MASTER IN COMPUTER SCIENCE

On Requirements Specification Products and Processes

Goffinet, Luc; Lejoly, Marc

Award date:
1989

Awarding institution:
University of Namur

On Requirements Specification Products and Processes

by

Luc GOFFINET & Marc LEJOLY

Promoter: Axel Van Lamsweerde

A thesis submitted in
conformity with the requirements
for the degree of
"Licencié et Maître en Informatique"

Facultés Universitaires Notre-Dame de la Paix
Namur - September 1989

ABSTRACT

ENGLISH

A great deal of research is currently carried on to introduce formal methods further in the software development process. But the use of formal specifications also involves the creation of automated tools to help managing them. Such tools require formal models to capture the semantics of their tasks. One of these tools deals with the gradual building of formal specifications from informal requirements. This is the field addressed by our work. It studies the processes underlying the gradual building of formal specifications, and sketches a model for representing such processes. Thus it provides a theoretical background for the formalization of the specification method that guides the specification process of a system. This has been done in two stages. The former consists in the study of a few formal languages based on an analysis of their expressive and deductive power. It makes up the PRODUCT LEVEL of this work. The latter is composed of a study of the specification method, followed in the solving of some case studies in some formal languages, and of an attempt at the modelling of this method. This makes up the PROCESS LEVEL of this work.

FRANÇAIS

De nombreuses recherches sont actuellement menées afin d'introduire des méthodes formelles dans le processus de développement de logiciels. Mais l'emploi de spécifications formelles implique aussi l'introduction d'outils automatisés permettant de gérer ces méthodes formelles. De tels outils nécessitent des modèles formels afin de décrire la sémantique de leurs tâches. Parmi ceux-ci nous retrouvons les outils concernant la construction progressive de spécifications formelles à partir de spécifications informelles. C'est dans ce cadre que s'inscrit notre mémoire. Nous étudierons donc les processus sous-jacents à l'élaboration progressive de spécifications formelles; après quoi nous tracerons une esquisse d'un modèle qui représentent ces processus. Donc, cela peut donner un référentiel théorique à la formalisation de méthodes de spécification qui guident le processus de spécification d'un système. Pour mener cela à bien, deux étapes ont été requises. La première a consisté à étudier quelques langages formels de spécification, étude basée sur une évaluation de leurs puissances expressives et déductives; ceci formant le PRODUCT LEVEL de ce mémoire. La seconde étape a consisté dans l'étude des méthodes de spécification que nous avons utilisées durant la résolution d'un cas représentatif, et dans une tentative de modélisation de ces méthodes, ce qui a constitué le PROCESS LEVEL de ce mémoire.

ACKNOWLEDGEMENTS

First of all, we would like to express our gratefulness to Dr. John Wordsworth and Steve Powell from Hursley Park IBM Laboratories, who have devoted a lot of their precious time to our education in Z. Their practical experience as well as pedagogical skills were extremely helpful.

Secondly, we wish to thank the people who have helped us in the writing of this essay, i.e. those we have interviewed, those who have read our drafts and made many interesting comments about them, and those who have showed some interest in our work. So, we would like to express our thankfulness to our promoter, Axel Van Lamsweerde, and his collaborators Philippe Dubois and M.S. Celitkin.

Next, we would like to thank our parents for their generous and constant support shown throughout all these years spent in college.

Finally, we would love to thank other people who have been a great help to us - although they have not been directly involved in the completion of this work. So, we owe a great deal of gratitude to the Beatles and their wonderful music, which has turned so many of our breaks into delightful moments, and therefore has renewed our energy to carry on with our work...

TABLE OF CONTENTS

The initials between curly brackets refer to the first names of the authors of this study. They point out the parts of this work on which they have worked especially. So (G) stands for Goffinet and (L) for Lejoly, whereas (L + G) denotes a close collaboration on a particular section.

1. THE NIGHT BEFORE CRISIS

"Twas the night before crisis, and all through the house,
Not a program was working, not even a browse.

The programmers were wrung out, too mindless to care,
Knowing chances of cutover hadn't a prayer.

The users were nestled all snug in their beds,
While visions of inquiries danced in their heads.

When out in the lobby there arose such a clatter,
That I sprang from my tube to see what was the matter.

And what to my wondering eyes should appear,
But a Super Programmer, oblivious to fear.

More rapid than eagles, his programs they came,
And he whistled and shouted and called them by name.

On Update! On Add! On Inquiry! On Delete!
On Batch Jobs! On Closing! On Functions Complete!

His eyes were glazed over, his fingers were lean,
From weekends and nights in front of a screen.

A wing of his eye, and a twist of his head,
Soon gave me to know I had nothing to dread.

He spoke not a word, but went straight to his work,
Turning specs into code, then turned with a jerk,

And laying his finger on the ENTER key,
The system came up, and worked perfectly.

The updates, updated; the deletes, they deleted;
The inquires, inquired; and the closing completed.

He tested each whistle, he tested each bell;
With nary an abend, and all had gone well.

The system was finished, the tests were concluded,
The client's last change were even included!

And the client exclaimed with a snarl and a taunt,
"*It's just what I asked for, but it's not what I want.*"

(Anonymous)

2. INTRODUCTION

It is becoming more and more obvious that formal specifications are bound to spread faster and faster in the early stages of the software development process, and therefore influence what will happen downstream of this process, namely the design, programming and maintenance phases.

As a matter of fact, formal specifications improve a great deal of the qualities required in a good software engineering process. As discussed by B. Meyer in [DI-3], rigour and formalism in specification prevents most of the mistakes being usually found in informal requirements. Such mistakes, sometimes referred to as the 7 sins of the specifier, create numerous and costly backtrackings in the development process. The sooner a specification error is detected, the less expensive it is to correct. So the more rigourous and automation-oriented approach implied by the use of formal specifications is an actual need, even if not yet a reality everywhere...

A great deal of research is currently carried on to introduce formal methods further in the software development process, and derive benefit from their expressive and deductive power. This can simplify and automate some part of this process. However, the handling of formal notations and their gradual elaboration from informal requirements remains a heavy and complicated process, reserved to a handful of analysts that have been expensively and lengthily trained. Thus, the necessity of developing tools so as to help them in their arduous task has been emphasized recently, and has given rise to much research work.

Such tools are of two kinds: "logistic" and "intelligent" ones. In the former category, we put graphical editors, report generators and syntactic analysers. In the latter, we find "expert" systems, which assist the specifier in the gradual building of formal specifications and the checking of completeness and consistency properties. These tools, with a greater process orientation, will be based on formal models of the method's steps and heuristics.

A great deal of recent research in formalizing and automating software specification concerns a computer-based specification assistant. For example, Fickas [PA-1,4,8] proposes a knowledge-based system called KATE which goes in that direction. This kind of tool requires models that involve a great deal of investigations in the following fields:

- knowledge representation for the assistant, which is an expert-system
- representation of requirements handled by the assistant, which cannot deal with informal requirements but needs formal specifications (and therefore formal languages)

- acquisition of specifications (through a dialogue with the analyst and/or user), and strategies or methods which are needed for guiding this process

- the semi-automation of further stages in the software development processes, which occur after the specification step. For example, the automatic generation of prototypes, the design of the system architecture, etc...

In this dissertation, we shall not tackle the problems related to knowledge representation, nor propose a new language for formally representing requirements either. However, we shall be interested in the methods for turning informal requirements into formal specifications.

So our work addresses the processes underlying the gradual building of formal specifications, but also the modelling of such processes. This thesis aims at sketching a theoretical background for the formalization of the specification method that guides the specification process of a system.

To achieve this goal, it is necessary to study different empirical methods for building formal specifications in different languages. So we shall first study a few formal languages - namely Z, RML and GIST - thereby acquiring some practice with them. And then a common case study will be treated in all these languages.

So, the first step was to gather some practical experience among analysts who write formal specifications from requirements every day. With this aim in view, we have been on a three-month training period at IBM U.K. Laboratories, where Z has been experimented as a specification language, to a certain extent, in the CICS design group (cfr [RS-1]).

Over there, we have raised some important issues concerning Z with experienced analysts. This has been the major and most interesting insight to our work, because of close contacts with the persons who conceived and practice this language. This was not possible for both RML and GIST, which are the other two languages we have studied afterwards.

This makes up the first part of this thesis, called the PRODUCT LEVEL. It also includes the study of these formal languages according to their expressive and deductive power. This evaluation, besides being very interesting in itself, will also help discovering some general features common to most formal languages. They will be used in our model later on.

As we would like to outline some basic principles for modelling a specification method, several different languages have been chosen. This is necessary for detecting the aspects of a specification method that are inherent in a given language, and those that are independent of it.

Thus, our practical experience gained in formal specifications will provide insight into the specification methods we have followed. This is the subject of the second part of this thesis, called the PROCESS LEVEL.

In a first time, we examine the empirical specification methods we have followed for the building of formal specifications in two languages. So, in both Z and GIST, we try to find out the processes and rationales that we have applied during the stepwise building of the specifications of the common case studies treated. This will be done most informally.

Next, we take a higher-level point-of-view and attempt to sketch a model for describing a specification method. This model first distinguishes different levels in a specification method - for example, the process and rationale levels. And secondly, it also tells apart some aspects of a specification method that depend on a given language and some that do not.

Finally, we make some comments about the work achieved and some prospects for future research.

3. REQUIREMENTS SPECIFICATION: THE PRODUCT LEVEL

3.1 INTRODUCTION

This section studies three formal languages for writing specifications (Z, RML, GIST) through the solving of a common case study (the Library Problem). This will enable us to have both a theoretical and a practical approach to these three languages. We should also gain experience in the knowledge of the process of turning informal requirements into formal specifications. This will be of valuable help for the next section as we shall attempt to find out those processes and rationales that intervened in our specification.

For each of these three languages the same stages will be followed:

> Presentation of an overview of the main features of the language.
>
> Treatment of the Library Problem, or at least of a part of it, in that language.
>
> Assessment of the expressive and deductive power of the language

It is obvious that the quality of our dealing with these three points and the relevance of our conclusions depend to a great extent on the level of practice we have acquired in these formal languages. Our experience in some of them has been somewhat limited, due to both a limited amount of documents at hand and the relative scarcity of knowledge available about these languages in our environment. We have been able to interview people about important issues concerning Z and work with their invaluable help. This was not possible in the case of RML and GIST, for which the only references were papers published by their authors.

Now, it is of some interest to give further details about the third point in our plan, viz. the expressive and deductive power of a formal language. What will be the criteria for the analysis that will be carried out on the three languages?

3.1.1 EXPRESSIVE POWER

The expressive power of a formal specification language is determined by the width of the field of what can be formalized within the limits of this language. But it is not only based upon the possibility of formalizing things within a given language, it also involves the easiness with which that can be done. That means that, if a language allows the description of

certain things, we must also examine more closely if this can be done without resorting to complex and somewhat cumbersome devices.

Thus, for each language, we shall try to answer the following questions: what can or cannot be formalized? what is the price to pay?

In short we shall examine the important issues of the **scope** of the language, its **ability at modelling static and dynamic aspects of requirements**, and **the specialization area** of the language in order to assess its expressive power.

3.1.1.1 The scope of the language

What types of requirements are addressed by the formal language? Requirements are divided into two general categories:

(i) **Functional** requirements, which refer to the services the system is expected to provide [DI-7]. It encompasses both the functions to be automated and the objects of interest to the system. The functions are defined by their arguments, their conditions of applicability and the description of their effect on the system. The objects are defined by the applicable operations that can be associated with them and assertions which constrain them.

(ii) **Non-functional** requirements, which restrict the types of solutions one might consider.

As to informal requirements, one may distinguish [DI-1]:

- *interface constraints*: define the ways the component and its environment interact. A program interacts with the operating system, database management systems and other packages, which provide services. So there is an interface language, with its syntax and constraints which is not taken into account by functional requirements.

- *performance constraints:* are concerned with a broad range of issues dealing with time/space bounds: response time, workload, throughput and available storage space. Performances constraints are increasingly important in specifications, seeing the possibility of simulation and quick prototyping provided by new tools.

- *reliability constraints*: are concerned with both the availability of physical components and the integrity of the information handled by the system.

- *security constraints*: deal with both physical and logical issues such as permissible information flows (e.g. for secure operating systems), and information inference (e.g. from statistical summaries about the database contents).

- *life-cycle constraints*: first, those who favour the better development of a software. Thereby we mean qualities desired for a better system design, which would reduce life-cycle costs: maintainability, enhanceability, portability, flexibility, reusability of components, compatibility. Next the development process is always limited by available resources and time, which are non-functional constraints.

- *economic constraints*: represent considerations relating to immediate and long term costs.

- *political constraints*: deal with policy and legal issues.

These are the non-functional constraints as classified by Roman. However most of them are not expected to be expressible in the formal languages we have practiced. They have been mentioned only for the sake of completeness, and do not play an important role (e.g. the economic and political constraints are too informal to be treated by Z, RML or GIST).

Some techniques limit themselves to functional requirements, others are concerned only with particular non-functional requirements (e.g. reliability) while others cover functionality and a selected subset of non-functional requirements.

3.1.1.2 Static vs. dynamic aspects

The scope of a formal language can also be examined on the basis of another criterion: its ability to model static and dynamic aspects of a system, both its objects and its behaviour.

The questions raised are: What kind of objects can be described in the language? What level of modularity does the language support? Are there powerful abstraction mechanisms such as generalization, aggregation, etc?

Another important issue in the framework of this thesis is the modelling of the temporal behaviour of the system. Can it be done? We shall examine whether or not there exists a possibility of specifying realtime operations or an ordering of the execution of these operations. In other words, we would like to know whether or not it is possible to model time-related questions within the formal language. Not only the *performance constraints* mentioned above - regarding time performance bounds - but also the dynamic scheduling of operations and historical aspects.

3.1.1.3 Domain-specificity vs. domain-independence

The expressive power also involves the determination of *specialization areas*, which are classes of problems for which the formalization in a given language is more appropriate. Current formal languages range from domain specific to domain sensitive and finally domain independent.

3.1.2 DEDUCTIVE POWER

The deductive power of a formal language is the ability to derive consequences from a set of axioms, which are phrases in a given language.

So we shall first have a look at the language **formal foundations**. The deductive power of a language very much relies on the proof theory associated with it.

Can the formal specifications be considered as a theory? Can theorems be deduced from it? Can one check whether a new phrase added to the existing specifications is a consequence of those already written, and so redundant? or in contradiction with them, and so creating inconsistencies?

Another issue related to the deductive power of a language is the *analyzability* of the requirements by mechanical or other means. Are there tools which could be used to analyze the requirements? Does the language lend itself to such a mechanical analysis? This issue can be dealt with more precisely by taking three aspects into consideration:

- the *verifiability* of a specification, i.e. the possibility of formally checking completeness and consistency.
- the *validation* of a specification, i.e. the possibility of submitting the specification to the "customer" for analysis. This includes the *executability* of the requirements. Can simulations and prototypes be constructed in a systematic way from its requirement specifications, prior to starting the design or implementation?
- the *modifiability* of a specification.

3.2 "Z": A FORMAL LANGUAGE AT THE PRODUCT LEVEL

The first sub-section below will be dedicated to an overview of the main features that are usable in Z. In the second one, we shall present the complete handling of a case study. An evaluation of the Z deductive and expressive power will then be built up.

3.2.1 AN OVERVIEW

This section is mainly based on the reports whose references are [DI-3], [ML-1], [ML-3] and [ML-10]. The objectives of this section are not of course to give an exhaustive definition of the Z syntax. We only want to concentrate on the concepts used in the following sections of this document.

3.2.1.1 Specification structure

The structure of a typical Z document is organized as a sequence of informal text followed by some formal text (Z text). The informal text is the translation of the formal one in natural language.

A Z text is composed of a sequence of Z phrases. These phrases refer to variables in order to define or constrain them. A Z phrase can also be a theorem.

The formal texts describe both the data which model the system states and the operations on these states.

The modelling of the system state creates the "data-space". It is generally divided into two parts:

(i) the data which will be manipulated,

(ii) the constraints made on those data.

On the other hand, the modelling of the operations creates the "operation-schemas".

We shall define what a Z schema is further (see p. 13).

3.2.1.2 Definitions in Z

The mathematical concepts used in Z are sets theory, functions, relations, sequences and first order predicate logic (quantifiers and propositions).

Each identifier used in a Z text has to be declared and typed.

Example 1: if sq is the name of a variable,then

$$\{ sq : N \mid 5 < sq < 10 \bullet sq2 \},$$

where "|" stands for "such that" and " •" stands for "for which", is a Z phrase which defines the following set: given sq, a natural number, such that sq is greater than 5 and lower than 10, the set formed of terms sq squared. We have thus defined the set:

$$\{36, 49, 64, 81\}$$

3.2.1.3 Types in Z

All usable types in Z are defined in the *Z Basic Library* by given sets, datatype definitions, or schematype definitions.

Given sets are sets which can be regarded as a parameter of a specification.

Datatype definition introduces a new datatype which is a set and which can be used in declarations.

A schematype is introduced by a schema(1) definition.

Example 2: N, R, Z are standard sets,

Example 3.1: BOOK is a given set.

Example 3.2: A given set definition can also be an identifier-list enclosed by square brackets: [and]. The identifiers enclosed in the brackets are the names of the given sets, i.e. [SUBJECT,AUTHOR,BOOK]

Example 4: MESSAGE ::= 'ok' | 'false' | 'true' | 'fail' is a datatype definition.

We can also build up a type using the powerset mechanism (**P** or **F**) or the cross product mechanism (**X**).

Example 5: if BOOK is the given set which contains all the books issued until today, then ***P*** *BOOK is the set of all the subsets of BOOK. We can then consider the set books* ***P*** *BOOK as the set of books which are in a particular library.*

We can use all the usual mathematical notations to handle sets, e.g. # for the cardinallity of a set, U for union of two sets, ∩ for intersection of two sets, \ for difference of sets and ∅ for empty set.

---

(1) The notion of "schema" is defined in the sub-section "3.2.1.5 The schema notation".

3.2.1.4 The function notation

Another useful Z mechanism is called "function". This one allows the specifier to establish an explicit relation between sets.

When defining a function, we have to specify its name, its domain and its range.

Example 6:

$$talk\_about : BOOK \leftrightarrow SUBJECT$$

In this example, "talk_about" is the name of our function, "BOOK" is its domain and "SUBJECT" is range.

Using this tool, we build sets of ordered pairs where the first element is selected from the function domain and the second from the function range.

A. Kinds of functions

In Z, a variety of notations can be used to accurately characterize the kind of relation existing between two sets. The following table summarizes these notations:

Notation	Meaning
$A \leftrightarrow B$	Relation
$A \rightarrow B$	Total function
$A \nrightarrow B$	Partial function
$A \rightarrowtail B$	Total injection
$A \nrightarrow B$	Partial injection
$A \twoheadrightarrow B$	Total surjection
$A \nrightarrow B$	Partial surjection

where A stands for the function domain and B for its range(1). These notations allow only the specification of binary relations.

Beside these seven types of functions we have the *sequence* type at our disposal. This one is a particular kind of function whose domain is always defined as N (natural number).

We can make use of four predefined sequence operators: **head** (to extract the first sequence element), **last** (to extract the last element of a sequence), **front** (to extract all but the last sequence elements) and **tail** (to extract all but the first sequence elements).

---

(1) Those notions are defined in the next sub-section "Domains and ranges".

Example 7: if we consider tq : seq TANKER where TANKER is a given set then we can consider tq a partial function from N to TANKER and if we need to refer to the penultimate tanker of tq, we proceed like this: interesting_tanker = last (front (tq)).

B. Domains and ranges

Let us consider the following relation: $R : A \leftrightarrow B$

Our work addresses not only the processes underlying the gradual building of formal specifications, but also the modelling of such processes, which should take into account aspects that are dependent and independent from a particular formal language.

The **domain** of R consists of those elements of A that are the first members of the pairs in R.

The **range** of R consists of those elements of B that are the second members of the pairs in R.

We can also use the four following notations in order to construct our specification:

(i)Domain restriction: $A \lhd B$

The result is a relation or a function derived from B by keeping only those pairs whose first members are in set A.

(ii) Domain subtraction: $A \lhd\!\!\!\!- B$

The result is a relation or a function derived from B by subtracting those pairs whose first members are in set A.

(iii)Range restriction: $A \rhd B$

The result is a relation or a function derived from A by keeping only those pairs whose second members are in set B.

(iv) Range subtraction: $A \rhd\!\!\!\!- B$

The result is a relation or a function derived from A by subtracting those pairs whose second members are in set B.

C. The overriding operator

With this operator, the specifier will be able to replace a pair in a relation by another one: he can construct a new relation from an existing one.

Example 8: if the function f is currently defined by

$$f = \{ 001 \mapsto a, 002 \mapsto b, 003 \mapsto c \}$$
and
$$g = \{ 002 \mapsto c, 003 \mapsto e, 007 \mapsto Bond \}$$
then
$$f \oplus g = \{ 001 \mapsto a, 002 \mapsto c, 003 \mapsto e, 007 \mapsto Bond \}$$

The symbol "$\mapsto$" must be read as "maps to" and it is used to consider a special pair of a given relation.

3.2.1.5 The schema notation

When the specifier wants to specify the "data-space" or the "operation-schemas" in Z, he has to elaborate Z schemas(1).

A schema can be displayed either in a vertical (see example 9a) or in a horizontal (see example 9b) format.

Example 9a:

Schema_1

< declaration part >
< predicate part >

Example 9b:

$$Schema\_1 \triangleq [\ < \text{declartion part} > \ |\ < \text{predicate part} > \ ]$$

A Z schema normally consists of two separated parts: the declaration part and the predicate part, but one of these parts may be omitted. The meaning of these parts are different, depending on whether we read a Z schema defining the data-space or a Z schema defining an operation.

Each schema also receives a name.

(1) Each of them defines a specification unit.

A. Schemas and data-spaces

In the declaration part, we find the declaration of every set and function that is needed in our data-space.

Example 10:

$$
\begin{array}{|l}
\hline
PERSON \\
\hline
first\_name : \mathbb{P}\ NAME \\
home\_add : \mathbb{P}\ ADDRESS \\
id\_number : \mathbb{P}\ NUMBER \\
user\_id : NAME \pfun NUMBER \\
\hline
: \\
: \\
\hline
\end{array}
$$

In this example, NAME, ADDRESS and NUMBER are given sets. On the one hand, three subsets have been declared: first_name, home_add and id_number and on the other hand, we specify a total function from NAME to NUMBER: user_id.

On the other hand, the predicate part states all the constraints that we have to define to meet the customer's requirements. Those constraints have to be verified after the execution of every operation. They are the specification invariants of the system.

Example 11: this example works toward the example 10 and constrains the domain and the range of the "user_id" function but also the size of the set "id_number" to a predefined value "max_size":

$$
\begin{array}{|l}
\hline
: \\
: \\
\hline
\#(id\_number) \leq max\_size \\
\mathbf{dom}\ user\_id \subseteq first\_name \\
\mathbf{ran}\ user\_id \subseteq id\_number \\
\hline
\end{array}
$$

A data-space can be used as a schematype. Using this mechanism we can define new variables or sets having all the characteristics of our schematype.

Example 12: if we consider the following schema:

$$
\begin{array}{|l}
\hline
Admit\_value \\
\hline
x : \mathbb{N} \\
\hline
18 < x < 65 \\
\hline
\end{array}
$$

then we can declare "age : ADMIT_VALUE". The value of "age" is defined as a natural number greater than 18 and lower than 65.

B. Schemas and operations

The predicate part will be devoted to the specification of which data-space schemas, which input parameters and which output parameters are needed, in order to execute the operation. If we need local variables, we also declare them in this part.

The second part can be divided into two new parts:

(i) A list of all the preconditions of the current operation. If one of these preconditions is not verified, the operation will not be "executed".

(ii) The predicates ensuring the accomplishment of the operation goals.

Before giving an example, we need to take a few notational conventions:

An undashed variable denotes the value of a variable before the operation execution.

A dashed variable denotes the value of a variable after the execution of the operation.

A variable followed by "?" denotes an input parameter of the operation.

A variable followed by "!" denotes an output parameter of the operation.

If the name of data-space schema is preceded by "Δ" then the operation will modify the value of this data-space.

If the name of data-space schema is preceded by "Ξ" then the operation will not affect the value of this data-space.

Example 13:

Add_a_user

$\Delta User\_file$	(1)
$new\_user? : NAME$	(2)
$mess! : MESSAGE$	(3)
$number\_of\_user < Max\_Users$	(4)
$new\_user? \notin file\_user$	(5)
$file\_user' = file\_user \cup \{new\_user?\}$	(6)
$mess!$ ="Ok_user_added"	(7)
$number\_of\_user' = number\_of\_user + 1$	(8)

Here are some explanations:
- (1) USERFILE is a data-space and the current operation will modify its value if the operation is executed.
- (2) new_user? is the only input parameter of this operation.
- (3) mess! is the only output parameter.
- (4),(5) are the two preconditions. We can see that the number of user should be lower than Max_Users and that the new user should not be in the set file_user. This set has been defined in the schema USERFILE.

- (6) This predicate models the addition of the new user to the user set (file_user).
- (7) This predicate models the way for assigning mess! a value
- (8) This predicate models required modifications to the number of users.

C. Schema operators

Z provides schema operators to build new schemas from existing ones. We shall describe here four useful Z schema operators.

a. Two binary schema operators: conjunction and disjunction

(1) conjunction:

The general notation of this operator:

$$\text{schema\_3} \triangleq \text{schema\_1} \ \wedge \ \text{schema\_2}$$

The effects of this mechanism are:

(i) The definition of the schema named schema_3

(ii) The declaration parts of schema_1 and schema_2 are merged to create the declaration part of schema_3. Duplicated variables are merged. Their types must correspond.

(iii) The predicate parts of schema_1 and schema_2 are joined by an "and" logical operator.

(2) disjunction:

The general notation of this operator:

$$\text{schema\_3} \triangleq \text{schema\_1} \ \vee \ \text{schema\_2}$$

The effects of this mechanism are:

(i) The definition of the schema named schema_3

(ii) The declaration parts of schema_1 and schema_2 are merged to create the declaration part of schema_3. Duplicated variables are merged. Their types must correspond.

(iii) The predicate parts of schema_1 and schema_2 are joined by an "or" logical operator.

b. The schema extension

The general notations of this operator are:

[schema_name | new_predicate] (1)

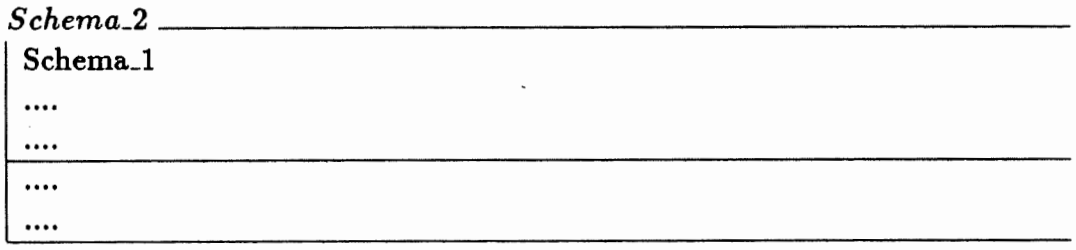
[schema_name ; new_declaration] (2)

Using the notation (1), we can add a new predicate to the predicate part of a schema.

The notation (2) allows a new declaration to be inserted in the declaration part of the schema.

c. The schema inclusion

When defining a schema, we can use existing ones. We only need to include their names in the declaration part of the new schema. So, if schema_1 has been defined in a previous part of a specification, it is possible to define schema_2 as



If we include a schema, e.g. schema_1, in another one, e.g. schema_2, its declaration part and its predicate part are copied in the corresponding part of the new schema.

As a matter of fact, schema_2 inherits each phrase of the declaration and predicate parts of schema_1.

