?lfunc.coded_in ?mod])):
[ ?lfunc.status = Maint ]
{}
[ ?mod.status = Maint ];

hide touchdown2 [ ?mod:MOD_PROG ]:
( exists L_STATE ?lst suchthat ( linkto [ ?lst.coded_in  ?mod ])):
[ ?lst.status = Maint ]
 {}
[ ?mod.status = Maint ];

hide touchdown [ ?dsg:PHYS_DESIGN ]:
 ( exists DESIGN ?dsg1 suchthat ( linkto [ ?dsg1.sap ?dsg ])):
 [ ?dsg1.status = Maint ]
  {}
 [ ?dsg.status = Maint ];

hide touchdown [ ?sql:SQL ]:
  ( exists PHYS_DESIGN ?dsg  suchthat ( member [ ?dsg.db_language ?sql ])):
  [ ?dsg.status = Maint ]
  {}
  [ ?sql.status = Maint ];

hide touchdown [ ?db:PHYS_DB ]:
  ( and ( exists PHYSARCH ?phy suchthat ( member [ ?phy.db ?db ]))
        ( exists SQL ?sql suchthat ( ancestor [ ?phy ?sql ]))):
  [ ?sql.status = Maint ]
  {}
  [ ?db.status = Maint ];

hide touchdown [ ?scr:MOD_SCR ]:
  ( exists L_SCREEN ?lscr suchthat ( linkto [ ?lscr.code  ?scr ])):
  [ ?lscr.status = Maint ]
  {}
  [ ?scr.status = Maint ];

hide touchup [ ?mod:MOD_PROG ]:
 ( exists MOD_PROG ?mod1 suchthat ( member [ ?mod.subroutines  ?mod1 ])):
```

```
[ ?mod1.status = Maint ]
{}
[ ?mod.status = Maint ];

hide touchup [ ?lst:L_STATE ]:
( exists L_FUNC ?lfunc suchthat ( linkto [ ?lfunc.uses_state  ?lst ])):
[ ?lfunc.status = Maint ]
{}
[ ?lst.status = Maint ];
```

APPENDIX D : Envelopes

ENVELOPE build;

```
SHELL sh;

INPUT
    text : thefile;
    binary       : main_object_f;
    set_of binary : object_code;
    set_of binary : scr_codes;
    binary : scr_code;
    test : results;
OUTPUT
   none;

BEGIN
   echo
   echo ----------------
   echo "now building..."
  echo " build ok ?"
  read  ok
  if [ "$ok" = "y" ]
  then
       echo "Build successful "
       ret_val=0
 else
       echo " errors " >> $results
       echo "Build failed (--> look with viewErr rule)"
       ret_val=1
   fi

RETURN "$ret_val";
END

#####################################################################
```

ENVELOPE change;

```
# The envelope Edit_file is used by the mod_design, mod_test_code
# rules. It pumps up an emacs and allows the modifications.

SHELL sh;

INPUT
   text : thefile;
OUTPUT
    none;

BEGIN
 clear
 SaveReport=`ls -l $thefile`
 ret_code=2

# Call the emacs editor
# --------------------
 emacs -fn 9x15 -geometry 80x24 $thefile
# --------------------
   NewReport=`ls -l $thefile`
```

```
   echo "Are you ready with maintenance y/n ? "
   read answer
   if [ "$answer" = "y" ]
   then
      if [ "$SaveReport" = "$NewReport" ]
      then
          ret_code=1
      else
          ret_code=0
      fi
    fi
  RETURN "$ret_code";
END

#################################################################
```

ENVELOPE compile;

```
# The envelope compile is used by the compile rule. It invokes a C
# compiler on the source_code ( from the module or the testpackage)

SHELL sh;

INPUT
    text : source_f;
    binary : object_f;
    text : results;

OUTPUT
   none;

BEGIN
   if [ -f $results ]
   then rm $results
   fi
   echo "--------------------------------"
   echo "compiling < `basename $source_f` > ..."
   echo "compile ok "
   read ok
#    cc -c $source_f -o $object_f > > $results 2>&1
  if [ "$ok" = "y" ]
  then
       echo "Compile successful "
       ret_val=0
  else
       echo "This are your compile errors " > > $results
       echo "Compile failed (--> look with viewErr rule"
       ret_val=1
   fi
RETURN "$ret_val";
END