Moreover, thanks to this mechanism, more than one schema can be included in another one. It allows multiple inheritance.

3.2.2 A COMPLETE EXAMPLE

In this part, we present the complete formulation in Z of the well known "Library problem" [EC-6].

After stating the problem, we shall begin our specification with the definition of the data-space. Then, we shall define all the operations that will be found in this case study.

3.2.2.1 Informal specification of "The library Problem"

Consider a small library database with the following transactions:

1. Checkout a copy of a book / Return a copy of a book;
2. Add a copy of a book to the library / Remove a copy of a book from the library;
3. Get the list of books by a particular author or in a particular subject area;
4. Find out the list of books currently checked out by a particular borrower;
5. Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers. Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The database must also satisfy the following constraints:

1. All copies in the library must be available for checkout or be checked out.
2. No copy of the book may be both available and checked out at the same time.
3. A borrower may not have more than a predefined number of books checked out at one time.
4. A borrower may not have more than one copy of the same book checked out at one time.

3.2.2.2 The State-space

A. Predefined elements

Here are given sets of objects that are to be dealt with by the new library system.

$$[SUBJECT,\ BOOK,\ AUTHOR,\ COPY,\ PERSON]$$

Note: The PERSON given set is a generalization of the STAFF and BORROWER sets that had first been spotted in the requirements.

B. The Users

In order to make a clear distinction of who can use what as far as the operations are concerned, two separate sets of persons are created for the purpose of recognizing what type of user is allowed to carry out an operation on the system.

USERS

$$staff: \mathbb{P}\ PERSON$$
$$borrowers: \mathbb{P}\ PERSON$$

$$staff \cap borrowers = \emptyset$$

The invariant part of this schema states that a staff member will not be allowed to use the library services. This separation is merely arbitrary, but it makes our task easier.

C. The main data-space schema

a. Sets, relations and functions

We need to informally define the sets, relations and functions which are required to understand our data-space schema:

copies: this set contains every copy of a book that belongs to the library we want to model. Such a copy can be either borrowed or not.

books: in this set we find the books which can have one or many copies in our library.

authors: in order to be able to issue queries about books, a set containing information about the author has been introduced.

subjects: we have introduced a set containing information about the subjects. This will be useful if we look for all the books corresponding to a subject.

talk_about: this relation will enable us to know which book(s) is related to which subject(s) and conversely.

written_by: this relation will enable to know which book(s) has been written by which author and conversely.

is_a_copy_of: this function maps each copy of a book to the corresponding book in the books set.

on_loan_to: this function maps a copy to its current borrower, if any.

on_last_loan_to: this function is used to map the copy of a book to the last person who has read it before the current borrower. We have to introduce this function since Z do not offer any tools for managing the time and the old states of the system. Otherwise, we could have used the function "on_loan_to".

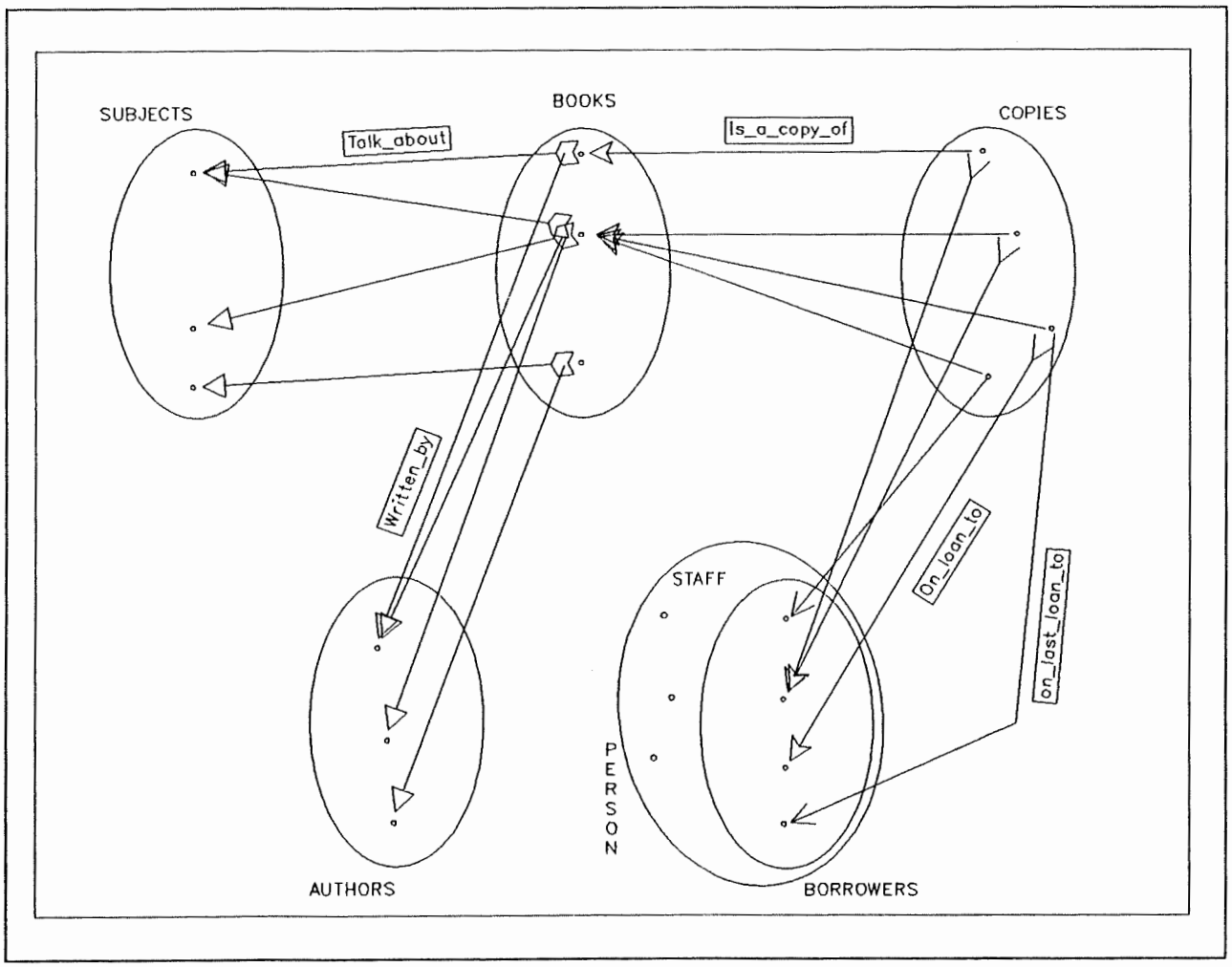


Figure Z.1. The Library Problem and its sets

b. The schema

LIBRARY

$$
\begin{array}{l}
\textit{USERS} \\
\textit{copies}: \mathbb{P}\ \textit{COPY} \\
\textit{books}: \mathbb{P}\ \textit{BOOK} \\
\textit{authors}: \mathbb{P}\ \textit{AUTHOR} \\
\textit{subjects}: \mathbb{P}\ \textit{SUBJECT} \\
\textit{talk\_about}: \textit{BOOK} \leftrightarrow \textit{SUBJECT} \\
\textit{written\_by}: \textit{BOOK} \leftrightarrow \textit{AUTHOR} \\
\textit{is\_a\_copy\_of}: \textit{COPY} \nrightarrow \textit{BOOK} \\
\textit{on\_loan\_to}: \textit{COPY} \nrightarrow \textit{PERSON} \\
\textit{on\_last\_loan\_to}: \textit{COPY} \nrightarrow \textit{PERSON} \\
\hline
\textbf{dom}\ \textit{talk\_about} = \textit{books} \\
\textbf{ran}\ \textit{talk\_about} = \textit{subjects} \\
\textbf{dom}\ \textit{written\_by} = \textit{books} \\
\textbf{ran}\ \textit{written\_by} = \textit{authors} \\
\textbf{dom}\ \textit{is\_a\_copy\_of} = \textit{copies} \\
\textbf{ran}\ \textit{is\_a\_copy\_of} \subseteq \textit{books} \\
\textbf{dom}\ \textit{on\_loan\_to} \subseteq \textit{copies} \\
\textbf{ran}\ \textit{on\_loan\_to} \subseteq \textit{borrowers} \\
\textbf{dom}\ \textit{on\_last\_loan\_to} \subseteq \textit{copies} \\
\textbf{ran}\ \textit{on\_last\_loan\_to} \subseteq \textit{borrowers} \\
\forall\ x : \textit{borrowers} \bullet \#\ (\textit{on\_loan\_to} \rhd \{x\}) \leq \textit{nbr\_max} \\
\forall\ x : \textit{borrowers}\ ,\ \forall\ a,b : \textbf{dom}\ (\textit{on\_loan\_to} \rhd \{x\}) \bullet \\
\quad \textit{is\_a\_copy\_of}(a) \neq \textit{is\_a\_copy\_of}(b)
\end{array}
$$

c. Assumptions and constraints

By lack of proper information, many assumptions have been made:

1. There is a relation called talk_about between the books set and the subjects set. It implicitly states that a book can refer to many subjects and, reciprocally, that a subject can be discussed in many books. We went further on in constraining each book to be at least related to one subject and vice versa (schema constraints 1 and 2). It implies that a subject and a book cannot exist on their own without having at least one relation with a member of the other set. So no operation can ever create or delete a book without caring about its subjects and vice versa.

2.There is a relation called written_by between the books set and the authors set. It implicitly states that a book is written by many authors and, reciprocally, that an author can have written many books. We went further on in constraining each book to be written by at least one author and vice versa (schema constraints 3 and 4). It implies that an author and a book cannot exist on their own without being related to at least one member of the other set. So no operation can ever create or delete a book without caring about its authors and vice versa.

3.There is a function called is_a_copy_of mapping the copies set to the books set. It states that a copy refers to one and only one book (schema constraint 5). How many copies can a book have? No range having been given, our library allows a book to have as many copies as the librarian could ever dream of, including the fact that there could be none (schema constraint 6).

4.There is a function called on_loan_to mapping the copies set to the borrowers set. That is a partial function. It asserts that a copy can either be borrowed by one borrower or be available in the library (schema constraint 7 and 8).

5.We interpreted the terms borrower and last borrower

A borrower is a person different from a staff member.

The last borrower of a copy is the latest person having borrowed it before the current borrower, if this copy is currently borrowed. If not then it's really the last borrower. And if it has never been borrowed, then the copy does not belong to the domain of the function on_last_loan_to. This function shares common features with the on_loan_to function (constraints 9 and 10)

The library problem constraints are dealt with as follows (in order of their appearance, cfr supra):

1.Whether a copy is available for checkout or not, can be traced back by means of looking at whether this copy belongs to the domain of the function on_loan_to or not. Thus, it doesn't require any extra schema constraint. That also means that a copy is available for checkout as soon as it belongs to the copies set, which is when it is added to the library database. There is thus no intermediary state where a copy could be present in the library but not yet available for checkout.

2.A copy of a book cannot either belong to the domain of on_loan_to or not ! Then, once again, no additional schema constraints are needed.

3.Borrowing limitations for a borrower (schema constraint 11). Thus, we need a predefined integer variable if we want to cope with the third constraint of the requirements: "A borrower may not have more than a predefined number of books checked out at one time". $nbr\_max: \mathbb{N}$

4.Borrowing limitations for one book by one borrower (schema constraint 12)

3.2.2.3 Operations Schemas

A. Introduction of the Userid notion

These schemas are intended to introduce the notion of 'User' to the system which will perform some operations only for certain users categories. They will be used as preconditions to subsequent operations that are to be carried out by either staff members or ordinary borrowers, or possibly both.

a. Staff identification

The STAFF_MEMBER operation just checks whether the userid parameter passed to the operation (id?) is the one of a staff member; it fails otherwise.

STAFF_MEMBER

$\Xi USERS$
$id?: PERSON$

$id? \in staff$

The STAFF_FAILURE operation just checks whether the userid parameter passed to the procedure (id?) is not the one of a staff member, it fails otherwise.

STAFF_FAILURE

$\Xi USERS$
$id?: PERSON$
$mess! : MESSAGE$

$id? \notin staff$
$mess! = this\_id\_isn't\_a\_staff\_id$

In case of failure, a message must be sent to the user. We thus need to introduce a datatype: "MESSAGE". It will be defined progressively. Its complete definition is given at the end of the specification.

b. Borrower identification

The BORROWER_MEMBER operation just checks whether the userid parameter passed to the procedure (id?) is the one of a borrower.

And it fails otherwise.

BORROWER_MEMBER

$\Xi USERS$

id?: PERSON

---

id? $\in$ *borrowers*

Note: No need arose for a BORROWER_FAILURE operation since it was not useful to later operations.

B. A useful schema

This schema resulted from the repetition of some predicates in some operations schemas. So they were taken away from them and replaced by including the following schema:

$$
\begin{array}{|l}
\textit{NOT\_MODIFIED} \\
\hline
\textit{talk\_about}' = \textit{talk\_about} \\
\textit{written\_by}' = \textit{written\_by} \\
\textit{books}' = \textit{books} \\
\textit{authors}' = \textit{authors} \\
\textit{subjects}' = \textit{subjects} \\
\textit{borrowers}' = \textit{borrowers} \\
\textit{staff}' = \textit{staff} \\
\hline
\end{array}
$$

C. Check out a copy

$$\textit{CHECK\_OUT} \triangleq \textit{CHECK\_OUT\_A\_COPY\_IF\_NO\_PROBLEM} \lor \textit{CHECK\_OUT\_A\_COPY\_IF\_PROBLEM} \lor \textit{STAFF\_FAILURE}$$

The checkout operation makes use of suboperations described in other schemas and joined through a disjunction operator. When everything is all right regarding input parameters (existing copy, and so on...) then there is only one way to carry out the operation. Otherwise another operation handles the exceptions and a third one deals with non staff members attempts to carry out this operation.

a. Check out no problem

$$
\begin{array}{|l}
\textit{CHECK\_OUT\_A\_COPY\_IF\_NO\_PROBLEM} \\
\Delta \textit{LIBRARY} \\
\textit{STAFF\_MEMBER} \\
\textit{NOT\_MODIFIED} \\
\textit{copy?: COPY} \\
\textit{bor? : BORROWER} \\
\textit{mess! : MESSAGE} \\
\hline
\textit{copy?} \in \textit{copies} \\
\textit{copy?} \notin \textbf{dom}\ \textit{on\_loan\_to} \\
\textit{bor?} \in \textit{borrowers} \\
\#\,(\textit{on\_loan\_to} \rhd \{\textit{bor?}\}) < \textit{nbr\_max} \\
\forall\, c : \textbf{dom}\,(\textit{on\_loan\_to} \rhd \{\textit{bor?}\}) \bullet \\
\qquad \textit{is\_a\_copy\_of}(c) \neq \textit{is\_a\_copy\_of}(\textit{copy?}) \\
\textit{on\_loan\_to}' = \textit{on\_loan\_to} \cup \{\, \textit{copy?} \mapsto \textit{bor?} \,\} \\
\textit{on\_last\_loan\_to}' = \textit{on\_last\_loan\_to} \\
\textit{copies}' = \textit{copies} \\
\textit{mess!} = \textit{ok\_check\_out} \\
\hline
\end{array}
$$

The preconditions of this procedure are that:

1.The checkout operation has been triggered off by a staff member.

2.The copy already exists in the database

3.This copy is not currently borrowed

4.The borrower is also in the database

5.He hasn't yet got either a copy of the book or too many copies at home (< nbr_max)

One effect upon the system is: to extend the domain of the function on_loan_to.

b. Check out with problem

CHECK_OUT_A_COPY_IF_PROBLEM

$$
\begin{array}{|l}
\hline
\Xi Library \\
STAFF\_MEMBER \\
copy?: COPY \\
Bor? : BORROWER \\
mess! : MESSAGE \\
\hline
(copy? \notin copies \wedge \\
mess! = bad\_copy\_identification) \\
\vee \\
(copy? \in \textbf{dom}\ on\_loan\_to \wedge \\
mess! = this\_copy\_is\_already\_on\_loan) \\
\vee \\
(bor? \notin borrowers \wedge \\
mess! = unknown\_borrower) \\
\vee \\
(\#(on\_loan\_to \rhd \{bor?\}) = nbr\_max \wedge \\
mess! = too\_much\_books\_checked\_out\_by\_this\_borrower) \\
\vee \\
(\exists c : \textbf{dom}\ (on\_loan\_to \rhd \{bor?\}) \mid \\
\qquad is\_a\_copy\_of(c) = is\_a\_copy\_of(copy?) \\
mess! = the\_borrower\_has\_already\_a\_copy\_of\_this\_book) \\
\hline
\end{array}
$$

The preconditions of this procedure are that:

1. The checkout operation has been triggered off by a staff member.
2. One of the preconditions 2,3,4,5 of the CHECK_OUT_IF_NO_PROBLEM procedure is not met.

Note: precondition 1 failure will be looked after by the STAFF_FAILURE procedure.

The output of this procedure is a message explaining the sort of problem faced.

D. Return a copy

$$RETURN\_COPY \triangleq RETURN\_A\_COPY\_OF\_A\_BOOK\_NO\_PROBLEM \vee RETURN\_A\_COPY\_OF\_A\_BOOK\_WITH\_A\_PROBLEM \vee STAFF\_FAILURE$$

When everything is alright regarding input parameters (existing copy, and so on...) then there is only one way to carry out the operation. Otherwise another operation handles with the exception treatments (in case of errors) and a third one deals with non staff members attempts to carry out this operation.

a. Return a copy: no problem

$$
\begin{array}{|l}
\hline
RETURN\_A\_COPY\_OF\_A\_BOOK\_NO\_PROBLEM \\
\Delta\, LIBRARY \\
STAFF\_MEMBER \\
NOT\_MODIFIED \\
copy? : COPY \\
bor? : BORROWER \\
mess! : MESSAGE \\
\hline
copy? \in copies \\
bor? \in borrowers \\
copy? \in \mathbf{dom}\ on\_loan\_to \\
on\_loan\_to(copy?) = bor? \\
on\_loan\_to' = \{ copy? \} \lhd on\_loan\_to \\
on\_last\_loan\_to' = on\_last\_loan\_to \oplus \{ copy? \mapsto bor? \} \\
is\_a\_copy\_of' = is\_a\_copy\_of \\
copies' = copies \\
mess! = return\_ok \\
\hline
\end{array}
$$

The preconditions of this procedure are that

1.The return operation has been triggered off by a staff member.

2.The copy already exists in the database

3.The borrower is also in the database

4.This copy has been borrowed

5.This copy has been borrowed by this borrower

The effect upon the system is: to restrain the domain of the function on_loan_to and redefine the function on_last_loan_to for the copy returned.

b. Return a copy with problems

$$
\begin{array}{|l}
\textit{RETURN\_A\_COPY\_OF\_A\_COPY\_WITH\_A\_PROBLEM} \\
\hline
\Xi \textit{LIBRARY} \\
\textit{STAFF\_MEMBER} \\
\textit{copy?} : \textit{COPY} \\
\textit{bor?} : \textit{BORROWER} \\
\textit{mess!} : \textit{MESSAGE} \\
\hline
(\ \textit{copy?} \notin \textit{copies} \land \\
\textit{mess!} = \textit{this\_copy\_doesn't\_exist}\ ) \\
\lor \\
(\ \textit{bor?} \notin \textit{borrowers} \land \\
\textit{mess!} = \textit{unknown\_borrower}\ ) \\
\lor \\
(\ \textit{copy?} \notin \mathbf{dom}\ \textit{on\_loan\_to} \land \\
\textit{mess!} = \textit{this\_copy\_is\_not\_currently\_borrowed}) \\
\lor \\
(\ \textit{on\_loan\_to(copy?)} \neq \textit{bor?} \land \\
\textit{mess!} = \textit{this\_copy\_hasn't\_been\_borrowed\_by\_this\_borrower}\ ) \\
\hline
\end{array}
$$

The preconditions of this procedure are that:

1. The return operation has been triggered off by a staff member.
2. One of the preconditions 2,3,4,5 of the RETURN_IF_NO_PROBLEM procedure is not met.

Note: precondition 1 failure will be looked after by the STAFF_FAILURE procedure.

The output of this procedure is a message explaining the sort of problem faced.

E. Add a copy of a book

$$ADD\_COPY \triangleq ADD\_COPY\_IF\_BOOK\_EXISTS \vee ADD\_COPY\_IF\_BOOK\_NOT\_EXISTS \vee ADD\_COPY\_WITH\_PROBLEM \vee STAFF\_FAILURE$$

When everything is alright regarding input parameters (existing copy, and so on...) then there are two ways to carry out the operation. The first one occurs when the book already exists in the library and the second one when it is a new book. In the latter case, we need to introduce all necessary information regarding this book (authors, subjects).

Otherwise another operation handles the exceptions treatment (in case of errors) and a fourth one deals with non staff members attempts to carry out this operation.

a. Add a copy of an existing book

ADD_COPY_IF_BOOK_EXISTS

Δ *LIBRARY*
STAFF_MEMBER
NOT_MODIFIED
copy? : COPY
book? : BOOK
mess! : MESSAGE

$book? \in books$
$copy? \notin copies$
$copies' = copies \cup \{ copy? \}$
$is\_a\_copy\_of' = is\_a\_copy\_of \cup \{ copy? \mapsto book? \}$
$on\_loan\_to' = on\_loan\_to$
$on\_last\_loan\_to' = on\_last\_loan\_to$
$mess! = ok\_copy\_added$

The preconditions of this procedure are that:

1.The add_copy operation has been triggered off by a staff member.

2.The copy does not already exist in the database

3.The book belongs to the database. Therefore all the information concerning this book (authors, subjects) do not have to be introduced in our database since it has been done previously.

The effect upon the system is to add a copy in the corresponding set and define the corresponding instance of the function is_a_copy_of.

b. Add copy of a non-existing book

ADD_COPY_IF_BOOK_NOT_EXISTS

$$
\begin{array}{l}
\Delta LIBRARY \\
STAFF\_MEMBER \\
copy?: COPY \\
book?: BOOK \\
mess! : MESSAGE \\
subjects?: \mathbb{P}\ SUBJECT \\
authors?: \mathbb{P}\ AUTHOR \\
\hline
book? \notin books \\
copy? \notin copies \\
authors? \neq \emptyset \\
subjects? \neq \emptyset \\
books' = books \cup \{ book? \} \\
copies' = copies \cup \{ copy? \} \\
is\_a\_copy\_of' = is\_a\_copy\_of \cup \{ copy? \mapsto book? \} \\
authors' = authors \cup authors? \\
written\_by' = written\_by \cup \\
\qquad \{ \forall a : authors? \bullet book?\ written\_by\ a \} \\
subjects' = subjects \cup subjects? \\
talk\_about' = talk\_about \cup \\
\qquad \{ \forall s : subjects? \bullet book?\ talk\_about\ s \} \\
on\_loan\_to' = on\_loan\_to \\
on\_last\_loan\_to' = on\_last\_loan\_to \\
staff' = staff \\
borrowers' = borrowers \\
mess! = ok\_added\_new\_book\_and\_first\_copy
\end{array}
$$

The preconditions of this procedure are that:

1.The add_copy operation has been triggered off by a staff member,

2.The copy does not already exist in the database,

3.The book does not belong to the database either,

4.This book must at least have one author and one subject,

The effect upon the system is to add a copy, a book, its author(s) and its subject(s) in their corresponding sets and define the functions is_a_copy_of, written_by, talk_about for them.

c. Problems with the add_copy operation

$$
\begin{array}{|l}
\textit{ADD\_COPY\_WITH\_PROBLEM} \\
\hline
\Xi \textit{LIBRARY} \\
\textit{STAFF\_MEMBER} \\
\textit{copy?: COPY} \\
\textit{book?: BOOK} \\
\textit{authors?: } \mathbb{P}\ \textit{AUTHOR} \\
\textit{subjects?: } \mathbb{P}\ \textit{SUBJECT} \\
\textit{mess! : MESSAGE} \\
\hline
(\textit{copy?} \in \textit{copies} \land \\
\quad \textit{mess!} = \textit{this\_copy\_is\_already\_is\_the\_library}) \\
\lor \\
(\textit{book} \notin \textit{books} \land \textit{copy?} \notin \textit{copies} \land \textit{authors?} = \emptyset \land \\
\quad \textit{mess!} = \textit{this\_book\_has\_no\_author}) \\
\lor \\
(\textit{book} \notin \textit{books} \land \textit{copy?} \notin \textit{copies} \land \textit{subjects?} = \emptyset \land \\
\quad \textit{mess!} = \textit{this\_book\_has\_no\_subject}) \\
\hline
\end{array}
$$

The preconditions of this procedure are that:

1. The add_copy operation has been triggered off by a staff member.
2. One of the preconditions of the ADD_COPY_IF_BOOK_EXISTS or ADD_COPY_IF_BOOK_NOT _EXISTS is not met

Note: precondition 1 failure will be looked after by the STAFF_FAILURE procedure.

The output of this procedure is a message explaining the sort of problem faced.

F. Remove a copy of a book

$$REMOVE\_COPY \triangleq REMOVE\_COPY\_NO\_PROBLEM \lor REMOVE\_COPY\_IF\_PROBLEM \lor STAFF\_FAILURE$$

When everything is alright regarding input parameters (existing copy, and so on...) then there is only one way to carry out the operation. It does not matter whether the copy is the last one of a book or not. Because we assumed that a book can survive in the library database even if no copy refers to it.

Otherwise another procedure handles the exception treatments (in case of errors) and a third one deals with non staff members attempts to carry out this operation.

a. Remove a copy: no problem

REMOVE_COPY_NO_PROBLEM

$\Delta LIBRARY$
STAFF_MEMBER
NOT_MODIFIED
copy?: COPY
mess! : MESSAGE

$copy? \in copies$
$copy? \notin \mathbf{dom}\ on\_loan\_to$
$is\_a\_copy\_of' = \{ copy? \} \lhd is\_a\_copy\_of$
$on\_last\_loan\_to' = \{ copy? \} \lhd on\_last\_loan\_to$
$copies' = copies \setminus \{ copy? \}$
$on\_loan\_to' = on\_loan\_to$
$mess! = ok\_removed$

The preconditions of this procedure are that:

1.The remove_copy operation has been triggered off by a staff member.

2.The copy already exists in the database

3.This copy is not on loan for the time being

The effect upon the system is to remove a copy from the copies set.

b. Trying to remove a non-existing copy

REMOVE_COPY_IF_PROBLEM

Ξ*Library*
STAFF_MEMBER
copy?: COPY
mess! : MESSAGE

(copy? $\notin$ *copies* $\wedge$
mess! = *bad_copy_identification)*
$\vee$
(copy? $\in$ **dom** *on_loan_to* $\wedge$
mess! = *this_copy_is_on_loan_and_cannot_be_removed)*

The preconditions of this procedure are that:

1. The remove_copy operation has been triggered off by a staff member.
2. One of the preconditions 2,3 of the REMOVE_COPY_NO_PROBLEM is not met.

Note: precondition 1 failure will be looked after by the STAFF_FAILURE procedure.

The output of this procedure is a message explaining the sort of problem faced.

G. Get the list of books written by some given author

$$GET\_LIST\_AUTHOR \triangleq GET\_LIST\_AUTHOR\_NO\_PROBLEM \lor GET\_LIST\_AUTHOR\_IF\_PROBLEM \lor STAFF\_FAILURE$$

When everything is alright regarding input parameters (existing author, and so on...) then there is only one way to carry out the operation. Otherwise another operation handles the exception treatments (in case of errors) and a third one deals with non staff members attempts to carry out this operation.

a. Get the list of an author's book

GET_LIST_AUTHOR_NO_PROBLEM

$\Xi LIBRARY$
$author?: AUTHOR$
$books!: \mathbb{P}\ BOOK$
$mess! : MESSAGE$

$author? \in authors$
$books! = dom\ (\ written\_by \rhd \{Author?\}\ )$
$mess! = enquiry\_ok$

The precondition of this procedure is that the author is in the system database

The output of the procedure is an unordered list of books.

b. Problem: The author is unknown

GET_LIST_AUTHOR_IF_PROBLEM

Ξ*LIBRARY*
author?: AUTHOR
mess! : MESSAGE

---

author? $\notin$ *authors*
mess! = *unknown_author*

The preconditions of this procedure are that:

1. The get_list_author operation has been triggered off by a staff member or by a borrower.

2. The author is unknown to the system

The output of this procedure is a message explaining the sort of problem faced.

H. Get the list of books talking about a given subject

$$GET\_LIST\_SUBJECT \triangleq GET\_LIST\_SUBJECT\_NO\_PROBLEM \lor GET\_LIST\_SUBJECT\_IF\_PROBLEM \lor STAFF\_FAILURE$$

When everything is alright regarding input parameters (existing subject, and so on...) then there is only one way to carry out the operation. Otherwise another operation handles exceptions treatment (in case of errors) and a third one deals with non staff members attempts to carry out this operation.

a. Get the list of books: no problem

GET_LIST_SUBJECT_NO_PROBLEM

$$\Xi LIBRARY$$
$$subject?: SUBJECT$$
$$books!: \mathbb{P}\ BOOK$$
$$mess! : MESSAGE$$

$$subject? \in subjects$$
$$books! = dom\ (talk\_about \rhd \{subject?\}\ )$$
$$mess! = enquiry\_subject\_ok$$

The precondition of this procedure is that the subject is in the system database

The output of the procedure is an unordered list of books.

b. Problem: The subject is unknown

$$
\begin{array}{|l}
\textit{GET\_LIST\_SUBJECT\_PROBLEM} \\
\hline
\Xi \textit{LIBRARY} \\
\textit{subject?}: \textit{SUBJECT} \\
\textit{mess!} : \textit{MESSAGE} \\
\hline
\textit{subject?} \notin \textit{subjects} \\
\textit{mess!} = \textit{unknown\_subject} \\
\hline
\end{array}
$$

The preconditions of this procedure is that the subject is unknown to the system

The output of this procedure is a message explaining the sort of problem faced.

I. Find out the list of book copies currently checked out by a borrower

$$FIND\_OUT \triangleq FIND\_OUT\_BY\_STAFF \lor FIND\_OUT\_PROBLEM \lor FIND\_OUT\_BY\_BORROWER$$

When everything is alright regarding input parameters (existing borrower, and so on...) then we must consider the userid of the person who is currently carrying out this operation. If he is a staff member then he has got access to any list of books borrowed by any borrower. If he is a borrower then he may just have a look at his own checkout list.

Otherwise a third procedure handles the exceptions treatment (in case of errors) when it is used by a staff member. The operation always succeeds when carried out by a borrower because we take his userid, thanks to which he is issuing this query, and then we make sure he does not try to look into the library database beyond his privileges, by giving him his checkout list without asking for another id.

Note: It would seem more natural to understand the requirements for this operation as producing a list of copies borrowed by a person, because we decided to make a difference between the concepts of copy and that of book.

But we made up our minds to stick to the requirements and deliver a list of books. Since a borrower cannot have more than one copy of a book, it yields a similar result anyway.

a. The list of books by a staff member

FIND_OUT_BU_STAFF

$$\Xi LIBRARY$$
$$STAFF\_MEMBER$$
$$borrower?: BORROWER$$
$$books!: \mathbb{P}\ BOOK$$
$$mess! : MESSAGE$$

$$borrower? \in borrowers$$
$$books! = \mathbf{ran}\ (\ \mathbf{dom}\ (on\_loan\_to \rhd \{borrower?\}\ ) \lhd is\_a\_copy\_of\ )$$
$$mess! = enquiry\_book\_borrower\_ok$$

The preconditions of this procedure are that:

1.The find_out operation has been triggered off by a staff member.

2.The borrower is in the system database

The output of the procedure is an unordered list of books, the borrower of which is given in the inputs

b. The list of books by a borrower

FIND_OUT_BY_BORROWER

$\Xi LIBRARY$
BORROWER_MEMBER
books!: $\mathbb{P}$ *BOOK*
mess! : *MESSAGE*

---

books! = **ran** (**dom** (*on_loan_to* $\rhd$ {*id?*}) $\lhd$ *is_a_copy_of*)
mess! = *enquiry_book_borrower_ok*

The precondition of this procedure is that the find_out operation has been triggered off by a borrower

The output of the procedure is an unordered list of books borrowed by the person who issued this query

c. Problem: The borrower is unknown

FIND_OUT_PROBLEM

$\Xi LIBRARY$
STAFF_MEMBER
borrower?: *BORROWER*
mess! : *MESSAGE*

---

borrower? $\notin$ *borrowers*
mess! = *unknown_borrower*

The precondition of this procedure is that:

1.The find_out operation has been triggered off by a staff member.

2.The borrower is unknown

The output of this procedure is a message explaining the sort of problem faced.

J. Find out the last borrower of a given book copy

$$FIND\_OUT\_LAST\_BORROWER \triangleq FIND\_OUT\_LAST\_NO\_PROBLEM \vee FIND\_OUT\_LAST\_IF\_PROBLEM \vee STAFF\_FAILURE$$

When everything is alright regarding input parameters (existing copy, and so on...) then there is only one way to carry out the operation. Otherwise another operation handles the exceptions treatment (in case of errors) and a third one deals with a non staff member attempt to carry out the operation.

a. Find out the last borrower if he (or she) exists

FIND_OUT_LAST_NO_PROBLEM

$\Xi LIBRARY$
STAFF_MEMBER
copy?: COPY
borrower!: BORROWER
mess! : MESSAGE

---

copy? $\in$ *copies*
copy? $\in$ **dom** *on_last_loan_to*
borrower! = *on_last_loan_to(Copy?)*
mess! = *enquiry_last_borrower_ok*

The preconditions of this procedure are that:

1.The find_out_last operation has been triggered off by a staff member.

2.The copy is in the system database

3.This copy has already been borrowed at least once in its life

The output of the procedure is: an unordered list of books.

b. Find out last borrower if something is going wrong

FIND_OUT_LAST_IF_PROBLEM

Ξ*LIBRARY*
STAFF_MEMBER
copy?: COPY
mess! : MESSAGE

(copy? $\notin$ *copies* $\wedge$
mess! $=$ *unknown_copy)*
$\vee$
(copy? $\notin$ **dom** *on_last_loan_to* $\wedge$
mess! $=$ *this_copy_has_never_been_borrowed)*

The preconditions of this procedure are that:

1. The find_out_last operation has been triggered off by a staff member.
2. The precondition 2 or 3 of the FIND_OUT_LAST_NO_PROBLEM procedure is not met.

Note: precondition 1 failure will be looked after by the STAFF_FAILURE procedure.

The output of this procedure is: a message explaining the sort of problem faced.

K. The MESSAGE data-type

We are now able to define the MESSAGE data-type:

MESSAGE :: = *"this_id_is_not_a_staff_id"* |
"ok_check_out" | *"bad_copy_identification"* |
"this_copy_is_already_on_loan" | *"unknown_borrower"* |
"too_much_books_checked_out_by_this_borrower |
"the_borrower_has_already_a_copy_of_this_book" |
"return_ok" | *"this_copy_does_not_exist"* |
"this_copy_has_not_been_borrowed_by_this_borrower" |
"ok_copy_added" | *"ok_added_first_copy"* |
"this_copy_is_already_in_the_library" |
"ok_removed" | *"this_back_has_no_author"* |
"this_book_has_no_subject" | *"bad_copy_id"* |
"enquiry_ok" | *"unknown_subject"* |
"enquiry_book_borrower_ok" | *"unknown_copy"* |
"this_copy_has_never_been_borrowed"

3.2.3 EVALUATING Z

In this section, we propose an evaluation of the Z language based on its expressive and deductive powers.

3.2.3.1 Expressive Power of Z

A. Scope of the language

Z can only be used by a specifier to model the functional aspects of the requirements.

So, if the analyst has to specify non-functional aspects of a problem, he can only express them in natural language and insert them into the formal specification.

This is especially true for the non-functional constraints dealing with the environment of the future system (Interface, performance, reliability, security, life-cycle, economics and political constraints).

B. Static and dynamic aspects

a. Dynamic modeling in Z

There is no standard Z feature for specifying a dynamic behaviour of the system. It is, for example, impossible to model explicitly that an operation A has to be executed before (or after, or in parallel with) another operation B.

Nevertheless, the specifier can use an artificial trick which consists in introducing a token (tok!). This token is assigned a special value (v1) in the operation A and the precondition of the operation B includes a test over the token value (tok? = v1). Thus, if the value of the token is not equal to v1, the operation B will not start.

Hence the two following schemas patterns:

A

....
$tok! : TOKEN$
....

....
....
$tok! = v1$

B

....
$tok? : TOKEN$
....

$tok? = v1$
....
....

Nonetheless, two weak points remain:

(i) when one needs to specify that the operation B cannot start before the operation A has been triggered for 10 minutes. This is closely related to the poor ability of Z to specify real-time problems. (see "Real time or historical aspects")

(ii) it is also very difficult to specify that an operation must (or can) be triggered every time an event or a special state transition occurs. This is closely related to the non-existence of modal operators in Z.

b. Static modeling in Z

It is possible to define all real world objects using the Z typing feature.

The basic types we have at our disposal are the given sets. They are sets whose values are to be determined at a later stage of the specification. Thus, they can be considered as parameters of the specification.

In the "Library problem", we have five such types: SUBJECT, BOOK, AUTHOR, COPY, PERSON. Beside given sets, we also have N, Z, R, and the data-types.

An example of data-type has been given in the "Library problem" when we have defined the MESSAGE type.

But usually, one needs to create more complex objects by using the schematype mechanism. If one creates a new schematype in Z, its internal semantics will be richer than the semantics of an Entity type in the Entity-Relationship Model because:

(i) its attribute types can be not only sets, sequences, relations, function but also other schematypes.

(ii) we are able to add constraints (expressed in formal language) into the schematype itself.

(iii) we can use mechanisms like aggregation and decomposition, specialisation and generalisation, classification. (see second part of this thesis)

In the "Library Problem", we have an important example of such features: the schema named "Library".

Thanks to the inclusion or conjunction of schema names, a schema can also inherit of sets, sequences, relations, functions and predicates which are defined in the schematype included.

c. The association concept

We also have a mechanism in Z which gives us the ability to establish an association between two Entity Types easily. We use a mathematical relation for that purpose.

The domain and range constraints of a relation define the connectivity of the relation.

d. Real time or historical aspects

In Z, there is no standard feature which provides the analyst with the possibility of handling time and historics in a natural way.

For real time problems, the analyst must create a new object which receives the CLOCK role, e.g.,

$$\begin{array}{|l} CLOCK \\ \hline time : \mathbf{N} \\ \hline \end{array}$$

An operation to initialize the clock and another one to update the time value have to be modelled. We have, for instance,

$$\begin{array}{|l} INIT\_CLOCK \\ \hline CLOCK' \\ \hline time' = 0 \\ \hline \end{array}$$

and,

$$\begin{array}{|l} TICK\_TOCK \\ \hline \Delta CLOCK \\ \hline time' = time + 1 \\ \hline \end{array}$$

Then the following Z phrases have to be included in each operation which uses real-time references:

$$\Xi CLOCK$$

On the other hand, for the specification of historics, the analyst must define a "complex" schematype like:

$$\begin{array}{|l} Hist\_data \\ \hline useful\_data : \mathbf{P}\ DATA \\ time : \mathbf{N} \\ \hline \end{array}$$

where time will record when the useful_data set has been recorded in the database. With this schematype we can create a new sequence:

$$recorded\_data : \mathbf{seq}\ Hist\_data$$

The analyst has to define all the operations for manipulating this sequence, because time management primitives are not available in Z.

C. Specialization areas

Z could be classified as a domain-independent formal languages.

However, because of its poor ability to express time related operations, the use of Z for specifying problems whose requirements do not involve time, events, parallelism,... should be limited.

Moreover, as Z is based on first order predicate logic, we can find other difficulties:

(i) it is impossible to define parameterized operations,

(ii) modalities, likes "possibility", "obligation", "interdiction",... can only be translated heavily.

D. New objects or relations

In this sub-section, we shall discuss the problem of introducing new objects into the formal specification, which did not appear in the informal one.

As a matter of fact, using the Z specification language, the specifier may have to introduce new objects or relations that did not appear in the informal specification.

As seen before, whenever the specifier has to specify a problem that needs a time counter, an object 'CLOCK' must be defined. We also need to define an object 'TOKEN' when we want the execution of an operation A to occur before the execution of an operation B.

But other types of things in Z could also be added:

(i) relation: in the "Library Problem", a function named "on_last_loan_to" has been added, although the function "on_loan_to" could have been more adequate. But since Z specifies only a snapshot of the system life, the introduction of this new function, whose purpose is to find a trace of the system past state, is needed.

(ii) schema: in the "Library Problem", the schema "NOT_MODIFIED" was introduced to simplify the other operation-schemas: the "visible" part of a schema being reduced to its most important components.

3.2.3.2 Deductive Power of Z

A. Type of deduction

The syntax used in Z is mainly based on predicate logic with sets and relations. This allows the specifier to deduce a great deal of information about the informal requirements from the formal specifications.

A Z formal specification can be considered as a theory where each Z schema is an axiom and has to respect every constraint defined in the data-space. Let us take an example from the "Library Problem":

We can prove that no constraints of the data-space are violated after the execution of the procedure "ADD_COPY_IF_BOOK_EXITS":

(i) constraints (1) to (4) and (7) to (12): OK!
(see "NOT_MODIFIED", "on_loan_to' = on_loan_to" and "on_last_loan_to' = on_last_loan_to");

(ii) constraints (5):

we must prove that **dom** is_a_copy_of' = copies'.
We have:

dom is_a_copy_of' = **dom**[is_a_copy_of **U** {copy?↣book?}]
dom is_a_copy_of' = **dom**[is_a_copy_of] **U** {copy?}
dom is_a_copy_of' = copies **U** {copy?}
dom is_a_copy_of' = copies'. QED

(iii) constraints (6):

we must prove that **ran** is_a_copy_of' books'.
We have:

ran is_a_copy_of' = **ran** [is_a_copy_of **U** {copy? ⤔ book?}]
ran is_a_copy_of' = **ran**[is_a_copy_of] **U** {book?}
ran is_a_copy_of' ⊆ books **U** {book?}
ran is_a_copy_of' ⊆ books
ran is_a_copy_of' ⊆ books'. Which was to be proved.

Theorems can thus be inferred deductively in the classical framework of first order logic with equality. Let us take an example:

For all c **in** copies,
exists b: BOOK | b = is_a_copy_of (c)

can be inferred from

dom is_a_copy_of = copies.

Moreover, the reader can understand the meaning the specifier has given to a part of the informal specification only by looking, for example, at a relation and at the definition of its domain and range in the data-space definition.

Let us examine the relation "is_a_copy_of" of the "Library Problem".

It is a partial function from COPY to BOOK. So, we consider only the copies of a particular library and certainly not the copies of all the libraries in the world!

It can also be deduced that each copy is related to one and only one book (see constraints on the function domain). On the other hand, a given book can be linked to zero, one or more copies (see constraint on the function range).

This last establishment could give rise to the specification of a new operation: ADD_A_NEWLY_PUBLISHED_BOOK.

Its effect is the insertion of some information about a book which has been recently published, even if there is no copy of it on the shelves of the library.

As seen for the incompleteness problem, the specifier can also deduce the preconditions of an error handler for a function from the preconditions of this operation.

B. Analizability of the formal specifications

a. Verifiability of a specification

Since a formal specification in Z can be viewed as a theory, it offers a background for the formal solving of both the inconsistency and incompleteness problems.

There are several kinds of tools one could think of. The first of them could be an automated tool for syntax checking. For example, it should be able to verify that each variable has been defined and typed according to the Z syntax.

A second level of consistency verification can be achieved. As seen before, the mathematical concepts used in Z are sets, relations and first order predicate logic. Thus, a computerized tool could also be used to verify that every Z schema is consistent and does not bring contradictions with regard to the rest of the specifications. One could, for example, check that the postconditions of a given schema do not violate a constraint of the data-space.

A tool checking completeness can also be considered. Among other examples, it should be capable of spotting missing postconditions.

b. Validation of a specification

To validate a Z specification, the specifier can use two methods:

(i) "Customer" analysis

Every canonical Z specification is made up with formal and informal text. The latter helps the "customer" to understand the specifications. Nevertheless, the specifier's presence remains necessary, because we are not convinced that Z specifications can be easily read by unaccustomed people.

(ii) Executability

Up to now, an automatic processor, which could compile a Z document in order to produce a prototype of the future information system, does not exist.

Nevertheless, a method for deriving procedures in Dijkstra's guarded command language from a Z specification exists. These procedures can obviously be easily translated in a Pascal-like language and be used as a prototype of the future system. (see [ML-4])

c. Modifiability of a specification

If the customer modifies his informal requirements (during or after the formalisation process), the specifier will be able to find out the modifications he has to do.

The complexity of these modification(s) is highly model-dependent: the specifier may have to start it all over again, or just proceed to some little alterations locally.

The specifier will have to scrutinize the set(s), the relation(s) and the operation(s) affected by the new customer needs. A good modularization of Z schemas can help the specifier in this task, because modifications may be located in some sub-schemas only. Thanks to the inheritance mechanism(1), these localized schema modifications will have consequential effects upon the entire specification.

Let us suppose that the customer of the "Library Problem" is not pleased with a system which memorizes only the last book checked out by a particular borrower. He may want a more powerful system that would record all the books which have been checked out by a particular borrower. The fifth transaction of the "Library Problem" could then be changed into:

"Give the list of borrowers who have already checked out a given copy of a book."

First the data-state must be modified because the partial function "on_last_loan_to" is not adequate any more: it needs to be redefined as a relation:

$$on\_last\_loan\_to : COPY \leftrightarrow PERSON$$

If one takes a look at the invariant, one will be able to see that nothing must be changed. The domain and the range of "on_last_loan_to" are already correctly specified.

Then, the operations affected by the data-space modification have to be reviewed. The first operation that uses this new relation is "RETURN_A_COPY_OF_A_BOOK_NO_PROBLEM": only the following Z sentence is replaced:

$$on\_last\_loan\_to' = on\_last\_loan\_to \oplus \{copy? \mapsto bor?\}$$

by the sentence:

$$on\_last\_loan\_to' = on\_last\_loan\_to \cup (copy?, bor?)$$

(1) Due to schemas inclusion, conjunction or disjunction.

The second operation that uses this new relation is "FIND_OUT_LAST_NO_PROBLEM". The result to produce is a list of borrowers, so in the declaration part, one will find:

$$borrower! : \mathbf{P}\ BORROWER$$

instead of

$$borrower! : BORROWER$$

According to the definition of "on_last_loan_to" one does not need to modify the predicate part of this schema. Nevertheless, to be consistent with the requirement of our customer, we should have to modify the name of our operation in "FIND_OUT_ALL_NO_PROBLEM".

3.3 A SHORTER ANALYSIS OF TWO OTHER SPECIFICATION LANGUAGES

After this detailed analysis of the Z language, we shall now approach two other formal languages for specifying requirements: RML and GIST. This analysis will only be based on the documents we have had at our disposal.

3.3.1 THE RML LANGUAGE

3.3.1.1 Introduction

This section is based on two papers by Greenspan [ML-6] and [ML-13]. It should be noted that the syntax and semantics of RML is somewhat different in the two papers. The second of these [ML-6] presents a revised but shorter version of the language in which some important features have disappeared. And some very interesting new possibilities are not explained enough for us to build a practical example of a RML specification as satisfactorily as well-trained RML specifiers.

Due to incomplete semantics and syntax we have been obliged to interpret some RML features our own way. So we cannot guarantee the correctness of these specifications, nor check it against a formal theoretical background of the language (like in Z for example). But it nonetheless remains a valuable exercise in elaborating formal specifications. Even though our way of approaching the language may be different from its designers, it is still representative of a formal language. We have tried to be consistent in our interpretations of the language, so that the analysis of specification building processes can still be carried out.

Just like in Z, we shall present an overview of the language features first, but with an application to the Library Problem at the same time. This mixed approach has been chosen on both efficiency and space grounds. Finally, we shall discuss the issues of expressive and deductive power of RML.

3.3.1.2 An overview

According to the authors, the most important thing regarding requirements is that they must capture our understanding of the environment within which the proposed system will function. They believe that this information is most appropriately presented in the form of a model of the real world, or more precisely our knowledge of the world.

Consequently, the constructs of the language have their intellectual roots in Artificial Intelligence research on the representation of knowledge, specifically on ideas used by

semantic networks and frame-based representation languages as well as object-centred languages such as Simula and Smalltalk.

The actual features of the RML language have been inspired by a number of basic yet powerful principles:

- a good modelling language must allow the representation of data to be manipulated by the system, the representation of operations that turn input data into output data, and the representation of various constraints on both operations and data. That has led to the modelling of specifications thanks to three main concepts: **entity, activity** and **assertion.**

- uniformity is an important characteristic of a language which is easy to learn and use. For this reason, RML adopts an object-centered view, where all information is recorded in terms of objects, inter-related by properties and grouped into classes;

- in order to structure large, complex descriptions, RML supports a structured organization based on widely used abstraction principles: aggregation, classification and generalization. Again, it applies to all three kinds of information;

- a fourth principle is to make it easier to state frequently occurring expressions and constraints.

A fundamental principle of RML is that everything that is to be described is an object, so that a world model consists of a collection of object descriptions. RML distinguishes entity, activity and assertion objects in order to help modelling different kinds of things in the world.

Such objects have relations between them. The latter can be:

(i) functional relations: binary functions mapping an object to another one are expressed through the notion of **property** or **attribute**. This means that the object belonging to the range of the function will be stated as a property of the object belonging to the domain of the function. An example can be found in Figure RML.01 where there exists a function from "BOOKS ⟶ NAMES" which is expressed by a property "name: NAMES" of BOOKS.

Other non binary functions will be expressed through **assertions**, which are predicates binding variables that are objects.

(ii) abstract relations: these are relations arising from abstraction mechanisms. We mean that there is a relation between a class of objects and a "specialisation" of it, etc... There are three abstract relations in RML: generalization, classification, aggregation.

IS_A enables the organisation of classes of objects into a hierarchy of subclasses and superclasses, with inheritance of properties down the hierarchy.

IN indicates that an object or a class of objects is a member of a class or meta-class of objects. This amounts to *Instance_of* relationship.

CLASS is a classification mechanism, it allows one to group objects that share common properties into a class and to talk about the class without mentioning any actual instance of it. It amounts to a typing mechanism.

Aggregation allows one to view an object as a collection of the objects to which it is related by properties.

In addition to being able to relate objects , we also need the ability to express constraints on the possible relationships between objects if we are to provide accurate models of the real world, distinguishing it from other possible worlds. Assertions will of course be suitable for this task, but we shall present a number of other ways of expressing special kinds of constraint. Their goal is to make it easier to state commonly occurring situations, and to make descriptions shorter and intelligible.

A. Entity Modeling

Information about entities, and about all other kinds of objects, is presented through class definitions expressing facts about their instances. This looks in many ways like semantic data modelling, but in requirements modelling there are concepts that are described even though they will never be actually implemented (ex: definition of "human", "time").

Let us illustrate entity modelling with a little example: we would like to model the fact that every book has one name, one author and may have one subject. This is accomplished by defining a class of objects, BOOKS, each instance of which has properties with suitable identifiers and restricted ranges of values.

```
entity class BOOKS with
        necessary part
                name: NAMES
                author: AUTHORS
        association
                subject: SUBJECTS
```

Figure RML.01 A class definition

In addition to giving the name or identifier of each property (name, author, subject), the specification of a property in a class definition introduces one or more constraints on the values of this property. First, they define a range for them. Second, a property can also be specified to belong to one or more property categories, which appear in bold face as prefixes to lists of properties. Each property category acts as a qualifier describing in more details some aspects of the functionality or the property.

In the above example, the range of name is specified to be the class NAMES and the property category **necessary** states that every object in the class must have a value for the property. The convention is that if a property does not have a value, it will be said to have the special value null.

Property categories provide a concise way of stating certain constraints which form part of the semantics of the relationship expressed by a property.

Here is a list of predefined property categories available in RML for all objects (entity, activity, assertion)

- **part**: property value is a component of the object and does not change with time
- **necessary**: property value cannot be null
- **initially**: property value cannot be null at the 'birth' of the instance in the class
- **finally**: property value cannot be null at the 'death' of the instance in the class
- **association**: property value is an entity object

It could be interesting to show, thanks to a table, what type of objects can be related to what other types of objects. In other words, what objects can be properties of what other objects. To achieve this goal, we shall put property categories in columns and objects in rows, so that we can tell at a glance what can be related to what and through which category. A

hyphen means that the semantics of the language is not so definite as to tell whether or not this property category is possible or makes sense for a certain object(1).

	part	**necessary**	**initially**	**finally**	**associa.**
Entity	Ent	Ent/Ass	Ent/Ass	Ent/Ass	Ent
Activity	Act	-	Ass	Ass	-
Assertion	Ass	Ass	-	-	-

Of course each object has also its own property categories which do not appear in this table. But, as they will be explained in detail later, it was not necessary to introduce them at this stage.

Here is a list of predefined property categories available in RML for entities:

- **producer**: property value is an activity that creates an object, new instance of the class

- **consumer**: property value is an activity that destroys an instance of the class

- **modifier**: property value is an activity that affects the relationship in which the instance participates but not its membership in the class

The Library problem provides a great deal of examples to illustrate these concepts. So we shall start to model entity classes right now. This will enable us to illustrate most of the features presented above.

```
entity class PERSON with
    necessary part
        name: NAME
        categ_person:
                        {'Staff,'Borrower,'Author}
```

Figure RML.02 A RML "necessary part" illustration

The entity class PERSON has two properties of the same category, the name which consists of two predefined category names (**necessary** and **part**). That means that the values for these properties can neither change, nor be null. So a person must have one name and nobody can change it. Second, a person must be an author or a staff member or a borrower,

(1) "Ass" stands for "assertion" and not "association".

not two or more of these. It is interesting to notice that a property is a function, and therefore maps an object to one and only one other object. In order to express non-binary relationships between entities, we should have had to resort to using assertions instead of properties.

The next three classes are specialization of the PERSON entity class, through the use of the well-known "IS_A" mechanism, available in RML too. Inheritance of properties down the hierarchy is supported.

```
entity class AUTHOR is_a PERSON

entity class STAFF_MEMBER is_a PERSON with
        necessary unique part
                staff_id: STAFF_ID
```

Figure RML.03 The "is_a" RML mechanism

This shows how classes can be related to each other by the IS_A relation. 'Subclasses' may have additional constraints on properties of the 'superclass', and may have other properties (or attributes) applicable to them. For example, an identifier has been added to STAFF_MEMBER. But no further information about AUTHOR has been considered useful here, in the context of the requirements stated above (in the chapter about Z).

```
entity class BORROWER is_a PERSON with
        necessary unique part
                borrower_id: BORROWER_ID
        necessary association
                nbr_copies: NUMBER
        initially finally
                no_copies_borrowed?: (nbr_copies = 0)
        modifier
                increment: CHECK_OUT_COPIES
                                (copy_borrower = this)
                decrement: RETURN_COPIES
                                (copy_borrower = this)
```

Figure RML.04 The is_a RML mechanism

In order to satisfy the Library Problem constraint on the maximum number of copies that can be checked out by one borrower, a counter for check-outs has been added to each entity of the BORROWER class. The value of this counter will be updated by two activities, CHECK_OUT_COPY and RETURN_COPY, which will increment and decrement it. These activities are those properties of the entity class BORROWER which appear in the **modifier** category.