#################################################################
```

ENVELOPE console;

```
# Run a shell in an OpenWindows terminal window

SHELL sh;
```

```
INPUT
   text : contents;

OUTPUT
   none;

BEGIN
    echo "This command tool server to construct the sql tables."
    echo "Type 'exit' or <CTRL>-d to stop"
    echo "-----------------------------"
    pwd
    cmdtool;
    echo "Are you ready with table construction ? y/n ?"
    read answer
    if [ "$answer" = "y" ]
    then
       ret_code=1
    else
       ret_code=0
    fi
RETURN "$ret_code";
END

#####################################################################
```

ENVELOPE edit;

```
SHELL sh;

INPUT
   text : thefile;
OUTPUT
    none;

BEGIN
  clear
# Call the emacs editor
   emacs -fn 9x15 -geometry 80x24 $thefile
    echo "File is ready to review y/n "
    read answer
    if [ "$answer" = "y" ]
    then
       ret_code=1
    else
      ret_code=0
    fi
  RETURN "$ret_code";
END

#####################################################################
```

ENVELOPE edit_file;

```
SHELL sh;

INPUT
   text : thefile;

OUTPUT
```

```
    none;

BEGIN
# Test to see if file already exists
# --------------------------------
  Created="Yes"
  SaveReport=`ls -l $thefile`
  if [ -f $thefile ]
  then
     Created="No"
  fi
#  echo "File needs to be created : "$Created
# Call the emacs editor
# --------------------
  emacs -fn 9x15 -geometry 80x55 $thefile
# ---------------------
    echo " Is the code ready for compiling  y/n "
   read answer
   if [ "$answer" = "y" ]
   then
      ret_code=1
   else
     ret_code=0
   fi
  RETURN "$ret_code";
END

#################################################################
```

ENVELOPE edit_doc;

```
SHELL sh;

INPUT
   text : thefile;

OUTPUT
   string : NAME,MODULE;

BEGIN
  echo " Please enter the object name "
  read NAME
  echo " For what module ?"
  read MODULE
  emacs -fn 9x15 -geometry 80x55 $thefile
# ---------------------
  RETURN "0":$NAME,$MODULE;
END

#################################################################
```

ENVELOPE graph2;

```
SHELL sh;

INPUT
   text : thefile;
   text : feedb_file;
```

```
OUTPUT
   none;

BEGIN
  EL_FILE=/tmp/review$$
  touch $EL_FILE
  # This places the feedback in the lower buffer, and places the
  # design in the upper buffer in read only moe.
  # ------------------------------------------------------
  echo \(find-file \"$thefile\"\) > > $EL_FILE
  echo \(split-window\) > > $EL_FILE
  echo \(find-file-read-only \"$feedb_file\"\) > > $EL_FILE
  emacs -fn 9x15 -geometry 80x55 -l $EL_FILE
  rm $EL_FILE
  echo " Is the file ready to review  y/n ?"
  read answer
  if [ "$answer" = "y" ]
  then
     ret_code=1
  else
     ret_code=0
  fi
RETURN "$ret_code";
END

#####################################################################
```

ENVELOPE ini;

```
# The envelope INI is used by the initiate rule and asks the manager if
# the project is ready to be scheduled.

SHELL sh;

INPUT
   none ;

OUTPUT
   none ;

BEGIN
clear
echo "Is the project ready to be started ? [y/n]: "
read answer
if [ "$answer" = "y" ]
then ret_val=0
else ret_val=1
fi
RETURN "$ret_val";
END

#####################################################################
```

ENVELOPE list;

```
SHELL ksh;

INPUT
  set_of string  : func_name;
```

```
  set_of enumerated  : f_status;
  set_of string   : f_user_name;
  set_of string   : f_reviewer;

  set_of string  : state_name;
  set_of enumerated  : st_status;
  set_of string   : st_user_name;
  set_of string   : st_reviewer;

OUTPUT
  none;