An entity can also be related to assertions, which are objects too in RML. Two assertions have been added, the first one in order to ensure that the counter is initialized at the creation of an entity, and the other one to make sure that BORROWER's cannot be deleted from the system if they still have copies checked-out. As the predicates of these assertions are the same, only one of them has been left. It belongs to both categories **initially** and **finally**. The reader will find more complete information about assertions in section 3 of this chapter.

```
entity class SUBJECT with
        necessary unique part
                keyword: KEYWORD


entity class BOOK with
        necessary unique part
                book_id: BOOK_ID
        necessary part
                title: TITLE
                authors: class_of AUTHOR
                subjects: class_of SUBJECT
        producer
                addition: ADD_BOOK (book = this)
```

Figure RML.05 Subject and book declaration

A BOOK is related to a TITLE, a class of AUTHOR and a class of SUBJECT.

```
entity class COPY with
        necessary unique part
                copy_id: COPY_ID
        necessary part association
                of_book: BOOK
        association
                borrowed_by: BORROWER
                last_borrowed_by: BORROWER
        initially
                unborrowed?: (borrowed_by = null) and
                        (last_borrowed_by = null)
        producer
                addition: ADD_COPY (copy = this)
        consumer
                deletion: REMOVE_COPY (copy = this)
        modifier
                return: RETURN_COPY (copy = this)
                checking_out:
                        CHECK_OUT_COPY (copy = this)
```

Figure RML.06 The copy declaration

A COPY has an identifier COPY_ID, it is related to one and only one book (*of_book*). It can be related to one borrower (*borrowed_by*). So if the value of this property is **null**, then it

simply means that the copy is not borrowed at the moment. There is also another relation, named *last_borrowed_by*, the purpose of which is to indicate the latest person that borrowed the copy before the current borrower.

At the creation of an instance in the class, an assertion must be verified (the one in the category **initially**). The latter asserts that at the instant a copy exists in the library, it is available for check-out. This was not explicitly stated in the informal requirements, but assumptions have to be made when silences arise from these ones (This is one of the reason for which formality helps building more accurate and sturdy specifications than the informal method)

An interesting feature is presented in this example: the activities creating, deleting or modifying an entity of the class are mentioned explicitly in properties of the entity. Several categories are available. First the **producer** category allows the specifier to explicitly state those activities that create a new entity. Likewise, the **consumer** category is for those activities that delete a new entity. And finally, the **modifier** category lists those activities that modify one or more relationships (property values) of the entity, without creating or deleting it.

This example also introduces another technique for stating constraints, namely *property binding*, which looks a little like parameter binding for procedures in programming languages. It is used to further specify the relationship of an object and its properties, by restricting some of the attributes or property values. To illustrate this, let us take the property *return* of the entity class COPY. Its value is an activity of the class RETURN_COPY, whose property *copy* will be the instance of the entity class COPY modified by this activity(1). Thus, this is a proper way of specifying which properties of an activity are "bound" (constrained) when that activity is related to some other object.

---

(1) This refers to prototypical instances of the class being defined, and will be viewed as a variable ranging over this class.

B. Activity Modelling

One can find in [ML-13] a definition of what an activity is in RML. An activity is used to represent something that happens in the world, something which has a start and end times. It captures information about events in the world and usually has as instances at any moment of time events which are taking place then.

Events are related by properties to other events (e.g. component activities which must occur as parts of this occurrence of an activity). Events are also related to entities participating in them, and to assertions constraining them.

Here is a list of additional property categories available in RML, specifically for activities:

- **input**: property value is an entity that participates in the activity being defined and is of interest at the start time of the activity; it is removed by this activity from its property value class

- **output**: property value is an entity that participates in the activity being defined and is of interest at the end time of the activity; it is added by this activity into its property value class

- **control**:property value is an entity that affects the activity being defined but whose properties and relationships are not altered by this activity

- **actcond**:property value is an assertion which becomes true at a point in time if and only if an instance of the activity being defined begins at that point in time

- **stopcond**:property value is an assertion which becomes true at a point in time if and only if an instance of the activity being defined ends at that point in time

Let us see how this all works on examples from the Library Problem.

```
activity class CHECK_OUT_COPY with
        input output
                copy: COPY
                copy_borrower: BORROWER
        control
                user: PERSON
        initially
                authorized?:
                        (categ_pers of user = 'Staff)
                not_borrowed_yet?:
                        (borrowed_by of copy = null)
```

```
not_same_book_and_borrower?:
                    (not exists x in COPY)
                    such that
                            (borrowed_by of x = copy_borrower)
                    and  (of_book of x = of_book of copy)
            under_maximum?:
                    (nbr_copies of copy_borrower
                     < NBR_MAXIMUM_COPIES_BY_BORROWER)
    finally
            checked_out?:
             (borrowed_by of copy = copy_borrower)
    part
            p1: ASSIGN_BORROWER ( c = copy,
                                  b = copy_borrower)
            p2: INCREMENT (counter = nbr_copies of
                                                  copy_borrower)
```

Figure RML.07 CHECK_OUT_COPY in RML

The first activity modelled is the CHECK_OUT operation. The attribute *copy* has been classified in both the **input** and **output** categories, because a copy exists before and after this activity has been carried out, but one of the attributes of this entity is modified by the activity (thus it could not be in the **control** category). This is the same for the entity *copy_borrower*. So these two entities are removed and re-added in their classes, after a modification of one or more of their property values.

The property *user* is an entity which is intended to represent the person that performs (or rather make the computer perform) the activity. This comes from a constraint stated in the Library Problem, which restricts the access to certain operations for certain categories of users. This one, for instance, is restricted to staff members.

The property category **initially** is very useful for activities. We know that an activity, like any other object in RML, has properties. These can be entities, other activities and assertions. So an activity can be related to assertions. If the assertion belongs to the property category **initially**, its value cannot be null at the start time of the activity. Since a non-null value for an assertion means that it is true, this true assertion corresponds to a condition that must be met before the execution of the activity, which is the definition of a precondition in fact. Therefore this is the place where we shall state the preconditions of the activity. (Note: in the less recent version of the language syntax, this property category was called **precond** for operations, so the meaning was very accurate already).

In our example, the first precondition is that the activity can be triggered by staff members only. So we have stated a property *authorized?* which is a predicate that has the value true if the attribute *categ_pers* of *user* has the value *'Staff*. Another precondition is that the copy must not be currently borrowed. The last two preconditions arise from two of the

Library Problem constraints, which give a maximum amount of copies available for a borrower to check out and prevent their borrowing several copies of the same book.

The property category **finally** is very similar to the property category **initially**, as far as their meanings are concerned. It consists of a set of assertions that must be true at the end of the activity, which means that they amount to postconditions. We only have one example at hand in this case, the property *checked_out?*. This one states that the attribute *borrowed_by* of *copy* has the value of the property *copy_borrower*. These two entities are therefore linked from now on.

Finally, the property category **part** allows the specifier to refine the description of the activity being specified, by defining subactivities which make it up. Note that these ones are not ordered in any way, it is up to the designer to state any constraints on their temporal ordering as explicit assertions. Here the activity is made up of two parts, one for "linking" a copy to its borrower, and another one to increment the counter for those copies already checked out by that person.

```
activity class RETURN_COPY with
    input output
        copy: COPY
        copy_borrower: BORROWER
    control
        user: PERSON
    initially
        authorized?: (categ_pers of user = 'Staff)
        borrowed?:
            (borrowed_by of copy = copy_borrower)
    finally
        checked_out?: (borrowed_by of copy = null)
        update_last_borrower?:
        (last_borrowed_by of copy = copy_borrower)
    part
        p1: DEASSIGN_BORROWER (c = copy)
        p2: MEMORIZE_LAST_BORROWER
            (c = copy, b = copy_borrower)
        p3: DECREMENT (counter = nbr_copies of
                                    copy_borrower)
```

Figure RML.08 RETURN_COPY in RML

This activity is very like the previous one and so does not require any additional comments.

```
activity class ADD_COPY with
        output
                copy: COPY
        control
                copy_identification: COPY_ID
                book: BOOK
                user: PERSON
        initially
                authorized?: (category of user = 'Staff)
                correct_id?: (not exists x in COPY
                                such that
                                        copy_id of x =
                                        copy_identification)
        finally
                add_copy?:
                  (copy_id of copy = copy_identification)
                  and (of_book of copy = book) and
                  (borrowed_by of copy = null) and
                  (last_borrowed_by of copy = null)
        part
                p1: GET_BOOK (b = book)
                p2: FIND_IDENTIFIER
                                (id = copy_identification)
                p3: CREATE_COPY (c = copy, b = book,
                                        id = copy_identification)
```

Figure RML.09 ADD_COPY in RML

The activity ADD_COPY has *copy* as output property, because one of its main action is the creation of that entity. Two assertions have proved necessary, one asserting that this operation must be executed by staff members only, and another one asserting that the *copy_identification* - found by the subactivity FIND_IDENTIFIER - must be different from all the identifiers of other existing copies (this assertion is the *correct_id?* property).

Note that the activity ADD_COPY will work properly if the corresponding book has been recorded before. Thus, if this is not the case, the activity ADD_BOOK - whose purpose is to register a book in the system memory - will have to be triggered off before ADD_COPY.

```
activity class ADD_BOOK with
        output
                book: BOOKS
        control
                book_identification: BOOK_ID
                book_title: TITLE
                book_subjects: class_of SUBJECT
                book_authors: class_of AUTHOR
                user: PERSON
```

```
initially
        authorized?: (category of user = 'Staff)
        correct_id?: (not exists x in BOOK
                              such that
                                    book_id of x =
                                    book_identification)
finally
        add_book?:
          (book_id of book = book_identification)
           and (title of book = book_title)
           and (authors of book = book_authors)
           and (subjects of book = book_subjects)
part
        p1: GET_BOOK_TITLE (t = book_title)
        p2: GET_BOOK_SUBJECTS (s = book_subjects)
        p3: GET_BOOK_AUTHORS (a = book_authors)
        p4: FIND_IDENTIFIER
                        (id = book_identification)
        p5: CREATE_BOOK (b = book, t = book_title,
                                b_auth = book_authors,
                                b_subj = book_subjects,
                                id = book_identification)
```

Figure RML.10 ADD_BOOK in RML

The activity ADD_BOOK is very similar to ADD_COPY. Thus no additional comments are needed.

As to the REMOVE_COPY activity it is fairly simple too.

```
activity class REMOVE_COPY with
        input
                copy: COPY
        control
                user: PERSON
        initially
                authorized?: (category of user = 'Staff)
                unchecked_out?:
                                (borrowed_by of copy = null)
        part
                p1: DELETE_COPY (c = copy)
```

Figure RML.11 REMOVE_COPY in RML

Now we shall deal with the query activities stated in the library problem. Thus we shall have to work out lists of books, given a certain filter: the books written by a given author, those dealing with a given subject and those borrowed by a given person. These operations are very much alike in RML. They all have a list of BOOK as output. The only important difference lies in the postconditions, which are assertions put into the **finally** property category. A predicate thus describes for each activity the condition for a book to be in the output list.

```
activity class GET_BOOKS_BY_AUTHOR with
        output
                list: class_of BOOK
        control
                author: AUTHOR
        finally
                list_found?: forall x in BOOK
                                        author in authors of x
                                        => x in list
```

Figure RML.12 GET_BOOKS_BY_AUTHOR in RML

```
activity class GET_BOOKS_BY_SUBJECT with
        output
                list: class_of BOOK
        control
                subject: SUBJECT
        finally
                list_found?: forall x in BOOK
                                        subject in subjects of x
                                => x in list
```

Figure RML.13 GET_BOOKS_BY_SUBJECT in RML

There are no preconditions to these two activities for they can be executed by anybody, either staff member or reader. The next one is different, because the query can be executed by a staff member or a borrower, but the latter may only have a look at their own check-out list. This is stated explicitly in the **initially** part of its specification by a combination (conjunction or disjunction) of simple conditions.

```
activity class GET_BOOKS_BY_BORROWER with
        output
                list: class_of BOOK
        control
                borrower: BORROWER
                user: PERSON
        initially
                authorized?: (category of user = 'Staff)
                        or ((category of user = 'Borrower)
                        and (name of user=name of borrower))
        finally
                list_found?: forall x in COPY
                                        borrowed_by of x = borrower
                                        => (of_book of x) in list
```

Figure RML.14 GET_BOOKS_BY_AUTHOR in RML

The last operation required for the new system is the one which finds out the last borrower of a copy. Interpretation of the requirements is necessary here too. What does "last

borrower" mean? This question shows once again a problem typical of informal requirements: ambiguities. To solve it we have decided that the last borrower is the current one if the copy is currently borrowed, otherwise it is the latest person that borrowed it.

```
activity class GET_BORROWER_OF_COPY with
    control
        last_borrower: BORROWER
        copy: COPY
        user: PERSON
    initially
        authorized?: (category of user = 'Staff)
    finally
        last_borrower_found?:
            (borrowed_by of copy = null)
            and (last_borrower =
                last_borrowed_by of copy)
         or (borrowed_by of copy <> null)
            and (last_borrower =
                borrowed_by of copy)
    part
        p1: GET_COPY (c = copy)
        p2: GET_LAST_BORROWER (c = copy,
                l = last_borrower)
```

Figure RML.15 GET_BORROWER_OF_COPY in RML

C. Assertion Modelling

An assertion class is a closed formula with free variables, whose values are entities from given entity classes. An assertion token, member of the class, is a closed formula derived from an assertion class by binding each of its free variables. Every assertion token has an argument referring to the time when the assertion is supposed to hold. So a token represents a specific fact that is true at a specific time.

Some commonly used constraints of a restricted form have been built into the RML notation and principles through such facilities as ranges for properties, property categories, binding assertions and rules about the IsA relation.

To deal with more general constraints, RML also provides a First Order Logic language with logical connectives and quantifiers such as **forall, exists, such that, not, and, or, implies,** etc...

Predicates can deal with time in RML: assertions may involve the following functions and predicates involving time:

- time comparators =, < and <=

- functions **start** and **stop**, specifying the start and end time of an activity token

- predicate **in**(x,y,t) indicating whether x is an instance of y at time t. This can also be written (x **in** y **at** t)

- function **pv**(x,y,t) returning the value of property y of object x at time t. This can also be written as (y **of** x **at** t)

In addition, RML assertions subsume the usual notations for the logic of arithmetic and strings, and may involve the special symbols **null** and **this**.

In order to represent more general relationships than functions we need assertions which play the role of predicates. Since uniformity is one of the guiding principles in designing RML, the language designers have chosen to also model assertions as objects organized into classes. In this case, an assertion class is to be interpreted as a predicate declaration. By analogy with entity and activity classes, instances of an assertion class will have zero or more attributes, which in this case include the free variables (arguments) of the predicate; these variables will be typed by the usual method of property definitions. If, for example, P is an assertion class with argument properties x1 and x2 say, then at time t each instance of P is assumed to hold for constants which make the formula P(x1,x2,t) true. Thus, instances of assertion classes represent propositions which are true at that moment, in the same way as

instances of entity classes represent existing entities, and instances of activity classes correspond to occurring activities. Therefore in RML, in order to require that some condition be true, one must state that the property relating this condition to some object has a non-null value. In other words, truth of conditions is replaced by the presence of objects in assertion classes, where logical formulas written by designers are assumed to implicitly define distinct assertion classes.

The understanding of an example will be made easier if we list property categories specific to assertions:

- **argument**: the property represents a free variable of the predicate corresponding to the class; hence, for each token of this class, the values of the arguments represent one particular set of variable bindings

- **asserter**: activity which makes this assertion true

- **denier**: activity which makes this assertion false

There were numerous examples of assertions in the properties of both activities and entities that have been modelled so far. The reader must certainly have noticed them. If not, here is a reminder of one of them, from the GET_BOOKS_BY_BORROWER operation (Figure RML.14):

```
finally
        list_found?: forall x in COPY
                                borrowed_by of x = borrower
                        => (of_book of x) in list
```

Figure RML.16 An assertion in RML

This illustrates how to express an assertion as a property of either an activity or an entity. But, for more complex and cumbersome assertions, and in cases where it is more suitable to treat assertions as objects (with classes and generalizations, and so on...), this could be expressed by an assertion on its own:

```
assertion class LIST_OF_BOOKS with
    arguments
        l: class_of BOOK
        b: BORROWER
    necessary
        list_found?: forall x in COPY
                                borrowed_by of x = b
                        => (of_book of x) in l
```

Figure RML.17 An assertion as a property

and be introduced in the operation (summarized here) in this fashion(1):

```
activity class GET_BOOKS_BY_BORROWER with
        output
                list: class_of BOOK
        control
                borrower: BORROWER
        finally
                list_found?: LIST_OF_BOOKS (l = list,
                                            b = borrower)
```

Figure RML.18 How to use a property?

As *list_found?* is a property whose value must be true at the end of the activity, it means here that the assertion LIST_OF_BOOKS may not have the value **null**. Thus, the predicate must be true. And so the variables l and b must be instantiated with values that will make this predicate true. As a result of property binding, b is assigned the value of *borrower* and so becomes bound, whereas l remains free and will be instantiated and therefore supply a value for *list*, which is an output of the activity.

(1) Actually, this is a kind of procedural abstraction considering assertions as operations with a boolean range.

3.3.1.3 Evaluating RML

A. Expressive power

a. Scope of the language

RML mainly deals with functional requirements. It seems that non-functional requirements are not addressed by RML. At least not explicitly, without resorting to complex devices, which we do not know about for lack of sufficient practice in RML.

b. Static and Dynamic aspects

RML allows the specification of entities and relations (binary through properties and n_ary through assertions), which has the power of an entity-relationship model. RML also has abstraction mechanisms current in data modelling languages: classification, aggregation, generalization. So these powerful features allow the building of objects in a stepwise fashion, which is of paramount importance for the specification of large systems. Here is an example, inspired by the Library Problem:

```
entity CHILD_BORROWER in PERSON_CLASS
    isa BORROWER with
        association
            teacher: TEACHERS
            school: SCHOOLS
```

This is the description of an instance of the class CHILD_BORROWER (classification), which is itself a member of the meta-class PERSON_CLASS (classification). Each member of the class CHILD_BORROWER is also a member of the class BORROWER, which means that they inherit the properties of the instances of the class BORROWER (specialization). Finally, an entity of the class CHILD_BORROWER is made up with properties, both specific and inherited (aggregation).

Moreover, these abstraction mechanisms are also available for operations and assertions. Thus there can also be hierarchies of operations, assertions and so on... The most interesting use of such hierarchies is when a hierarchy of operations or assertions is associated with a hierarchy of entities. And so, when the specifier gradually builds the specification of the system objects, he can refine the operations working upon them at the same time.

For example, if the specifier considers the rough entity class BORROWER, in a first time he/she will define the activity class ADMIT_BORROWER accordingly. But later, after refining BORROWER into CHILD_BORROWER and GROWN-UP_BORROWER, he will need to refine ADMIT_BORROWER into ADMIT_CHILD_BORROWER and ADMIT_GROWN-UP_BORROWER. The resulting piece of specification will be:

```
activity class ADMIT_BORROWER with
        output
                b: BORROWER
        <...>
        part
                p1: FIND_CODE_NUMBER
                <...>

activity class ADMIT_CHILD_BORROWER
    isa ADMIT_BORROWER with
        output
                b: CHILD_BORROWER
        <...>
        part
                p0: RECORD_TEACHER_AND_SCHOOL
                p1: FIND_CODE_NUMBER_CHILD
                <...>
```

This activity is a specialization of the ADMIT_BORROWER activity. It should be mentioned that properties are inherited by default, but can be redefined. In this example, the output property *b* is redefined otherwise in the specialized activity class ADMIT_CHILD_BORROWER. As to the property p0, it is specific to a child borrower and so must be added, whereas the property p1 just needs to be adapted to fit in the refined activity class.

To our knowledge, there is no inhibition mechanism in RML that would allow to drop some undesired properties. They can only be redefined.

Dynamic aspects of the system can be defined through assertions. As activity preconditions depend to a great extent on assertions, it is possible to define a sort of behaviour of the system. These activities modify in turn assertions, namely make them true or false, and so some other activity preconditions can become true.

Now what about the *temporal behaviour* of the system?

It is possible to define a standard calendar time in RML, which is very useful for many applications and which would allow the modelling of the dynamic scheduling of operations for example.

The solution proposed in [ML-6] is to overlay on top of the linear time a temporal scheme based on intervals associated with activity occurrences: the beginning and end of an activity token demarcate an interval.

The triggering of activities will depend on assertions that will constrain them. These assertions make use of time predicates including time comparators and functions **start** and **stop**. They make up a tree-structure, the root of which is the less constraining assertion between two events, i.e. that they are just temporally related.

```
assertion class Temporally_Related with
        arguments
                first: AnyEvent
                second: AnyEvent
```

This assertion is the basis of the temporal system that we intend to build. Its predicate is always true, since there is none, and the argument can therefore be instantiated with any activity in the system. So it actually creates a relationship between the two activities of the system.

Now, we can start from this basic assertion and build up more meaningful and constraining assertions that will constrain activities. This will give a structure of assertions, organized in an isa hierarchy, an example of which could be:

```
TemporallyRelated <—isa— Overlaps <—isa— Occurs_during
                                          <—isa- ....
                  <—isa— Non_overlapping  <—isa- Earlier
                                          <—isa- Later
```

```
assertion class Overlaps isa TemporallyRelated with
        necessary
                atOneEnd: (x = first of this at ft)
                          and (y = second of this at st)
                          and (x <> null) and (y <> null)
                  => ( stop(x) > start(y) or
                          stop(y) > start(x) )
```

This assertion constrains two temporally-related activities further, and will be added as a **necessary** property in both activities involved, in the following:

```
activity class ADMIT_BORROWER
        output
                b: CHILD_BORROWER
        necessary
                overlapping: (first = this,
                                second = CHECK_BORROWER_AGE)
        <...>
```

We have thus stated that the beginning of the activity CHECK_BORROWER_AGE must begin before the end of ADMIT_BORROWER. (Of course this is just for the sake of the example and does not have any actual meaning in any actual library)

This is a convenient way for scheduling activities, but it is not suited for definite time intervals and start times. If one wants to model the fact that an activity starts at 5 p.m. or 10 minutes after another one, one needs a clock. The modelling of such a clock in RML can be found in [ML-6].

For example, having defined *ClockHour*, to specify that the admission of borrowers can only begin at nine o'clock in the morning, we could attach the assertion:

```
initially
        when: (exists h) (h in ClockHours)
                and (clockReading of h = 9)
```

However, even though time plays an important role in RML, we do not think historical aspects can be dealt with easily. The only way to take into account an historical view for entities seems to add a property for the memorization of the date of creation, given a clock previously defined.

c. Specialization areas of the language

RML is rather domain-independent; no particular application domain is addressed. But this is restricted to sequential systems, as there are no facilities for specifying parallel systems.

B. Deductive power

The semantics of RML is described by presenting a method for translating an arbitrary RML specification into a set of assertions in predicate calculus. Assertions were already expressed in a first order logic language; the rest of RML phrases is translated into a similar form.

a. Verifiability of a specification

This 1st order logic background provides ready-made answers for consistency-checking of specifications and deductions from them, because the proof-theory of first-order logic can be used, and the considerable work in the field of automatic deduction can be incorporated into computer tools that will assist users in developing requirements.

There are tools currently in the process of development. These include consistency checkers and database facilities for storing/retrieving RML models. So the *analyzability* of requirements is supported.

b. Validation of a specification

Unlike Z, RML does not explicitly allow the introduction of informal comments within a specification. So this makes specifications less legible for the final user, although the notations used are less mathematical than in Z.

Executability is not straightforward either. A RML model cannot be translated into a prototype directly. In fact RML is not preoccupied with the design phase. Programs describing (at a high level) an information system are written in a language called TAXIS. There must be a manual translation from RML specifications into TAXIS programs. Only the latter can be executed to produce prototypes. Up to now there has been no computer-aided support for the design of TAXIS programs from RML models, so we cannot really assert that RML specifications lend themselves to executable products.

c. Modifiability of a specification

The abstraction mechanism of specialization, with inheritance of properties down the hierarchy, makes it easier to locate changes. But interesting modularity features, such as the possibility of including ready-made pieces of specifications (like in Z), is missing.

3.3.2 THE GIST LANGUAGE

We will first present an overview of the major aspects of GIST formal specification language. This part is mainly based on [ML-2], [ML-7], [ML-11], [ML-12]. Each new theoretical notion will be illustrated using examples coming from the Library Problem.

A tentative evaluation of GIST will then be proposed, based on its expressive and deductive power.

3.3.2.1 An overview

The objective of a GIST specification consists in the definition of a class of behaviours for the system being considered. Each behaviour can be considered as a sequence of situations and represents, by definition, one possible behaviour of the specified system.

A GIST specification is executable; it can be used as a prototype to access the specified system behaviour. But, because of the intolerably slow evaluation of the GIST specification, it is not possible to use the GIST prototype directly as a system implementable.

A GIST canonical prototype should always be divided into the three following parts:

(i) *Structural declarations*, which define a space of potential states of the system,

(ii) *Operations*, which define situations which initiate activity and the range of behaviours ensuing from those situations,

(iii)*Constraints*, which prune the space of possible behaviours by the elements of the two previous parts. This "pruned" space of behaviours will constitute the system defined by a GIST specification.

We shall now study those three parts in a more accurate way.

A. Structural declarations

a. Relational model of information

When introducing a new object in a GIST specification, we have to give it a type.

The Library Problem domain, for example, involves objects of type BOOK, AUTHOR, SUBJECT, COPY, BORROWER, and STAFF_MEMBER. Taking the GIST syntax, we have:

type BOOK; **type** AUTHOR; **type** SUBJECT, and so on.

Figure GIST.01 GIST types

Type hierarchies are also possible in GIST thanks to the **supertype** mechanism. We can, for example, consider that the borrowers and the staff members are more generally the users of our system. Thus, we have:

type USERS() **supertype_of** <BORROWER;STAFF_MEMBER>

Figure GIST.02 GIST supertypes

Next to this mechanism, GIST provides also other type constructors, like **set_of**, and built-in relations among sets such as **is_a_member_of** or **cardinality**.

We also need to set up relations among each of those objects typed to model information about the system to specify.

In the Library Problem, we must specify, for example, that a copy of a book can be checked out by a borrower:

relation on_loan_to (COPY, BORROWER).

For the sake of completeness, we introduce the four following relations, whose semantics has been defined in the Z specification (see pages 19 and 20):

relation on_last_loan_to (COPY, BORROWER);
relation is_a_copy_of (COPY, BOOK);
relation written_by (BOOK, AUTHORS);
relation talk_about (BOOK, SUBJECT);

Figure GIST.03 GIST relations

Binary relations such as the previous ones, are a frequently used form of n-ary relations. Thus, GIST has provided a syntactic shorthand to declare and access binary relations more easily. This shorthand takes the form of "attributes" associated with types.

If we consider, for example, the relation written_by which is a binary relation between types BOOK and AUTHOR, it becomes an attribute of type BOOK and of type AUTHOR. The simultaneous declaration of types and attributes becomes:

```
type BOOK   (is_a_copy_of| COPY,
             written_by⌈AUTHOR:any::any,
             talk_about| SUBJECT:any::any );   (1)
```

Figure GIST.04 GIST types and attributes

The same mechanism can be applied to each "subtype" which makes up the supertype USER:

```
type USER() supertype_of < BORROWER (on_loan_to|COPY,
                                     on_last_loan_to|COPY);
                           STAFF_MEMBER()
                         >
```

Figure GIST.05 GIST supertypes and attributes

This relational model of information allows the specifier to use a descriptive reference to an object. So the specifier does not need to care about data access paths since objects are defined via their relationships with other objects. Thus the access problem has been solved, because any of the relationships in which an object participates (or any combination of them) can be used to access that object: the relationships are fully associative.