BEGIN
# ----------------------------------------------------------
# List for all functions
# ----------------------------------------------------------
flag1=""
flag2=""
flag3=""
flag4=""
for i in $func_name
 do
  flag1="$flag1 0"
  flag2=""
  for j in $f_status
  do
   flag2="$flag2 0"
   flag3=""
   for k in $f_user_name
    do
    flag3="$flag3 0"
    flag4=""
    for l in $f_reviewer
      do
      flag4="$flag4 0"
      if [ "$flag1" = "$flag2" -a "$flag2" = "$flag3" -a "$flag3" = "$flag4" ]
       then
# Body of your envelope. I just print them out .
      echo ------------------------
      echo "Name of function    : "$i
      echo "actual status       : "$j
      echo "engineer            : "$k
      echo "reviewer            : "$l
      echo ------------------------
     fi
   done
  done
 done
done
# ----------------------------
#list for all state
# ----------------------------
flag1=""
flag2=""
flag3=""
flag4=""
for i in $state_name
 do
  flag1="$flag1 0"
  flag2=""
```

```
 for j in $st_status
 do
  flag2="$flag2 0"
  flag3=""
  for k in $st_user_name
   do
   flag3="$flag3 0"
   flag4=""
   for l in $st_reviewer
    do
    flag4="$flag4 0"
    if [ "$flag1" = "$flag2" -a "$flag2" = "$flag3" -a "$flag3" = "$flag4" ]
     then
# Body of your envelope. I just print them out .
     echo ----------------------------
     echo "Name of state      : "$i
     echo "actual status      : "$j
     echo "engineer           : "$k
     echo "reviewer           : "$l
     echo ----------------------------
    fi
   done
  done
 done
done
RETURN "0";
END

#####################################################################
```

ENVELOPE list1;

```
SHELL sh;

INPUT
  set_of string  : lfunc_name;
  set_of enumerated  : lf_status;
  set_of string   : lf_user_name;
  set_of string   : lf_reviewer;

  set_of string  : lstate_name;
  set_of enumerated  : lst_status;
  set_of string   : lst_user_name;
  set_of string   : lst_reviewer;

  set_of string  : lscr_name;
  set_of enumerated  : lscr_status;
  set_of string   : lscr_user_name;
  set_of string   : lscr_reviewer;

  set_of string  : dsg_name;
  set_of enumerated  : dsg_status;
  set_of string   : dsg_user_name;
  set_of string   : dsg_reviewer;

OUTPUT
  none;

BEGIN
# ----------------------------------------------------------
```

```
# List for all lfunctions
# ------------------------------------------------------
  clear
  echo " LIST OF PROGRESS IN LOGICAL ARCHITECTURE "
  echo " ------------------------------------ "
flag1=""
flag2=""
flag3=""
flag4=""
for i in $lfunc_name
 do
  flag1="$flag1 0"
  flag2=""
  for j in $lf_status
  do
   flag2="$flag2 0"
   flag3=""
   for k in $lf_user_name
    do
    flag3="$flag3 0"
    flag4=""
    for l in $lf_reviewer
      do
      flag4="$flag4 0"
      if [ "$flag1" = "$flag2" -a "$flag2" = "$flag3" -a "$flag3" = "$flag4" ]
       then
# Body of your envelope. I just print them out .
      echo -------------------------
      echo "Name of lfunction    : "$i
      echo "actual status        : "$j
      echo "engineer             : "$k
      echo "reviewer             : "$l
      echo -------------------------
     fi
   done
  done
 done
done
# ----------------------------
#list for all lstate
# ----------------------------
flag1=""
flag2=""
flag3=""
flag4=""
for i in $lstate_name
 do
  flag1="$flag1 0"
  flag2=""
  for j in $lst_status
  do
   flag2="$flag2 0"
   flag3=""
   for k in $lst_user_name
    do
    flag3="$flag3 0"
    flag4=""
    for l in $lst_reviewer
      do
      flag4="$flag4 0"
```