(1) The cardinality constraints are explained in the sub-section "Constraints and non-determinism".

b. Predicates and expressions

Using predicates and expressions, the specifier can draw out information from the current state. Those predicates and expressions will only refer to objects of the current state.

The three main types of expressions are provided in GIST:

(i) A book in the library domain: a book

In GIST, a variable name that is also the name of the type can be used instead of the form <var name> | <type>. In the previous example, "a book" is a shorthand for "book | BOOK". We can declare, using the same mechanism: a copy, a user, a subject, a author.

One can already observe that we are able to define an undeterminate book. We just need an object satisfying an accurate description. In such a case, we say that the expression is non-deterministic.

(ii) The copies of a book b: b: is_a_copy_of

This GIST sentence denotes the objects related by "is_a_copy_of" to "b", in this case, its copies.

(iii) A book written by the author "E.A. Poe":

a book || (book: written_by = "E.A. Poe")

The construct used here takes the form a <type name> || <predic> and denotes an object of that type satisfying the predicate.

Here also, we have a non-deterministic expression.

It is also possible to express predicates in GIST: "Is every book related to a copy?" will be translated in GIST as:

```
for all book || (
        exist copy || (copy:is_a_copy_of = book))
```

Figure GIST.06 GIST predicates and expressions

It can only be either true or false at a given time of the system life.

As can be seen, we can use in GIST existential and universal quantification over objects of a given type.

B. Operations

a. Change in the domain and procedures

As introduced before, a GIST specification is a collection of "behaviours" which are sequences of states connected by transitions.

These transitions are modelled in the domain not only by objects creation and deletion, but also by the insertions and the deletions of relations between different objects. Each such primitive change primitive causes a transition to a new state.

These primitives can be composed using forms familiar from programming languages like sequencing, conditional execution, iteration,...

We can, for example,

(i) Create a new author:

create author;

(ii) Assign the author "Eleanor Rigby" as the writer of the book "Paul McCartney, musical genius of our century":

insert "Paul McCartney, musical genius of our century": written_by = "Eleanor Rigby"

Figure GIST.07 How to change the domain value

To include within a single transition several such primitive changes, we have to put them together inside a GIST **atomic** construct. Let us consider the following example: "Change the last borrower of a book!"

```
atomic
        delete copy: on_loan_to = borrower;
        insert copy: on_last_loan_to = borrower;
end atomic
```

Figure GIST.08 The atomic feature

One or more GIST actions can be defined within a GIST procedure construct. A procedure is parameterized and can be called from everywhere in the specifications. Each such a call instantiates the procedure formal parameters with the actual ones and executes the defined action.

We can illustrate this by the following procedure which adds a copy of an existing book into our library system:

```
procedure ADD_A_COPY (book,copy)
        insert book: is_copy_of = copy
        where always required exists book and
                                        not exist copy
```

Figure GIST.09 The ADD_A_COPY procedure

b. Temporal reference

We can see a "behaviour" as a sequence of states connected by transitions leading "to" and "from" the current state.

GIST gives the specifier the ability to extract information from any state in the behaviour thanks to the temporal reference mechanisms. These mechanisms enable the specifier to describe what information is needed from earlier and later states. If the specifier does not add such expressions, the request will be processed on the current state of the system.

(i) **asof everbefore**

Thanks to this feature, the specifier can refer to the previous states of the specified system.

The following GIST predicate gives an example of such a temporal reference. Has a borrower already borrowed a given copy of a book?

((copy: on_loan_to = borrower) **asof everbefore**)

Figure GIST.10a GIST asof everbefore

(ii) **ordered temporally**

Thanks to this feature, the specifier has the ability to produce an ordered sequence of objects.

The following GIST procedure produces the list of books currently checked out by a given borrower. This list of the books is ordered in their borrowing order.

```
procedure FIND_OUT (borrower)
      for all (book || book is_a_copy_of c and
                  c on_loan_to borrower)
                                          order temporally
      where always required exists borrower
```

Figure GIST.10b GIST order temporally

(iii) **asof evermore**

Using this feature, a piece of specification can make references to the future states of the system.

It is important to notice that the specifier does not need to worry about the details of how this temporal information might be made available: GIST assumes the responsibility for remembering all previous states so that each temporal request can be satisfied.

c. Daemons

Daemons are a GIST mechanism for providing data-driven invocation of processes. A "daemon's trigger" is a predicate which triggers the "daemon's response" whenever a state change induces a change in the value of the trigger predicate from false to true.

```
procedure ADD_A_COPY (copy,book,author,subject)
      insert book: is_a_copy_of = copy

daemon D_ADD_A_COPY (copy,book,author,subject)
      trigger       not exists copy and
                    exists book and
                    exists author and
                    exists subject and
                    RANDOM()
      response ADD_A_COPY (copy,book,author,subject)
```

Figure GIST.11 The daemon mechanism

As one can see, daemons are a convenient specification construct for use in situations in which we wish to trigger an activity upon some particular change of state in the environment modelled. The **RANDOM()** condition states that an event coming from the environment of the system will trigger the daemon which triggers the procedure "ADD_A_COPY". The event is, in this example, an update of the system by a user using his terminal.

Daemons also save us from the need to identify the individual portions of the specification where actions might cause a change and the need to insert into such places the additional code necessary to invoke the response accordingly.

C. Constraints and non-determinism

A specification denotes those behaviours which do not violate the constraints specified, and only those. Before adding the constraints, the specification defines an infinite space of possible situations and behaviours ensuing from them.

Thus, we have to use constraints in order to model the integrity conditions that must remain satisfied throughout the life of the specified system. One of the most interesting features of these constraints is that we are able to constrain either the past or the future state of the system(1).

They will be used in order to describe both the limitations of the domain and the behaviour restrictions.

Let us consider two examples. The former restricts the changes that may occur to the system state and the latter restricts the number of objects related to other objects in a relationship.

The 4th constraint of the Library Problem, which states that a borrower may not have more than one copy of the same book checked out at one time, will be translated in GIST as follows:

always prohibited for all book, borrower, copy.1, copy.2,

||

(copy.1: on_loan_to = borrower) **and**
(copy.2: on_loan_to = borrower) **and**
(book: is_a_copy_of = copy.1) **and**
(book: is_a_copy_of = copy.2)

Figure GIST.12 A GIST constraint

(1) In Z or in RML, they constrain the entire life of the system.

Special notations are provided for cardinality constraints. E.g., we can specify that every copy is linked to one and only one book:

type COPY(is_a_copy_of| BOOK:**unique::any**,...)

Figure GIST.13 A GIST cardinality constraints

In this declaration, we have also specified that a book in our library system can be related to zero, one or more copies.

Syntax: the keyword following ":" constrains how many objects of the attribute type (BOOK) can be attributed to the type being defined (COPY). On the other hand, the keyword following "::" constrains how many objects of the defined type (COPY) can have as attribute an object of the attribute type (BOOK). We have the choice between the four following keywords:

any for 0,1,N;

unique for 1 and only 1;

multiple for greater or equal to 1;

and **optional** for 0,1.

3.3.2.2 Evaluating GIST

We are now able to propose an evaluation of the GIST expressive and deductive power. We shall analyze them by referring to the analysis grid we proposed in the introducing section of this part.

A. Expressive power

a. Scope of the language

GIST addresses not only functional requirements but also some of the non-functional ones(1).

Complete examples of GIST specifications are given in [ML-2], where one can find a GIST environment description, and in [EC-5].

Non-functional requirements are modelled thanks to special GIST features like : **environment**, **agent**(2), **event** and **RANDOM()**.

b. Dynamic modeling in GIST

The dynamics of a system specified in GIST can be managed using temporal referencings.

If we consider that "<predicate> **asof everbefore**" is a constraint upon an operation, then this operation will only be executed when the <predicate> is true. But in this expression, the specifier has to handle himself the objects which must be taken into account for the evaluation of the <predicate>.

However, one might regret that such underlying principles cannot be expressed in a more explicit way.

c. Static modelling in GIST

The three GIST keywords **type**, **supertype** and **set_of** allow the specifier to define the objects of the system he has to specify.

---

(1) Concerning the environment of the system.
(2) See further the sub-section "Language extensions".

Thanks to the **supertype** and **set_of** features, it is feasible to handle type hierarchies. Thus specialization or generalization is supported.

The specifier will also be able to manage the inheritance of relations and operations which define each subtype: when one references a supertype, the selection of the relevant subtype is automatically managed by the context in which the supertype is used.

Examples are given in Figures GIST.01 to GIST.05.

d. The association concept

GIST enables the specifier to create associations among any types declared thanks to either **relation** declarations or the **type** shorthand.

As seen before, it is also possible to specify the cardinality of these relations using the keywords **any, unique, multiple** or **optional**. But, these keywords do not allow the specifier to express more precise cardinality constraints directly. See, for an example, Figure GIST.13.

Nevertheless, one of the most important advantages of these associations is that they are fully reversible. Thus, the specifier must not worry about the definition of access paths: this problem is to be solved only during the implementation phase.

e. Real time and historical aspects

We have seen that GIST supports historical referencing thanks to keywords like **asof everbefore, ordered temporally** and **asof evermore**. (see Figures GIST.10a and GIST.10b) In fact, GIST assumes the responsibility for remembering all previous states so that each historical information request could be satisfied.

Thus, using such predefined keywords, the specifier can define predicates, operations, daemons and constraints which refer to the entire history (past and future) of the specified system without knowing "HOW" it would satisfy this functionality.

GIST provides also the **RANDOM()** facilities to express, in an unnatural way, realtime events creation. ([RS-9], p. 21)

On another hand, GIST provides the specifier with no primitives dealing with realtime considerations. Thus, the specifier has to introduce a new object 'CLOCK' and some operations to manage it, e.g.,

```
type CLOCK with
relation WHAT_TIME(integer)
daemon TICK_TOCK[]
        triggers always true
        response WHAT_TIME(?) := WHAT_TIME(?)+1
```

Figure GIST.14 A GIST clock specification

Having defined such a type, realtime-oriented primitives should have a precondition like:

when (WHAT_TIME = <time>) **and** (<other preconditions>)

f. Language extensions

The GIST syntax has been recently upgraded by the **Agent** feature and the responsibility concept. They are used to model components. Agents partition the generative portion of a GIST specification. Thus every primitive procedure is within the scope of some agent. Each autonomous process in the domain being specified will typically be a separate component. ([ML-12] and [RS-9])

Thereby, we could specify that a borrower can only make a request to the database when he wants to know which book he has borrowed.

```
agent borrower {
                procedure FIND_OUT_BY_BORROWER ( borrower )
                  || write copy: on_last_loan_to(borrower)
                        }
```

Figure GIST.15 A GIST agent specification

g. Specialization areas

GIST seems to be a good example of a domain-independent formal specification language.

As seen before, using GIST, it seems to us that the specifier can specify any type of requirements want he has got not only tools for managing complex data types and relations, but he can also easily manage time related problems (real-time or historical ones). Another GIST interest consists of its ability for specifying both functional and non-functional requirements.

B. Deductive power

a.Verifiability of a specification

Unlike Z and RML, the theoretical background of GIST is the relational model which has been upgraded with useful statements like create, destruct, insert and delete, to model state transitions.

A computerized tool for verifying whether a specification respects the GIST syntax could be used by a specifier.

Another tool to verify if every GIST procedure is used by a daemon, could be also used, because a GIST "independent" procedure will never be used.

b.Validation of a specification

Like Z, GIST explicitly allows the introduction of informal comments within a specification, thanks to the GIST keywords:

Spec comment ..... end comment

This could make specifications more legible for the final user. But, we remain convinced that the level of abstraction and the existence of "unusual" features, like a daemon, leave the specifications hardly understandable for people that are not familiar with such abstractions.

c.Executability of a GIST specification

Up to now, no GIST compiler seems available. Nevertheless, GIST designers describe GIST as being "symbolically executable". Thus, feedback can be used to show incomplete or ambiguous portions of the requirements. [ML-11]

But the major difficulty is that the evaluation of a GIST specification would be intolerably slow.

d.Modifiability of a specification

The level of modularity that can be reached in GIST should offer the specifier a good support when he has to modify a piece of formal specifications.

This section proposes a comparison of the three formal specification languages studied. First of all, we shall build up a summary table for the conclusions regarding the expressive and deductive powers of the formal languages. Then, for each one of them, we shall make some comments.

3.4.1 EXPRESSIVE POWER

	Z	RML	GIST
Scope	Functional	Functional	Functional Non-functional
Objects	DataSpace schema	Entity	Type/Supertype.
Relations	2-ary (relation) n-ary (schema)	2-ary (property) n-ary (assertion)	relation
Operations	Operation schema	Activity	Procedure
Constraints	Predicates	Assertion	Constraints
Modularity	Schemas inclusion	(cfr abstraction mechanisms)	
Abstraction mechanisms			
- **classif.**	Sch.Type(obj,op) given set	class(obj,op,as)	type (obj) proced. (op) set_of (obj)
- **aggreg.**	Sch.Type(obj,op)	in (obj,op,ass)	daemon (op) relation (obj)
- **gener.**	Sch.Type(obj,op)	isa (obj,op,ass)	supertype (obj)
Dynamical aspects	No standard	time relation, clock, calendar	temporal referencing
Historical	Not explicitly	Not explicitly	Not powerful

All three languages focus on functional requirements, and non-functional requirements such as performances, reliability, etc... do not really fit in those models very well. Nevertheless, GIST allows the specification of the environment.

As to the modelling of objects, RML remains very close to classical entity/relationship models, cardinality constraints are mainly expressed through the notion of property category (ex: **necessary** = one and only one value for a property). GIST would rather stick to the well-known relational model, whereas the representation of objects in Z is based upon mathematical notions such as sets and functions.

The three languages support the description of operations, plus some constraints upon them.

Constraints are expressed through 1st order predicates in RML, GIST and Z, which is a powerful feature as it will be explained below. In Z, they are directly attached to objects and operations, whereas in RML they can be expressed separately and be treated as objects too, with hierarchies and so on. Finally in GIST, they can be expressed separately (see figure GIST.12). In this case, they constrain all the system being specified. But, they can also be attached to a type, a relation, a daemon or a procedure and then have a limited scope.

Abstraction mechanisms do not exist explicitly in Z, but they can be found behind mechanisms such as schema inclusion, union, etc... The typing mechanism is common to all of the three languages. The most interesting features related to abstraction mechanisms is to be found in RML, which gives them a prominent part. This is not surprising, as the roots of RML lie in knowledge representation work. GIST appears to be the poorest language among the three studied, with regards to abstraction and structuring mechanisms. Perhaps this is due to some extent to the relational model used as a theoretical foundation; this model being not considered as the richest one.

Thus RML treats entities, operations and constraints as objects into classes which can be organized in hierarchies and in meta-classes. GIST only allows the decomposition of procedures thanks to daemons, the generalization of objects thanks to supertypes, the classification of objects tanks to the **type** and **set_of** feature, and the aggregation thanks to relations. Z has an intermediate approach because objects and operations are treated the same way. Therefore, if an object can be aggregated, so can an operation. However, this cannot be done explicitly.

In short, RML provides a unified approach to abstractions, whereas Z and GIST (especially the latter) are more uneven. But Z provides a unified approach with regards to the language used, everything being expressed mathematically, whereas RML uses an eclectic approach based on both a 1st order logic and other notations.

Aggregation/Decomposition and Generalization/Specialization facilities greatly favour modularity in a specification written in a given language. They enable the refinement of both objects and operations, which is fundamental for conveniently specifying large systems. So the more present these abstraction mechanisms are, the better it is.

But modularity also concerns the possibility of arbitrary splitting specifications into pieces, in order to either reuse them or make specifications less bulky. For this purpose Z offers the most interesting features, thanks to the schema inclusion mechanism. It enables the specifier to put a set of Z phrases(1) in a sub-schema that can be reused in any other schema(2). Neither RML, nor GIST easily provide the same facilities.

Dynamics modelling is one of the main weaknesses of Z. Nothing regarding the dynamic behaviour of a system can be easily described within this formal language. RML is more suited for that, thanks to its time predicates and assertions. In GIST, the system dynamics could be modelled in an unnatural(3) way thanks to temporal references.

As to the historical aspects of an information system, none of the three formal language designers have devoted a great deal of their attention to this problem. So there is no ready made approach for this issue and the specifiers must craft their own solutions. Nevertheless, if the historical requirements are not too complex, GIST provides some appropriate built-in tools (see Figures GIST.10a and GIST.10b).

The discussion above may sound a bit like a tribute to RML against Z and GIST. We would like to say, however, that even if RML seems to offer more powerful means for expressing things, this does not necessarily mean that the other two languages are lacking in expressive power. Z certainly seems the best language to mathematicians, who are used to mathematical notations and do not share the preoccupations of computer scientists. And it is also important to notice for RML that what is gained in expressive power is lost in deductive power...(4) and conversely for Z and GIST. It appears that RML is more of a language very convenient for representing knowledge, whereas Z is more of a language for rigourous inferencing.

---

(1) either declarations or predicates, or both.
(2) Examples have been given infra (cfr "A useful schema" in the Library Problem).
(3) this means that this is not conceptually clean and easily understandable.
(4) cfr. infra, in the next section titled deductive power

An important restriction to the three languages is that their use of 1st order logic makes it impossible to parametrize specifications with predicates. This means, for example, that an operation filtering data cannot have a predicate defining its filter as input.

Another restriction is inherent in their inability to deal with modal logic constraints (and other sorts of logics as well). This means, for example, that the difference in meaning between "an operation must occur" and "an operation can occur" cannot be expressed.

3.4.2 DEDUCTIVE POWER

	Z	RML	GIST
Deduction	1st O. Predic. logic	1st O. Predic. logic	Relational notations
Consistent	Yes	Yes	Yes
Completeness	Yes	Yes	Yes
Executability	No But, systematically derivable in guarded commands	No But, manually derivable into Taxis programs	Yes But, inefficient

Z's formal background is 1st order predicate logic which is a major advantage because it can benefit from all the framework provided by predicate calculus: proof-theory, etc...

RML, after some translation stage, can also derive benefit from 1st order logic. Thus, the difference is that RML is more distant from axioms and these sorts of things, and so loses a little of its deductive power. RML offers predefined syntactic phrases (for example, categories of properties, etc...) and restrict the specifier's initiative, unlike in Z where the specifier can write any predicate. Besides, it offers more basic bricks to build specifications and, in a way, gives more power to the specifier.

As seen before, GIST's theoretical background is the relational model theory.

All three languages allow consistency and completeness checking, as well as checking whether a phrase is a consequence of other phrases written in the same language (these issues are discussed in each evaluation section of Z, GIST and RML).

As to the executability of formal specifications, GIST has an edge over its competitors. It is the only one which can be considered as yielding prototypes, even though it is not directly executable as such. RML specs can be translated into Taxis program, which are executable, and Z specs can be translated into prolog programs(1). But there is a great deal of work left to do for all of these languages, because this translation is far from being trivial and straightforward. And the problem of

(1) the particularity of being translatable in prolog is a feature shared with most formal specification languages using 1st order logic and relations.

automatically translating predicates into prolog phrases is a complex one, because there is no efficient way to do it: the programs resulting from the translation, if large, may turn out to be incredibly slow, or may even never finish in reasonable time.

3.4.3 OTHER COMMENTS

Another interesting issue raised in the context of this chapter is the place of these formal languages in the software life cycle.

Z stretches beyond the specification phase, because it can be used for design as well, up to the latest stage in this process, namely the writing of symbolic algorithms in the guarded command language of Dijkstra(1) . In a stepwise refinement manner, Z specifications can get closer and closer to design modules. So Z can be applied at later stages in the software building process. Therefore, it offers an integrated approach.

But, on the other hand, this tends to show that Z is more of a design specification language than a pure requirement specification language.

RML situates itself between SADT schemas and Taxis Programs, but no integration and computer-aided translation from one stage to the other has been implemented yet. So there are three languages in one software development process.

As to GIST, if its non-determinism is lifted a Data Manipulation Language can be obtained, in which one can program. So it could be used at later stages in the software development process as well.

(1) for example, cfr [ML-3].

4. REQUIREMENTS SPECIFICATION: THE PROCESS LEVEL

4.1 INTRODUCTION

Whereas the PRODUCT LEVEL chapter dealt with a description and evaluation of some typical formal specification languages, this chapter addresses the processes underlying the construction of formal specifications from informal requirements. It is organized as follows:

First, we shall try to suggest typical processes that the specifier, unconsciously or not, applies when constructing formal specifications. We shall do this for both Z and GIST, but informally. Thus, the main questions will be: what can be done and what are the rules for deciding what to do?

The reasons for this choice are quite simple. First, as the scope of this thesis is limited, we cannot study in details more than two languages. Otherwise we would run out of time and never do anything worthwhile as a result. Second, we have chosen Z and GIST because Z is the language in which we have gained the most experience, and because Z and GIST appear to be representative of two distinct specification techniques.

Thus the more different the techniques are from each other, the better it is for telling apart the characteristics of the specification process that are independent or not from the language. This is also one of the reasons why the two studies have been carried out independently. For example, one can find rules from the rationale level that have been discovered in Z and which are not mentioned for GIST, and conversely. This has been done voluntarily, so as to be more beneficial for the development of a model for specification methods.

Finally, we shall study the specification processes suggested for both Z and GIST more formally. This study will consist of a comparison of the methods and ways of specifying in each language and, afterwards, we shall try to tell apart those aspects that are language-dependent and those that are not. Thus, the main question will be: what depends or not on the language?

4.2.1 INTRODUCTION AND CONCEPTS

The task of specifying a system within the framework of a formal language can be achieved with the help of models. At this level there are plenty of models available for describing objects, operations, constraints, the dynamics of the system and so on... These models allow the specifier to define the structure and the behaviour of the future I.S. within the limits of the language expressive power, as discussed above for three of them (Z, RML and GIST).

But what about the specification process itself? Can the activity of specifying lend itself to a formalization too? Many scientists have expressed their belief in the existence of models for the software specification activity (among whom [RS-2], [RS-3], [RS-4], [RS-5]). Also, we are convinced that at least some part of the specification process is mechanical, and so can be formalized in one way or another. Now we have to explore this path leading to a further formalization of the specification process.

Before formally defining models to deal with the various levels of the specification process, which will be done in the second and third parts of this section, we shall resort to more simple concepts. They will help us to approach the specification process in a given language.

A distinction is made between the notions of process and rationale. These concepts will allow us to make a clear distinction between what the specifier does and why he/she does it. This will be a basis for the definition of levels and models in the specification process more formally later, in the next part of this section.

In our analysis of specification processes in Z and GIST, we shall thus introduce the two following levels:

(i) **The processus level**: this level refers to what the specifier does, what he can do with the fragments of the specification already built, what kind of refinements he can do. This encompasses all the operations the specifier can apply to specifications: extensions, reuse, changes, verification, validation, etc...

(ii) **The rationale level**: this level refers to goals, tactics and strategies underlying the application of operation at the process level. In some cases, rationales may be expressed as rules telling which operations of the process level should be used in which cases.

These two levels will be studied for both Z and GIST.

4.2.2 SPECIFICATION PROCESSES IN Z

4.2.2.1 The Z process level

First we will identify typical primitives needed to manipulate Z specifications; these enable the handling of Z objects such as types, schemas, predicates, etc...

All such primitives will be characterized by a name, a list of parameters, an informal definition of their effect and one or more conditions of applicability (when required). Basic primitives add or modify something basic in the whole specification. But there will also be macro-primitives using basic ones. This will be mentioned explicitly whenever this is the case.

A. Primitive for set handling

CREATE_GIVEN_SET (name,given_set)

applicability: the name of the given_set is not used anywhere else in the specification for any other given_set

effect: produces the definition of a given set:

[name]

B. Primitive for variables handling

CREATE_VARIABLE (name, type, variable)

applicability: name is not used anywhere else in the specification for any other variable, etc...

effect: produces the definition of a variable:

name : type

C. Primitives for datatypes handling

CREATE_DATA_TYPE (data_type_name, list of alternatives, datatype)

applicability: name of the data_type is not yet used in the specification

effect: produces the definition of a datatype:

datatype ::= alt-1 | alt-2 | alt-3 | ... | alt-n

where alt-i are the constants that define the type *in extenso*

ADD_ALT_to_DATA_TYPE (datatype, alt')

applicability: alt' is different from alt-1, ..., alt-n

effect: adds an alternative to the definition of a datatype:

datatype ::= alt-1 | alt-2 | ... | alt-n | alt'

D. Primitives for schematypes handling

CREATE_SCHEMA (name, declarations(1) , predicates, schematype)

applicability: name is not used anywhere else in the specification for any other schematype

effect: produces the definition of a schematype, which can be written in two ways:

name $\triangleq$ [declarations | predicates]

or, using the Z schema notation:

name

declarations
predicates

MODIFY_DECLARATION_SCHEMA (schema, old_decl, new_decl)

applicability: new declarations are consistent with one another; all the variables defined in old declarations and used in the predicate part of the schema must be redefined in new declarations.

effect: modifies a set of declarations in a schematype:

schema

$declarations + \{new\_declarations\} \setminus \{old\_declarations\}$
predicates

MODIFY_PREDICATE_SCHEMA (schema, old_pred, new_pred)

applicability: new predicates are consistent with one another; all the new predicates must use variables defined in the declaration part of the schema.

effect: modifies a set of predicates in a schematype:

schema

declarations
$predicates + \{new\_predicates\} \setminus \{old\_predicates\}$

(1) declarations are variables + their types

The following primitives are not basic ones. They are typical compositions of some basic primitives previously defined.

EXTEND_SCHEMA (schema, new_declarations, new_predicates)

applicability: includes the conditions of applicability of the primitives used by this macro-primitive.

effect: adds a list of new declarations and new predicates to a schematype (with the possibility of one of these two lists being empty):

$$
\begin{array}{|l}
\hline
schema \\
\hline
declarations + \{new\_declarations\} \\
\hline
predicates + \{new\_predicates\} \\
\hline
\end{array}
$$

macro-primitive: EXTEND_SCHEMA is equivalent to

MODIFY_DECLARATION_SCHEMA (schema,{},new_decl)
MODIFY_PREDICATE_SCHEMA (schema,{},new_pred)

RESTRICT_SCHEMA(schema,old_declar,old_predic)

applicability: includes the conditions of applicability of the primitives used by this macro-primitive.

effect: subtracts a list of declarations and predicates from a schematype (with the possibility of one of these two lists being empty):

$$
\begin{array}{|l}
\hline
schema \\
\hline
declarations \setminus \{old\_declarations\} \\
\hline
predicates \setminus \{old\_predicates\} \\
\hline
\end{array}
$$

macro-primitive: RESTRICT_SCHEMA is equivalent to

MODIFY_DECLARATION_SCHEMA (schema,old_decl,{})
MODIFY_PREDICATE_SCHEMA (schema,old_pred,{})

AND_MERGE_OF_SCHEMAS (schema1, schema2, name, newschema)

applicability: the declaration parts of schema1 and schema2 do not contain contradictory declarations of the same variable; the conjunctive composition of the predicates of schema1 and schema2 do not produce predicates whose values are always "false".

effect: creates a new schema, whose name is given, as the conjunctiive composition of two other schematypes[1] :

new_schema

declaration of schema1
declaration of schema2
predicates of schema1
AND
predicates of schema2

macro-primitive: AND_MERGE_OF_SCHEMAS is equivalent to

CREATE_SCHEMA (name,
declarations = {declarations of schema1} U
{declarations of schema2}
predicates = (predicates of schema1) AND
(predicates of schema2)
newschema)

OR_MERGE_OF_SCHEMAS (schema1, schema2, name, newschema)

applicability: the declaration parts of schema1 and schema2 do not contain contradictory declarations of the same variable; the disjunctive composition of the predicates of schema1 and schema2 do not produce a predicate whose value is always false.

effect: creates a new schema, whose name is given, as the disjunctive composition of two other schematypes:

new_schema

declaration of schema1
declaration of schema2
predicates of schema1
OR
predicates of schema2

macro-primitive: OR_MERGE_OF_SCHEMAS is equivalent to

CREATE_SCHEMA (name,
declarations = {declarations of schema1} U
{declarations of schema2}
predicates = (predicates of schema1) OR
(predicates of schema2)
newschema)

(1) the primitives AND_MERGE, OR_MERGE and following could easily be adapted for n schemas, without too much trouble

INCLUDE_SCHEMA (schema, sub_schema)

applicability: the declaration part of schema and the subschema do not contain contradictory declarations of the same variable; the conjunctive composition of the predicates of the schema and the subschema do not produce a predicate whose value is always false.

effect: includes a schema into another one, which means that it will inherit both its declarations and its predicates:

schema

declarations
sub_schema
predicates

E. Primitives for modifying variables in schemas

RENAMING_VAR_IN_SCHEMA (schema,oldname,newname)

applicability: newname is not yet used in any other declaration of the schema.

effect: renames a variable in a schema, i.e. changes its name in both the declaration and the predicate parts:

schema [oldname / newname]

HIDING_VAR_IN_SCHEMA (schema,variable,newschema)

effect: hides a variable appearing in the declaration part of the schema, i.e. yields an equivalent schema where the declaration and the predicates concerning the given variable do not appear

newschema $\triangleq$ schema \ variable

DASH_VAR_IN_SCHEMA (schema)

effect: decorates a schema, i.e. adds a dash to each of its variable appearing in it. This dashed variables denote the states of the variables after an operation.

schema

declarations'
predicates'

4.2.2.2 The Z rationale level

This section is based upon our practice in Z specifications building. We shall try to put down in words the process that we followed when we treated the problems of the Library Problem (cfr chapter I) and the Oil Terminal (cfr Annex I). This "knowledge" about the process of building formal specifications from scratch will be expressed through both informal algorithms and rules. Thus, our approach will be both procedural and declarative, which seems to strengthen the idea of an expert system for assisting in the elaboration of formal specifications.