```
    if [ "$flag1" = "$flag2" -a "$flag2" = "$flag3" -a "$flag3" = "$flag4" ]
     then
# Body of your envelope. I just print them out .
     echo ------------------------
     echo "Name of log_state   : "$i
     echo "actual status       : "$j
     echo "engineer            : "$k
     echo "reviewer            : "$l
     echo ------------------------
    fi
   done
  done
 done
done
# ----------------------------
#list for all screens
# ----------------------------
flag1=""
flag2=""
flag3=""
flag4=""
for i in $lscr_name
 do
  flag1="$flag1 0"
  flag2=""
  for j in $lscr_status
  do
   flag2="$flag2 0"
   flag3=""
   for k in $lscr_user_name
    do
    flag3="$flag3 0"
    flag4=""
    for l in $lscr_reviewer
     do
     flag4="$flag4 0"
     if [ "$flag1" = "$flag2" -a "$flag2" = "$flag3" -a "$flag3" = "$flag4" ]
      then
# Body of your envelope. I just print them out .
     echo ------------------------
     echo "Name of screen      : "$i
     echo "actual status       : "$j
     echo "engineer            : "$k
     echo "reviewer            : "$l
     echo ------------------------
    fi
   done
  done
 done
done
# ----------------------------
#list for all design
# ----------------------------
flag1=""
flag2=""
flag3=""
flag4=""
for i in $dsg_name
 do
  flag1="$flag1 0"
```

```
 flag2=""
 for j in $dsg_status
 do
  flag2="$flag2 0"
  flag3=""
  for k in $dsg_user_name
   do
   flag3="$flag3 0"
   flag4=""
   for l in $dsg_reviewer
    do
    flag4="$flag4 0"
    if [ "$flag1" = "$flag2" -a "$flag2" = "$flag3" -a "$flag3" = "$flag4" ]
     then
# Body of your envelope. I just print them out .
    echo ----------------------------
    echo "Name of design     : "$i
    echo "actual status      : "$j
    echo "engineer           : "$k
    echo "reviewer           : "$l
    echo ----------------------------
    fi
   done
  done
 done
done
RETURN "0";
END

#################################################################
```

ENVELOPE list2;

```
SHELL sh;

INPUT
  set_of string  : dsg_name;
  set_of enumerated  : dsg_status;
  set_of string   : dsg_user_name;
  set_of string   : dsg_reviewer;

  set_of string  : sql_name;
  set_of enumerated  : sql_status;
  set_of string   : sql_user_name;
  set_of string   : sql_reviewer;

  set_of string  : db_name;
  set_of enumerated  : db_status;
  set_of string   : db_user_name;
  set_of string   : db_reviewer;

  set_of string  : scr_name;
  set_of enumerated  : scr_status;
  set_of string   : scr_user_name;
  set_of string   : scr_reviewer;

  set_of string  : mod_name;
  set_of enumerated  : mod_status;
  set_of string   : mod_user_name;
  set_of string   : mod_reviewer;
```

```
OUTPUT
  none;

BEGIN
# ----------------------------------------------------
# List for all phys_design
# ----------------------------------------------------
flag1=""
flag2=""
flag3=""
flag4=""

for i in $dsg_name
 do
  flag1="$flag1 0"
  flag2=""
  for j in $dsg_status
  do
   flag2="$flag2 0"
   flag3=""
   for k in $dsg_user_name
    do
    flag3="$flag3 0"
    flag4=""
    for l in $dsg_reviewer
     do
     flag4="$flag4 0"
     if [ "$flag1" = "$flag2" -a "$flag2" = "$flag3" -a "$flag3" = "$flag4" ]
      then
# Body of your envelope. I just print them out .
      echo ------------------------
      echo "Name of screen     : "$i
      echo "actual status      : "$j
      echo "engineer           : "$k
      echo "reviewer           : "$l
      echo ------------------------
     fi
   done
  done
 done
done
# ----------------------------
#list for all sql
# ----------------------------
  clear
  echo " LIST OF PROGRESS IN LOGICAL ARCHITECTURE "
  echo " ------------------------------------ "
flag1=""
flag2=""
flag3=""
flag4=""
for i in $sql_name
 do
  flag1="$flag1 0"
  flag2=""
  for j in $sql_status
  do
   flag2="$flag2 0"
   flag3=""
```

```
   for k in $sql_user_name
    do
    flag3="$flag3 0"
    flag4=""
    for l in $sql_reviewer
     do
     flag4="$flag4 0"
     if [ "$flag1" = "$flag2" -a "$flag2" = "$flag3" -a "$flag3" = "$flag4" ]
      then
# Body of your envelope. I just print them out .
      echo ------------------------
      echo "Name of design    : "$i
      echo "actual status     : "$j
      echo "engineer          : "$k
      echo "reviewer          : "$l
      echo ------------------------
     fi
   done
  done
 done
done
# ---------------------------
#list for all phys_db
# ---------------------------
flag1=""
flag2=""
flag3=""
flag4=""
for i in $db_name
 do
  flag1="$flag1 0"
  flag2=""
  for j in $db_status
  do
   flag2="$flag2 0"
   flag3=""
   for k in $db_user_name
    do
    flag3="$flag3 0"
    flag4=""
    for l in $db_reviewer
     do
     flag4="$flag4 0"
     if [ "$flag1" = "$flag2" -a "$flag2" = "$flag3" -a "$flag3" = "$flag4" ]
      then
# Body of your envelope. I just print them out .
      echo ------------------------
      echo "Name of db_language : "$i
      echo "actual status       : "$j
      echo "engineer            : "$k
      echo "reviewer            : "$l
      echo ------------------------
     fi
   done
  done
 done
done
# ---------------------------
#list for all screens
# ---------------------------
```

```
flag1=""
flag2=""
flag3=""
flag4=""

for i in $scr_name
 do
  flag1="$flag1 0"
  flag2=""
  for j in $scr_status
  do
   flag2="$flag2 0"
   flag3=""
   for k in $scr_user_name
    do
    flag3="$flag3 0"
    flag4=""
    for l in $scr_reviewer
     do
     flag4="$flag4 0"
     if [ "$flag1" = "$flag2" -a "$flag2" = "$flag3" -a "$flag3" = "$flag4" ]
      then
# Body of your envelope. I just print them out .
      echo ---------------------------
      echo "Name of database    : "$i
      echo "actual status       : "$j
      echo "engineer            : "$k
      echo "reviewer            : "$l
      echo ---------------------------
     fi
   done
  done
 done
done
# ------------------------------
#list for all modules progr
# ------------------------------
flag1=""
flag2=""
flag3=""
flag4=""
for i in $mod_name
 do
  flag1="$flag1 0"
  flag2=""
  for j in $mod_status
  do
   flag2="$flag2 0"
   flag3=""
   for k in $mod_user_name
    do
    flag3="$flag3 0"
    flag4=""
    for l in $mod_reviewer
     do
     flag4="$flag4 0"
     if [ "$flag1" = "$flag2" -a "$flag2" = "$flag3" -a "$flag3" = "$flag4" ]
      then
# Body of your envelope. I just print them out .
      echo ---------------------------
```

```
    echo "Name of module     : "$i
    echo "actual status      : "$j
    echo "engineer           : "$k
    echo "reviewer           : "$l
    echo ----------------------------
   fi
  done
 done
done
done
RETURN "0";
END
```

##

ENVELOPE mail_eng;

```
SHELL sh;

INPUT
   string : name;
   string : adr;
   set_of text   : comments;
   set_of int    : beg;
   set_of int    : end;

OUTPUT
   none ;

BEGIN
echo "now sending mail ... "
# construction of the message
echo " Schedule for : " $name > > message$$
echo " ----------" > > message$$
echo "            " > > message$$
echo " Begin date : " $beg > > message$$
echo " End date : " $end > > message$$
echo " Description of task :" > > message$$
echo > > message$$
cat $comments > > message$$
# mailing
mail $adr < message$$
rm message$$
RETURN "0";
END
```

##

ENVELOPE mail_man;

```
SHELL sh;

INPUT
   string : name;
   string : adr;

OUTPUT
   none ;
```

```
BEGIN
echo " The function ", $name ,"is ready to review but there is no reviewer assigned yet"  > > message$$
mail $adr < message$$
rm message$$
RETURN "0";
END

#################################################################
```

ENVELOPE mail_rev;

```
SHELL sh;

INPUT
    string : adr;
    string : name;

OUTPUT
    none ;

BEGIN
echo " The function ", $name , "is ready to review"  > > message$$
mail $adr < message$$
rm message$$
RETURN "0";
END

#################################################################
```

ENVELOPE print;

```
SHELL sh;

INPUT
   text : thefile;

OUTPUT
    none;