We chose this approach because, obviously, a pure procedural approach was felt far too rigid. We have chosen to separate the process into dataspace and operations modelling, but this does not necessarily mean that these modellings are independent. They are not, of course. This is just to classify matters.

The rules that will be stated will consist of 3 parts:

- a name: a short description of the rule purpose
- a premise: an informal text explaining the conditions for the rule to be triggered
- a consequence: a (set of) primitive(s) from the process level which will be applied if the rule is selected with its premise being satisfied, or some other consequences expressed informally.

First of all, we shall discuss dataspace modelling, and next operations modelling.

A. Modelling the data space

We have chosen to model the data space first, because it seems to us of the utmost importance as it lays the foundation for all what follows. We are convinced that it remains the keystone of our system. A good data model seems to lead to simpler and clearer operation structures, whereas an unsuitable may complicate them uselessly.

We first attempt to find meaningful objects; by this, we mean those that are significant to the future system. We carefully read the requirements and spot some relevant objects. Having done this, we select those which are useful and drop the useless ones, according to whether or not they are playing a part in the operations or not (1).

(1) this tends to show that no process is ever purely operation or object oriented...

The informal rule followed is thus:

rule n°1: selection of relevant objects

IF object is referred to in an operation
THEN object is relevant for specifications

Now that we have objects, what are we going to do with them? There exist several ways of modelling them: they can be unique, or there may be a set of them. They can express twice the same thing. They can be a part of another more general object. All these possibilities define different ways of modelling.

Here are some rules to help deciding what is the most convenient way for specifying them. These rules make use of primitives from the process level.

rule n°2: associating a given_set with an object identified

IF object is basic and cannot be decomposed
any further, and there are several
instances of it
THEN CREATE_GIVEN_SET (object_name)

example: BOOK, SUBJECT, AUTHOR, BORROWER were irresolvable objects in the Library Problem, unlike LIBRARY.

rule n°3: associating a datatype with an object identified

IF there exists a predefined finite amount of values for a unique object (alternatives)
THEN CREATE_DATA_TYPE (obj_name,alternatives)

example: MESSAGE ::= "enquiry_ok" ...

rule n°4: associating a variable with an object identified

IF object has one instance among a class of possible instances
THEN CREATE_VARIABLE (obj_name,type)

example: nbr_max : N

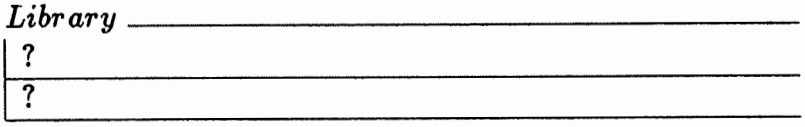
These rules create basic things in Z. This is not sufficient however to create more complex objects, such as schematypes, and, above all, the constraints attached to them. Whenever there is a variable or schematype creation, conditions on them often have to be made precise.

So let us consider the building of more complex objects, namely *schematypes*.. The first step is to spot a non basic object in the requirements:

rule n°5: associating a schematype with an object identified

IF object has not the same meaning as another object already defined
AND other objects seem to depend on it for their existence
THEN CREATE_SCHEMA_TYPE (obj-name,{},{})
(empty lists of declarations and predicates)

example: LIBRARY, OIL_TERMINAL are complex objects



Next, one checks if this object does not include any other existing object:

rule n°6: introducing inclusion relationships between schemas

IF object appears to include all the characteristics of another object
THEN INCLUDE_SCHEMA (object, subobject)

example: LIBRARY ≙ [LIBRARY ; USERS]. The library object includes the users.

If rule 5 or 6 has passed, we shall apply a macro-operator to include all the additional variables required in the schematype, among which are sets and relations.

foreach variable
do determine type;
declaration := <variable : type>;
EXTEND_SCHEMA (object,declaration,{})
determine and add predicates;
od

a. Determining the type of a variable?

Here are some rules for that purpose

rule n°7:

IF the value of the variable is an instance of an object
THEN type is the type of the object

example: nbr_max : N
(N is a predefined type in Z which refers to integer objects)

rule n°8:

IF the value of the variable is a set of instances of an object
THEN type is the powerset of the type of this object

example: copies: P COPY

rule n°9:

IF the value of the variable is a n-tuple of instances of n objects
THEN type is the cross product of the types of these objects

example: birthdate: DAY * MONTH * YEAR

rule n°10:

IF the value of the variable is one or more pairs of instances of two objects
THEN type is the type of a binary relation between these objects

example: $R : A \leftrightarrow B$

Of course, these rules can be applied recursively for the building of more complex types. For example:

marriage_celebration: MAN * WOMAN * (P GUESTS)

This means that the celebration of a marriage requires a man, a woman, and a set of guests. This has been obtained after applying **rule n°9** (cross product) and **rule n°8** (powerset). And we could apply once again the **rule n°8** to get:

$mariage\_in\_Liverpool : \mathbf{P}\ mariage\_celebration$

Whenever such rules are applied, and types are thereby defined, questions should arise in our minds. These should help detecting constraints upon variables, and thus finding relevant predicates to add to the schematype. So this answers a great part of the following question:

b. Attaching predicates?

rule n°11:
IF type of variable is a Powerset
AND IF there is a lower or upper limit, say l or L, to the size of the set
THEN EXTEND_SCHEMA (object,{},
predicate = { l < card(var) < L }

example:

$$
\begin{array}{|l}
\textit{object} \\
\hline
\textit{copies} : \mathbf{P}\ \textit{COPY} \\
\ldots\ldots \\
\hline
10 < \#\textit{copies} < 1000 \\
\ldots\ldots \\
\hline
\end{array}
$$

This was the question arising from a powerset type. Now here are the questions arising from the definition of a relation, they enable us to zero in on the nature of a relationship.

In Z, as soon as a relation is found, its domain and its range types must be defined accurately. And afterwards define the sets that will be related to each other this way:

$$
\begin{array}{|l}
\hline
\text{set\_A} \\
\text{set\_B} \\
R : A \leftrightarrow B \\
\hline
\mathbf{dom}\ R \subset \textit{set\_A} \\
\mathbf{ran}\ R \subset \textit{set\_B} \\
\hline
\end{array}
$$

What more could we learn about this relation? We are going to try and define it in such a way that no ambiguities whatsoever will remain regarding its nature.

The first five rules will permit to refine the rough statement made in the declaration part of our Z schema:

$$R : A \leftrightarrow B$$

Their right hand side will thus include a primitive to modify the statement of the relation in the declaration part of the schematype:

MODIFY_DECL_SCHEMA (object, old_decl={ R: A ↔ B}, new_decl={ R: A ? B})
(? stands for the definite relation)

rule n°12:

IF any occurrence of type A can be related to more than one occurrence of type B
THEN R is a relation and nothing must be modified
ELSE R is a function
MODIFY_DECL_SCHEMA
(object, old_decl={R:A↔B}, new_decl = { R: A→ B})

rule n°13:

IF any occurrence of type A must be related to at least one occurrence of type B
AND IF R is a function
THEN R is a total function
MODIFY_DECL_SCHEMA
(object, old_decl = { R: A↔B}, new_decl = { R: A→B})
ELSE R is a partial function
MODIFY_DECL_SCHEMA
(object, old_decl = { R: A↔B}, new_decl = { R: A⇸B})

rule n°14:

IF any occurrence of type B must be related to at least one occurrence of type A
AND IF R is a function
THEN R is a surjection
MODIFY_DECL_SCHEMA
(object, old_decl = { R: A↔B}, new_decl = { R: A↠B})

rule n°15:

IF any occurrence of type B must be related to at most one occurrence of type A
AND IF R is a function
THEN R is an injection
MODIFY_DECL_SCHEMA
(object, old_decl = { R: A↔B}, new_decl = { R: A↣B})

rule n°16:

IF R is an injection and a surjection
THEN R is a bijection
MODIFY_DECL_SCHEMA
(object, old_decl = { R: A↔B}, new_decl = { R: A⤖B})

The next four rules will permit to find out relevant cardinality constraints upon relations. We shall thus refine the predicate part of the schematype in their right hand side, using a primitive to extend it:

rule n°17:

IF a member of set_a cannot be related to more than N members of set_b (N finite)
THEN EXTEND_SCHEMA (object,{}, new_predicate is
(forall a: set_a * card ({a} ◁ R) <= N)

rule n°18:

IF a member of set_a must be related to at least N members of set_b (N finite)
THEN EXTEND_SCHEMA (object,{}, new_predicate is
(forall a: set_a * card ({a} ◁ R) >= N) and (dom R = set_a)

rule n°19:

IF a member of set_b cannot be related to more than N members of set_a (N finite)
THEN EXTEND_SCHEMA (object,{}, new_predicate is
(forall b: set_b * card (R ▷ {b}) <= N))

rule n°20:

IF a member of set_b must be related to at least N members of set_a (N finite)
THEN EXTEND_SCHEMA (object,{}, new_predicate is
(forall b: set_b * card (R ▷ {b}) >= N) and (ran R = set_b)

An example from the Library.

Several relations were spotted and we shall pick an example: the written_by relation. We had decided that a book had at least one author and, conversely, that an author had written at least one book, so the inputs to the "specification processor" are:

R = written_by
A = BOOK
B = AUTHOR
set_a = books
set_b = authors

And the following deductions could have been made, thanks to applicable rules:

rule n°12 ==> written_by is a relation
rule n°18 ==> **dom** written_by = books
rule n°20 ==> **ran** written_by = authors

This would have yielded the following schematype:

$$
\begin{array}{|l}
\hline
books : \mathbb{P}\ BOOK \\
authors : \mathbb{P}\ AUTHOR \\
written\_by : BOOK \leftrightarrow AUTHOR \\
\hline
\mathbf{dom}\ written\_by = books \\
\mathbf{ran}\ written\_by = authors \\
\hline
\end{array}
$$

After having defined variables and constraints associated with them, one should now have a look at the variables sharing the same types within a same schematype. The constraints we have defined so far only bear upon a variable itself, overlooking its environment within the declaration part of the schema. There should be constraints implied by interferences between variables who share the same types. This will be made clearer by an example from the Library:

users

$$
\begin{array}{|l}
\hline
staff : \mathbb{P}\ PERSON \\
borrower : \mathbb{P}\ PERSON \\
\hline
\end{array}
$$

This states that the object USERS is made up with two other objects, a set of staff and a set of borrowers, who are both defined upon the same type PERSON. So a comparison is possible between them. Then the following question should arise: can a member of the staff set be a member of the borrowers set as well? If the answer is no, then there should be an additional constraint to the schema:

$$staff \cap borrower = \emptyset$$

The general rule for managing these interferences between variables will be:

rule n°21:

IF two or more variables, whose types are sets, have the same type
AND IF the intersection or union of some of them must be an empty or a particular set
THEN EXTEND_SCHEMA (object,{},new_predicates)

The new predicates should be refined correspondingly through more precise rules, which we shall not explain further here.

B. Modelling the operations

If we want to satisfy the customer's needs, the first thing to do is to read the requirements in order to construct the list of all the operations required.

Example from the Oil Terminal Control System[1] :

We saw the system functioning through the controller eyes and identified the events that trigger the main operations. Apparently, four major things set the system in motion and define its main operations:

1. The controller turns the computer on, causing its initialization.
2. He switches on the "arrival button" to signal an approaching tanker to the system and waits for instructions from it.
3. He depresses the "departure button"
4. He wants information about tankers and berths

Example from the Library Problem:

In this case, our search for operations was much easier since the requirements were shaped accordingly. We just have to take operations straight away from the text: check-out-copy, return-copy, etc... So it all depends on the way requirements are presented. Some presentation are more suited for spotting objects and some others for spotting operations.

Next we must build each operation, step by step.

CREATE_SCHEMA (operation,{},{})

We can decompose an operation using two different strategies: the former consists in a study of the precondition of the current operation, and the latter is based on a study of the objectives of the operation.

---

[1] Annex 1.

The following procedure is a rough description of the process we have followed in Z in order to build operation schematypes. It mainly applies a **forward top-down strategy**[1]. This procedure is recursive as it sometimes leads to a decomposition of operations which will have to be defined in turn.

```
foreach operation
 do
    determine dataspace
    EXTEND_SCHEMA (operation,dataspace,{})
    determine inputs
    determine preconditions (= predicates on the data-space and on Inputs)
    if different preconditions  (cfr infra)
          then decompose suboperation
          else determine outputs
               determine postconditions (= pred. on I/O)
 od
```

a. How to determine the dataspace?

What is the part of the dataspace useful to the operation? The specifier will chose the schematype which encompasses all the data necessary to the operation.

What will be the status of the schematype in this operation?

rule n°22:

IF the operation modifies the dataspace
THEN EXTEND_SCHEMA (operation, new_decl = { Δ dataspace},{})
ELSE EXTEND_SCHEMA (operation, new_decl = { Ξ dataspace},{})

b. How to determine the inputs?

Reading the requirements regarding the operation enables to find out the input parameters. Each one of these is declared as a variable with an interrogation mark behind:

example: EXTEND_SCHEMA (operation, {copy? : COPY}, {})

---

(1) forward means that the way in which operations are defined progressively depends only on properties of arguments (results for a backward strategy); top-down means that the operation is progressively specified thanks to decompositions (aggregations for bottom-up).

c. How to determine the preconditions?

Knowing all the input parameters and which parts of the data state we need, we concentrate on the possible values of our input parameters and on the properties of the data space.

With those possible values, we can define the preconditions of our schema. But, beforehand, we need to construct an "analysis table" to find out the exhaustive list of possible preconditions (PRE_i).

Let us consider a simple example where <var1> and <var2> are both input parameters of an operation. <var1> can satisfy or not a condition, say <v1>. Likewise, <var2> can also be constrained or not by a predicate, say <v2>.

The following possibilities thus can arise:

PRE_1 = (v1) **and** (v2),
PRE_2 = (v1) **and not** (v2),
PRE_3 = **not** (v1) **and** (v2),
PRE_4 = **not** (v1) **and not** (v2).

Finally, the precondition PRE of this operation is then defined by taking the disjunctive composition of all the PRE_i.

Nevertheless, it remains possible to regroup some of those preconditions by taking their disjunctive composition, in case a common unique schema is to be associated with them. Thanks to this regrouping, we can, for example, determine the preconditions of an error handling operation.

d. How to decompose an operation?

Now, we can define a new operation (OP_i) corresponding to each PRE_i we have expressed:

```
foreach <PRE_i>
  do
        if operation_i does not exist
        then CREATE_SCHEMA
                (operation_i,{dataspace, inputparam},{<PRE_i>})
        endif
        OR_MERGE_OF_SCHEMA (operation, operation_i, operation)
  od
```

Note that an operation$_i$ could have been modelled during the specification of a previous operation[1]; thus, we need to verify if the sub-operation$_i$ has not been modelled yet.

Example: Using this rationale, the operation "RETURN_COPY" of the "Library Problem" was built. This operation is the result of the disjunctive composition of three other operations:

(i) "RETURN_A_COPY_NO_PROBLEM" corresponds to a possibility PRE$_i$,

(ii) "RETURN_A_COPY_WITH_A_PROBLEM" corresponds to the logical conjunction of all the other PRE$_i$ without taking into account the userid condition, and

(iii) "STAFF_FAILURE" corresponds to the violation of the userid condition. But, we didn't need to specify it: it was already done for the procedure "CHECK_OUT".

If we had followed a **backward** strategy, the decomposition of the operation would have been driven by the structure of the results.

Knowing the objective of the operation, we try to break it up in a series of sub-operations (SUBi) which could be defined separately and which together could achieve the objective. The operation will be defined as the conjunctive composition of each Z sub-operation. Thus, we have:

foreach <sub-operation$_i$>
do
if sub-operation$_i$ does not exist
then CREATE_SCHEMA(sub-operation$_i$,{},{});
endif
AND_MERGE_OF_SCHEMAS (operation, sub-operation$_i$,operation)
od

A sub-operation$_i$ could have been modelled during the specification of a previous operation[2]. Thus, we need to verify if the sub-operation$_i$ has not been modelled yet.

Example: Using this rationale, we built the operation "CHECK_OUT_A_COPY_IF_NO_PROBLEM" of the "Library Problem". We specified first the operation "CHECK_OUT_A_COPY_IF_NO_PROBLEM" without taking into account the userid condition. Then, we included the schema "STAFF_MEMBER" which looks whether the user of the operation is a staff member or not.

---

(1) see for example, the operation STAFF_FAILURE which is used in a lot of operations of the "Library Problem".

(2) see for example, the operation STAFF_MEMBER which is used in a lot of operation of the "Library Problem".

e. How to determine the outputs?

Reading the requirements regarding the operation enables to find out the output parameters. Each one of these is declared as a variable with an exclamation mark behind. But, we also need to express those Z predicates that will fix the value of all the output parameters.

These two rationales operations define the following procedure:

foreach <outpar>
do
determine type of <outpar>: <outtype>
EXTEND_SCHEMA (operation,{<outpar>!:<outtype>},{})
determine Z predicate which can fix the <outpar>! value: <predic>
EXTEND_SCHEMA (operation,{},<predic>)
od

Example: The execution of this procedure gives for the operation "GET_LIST_AUTHOR_NO_PROBLEM" of the "Library Problem":

EXTEND_SCHEMA (operation,{books!:**P** BOOK},{});
EXTEND_SCHEMA (operation,{}, { books! = **dom** (written_by {author?}) });
EXTEND_SCHEMA (operation,{mess!:MESSAGE},{});
EXTEND_SCHEMA (operation,{},{mess!:enquiry_ok});

f. How to determine the postconditions?

Reading the requirements regarding the operation enables to find out the postconditions. Each one of these will translated using a Z predicate. For the sake of completeness, we need to study each set and relation defined in the dataspace part used:

foreach <set> in dataspace
do
determine effect of the operation on <set>
translate effect into a Z predicate: <predic>
EXTEND_SCHEMA (operation, {},<predic>)
od

and in the same way for the relations,

foreach <relation> in dataspace
do
determine effect of the operation on <relation>
translate effect into a Z predicate: <predic>
EXTEND_SCHEMA (operation, {},<predic>)
od

g. Conclusions

We have followed processes that seem to be applicable to any problem, but this is not always the case. These worked for the problems we dealt with, but they may not suit any problem. Our limited experience in the art of specifying cannot yield any definite conclusions at this stage.

One could have expected the strategies suggested for the decomposition of operations to be based upon the structure of the input or output types. But our study relies heavily on the practical examples we have treated in Z and, as they are not the same size as large-scale problems, most of these types were simple. Thus, no interesting rationale has been discovered at this stage. But, this question will be tackled later, when a model for the methods of specification is considered more formally. The same remark applies to the lack of strategies concerning specialization and generalization of objects. This will also be considered later on.

4.2.3 SPECIFICATION PROCESSES IN GIST

We shall begin our analysis of the GIST specification process by looking for the GIST basic primitives belonging to the process level[(1)].

Only the more important operators are proposed since we do not have a complete definition of the language at our disposal. Moreover we should also add that our command of GIST has not reached as high a level as the one we have reached in Z. After this operator identification step, we shall try to define those rationales which help the specifier formalizing requirements.

4.2.3.1 The GIST process level

We propose a list of primitives which will be necessary for us to be able to build a GIST rationale level.

For each primitive, we shall give its name and both its input and output parameter(s). Then we shall informally define the goal of the primitive and its effect on the specification. If our primitive is constrained by a precondition, we shall also state it informally.

Some of these primitives can be "run" automatically and some other ones require the specifier's assistance when a decision has to be taken.

A. Add_comments <text>

This operator enables the specifier to create an occurrence of the following GIST phrase:

Spec comment
<text>
end comment ;

where <text> is an input parameter of the operator.

This occurrence will be inserted after the last element specified.

(1) "primitives" of the process level can also be referred to as "operators".

B. Exist_type_name <type_name>

This operator inspects the current GIST specification to know whether a given type name is already used or not.

if <type_name> already defined,
then Exist_type_name:= true
else Exist_type_name:= false;

If it does not currently exist, "Exist_type_name" takes the value "false", otherwise it will take the value "true".

C. Create_type <type_name>

This operator creates a new instance of the following GIST phrase:

type <type_name> ;

where <type_name> is an input parameter of the operator.

Precondition: Exist_type_name<type_name> must have false as value. Otherwise, it will introduce inconsistencies.

D. Delete_type <type_name>

This operator deletes the instance of the following GIST phrase from the specification:

type <type_name> ;

where <type_name> is an input parameter of the operator.

Precondition: Exist_type_name<type_name> must be equal to false.

E. Create_supertype <list_of type_name, supertype_name>

Using this operator, the specifier can create a new instance of the following GIST phrase:

<supertype_name>() **supertype_of** <list_of type_name> ;

where <list_of type_name> and <type_name> are both input parameters.

To respect the GIST syntax, each member of the "list_of type_name" must be separated from the following one by a comma.

If Exist_type_name<supertype_name> is equal to false, we create the whole GIST phrase. Otherwise, we only add the <list_of type_name> to the existing one.

Precondition: For each member of <list_of type_name>, Exist_type_name<type_name> must be true.

F. Create_set_of <type_name_1, type_name_2>

Using this operator, the specifier can create a new instance of the following GIST phrase:

type <type_name_2> = **set_of** <type_name_1> ;

where <type_name_1> and <type_name_2> are both input parameters.

Precondition: Exist_type_name<type_name_2> must be equal to false and Exist_type_name<type_name_1> must be equal to true. Otherwise, it will also introduce inconsistencies.

G. Exist_relation_name <relation_name>

This operator inspects the current GIST specification to know whether a given relation name <relation_name> is already used or not.

if <relation_name> already defined,
then Exist_relation_name:= true
else Exist_relation_name:= false;

If it does not presently exist, "Exist_relation_name" takes the value "false", otherwise it will take the value "true".

H. Create_Relation <list_of type_name, relation_name>

Using this operator, the specifier can create a new instance of the following GIST phrase:

relation <relation_name> (<list_of type_name>);

where both <list_of type_name> and <relation_name> are input parameters.

To respect the GIST syntax, each member of the "list_of type_name" has to be separated from the following one by a comma.

If Exist_relation_name<relation_name> has false as value then the GIST relation is created. Otherwise, it <list_of type_name> is only added to the existing one.

I. Specify_cardinality <type_1, rel_name, type_2, card1, card2>

Using this operator, the specifier can modify the connectivity constraint defined upon the relation <rel_name> linking <type_1> to <type_2>. The other relations using <type_1> or <type_2> are left unchanged.

Thus, we have the following GIST phrase:

if <card1> or <card2> are equal to a GIST predefined value
then
type <type_1> (<rel_name>|<type_2>:<card1>::<card2>);

where <type_1>, <rel_name>, <type_2>, <card1> and <card2> are all input parameters.

But, if both <card1> and <card2> are not equal to a GIST predefined value, the specifier will insert GIST constraints such as, e.g., the following ones:

always required for all t1: type1
cardinality rel_name(t1,*) <= <card1>
and
cardinality rel_name(t1,*) >= <card2>

and,

always required for all t2: type2
cardinality rel_name(*,t2) <= <card1>
and
cardinality rel_name(*,t2) >= <card2>

where w,x,y,z have to be determined by the specifier reading <relation_text> corresponding to <rel_name>.

Preconditions:

(i): The value of <card1> or <card2> must be either **any, unique, multiple** or **optional** or **undefined.**

(ii): The three following functions must return the value "true": Exist_type_name<type_1>, Exist_type_name <type_2> and, Exist_relation_name<rel_name>.

J. Exist_procedure_name <procedure_name>

This operator inspects the current GIST specification to check whether a given procedure name is used or not:

if <procedure_name> already defined,
then Exist_procedure_name:= true
else Exist_procedure_name:= false;

If it does not presently exist, "Exist_procedure_name" takes the value "false", otherwise it will take the value "true".

K. Init_a_procedure <proc_name>

Thanks to this operator, the specifier will be able to create an instance of the following framework of a GIST procedure:

procedure <proc_name>
atomic
<?>
end_atomic ;

where <proc_name> is an input parameters.

Precondition: Exist_procedure_name <proc_name> must have false as value. Otherwise, it will introduce inconsistencies.

L. Exist_daemon_name <daemon_name>

This operator inspects the current GIST specification to know whether a given daemon name is already used or not.

if <daemon_name> already defined,
then Exist_daemon_name:= true
else Exist_daemon_name:= false;

If it does not presently exist, "Exist_daemon_name" takes the value "false", otherwise it will take the value "true".

M. Init_a_daemon <dem_name>

Thanks to this operator, the specifier will be able to create an instance of the following framework of a GIST daemon:

daemon <daem_name>
trigger <?>
response <?> ;

where <daem_name> is an input parameter.

Precondition: Exist_daemon_name <daem_name> must be have false as value. Otherwise, it will also cause consistency problems.

N. Exist_agent_name <agent_name>

This operator inspects the current GIST specification to know if a given agent name is used or not.

if <agent_name> already defined,
then Exist_agent_name:= true
else Exist_agent_name:= false;

If it does not currently exist, "Exist_agent_name" takes the value "false", otherwise it will take the value "true".

O. Add_an_agent <agent_name, list_of_proc, list_of_dem>

Thanks to this macro-operator, the specifier will be able to create an instance of the following framework of a GIST agent:

agent <agent_name>
{
<list_of_proc>
<list_of_dem>
};

where <agent_name>, <list_of proc> and <list_of dem> are all input parameters of the operator.

If Exist_agent_name<agent_name> is equal to false then the GIST framework agent is created. Otherwise, both the <list_of proc> and <list_of dem> are added to the existing one.

Then, for each procedure of <list_of_proc>, we have to apply the operator "Init_a_procedure <proc_name>" before specifying the contents of the GIST procedure.

And finally, for each daemon of <list_of_daem>, we apply the operator "Init_a_daemon <dem_name>" and specify both what triggers the daemon and what its effect on the system is.