BEGIN
   echo "now printing ... "
   rubens  $thefile
RETURN "0";
END

#################################################################
```

ENVELOPE review;

```
SHELL sh;

INPUT
   text : thefile;
   text : feedb_file;
   string : name;
OUTPUT
    none;

BEGIN
```

```
  if [ -f $feedb_file ]
  then
    rm $feedb_file
  fi

  EL_FILE=/tmp/review$$
  touch $EL_FILE
  # This places the feedback in the lower buffer, and places the
  # design in the upper buffer in read only mode.
  # ------------------------------------------------------
  echo \(find-file \"$feedb_file\"\) > > $EL_FILE
  echo \(split-window\) > > $EL_FILE
  echo \(find-file-read-only \"$thefile\"\) > > $EL_FILE
 emacs -fn 9x15 -geometry 80x55 -l $EL_FILE
  rm $EL_FILE
  # test to see if the feedback file exists. If yes, it is mailed,
  # otherwise nothing happens.
  if [ -f $feedb_file ]
  then
    mail $name < $feedb_file
    ret_code=0
 else
    ret_code=1
  fi
RETURN "$ret_code";
END

#####################################################################
```

ENVELOPE schedule;

```
SHELL sh;

INPUT
      text   : comments;
      string : person;

OUTPUT
      string : O_NAME;
      int : TIME_B, TIME_E;

BEGIN
  clear
  echo ------------------------------------
  echo --------  S C H E D U L E R  ----------
  echo ------------------------------------
  if [ -f $comments ]    # test to see if file exists or not
  then
    rm $comments
  fi
  echo "Now scheduling : $person"
  echo
  echo "Please enter object name :"
  read O_NAME
  echo "Please enter dates as follows :"
  echo
  echo "Monday December 2, 1991 as : 911202"
  echo
# Get Beginning Date
# ----------------
```

```
  echo "Beginning Date [YYMMDD] : "
  read TIME_B
# Get Ending Date
  echo "Ending Date [YYMMDD] : "
  read TIME_E
# Get comments
  answer=""
  echo "Type <.> to finish"
  echo "Enter text here ...          "
  echo ------------------------------
  read answer
  while [ "$answer" != "." ]
  do
    echo $answer >> $comments
    read answer
  done
  echo ------------------------------
RETURN "0": $TIME_B, $TIME_E, $O_NAME;
END

#####################################################################
```

ENVELOPE compile;

```
SHELL sh;

INPUT
    text : source_f;
    binary : object_f;
    text : results;

OUTPUT
   none;

BEGIN
   if [ -f $results ]
   then rm $results
   fi
   echo "--------------------------------"
   echo "compiling <`basename $source_f`> ..."
   echo "compile ok "
   read ok
#    cc -c $source_f -o $object_f >> $results 2>&1
#    CSTATUS=$?
  if [ "$ok" = "y" ]
  then
       echo "Compile successful "
       ret_val=0
  else
       echo "This are your compile errors " >> $results
       echo "Compile failed (--> look with viewErr rule"
       ret_val=1
   fi
RETURN "$ret_val";
END

#####################################################################
```

ENVELOPE touch;

```
SHELL sh;

INPUT
   text : thefile;

OUTPUT
    none;

BEGIN
  clear
  ret_code=1
  SaveReport=`ls -l $thefile`
  echo " W A R N I N G "
  echo " ------------ "
  echo " Modification of this file will invoke consistency chainings "
  echo " Do you really want to modify this file (y/n) "
  read answer
  if [ "$answer" = "y" ]
  then
     # Call the emacs editor
     # --------------------
     emacs -fn 9x15 -geometry 70x40 $thefile
     # ---------------------
     NewReport=`ls -l $thefile`
     if [ "$SaveReport" = "$NewReport" ]
     then
        echo "File not changed"
        ret_code=1
     else
        echo "File changed, consistency chainings ..."
        ret_code=0
     fi
  else
    echo "File not changed"
  fi
  RETURN "$ret_code";
END

#################################################################
```

ENVELOPE viewerr;

```
SHELL sh;

INPUT
        text      : compile_log;

OUTPUT
        none ;

BEGIN
echo "!========= compile errors are printed ==================="
rybens  $compile_log
RETURN "0" ;
END
```