One can define this macro-operator more formally as the sequential composition of the following operators:

```
if not Exist_agent_name<agent_name>
        then Create the agent framework;


for each procedure in list_of_proc
        do
                Init_a_procedure <proc_name>
                specify procedure
        od


for each daemon in list_of_daem
        do
                Init_a_daemon <daem_name>
                specify what triggers the daemon
                specify the daemon's response
        od
```

P. Add_predicate <object_name, constraints>

This operator adds a constraint on a GIST object whose name is <object_name>.

Precondition: This constraint must have one of the following form:

Always required <object_name>
<constraints>

Or,

Always prohibited <object_name>
<constraints>

Q. Set_Assumption <(type or rel or oper or envir), text>

This operator records the informal text of the assumption <text> made by the specifier about a specification component: <type>, <rel>, <oper> or <envir>.

R. Remove_Assumption <(type or rel or oper or envir), text>

This operator erases the informal text of the assumption <text>, made by the specifier about a specification component: <type>, <rel>, <oper>, or <envir> out the specifier memory.

S. Consult_Assumption <(type or rel or oper or envir), text>

Thanks to this operator, the specifier will be able to select an assumption he has made on a specification component.

If an assumption has been recorded, it will be placed into <text> otherwise, <text> receives the value "no assumption".

4.2.3.2 The GIST rationale level

Using the previous (macro-)operators, we should be able to propose a study of the GIST rationale level. We shall analyze what the rationales are behind the modelling of not only the data and relations but also the operations and the environment.

However, our GIST experience and readings (see [ML-12], [EC-5]) also give us the feeling that the assumptions, which we can do during the entire specification process, play a determinant role in the quality of the specifications. By removing them, the specifier gradually elaborates the final specifications. Therefore we shall take into account another view point: "Model: Set/Remove Assumptions"

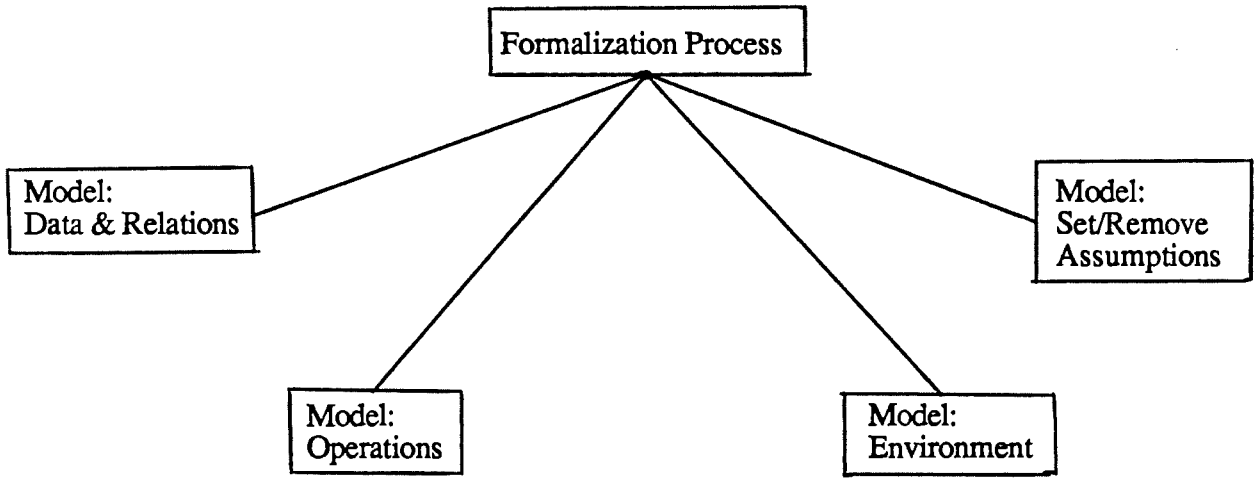


Figure GIST.16 The GIST rationales

These four view points will be presented sequentially, but the GIST specifier will actually use them **always concurrently.**

A. Model Data & Relations

To model an object spotted in the informal requirements, the GIST specifier must choose between three operators defined at the GIST process level. An object will only be selected if it plays a part in the specification of the GIST procedures or daemons.

Thus, a type related to the given <object_text> will be created or modified. The specifier has to find out a good <type_name>: it must be meaningful to improve the readability of the specifications.

rule n°1:

if object is related to a procedure or a daemon
then object is selected **and** named.

According to the current state of the data specifications, the specifier must decide what the most convenient way for modelling the selected object is. At the GIST process level, we have defined three operators to model objects:

(i) Create_type <type_name>,

This one is used when, on looking all the existing data-type, the specifier cannot found any other type which can be semantically related to the new type.

(ii) Create_supertype <list_of typename, type_name>,

If the specifier is able to establish a semantic link between the existing types and the new one, this operator can be called.

The specifier will create the **aggregation** of a list of types.

(iii) Create_set_of <type_name_1, type_name_2>.

If the specifier is able to establish a semantic link between the existing types and the new one, this operator could also be called.

Using this operator, he will be able to create a new type by taking the **classification** of a simple one.

Thus, we have the three following rules:

rule n°2:

if <type_name> cannot be linked to any other one
then begin
Create_type <type_name>;
Add_comments <object_text>
end

rule n°3:

if <type_name> can be aggregated to another one
then begin
Create_supertype <list_of typename, type_name>,
Add_comments <object_text>
end

rule n°4:

if <type_name> is a set of objects which are already defined by <type_name_1>
then begin
<type_name_2> := <type_name>,
Create_set_of <type_name_1, type_name_2>,
Add_comments <object_text>
end

During the specification process, the GIST specifier will also create or modify relations between existing object types. If it is a new relation, the specifier will have to find out a good <relation_name>: it must also be meaningful.

rule n°5:

if a relevant relation is found
then Create_Relation <list_of_type_name,relation_name>

Thus, once this <relation_name> is chosen, the specifier uses the operator "Create_Relation <list_of_type_name, relation_name>". In order to do this, he has also to translate the <list_of objects> into <list_of typename>.

If no assumption has been done on the cardinality of the relation, the specifier will also study it. To know if assumptions have been done, the operator "Consult_Assumption <relation_text, text>" must be used.

Thus, we define the following rules:

rule n°6:

if one and only one object of <type2> can be related to <type1> being defined,
then Specify_cardinality <type_1, rel_name, type_2, **unique**, ?>

rule n°7:

if <type2> must be related to <type1> at least once,
then Specify_cardinality <type_1, rel_name, type_2, **multiple**, ?>

rule n°8:

if <type2> is attributed either to 0 or 1 <type1>,
then Specify_cardinality <type_1, rel_name, type_2, **optional**, ?>

rule n°9:

if <type2> is attributed to an undefined number of <type1>,
then Specify_cardinality <type_1, rel_name, type_2, **any**, ?>

rule n°10:

if <type1> can have as attribute one and only one object of <type2>,
then Specify_cardinality <type_1, rel_name, type_2, ?, **unique**>

rule n°11:

if <type1> can have as attribute at least one object of <type2>
then Specify_cardinality <type_1, rel_name, type_2, ?, **multiple**>

rule n°12:

if <type1> can have as attribute either 0 or 1 object of <type2>
then Specify_cardinality <type_1, rel_name, type_2, ?, **optional**>

rule n°13:

if <type1> can have as attribute an undefined number of object of <type2>
then Specify_cardinality<type_1, rel_name, type_2, ?, **any**>

Nevertheless, the specifier could be unable to determine either <card1> or <card2> after following the "algorithm", i.e. if <card1> should be greater than 3 and smaller than 7.

In such cases, the next two rules are necessary:

rule n°14:

if <card1> is more complex than the predefined values
then
begin
find both lower and upper limits
Specify_cardinality <type_1, rel_name, * , lower, upper>
end

rule n°15:

if <card2> is more complex than the predefined values
then
begin
find both lower and upper limits
Specify_cardinality <*, rel_name, type_2, lower, upper>
end

B. Model Operations

The first thing to do is to read the requirements in order to construct an exhaustive list of the operations which must be modelled to satisfy the customer's needs.

rule n°16:

if there remains a non-selected operation
then add this operation to the list of accurate operations.

Then for each operation of the list, we have to specify a GIST procedure. But, at this stage, we only need to specify the post-conditions of this one.

macro-operator n°1:

```
for each operation in the list
  do
        choose a name for this operation --> <procedure_name>
        if not Exist_procedure_name <procedure_name>
              then Init_a_procedure <procedure_name>
        endif
        specify the post-conditions
        Add_comments <operation_text>
  od
```

But, these GIST procedures should be used as a daemon's response, if we want to be able to "execute" them. The first thing we define as triggering the daemon is **RANDOM()**. It is a very crude approximation of the pre-conditions of the operation. Thus, we define the following rule:

macro-operator n°2:

```
for each operation in the list
  do
        select a <procedure_name>
        if not Exist_daemon_name <"D_"+procedure_name>
              then Init_a_daemon <"D_"+procedure_name>
        endif
        init the trigger part of the daemon with RANDOM()
        daemon's response := <procedure_name>
        Add_comments <daemon_text>
od
```

As a matter of fact, thanks to this last rule, the following GIST pattern has been defined:

Daemon D_<procedure_name>
Trigger RANDOM()
Response <procedure_name>;

C. Model Environment

Now, the specifier must focus his attention on the environment features which are described in the informal requirements. If environment features are found then they should be modelled using a GIST agent. In [EC-5], such features have been imagined and modelled.

Thus, the following rationale level rule has to be introduced:

rule n°17:

if an environment feature is described
then
begin
look for a name ⟶ <agent_name>
Add_an_agent <agent_name, {}, {}>
end

But, we must add to the new agent the procedures and daemons required by its modelling:

rule n°18:

if procedures are under the responsibility of the agent
then
begin
look for procedures ⟶ <list_of_proc>
foreach d in list_of_daemon
do d = "D_"+ <corresponding_proc> **od**
Add_an_agent <agent_name, {list_of_proc}, {list_of_daemon}>
end

Thanks to the application of the macro-operator n°1 and n°2, we can also specify the contents of the procedures in the list and the list of daemons.

D. Model Set/Remove Assumptions

During the GIST specification process, assumptions must be made in order to concentrate all the specifier's efforts on an idealized and simplified version of the future system. Thus, we need to define the following rules to set these assumptions:

rule n°19:

if we define a type_name
then Set_assumption (<type_name>, "No relation with other existing type")

We assume that the variables of this type do not have an influence on other variables, e.g., constraining the existence of another one.

rule n°20:

if we define a type_name which is a **set_of** <another_type_name>
then
Set_assumption (<type_name>,
"Nothing constrains the size of the variables of this new type.")

rule n°21:

if we define a type_name which is a **supertype_of** <other_type_names>
then
Set_assumption (<type_name>, "Nothing news constrains the new_type.")

rule n°22:

if we define a relation
then
Set_assumption (<relation_name>, The cardinality is not yet determined.")

rule n°23:

if we define a procedure
then
Set_assumption (<procedure_name>, "The pre-conditions are met.")

The previous list of rules is probably not exhaustive, but it sketches the kind of rules that are required to set the assumptions during the specification process.

But, all these assumptions have to be removed to adjust the specification and finally specify the real customer's needs. Therefore, the following macro-operator is required to select which assumption we want to remove:

macro-operator n°3:

begin
select one "things defined in GIST "
Consult_assumption ("things defined in GIST", <text>)
if <text> **no equal to** "no assumption"
then select this text ⟶ <assumption>
end

The selection of the "things defined in GIST" is submitted to a specifier heuristic like:

choose first a "type",

if no assumption on "type" **then** choose a "relation",

if no assumption on "relation" **then** choose a "procedure",

if no assumption on "procedure" **then** choose a "daemon".

Once an assumption has been selected, the following macro-operator must be applied to refine the specification and remove the assumption:

macro-operator n°4:

begin
Remove_assumption ("things defined in GIST", <assumption>)
study the <assumption> to determine what it changes
update the specification
end

The updating of the specification could consist of the application of either the rationale level rules n°6 to 15 or the following ones:

rule n°24:

if (we want to remove an assumption on a <Gistset> which is a **set_of**)
and
(<assumption> = "Nothing constrains the size of the variables of this new type.")

then
begin
study the cardinality of <Gistset> ⟶ lower and upper values
Add_predicate (<Gistset>,"lower < **cardinality_of** <Gistset>")
Add_predicate (<Gistset>,"**cardinality_of** <Gistset> < upper")
end

rule n°25:

if (we want to remove an assumption on a **procedure**)
and
(<assumption> = "The preconditions are met.")
then
begin
find the corresponding daemon ⟶ <daemon>
determine the preconditions ⟶ <pre-conditions>
"trigger part" of <daemon> =
"trigger part" of <daemon> + "AND" + <pre-conditions>.
end

The application of this last rationale level rule was necessary to build the GIST procedure "ADD_A_COPY" of the figure GIST.11 (p. 82). We should also profit by the application of this rule to create another daemon triggered by the violation of one of the preconditions.

The previous list of rules is not exhaustive either. But it sketches the kind of rules which are required to remove the assumptions.

This section attempts to model the gradual building of formal specifications. Thanks to our little yet interesting exercises in both Z and GIST, we have gathered some hints and ideas about the modelling process.

We shall base our work mainly on the model proposed by Dubois-Van Lamsweerde in [RS-2], adapting it slightly. We shall therefore recall the principles of it and next, explain the adaptation. Actually we have extended it so far as to be applicable to Z and GIST too, and so become more general.

4.3.1 THE DUBOIS-VAN LAMSWEERDE MODEL[1]

The authors make a clear distinction between the three following levels:

(i) the specification product level, at which the various operations and object types of the system are defined,

(ii) the process level, at which the various operations of the specifier are defined. These are the meta-operations applied successively by the specifier when he/she incrementally constructs specification fragments from the level below.

(iii) the method or rationale level, at which the reasons underlying the choice and the application of each specifier's operation from the level below are defined.

Two models are proposed based on that distinction:

(i) **a process model** for capturing the description of the application of the specifier's operations;

(ii) **a method or rationale level** for capturing the description of the control of these operation applications.

[1] title by default !

4.3.2 THE PROCESS MODEL REVISITED

We propose to split the process model into two sub-models, one which is language-specific and the other which is language-dependent. We have found this more suitable so as to take into account those aspects of this process that depend on the specific language being used and those that do not.

4.3.2.1 The language-specific process sub-model

This model consists of a set of those operators handling the structures and concepts of a formal language which depend on the syntax of the language; thus all of the language-dependent features are found here.

In the same way as in a logical architecture, built according to the principles of methodical software development, these primitives will be "used" (as defined in [DI-8]) by the operators of the language-independent level above.

These primitives are the ones identified informally for the process level in the previous sections(1). Now this should be done more rigourously (in the event of an actual application of this model). This means that the parameters of the primitives should be defined more accurately, as well as the syntax and semantics of these primitives. Also it could be worth defining other non-basic primitives or macros, derived from basic primitives. This is due to their recurring use in algorithms of the above level primitives.

Here are some examples of such primitives:

A. Primitives dealing with Z structures

CREATE_GIVEN_SET (name, given_set)
CREATE_VARIABLE (name, type, variable)
CREATE_DATA_TYPE (data_type_name, list_of_alternatives, datatype)
CREATE_SCHEMA (name, declaration, predicates, schematype)
EXTEND_SCHEMA (schema, new_declarations, new_predicates)
RESTRICT_SCHEMA (schema, old_declar, old_predic)
MODIFY_DECLARATION_SCHEMA (schema, old_decl, new_decl)
MODIFY_PREDICATE_SCHEMA (schema, old_pred, new_pred)

(1) "The Z process level" and "The GIST process level".

AND_MERGE_OF_SCHEMAS (schema1, schema2, name, newschema)
OR_MERGE_OF_SCHEMAS (schema1, schema2, name, newschema)
INCLUDE_SCHEMA (schema, subschema)
RENAMING_VAR_SCHEMA (schema, oldname, newname)
HIDING_VAR_SCHEMA (schema, variable)

B. Primitives dealing with GIST structures

ADD_COMMENT (text)
EXIST_TYPE_NAME (type_name)
CREATE_TYPE (type_name)
CREATE_SUPER_TYPE (list_of_type_names, type_name)
CREATE_SET_OF (type_name_1, type_name_2)
EXIST_RELATION_NAME (relation_name)
DELETE_RELATION (relation_name)
CREATE_RELATION (list_of_type_names, relation_name)
SPECIFY_CONNECTIVITY (type1, rel_name, type2, con1, con2)
ADD_PREDICATE (object_name, constraints)
EXIST_PROCEDURE_NAME (procedure_name)
INIT_PROCEDURE (procedure_name)
EXIST_DAEMON_NAME (daemon_name)
INIT_DAEMON (daemon_name)
EXIST_AGENT_NAME (agent_name)
ADD_AN_AGENT (agent_name, list_of_proc, list_of_dem)
SET_ASSUMPTION ({type or rel or oper or envir}, text)
REMOVE_ASSUMPTION ({type or rel or oper or envir}, text)
CONSULT_ASSUMPTION ({type or rel or oper or envir}, text)

4.3.2.2 The language-independent process sub-model

At this level one can find general primitives used by the rationale level, e.g., general abstraction mechanisms that allow the stepwise elaboration of formal specifications for large systems.

As a matter of fact, all the formal languages that we have studied so far permit the use of well-known mechanisms such as decomposition, generalization, classification, etc... But most of the time, not explicitly. For some of them this is quite straightforward and natural, but for some others these ideas of abstraction lurk beneath the surface of the language structures.

This is the reason why we have chosen to introduce this conceptual "layer" between the rationale level and the language specific process level. This bridges the gap existing between languages in which abstraction plays an important part, and those who are further from it. For example, Z does not provide a direct generalization mechanism whereas GIST does. But the language-independent process sub-model will "emulate" this mechanism in Z. It will utilize primitives of the language-dependent process sub-model, which will implement it thanks to schema inclusion.

Thus we have a set of primitives that provides a stable background for the rationale level and hides the particularities of a language as much as possible.

Of course, this will not make the rationale level totally independent of the language. This is mainly due to the fact that, even though we shall try to have abstraction mechanisms at our disposal for every language, it is nevertheless clear that the result of the application of an abstraction mechanism may not always give interesting results in that language. We may not obtain specifications that have interesting virtues such as readability or easy understanding.

Now we shall try and define the primitives and their contents. Afterwards, we shall illustrate how they "use" the language-specific process level primitives written for both Z and GIST. Some of them will be fairly simple, in view of their availability in the target formal language.

For each primitive we will present:

1. ITS NAME
2. THE INPUTS AND PRECONDITIONS
3. THE OUTPUTS AND POSTCONDITIONS
4. AN "ALGORITHM" written in terms of Z specification-level primitives
5. AN "ALGORITHM" written in terms of GIST specification-level primitives

A. The primitives

First, we shall state those primitives that deal with object types, namely creation, specialization, etc...

a. CREATE_OBJECT_TYPE

Input: name, values = {known a priori, indefinite},
list_of_values (if known a priori)

Precondition: name is not the name of any other existing object

Output: initial specification of object_type

Postcondition: the new resulting specification state contains an initial definition of the object type.

algorithm in Z:

```
begin
    if values = known a priori

    then CREATE_DATA_TYPE
                (data_type_name = name,
                 list_of_alt = list_of_values,
                 datatype = object_type)

    else CREATE_GIVEN_SET
                (name = name, given_set = object_type)
end
```

algorithm in GIST:

```
begin
    CREATE_TYPE (type = name)
end
```

At this level we shall also have a table of correspondences between object types and their actual representations in a target formal language. This is necessary for these primitives to work properly on object types of the level above. They must know how object types are represented, because they can be modelled in many different ways.

An example of such a table could be:

Object Types	Representation in Z	Representation in GIST
BOOK	[BOOK] or BOOK ::= . | ...	**type** BOOK
class of BOOK	┌─BOOKS ────── set_book:P BOOK └────────	BOOKS = **set_of** BOOK
...	...	...

b. AGGREGATE_OBJECT_TYPES

Input: list_of_object_type, name_new_object_type

Precondition: the name of the new object is not the name of any other existing object; all the object types in the list have been defined previously.

Output: new_object_type

Postcondition: the new resulting specification state contains a new object type which is the aggregation of several other object_types given as inputs

algorithm in Z: creation of a new schema with the list of object types in its declaration part, with the following syntax: { variable = name of object type in small print, type = name of object type in capital print}

```
begin
    CREATE_SCHEMA
        (name = name_new_object_type,{},{},
         schematype = new_object_type)
    foreach object_type in list_of_object_types
        do EXTEND_SCHEMA
            (schema = new_object_type,
             new_decl = { object_type : OBJECT_TYPE},{})
        od
end
```

algorithm in GIST:

```
begin
    CREATE_RELATION
        (list_of_type_names = list_of_object_types,
         relation_name = name_new_object_type)
end
```

c. DECOMPOSE_OBJECT_TYPE

Input: object_type, list_of_new_names,
list_of_values = {known a priori, indefinite}

Precondition: all the names from the list_of_new_names are not the name of any other existing object

Output: list_of_new_object_types

Postcondition: the new resulting specification state contains a list of new object types which result from the decomposition of object_type

algorithm in Z: if the object type is not a schema[(1)], then creation of a new schema with the list of object types in its declaration part, with the following syntax: { variable = name of object type in small print, type = name of object type in capital print}

```
begin
    if object_type <> schema
    then CREATE_SCHEMA
       (name = name of object_type, {}, {}, schema = object_type)
    endif
    foreach n in list_of_new_names
        do CREATE_OBJECT (name = n, ...)
          EXTEND_SCHEMA
              (schema = object_type, new_decl = { n : N}, {} )
        od
end
```

algorithm in GIST:

```
begin
    if object corresponds to a existing type
  then DELETE_TYPE (type_name = object_type)
  endif
    CREATE_RELATION
     (list_of_type_names = list_of_new_names,
      relation_name = object_type)
end
```

---

(1) this can be easily checked thanks to the table described above.

d. SPECIALIZE_OBJECT_TYPE

Input: object_type, new_name

Precondition: new_name is not the name of any other existing object; object_type already exists in the specification

Output: new_object_type

Postcondition: the new resulting specification state contains a new object type which results from the specialization of object_type

algorithm in Z:

```
begin
    CREATE_SCHEMA (name = new_name, {},{},
                                schema = new_object_type)
    if object_type = given_set
    then DELETE_GIVEN_SET
                    (name = name of object_type)
         CREATE_SCHEMA
                    (name = name of object type,
                     declarations = {obj:OBJ},{},
                     schema = object_type)
    endif
    INCLUDE_SCHEMA (new_object_type, object_type)
end
```

algorithm in GIST: if the object is a type, then we must delete it before creating the supertype.

```
begin
    if object_type = "type"
         then DELETE_TYPE (type_name = name of object_type)
    endif
    CREATE_SUPER_TYPE
         (list_of_type_names = new_name,
          type_name = name of object_type)
end
```

e. GENERALIZE_OBJECT_TYPE

Input: object_type, new_name

Precondition: new_name is not the name of any other existing object; object_type already exists in the specification

Output: new_object_type

Postcondition: the new resulting specification state contains a new object type which results from the generalization of object_type

algorithm in Z: same as specialization, there is no difference a priori between the schematype of a generalized object and this of a specialized object, as long as no further declarations or predicates are added.

algorithm in GIST:

```
begin
    CREATE_SUPER_TYPE
        (list_of_type_names = name of object_type,
         type_name = new_name)
end
```

f. CLASSIFY_OBJECT_TYPE

Input: object_type[1]

Precondition: object already exists in the specification

Output: new_object = assignment of object to class_of_object

Postcondition: the new resulting specification state contains a definition of an object which is a class of objects whose type is given as input by object_type.

algorithm in Z: this primitive amounts to the creation of a new schema with a declaration of a powerset of the object to be specified.

```
begin
    CREATE_SCHEMA
        (name = "class_of_" + name of object_type,
         declarations = {"set_of"+
                                {object_type}: P OBJECT_TYPE},
         predicates = {},
         schema = new_object)
end
```

algorithm in GIST:

```
begin
    CREATE_SET_OF
        (type_name_1 = "class_of_"+ name of object_type,
         type_name_2 = new_object)
end
```

---

(1) the operator CLASSIFY_OBJECT, which is a typing mechanism, is implied by the operator CREATE_OBJECT_TYPE

Those were the primitives dealing with the gradual building of object types. Now we also need the ability to express constraints on such object types. The right time to state these constraints seems to be whenever one of the primitives above is used. For example, when one classifies an object type, constraints on the cardinality of the class should be expressed and attached to the new object type "class_of_object". Thus every application of an abstraction mechanism involves questions that can generate constraints, depending upon the answers being provided. When to use this primitive depends on rationales of the level above.

Just like for objects, there will be "high-level" constraints between objects which capture concepts that can be found in most formal languages, and "low-level" constraints, inherent in a given target language. The aim of this level is therefore to provide a translation from high-level constraints of the method level into, for example, predicates available in Z and GIST at the specification level. So there will also be tables of correspondences, in order to supply equivalents between high and low level constraints.

We shall only develop one general procedure, overlooking the process of translating high-level constraints into predicates.

g. ATTACH_CONSTRAINT_OBJECT_TYPE

Input: constraint, object_type

Precondition: object_type already exists in the specification

Output: none

Postcondition: the new resulting specification state contains a new constraint, attached to the object type given as input. So this operator receives a general constraint, expresses it in the target formal language and attaches it to the object type it refers to.

algorithm in Z:

```
begin
    translate constraint → predicate(s)
    EXTEND_SCHEMA (schema = object_type, {},
                   new_predicates = predicate(s))
end
```

algorithm in GIST:

```
begin
    translate constraint ⟶ predicate(s)
    if constraint concerns an operation to be
    triggered
    then ADD_DAEMON (...)
    else ADD_PREDICATE (name = object_type,
                        constraint = predicate(s))
    endif
end
```

But, in a similar fashion as for objects, there could be abstraction mechanisms for constraints as well. Our modelling practice in RML hints at considering some hierarchies of constraints.

In short, the rationales of the method level will help to spot constraints in the specifications. But they will have to know about the expressive power of the target language, so as to know whether or not a given kind of constraints can be written in it. All languages do not allow all the constraints a specifier could think of, or, at least, they cannot be written in a direct way.

Now, we shall have a look at operations, whose abstraction mechanisms are quite similar to those concerning the objects. This is the reason why we shall not make an exhaustive study of them.

h. CREATE_OPERATION

Input: name, list_of_inputs, list_of_outputs

Precondition: name is not the name of any other existing operation

Output: operation

Postcondition: the new resulting specification state contains an initial definition of an operation, which has all the objects and their types that it requires to be defined properly as inputs, and all the inputs modified as output. The preconditions and postconditions will be attached as constraints to this operation by: ATTACH_CONSTRAINT_OPERATION

algorithm in Z:

```
begin
    CREATE_SCHEMA (name = name,
        declarations = {list_of_inputs+"?"}+ {list_of_outputs+"!"},
        predicates = {},
        schematype = operation)
end
```

In order to express preconditions and postconditions, which are always constraints upon inputs and outputs, the following primitive will be useful:

i. ATTACH_CONSTRAINT_OPERATION

Input: constraint, operation

Precondition: operation already exists in the specification

Output: none

Postcondition: the new resulting specification state contains a new constraint, attached to the operation given as input. So this operator receives a general constraint, expresses it in the target formal language and attaches it to the operation it refers to.

algorithm in Z:

begin
translate constraint $\longrightarrow$ *predicate(s)*
EXTEND_SCHEMA (schema = operation, {},
new_predicates = *predicate(s)*)

end

Like constraints attached to objects, those attached to operations will be different at the specification level and the rationale level. Thus, likewise, the translation will be done at this level, which keeps a sort of a "secret".

The process of an operation refinement is in fact the building of a tree of conjunctive and disjunctive compositions of suboperations. Thus, an operation is completely defined by the composition of its final suboperations, i.e. the leaves of the tree. The input/output declarations will be shared by all the suboperations. These will also have predicates attached to them, a predicate being shared by the operations belonging to the sub-tree starting from this sub-operation.

This explains how the following decomposition primitives will work:

j. AND_DECOMPOSE_OPERATION

Input: operation, list_of_new_names

Precondition: all the names from the list_of_new_names are not the names of any other existing operation

Output: list_of_new_operations

Postcondition: the new resulting specification state contains a list of new operations which result from the decomposition of the operation given as input. So, an operation is decomposed into several suboperations, whose names are given as inputs. The original operation is a *conjunctive* composition of these suboperations. Note: this is a structural decomposition.

algorithm in Z: for each suboperation, a schematype will be created and all the input/output declarations will be transferred to each one of them.

```
begin
    foreach n in list_of_names
        do CREATE_SCHEMA (name = n,
            declarations = declaration part of operation,
            predicates = {},
            schematype = new_operation)
          list_of_new_operations :=
            list_of_new_operations + new_operation
        od
    AND_MERGE_OF_MULTIPLE_SCHEMAS
        (list_of_new_operations, operation)
end
```

The operator AND_MERGE_OF_MULTIPLE_SCHEMAS, which creates an operation as the conjunctive composition of several sub-operations is a recursive application of the operator AND_MERGE_OF_SCHEMAS, which only works for the conjunctive composition of two schemas.

k. OR_DECOMPOSE_OPERATION

Input: operation, list_of_new_names

Precondition: all the names from the list_of_new_names are not the name of any other existing operation

Output: list_of_new_operations

Postcondition: the new resulting specification state contains a list of new operations which result from the decomposition of the operation given as input. So, an operation is decomposed into several suboperations, whose names are given as inputs. The original operation is a *disjunctive* composition of these suboperations.

algorithm in Z: for each suboperation, a schematype will be created and all the input/output declarations will be transferred to each one of them.

```
begin
    foreach n in list_of_names
        do CREATE_SCHEMA (name = n,
            declarations = declaration part of operation,
            predicates = {},
            schematype = new_operation)
          list_of_new_operations :=
            list_of_new_operations + new_operation
        od
    OR_MERGE_OF_MULTIPLE_SCHEMAS
        (list_of_new_operations, operation)
end
```

The operator OR_MERGE_OF_MULTIPLE_SCHEMAS, which creates an operation as the disjunctive composition of several sub-operations is a recursive application of the operator OR_MERGE_OF_SCHEMAS, which only works for the disjunctive composition of two schemas.

The AND_AGGREGATION and OR_AGGREGATION primitives for operations are very simple and so will not be explained here. As to the classification primitive, although it is conceptually interesting for operations, it is not of prime necessity. Therefore, we shall not tackle the question of its formalization in this context.

B. Conclusions

We are aware that our approach is not very much refined, since it reduces a great deal of the aspects of formal languages. Its simplicity leaves in the shade many interesting features of some languages. Actually, it would be necessary to take a higher-level point-of-view, after an exhaustive study of the characteristics of most formal languages. What we would need is a meta-model describing the formal language models, so that we could design more elaborate *process primitives* working on it.

Then again, the results we obtain after application of our *process primitives* provide Z schemas that do not give as good a modelling of the problem as the ones written by a craftsman specifier. For example, in Z, our primitives model relations as aggregations of objects into a schematype. This is mainly due to reasons of simplicity and conceptual cleanliness of our procedures. However, this turns out to be far less clear than the good old Z relations... and, therefore, it endangers the readability and other qualities of good Z specifications.

Here is a practical example in Z. The first schema is the result of the application of our primitives, whereas the second Z declaration is the "natural" way of modelling a relation in Z.

talk_about
book: BOOK
subject: SUBJECT

talk_about: BOOK $\leftrightarrow$ SUBJECT

It is clear that the latter modelling of the relation *talk_about* is much easier to handle than the former. For example, writing a predicate that "describes a set of books dealing with a given subject" is more straightforward in the latter case (it is just the relation with a range restriction).

By systematizing the specification process in this way, the wealth of Z and GIST formalisms is somehow lost. And we may end up with poor schemas, which would use only a subset of the possibilities offered by such languages. So there is a need for a tool that would read through the requirements produced by our protocols and restate some objects and constraints in a less artificial way.

But of course, the main effort remains to be done on those primitives such as the ones we stated, so as to improve and enrich them while remaining at a very general level.

Finally, we would like to point out that other primitives can be considered as well, such as, for instance, those dealing with the environment or "agents" in GIST.

4.3.2.3 The rationale model

The specification process can be seen as a sequence of transitions between specification states: by starting with the informal requirements, the specifier will progressively reach the formal specification thanks to the application of those operators such as those defined in section "4.3.2.2 The language-independent process sub-model".

The selection of a process-level operator is determined by "control strategies". In [RS-2], Dubois and Van Lamsweerde have made a clear distinction between local strategies and global ones:

> "**Local control strategies** determine the direction in which a next specification state is reached from the current one, e.g., *up, down, backward, forward*".
>
> On the other hand, "**global control strategies** determine the direction in which a whole sequence of next specification states are reached from the current one".

Thus, in this section, we shall try to explain when and why we use a strategy rather than another one. This should help us defining some parts of our specification meta-algorithm. Therefore, we shall try to find out some meta-rules (local strategies) and a way to arrange them into global strategies.

A. Local strategies

As we are interested in the process of turning informal requirements into formal specifications, we must start from the informal expression of the specifications, and extract objects and operations from in there. We shall therefore read through the requirements and spot relevant objects and operations.

Then, we shall state rules that determine if the application of an abstraction mechanism can prove useful in a given situation. At this level we can only propose a set of rules, whose order of examination will be fixed in global strategies.

Here is a sample of such rules.

IF object is of some use to the specification, i.e. if it is an entity and has an existence of its own

THEN CREATE_OBJECT_TYPE (name = name given in the requirements, values={indefinite})

IF object is composed of two or more other existing objects

THEN AGGREGATE_OBJECT_TYPE (list_of_objects, new_name, new_object)

IF object has all the properties of some other object, plus some others of its own

THEN SPECIALIZE_OBJECT_TYPE (object, new_name, new_object)

IF object has properties that are of some use to an operation, these properties being not known yet

THEN DECOMPOSE_OBJECT_TYPE (object, list_of_names, list_of_new_object)

IF there is more than one occurrence of an object

THEN CLASSIFY_OBJECT_TYPE (object, class_of_object)

As explained above, the constraints upon object types will be expressed whenever we apply a process-level primitive to manipulate these object types. So there will be meta-rules that help finding out constraints on object types, according to the primitive used.

IF CLASSIFY_OBJECT_TYPE has been used

THEN check whether there exists a lower or an upper bound (or both) to the cardinality of the objects class

IF (lower bound > 0) and (upper bound finite)

THEN ATTACH_CONSTRAINT_OBJECT (object, constraint = { cardinality [l:L] })

IF AGGREGATE_OBJECT_TYPE has been used

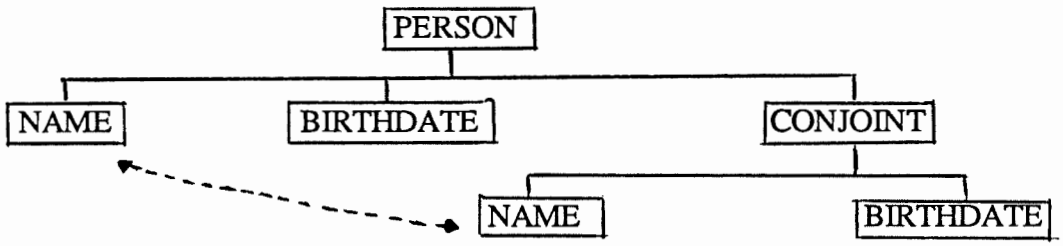
THEN examine the tree decomposition structure of the new object type, and find out identical types in different subtrees (if any)

FOREACH pair of such same object types

DO ATTACH_CONSTRAINT_OBJECT _TYPE (object_type,
constraint is { instance_of(object1_type) **?** instance_of(object2_type) })

where the **?** stands for a logical operator (≠,=, included,...).

An example will facilitate the understanding of the latter rule. Here is an object type aggregated from three other object types. We have thus a tree, which is an aggregation of three subtrees (two of them being only "leaves"):



Owing to this aggregation, NAME can be found at two different places in the tree of the object type PERSON. This enables comparisons. Thus, the question is: is there any constraint upon the objects (NAME of PERSON) and (NAME of CONJOINT of PERSON)? Of course, the answer is yes, the names must be the same!

This suggests how a constraint, which will be specified naturally by the specifier, can also be discovered by an systematic process questioning its user.

Many other rules like these ones can be found for the other abstraction principles applied to objects. Therefore, we shall not go any further in this direction, because we would rather concentrate on rules regarding operations now.

There exist local strategies for operations. For example, one can determine the abstraction mechanism to apply in order to refine an operation by just looking at the input and output objects and their structure. Of course, this also depends on which strategy is followed - i.e. down, up, backward, forward. A more precise definition of these strategies can be found in [RS-2].

We often have favoured a down backward strategy, because starting from the results is an approach advised in many "problem solving methods". This choice induces rules, some of which are:

IF output object of an operation is aggregation of subobjects
THEN AND_DECOMPOSE_OPERATION (operation,
list_of_new_names, list_of_new_operations)

IF output object of an operation is specialization of a more general object for which a corresponding operator has been specified
THEN OR_DECOMPOSE_OPERATION (operation, new_name, new_operation)

The contents of the operation will be expressed through the use of constraints attached to the basic suboperations at the bottom of the tree (i.e. the leaves). Thus, the total specification of an operation will be an expression containing conjunctions and disjunctions of basic suboperations.

After applying one of the previous rules to decompose operations till we obtain basic output objects for each suboperation, other rules will be looked at. These determine the constraint tying an input to an output. A simple example might be the following:

IF output and input have the same structure and there is a simple rule of transformation between them (i.e a predefined basic function on these objects)
THEN ATTACH_CONSTRAINT_OPERATION (suboperation,
constraint = { output = function(input) })

To conclude with, there will also be rules that reflect some knowledge about the expressive power of the formal language, i.e. which abstraction mechanisms should be used preferably or not in which language. This is due to the fact that all languages do not support abstraction mechanisms in the same way, and some are more suited for some local strategies, whereas they may not be convenient for some other ones. This will also be true at the global level.

B. Global strategies

We reckon it is of some use to state the most widespread global strategies existing to date. These are:

- operation- or object-oriented
- depth-first or breadth-first

Our experience in using Z, RML and GIST has led us to favour an object-oriented and depth-first global strategy. Therefore, the meta-algorithms we use are very much alike. This means that we have always begun with the specification of objects, as long as there was something to model, and that, only afterwards, we start modelling the operations. We have also favoured a depth-first approach because it is more natural to "human beings". One usually likes to complete a task before starting a new one, and the same applies to specification processes. It seems far more natural to refine an object/operation completely before specifying anything else.

We would like to make some further comments.

First, we have noticed that there was some backtracking in our specification process, because operations forced us to specify new objects that we had not realized were important before. Thus, a mixed approach seems far more realistic.

Second, the size of the problem we dealt with was not like the one of real-world systems. Thus, it was not really difficult to have a global view of the system to specify.

Finally, as it was noticed by Dubois-Van Lamsweerde in [RS-2], which strategy to follow seems to depend heavily on the type of problem considered and on the way the specifier sees the problem.

The meta-algorithm of the object-oriented strategy we have followed is made up with two "coroutines" applied concurrently: one for the refinement of objects and one for operations.

Coroutine n°1:

SPOT Objects {i.e. scan the requirements and spot objects}

FOREACH Object

DO

REFINE Objects {i.e. aggregate, decompose, specialize,... them recursively, according to rules which are language-dependent or independent)}

OD

Coroutine n°2:

SPOT Operations {i.e. scan the requirements and spot operations}

FOREACH Operation

DO

DEFINE Inputs, Outputs

IF there are new objects in inputs/outputs

THEN DEFINE (or REFINE) new objects

REFINE Operation {i.e. according to inputs or outputs structure, and this must be done recursively, until we have a structure of basic operations}

OD

Other meta-algorithms can be written in the same fashion. However, our experience is still too limited for us to formulate rules that could help the specifier chosing a relevant global strategy as a function of the problem considered, its size, the target formal language chosen, etc.

In the PRODUCT LEVEL part of this work, we have presented three languages (Z, RML and GIST), which we consider as being representative of what a formal specification language is all about. An assessment of their pros and cons has been undertaken, and led to bring the strengths and weaknesses of each one of them to light.

It could be interesting to extend such a comparison further to other languages. As formal languages can be classified[(1)], one member, representative of each class, could be chosen. And, as some problems require certain expressive qualities from a formal language, this could help specifiers chosing one of them accordingly. Likewise, if automatic consistency checks must be made absolutely, it would be wiser to chose a language that supplies such facilities easily.

In the PROCESS LEVEL part of this work, we have investigated the underlying processes making up the process of turning informal requirements into formal specifications. We started with the informal description of specification processes in both Z and GIST, and, next, attempted to generalize it as much as possible. Of course, this attempt has been biased to a certain extent because it is based on just two particular kinds of specification processes in two particular formal languages.

Thus, here also, the investigation of such processes in more than two languages would be required. Only then would we be able to talk about a real "specification process model". We have also emphasized the need for a model describing formal languages. This aims at a better understanding of those aspects of the specification process that depend on a given formal language and those that do not.

Also, this work requires a great deal of practical experience in the use of formal languages, which students typically do not have. So it is better to entrust research workers with this task.

This specification process model also relies heavily on rules that express the knowledge of the analyst, i.e. the strategies he/she follows. Thus, an empirical approach should be considered in the future, with on-the-ground interviews and collaborations.

September 1989

(1) for example algebraic, knowledge-oriented languages,etc...

6. BIBLIOGRAPHY

We have built up our bibliography using the following typology:

[DI-*] = Domain general introduction,

[ML-*] = Major formal specification languages,

[RS-*] = Requirements gathering strategies,

[AP-*] = Acquisition process,

[EC-*] = Case Studies.

According to this typology, a same paper could be found in several places.

GENERAL REFERENCES

[DI-1] Gruia-Catalin Roman, "A Taxonomy of Current Issues in Requirements Engineering", Washington University, St. Louis 1985.

[DI-2] Barbara H. Liskov and Valdis Berzins, "An Appraisal of program Specifications", Massachusetts Institute of Technology in "Software Specification Techniques", Addison-Wesley Publishing Company 1986 pp 3,23.

[DI-3] B. Meyer, "On formalism in specifications".

[DI-4] L. Osterweil, "Software Processes are software too", 1987

[DI-5] Kaizhi Yue, "What does it mean to say that a specifi-cation is complete?", Fourth International Workshop on Software Specification and Design, April 1987, pp. 42-49

[DI-6] Anthony Finkelstein, Colin Potts, "Building Formal Spe-cifications Using 'Structured Common Sense'", Fourth International Workshop on Software Specification and Design, April 1987, pp. 108-113

[DI-7] E. Dubois, J.P. Finance, J. Souquières, A. Van Lamsweerde, "First description of the Icarus language kernel for the product level", June 1989, Esprit Project 2537, SpecFunc-003-R, ICARUS.

[DI-8] Parnas, "Designing Specification for Ease of Extension and Contraction", IEEE Transaction on Software Engineering.

MAJOR FORMAL SPECIFICATION LANGUAGES

[ML-1] J.M. Spivey, "An Introduction to Z and Formal Specifications", Oxford University Computing Laboratory, August 1987.

[ML-2] R.M. Balzer, D. Cohen, M.S. Feather, N.M. Goldman, W. Swartout, D.S. Wile, "Operational Specification as the Basis for Specification Validation", USC/Information Sciences Institute, California 90291.

[ML-3] J.E. Nicholls, "Z user Manual", IBM United Kingdom Laboratories Ltd, Hursley Park, Winchester, May 1987.

[ML-4] J.B. Wordsworth, "A Z Development Method", IBM United Kingdom Laboratories Ltd, Hursley Park, Winchester, December 1987.

[ML-5] J.M. Wing, "A Study of 12 Specifications of the Library Problem", June 1987, July 1988.

[ML-6] Greenspan, Borgida, Mylopoulos, "A Requirements Modeling Language and its Logic".

[ML-7] R. Balzer, N. Goldman, D. Wile, "Operational Specification as the Basis for Rapid Prototyping", Software Engineering Notes -ACM- Vol 7, n°5 December 82, pp 3-16.

[ML-8] Wasserman, "Extending State Transition Diagrams for the Specification of Human-Computer Interaction", August 1985.

[ML-9] P. Zave, "A operational Approach to Requirements Specification for Embedded Systems", 1982.

[ML-10] Eric Dubois, Jacques Hagelstein, "Reasoning on formal requirements: A lift control system", Fourth International Workshop on Software Specification and Design, April 1987, pp. 161-167

[ML-11] "The Gist Specification Language", pp 37-51

[ML-12] M. S. Feather, "Constructing Specifications by combining Parallel Elaboration", January 1987 version.

[ML-13] Sol. J. Greenspan, "Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition", Technical Report CSRG-155, March 1984

REQUIREMENT GATHERING STRATEGIES

[RS-1] B.P. Collins, J.E. Nicholls, I.H. Sorensen, "Introducing Formal Methods: The CICS Experience with Z". IBM United Kingdom Laboratories Ltd, Hursley Park, Winchester and Programming Research Group, Oxford University, August 1987.

[RS-2] E. Dubois, A. Van Lamsweerde, "Making Specification Processes Explicit", 1987

[RS-3] E. Dubois, A. Van Lamsweerde, "A constructive Approach to Requirements Specification", Facultés Universitaires Namur, Institut d'Informatique, Research Paper, n°1/85

[RS-4] "Icarus Project".

[RS-5] L. Osterweil, "Software Processes are software too", 1987

[RS-6] M. Feather, S. Fickas, W. Robinson, "Design as Elaboration and Compromise", Computer Science Department, University of Oregon, June 1988.

[RS-7] S. Fickas, P. Nagarajan, "Being Suspicious: Critiquing Problem Specifications", Computer Science Department, University of Oregon, 1988

[RS-8] M. Feather, "Constructing Specification by Combining Parallel Elaborations", IEEE Transactions on Software Engineering, Vol 15 no 2, February 1989

[RS-9] S. M. Celiktin, E. Dubois, J. Vangeersdael, "Specifying with Gist Language", May 1989, ESPRIT project 2537 -ICARUS-

ACQUISITION PROCESSES

[PA-1] S. Fickas, "Automating the Specification Process", Computer Science Department, University of Oregon, CIS Technical Report 87-05, November 1987.

[PA-2] A.Malhotra, "Design Criteria for a Knowledge-based English Language System for Management: an Experimental Analysis", MIT/LCS/TR-146, February 1975.

[PA-3] K.A. Ericsson, H.A. Simon, "Verbal Reports as Data", Massachusetts Institute of Technology, November 1983.

[PA-4] S. Fickas, Collins, Olivier, "Problem Acquisition in Software Analysis: A preliminary Study".

[PA-5] S. Fickas, "Automating the Analysis Process: an Example", 1987.

[PA-6] C. Rich, Waters, Reubenstein, "Toward a requirements Apprentice", 1987.

[PA-7] M. Feather, S. Fickas, W. Robinson, "Design as Elaboration and Compromise", Computer Science Department, University of Oregon, June 1988.

[PA-8] S. Fickas, P. Nagarajan, "Being Suspicious: Critiquing Problem Specifications", Computer Science Department, University of Oregon, 1988

CASE STUDIES

[EC-1] J.B. Wordsworth, "Oil Terminal Control System: an Exercise for Readers of Z", IBM United Kingdom Laboratories Ltd, Hursley Park, Winchester, August 1987.

[EC-2] I. Hayes (Editor), "Specifications Case Studies", Prentice-Hall International Series in Computer Science, February 1986.

[EC-3] I. Hayes, "Transaction Processing".

[EC-4] "A Tasteful Variety of Specification Case Studies", FUNDP-NAMUR, January 1989, ESPRIT project 2537 -ICARUS-

[EC-5] S. M. Celiktin, E. Dubois, J. Vangeersdael, "Specifying with Gist Language", May 1989, ESPRIT project 2537 -ICARUS-

[EC-6] Call for Papers. Problem Set for the 4th International Workshop on Software Specification and Design. ACM Software Engineering Notes, April, 1986.

ANNEX 1:
A CASE STUDY "THE OIL TERMINAL CONTROL SYSTEM"

General description

The following description has been copied from a document published by IBM United Kingdom Laboratories Ltd.

An oil terminal has a number of berths at which tankers can discharge their cargoes.

When an approaching tanker asks for permission to dock, the controller will ask the system to allocate a berth for it to use. If no berth is free, the system will tell the controller so, and the tanker will be queued in the approach to the terminal. The system assumes that there will be enough room for any number of waiting tankers.

On docking, a tanker occupies the allocated berth, unloads its cargo, and so on. When it is ready to leave, the controller will notify the system so the berth is available for reuse and the tanker is deleted from the system. A tanker's leaving a berth might mean that a queing tanker can come and occupy it. The system will identify the tanker at the head of the queue to the controller, and allocate that berth to the tanker.

The system has enquiry facilities so that the controller can get information about which tankers are queuing, which berths are occupied and by which tankers, and which berths are free.

The Oil Terminal: state-space

Needed sets

The following schemas describe the declarations and the constraints we need in order to formalize the Oil Terminal specifications.

The given sets are : TANKER and BERTHS where TANKER is the set of tankers and BERTHS of berths. So, we have:

$[TANKER, BERTHS]$

We need a Berths subset which contains all the berths that are available in our Oil Terminal. Thus we declare:

$berths : \mathbb{P}\ BERTHS$

State data: the schema

OIL_TERMINAL

$t\_q : \mathbf{seq}\ TANKER$
$t\_un : \mathbb{P}\ TANKER$
$t\_s : \mathbb{P}\ TANKER$
$using : TANKER \rightarrowtail\!\!\!\!\!\!\rightarrow\!\!\!\!\!+\ BERTHS$

$TANKER = (\ \mathbf{ran}\ t\_q \cup t\_un \cup t\_s\ )$
$\mathbf{ran}\ t\_q \cap t\_un = \emptyset$
$\mathbf{ran}\ t\_q \cap t\_s = \emptyset$
$t\_un \cap t\_s = \emptyset$
$\forall\ i,j : \mathbf{dom}\ t\_q \mid i \neq j \bullet t\_q(i) \neq t\_q(j)$
$\mathbf{ran}\ using \subset berths \Rightarrow t\_q = \emptyset$
$\mathbf{dom}\ using = t\_un$
$\mathbf{ran}\ using \subseteq berths$

Some comments

We have chosen to model the Oil_Terminal using two entities i.e. TANKER and BERTHS. Thus we have overlooked certain features described in the informal text like: "cargo", as it played no significant role here.

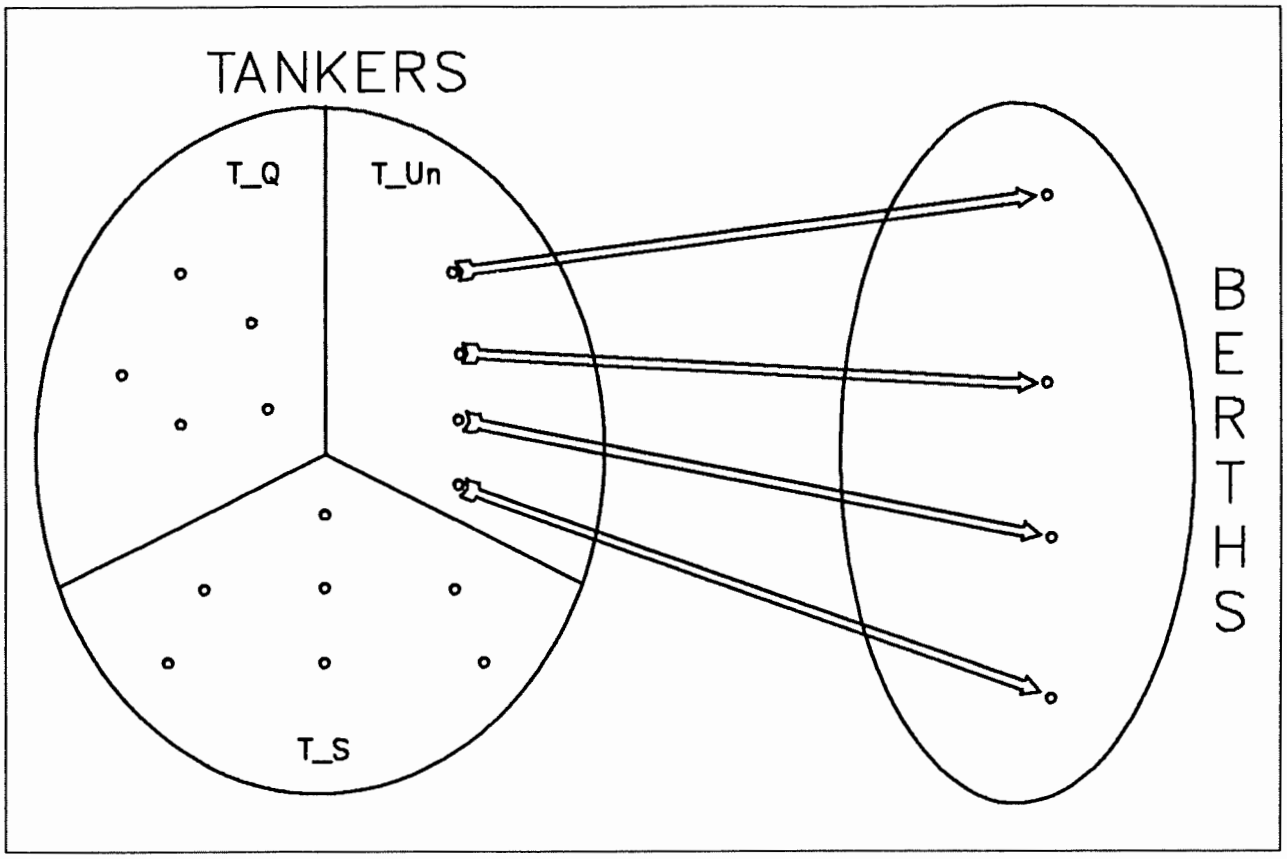


Figure 1. Oil Terminal Control System: the sets.

We have then created a partition of the TANKER set assuming a real world situation which seemed natural to our minds but which nonetheless needs to be discussed with the customer. This partition relies on three states: a tanker can be queuing (t_q), unloading (t_un) or sailing (t_s). This leaves sufficient room for further refinements. Moreover the "queuing state" has been modeled as a sequence assuming order is important and the system picks up a tanker according to a "first in, first out" strategy.

Because of the specifications words "allocated berths," we were led to think of a function from TANKER to BERTHS. This function is injective because a berth cannot accomodate more than one tanker and partial because only some tankers are unloading at the same time.

Our constraints describe the following facts:

1. The first four contraints relate to the TANKER partition.
2. While the next one refers to the assumed unicity of sequence members in the queue.
3. The following one ensures that no tanker could be queued if a berth is free.
4. And the last two restrain the "using" function domains.

The questions that arose in our minds were the followings:

1. Are all tankers known to the system beforehand? Or are they created or deleted as soon as they arrive or leave the harbour? If so, there could not possibly be any "sailing tanker" set and checks carried out on the tankers would be different.

2. Must we keep a record of all the comings and goings of tankers or drop information about actions once they have been completed?

3. We have used a lot of our own knowledge about the concepts of "TANKER" and "BERTHS" and actions related to them. Thus we have implicity answered questions that might have been settled otherwise. It's not beyond our grasp to fancy a tanker unloading while queuing, thanks to a floating and moving berth. Why not? We have to make sure that the partition we've adopted is a real partition with non overlapping classes. It's up to the customer to decide, and we have to find out what he fancies.

Initialization of the Oil_Terminal

INITIALIZATION

OIL_TERMINAL

$t\_q' = \emptyset$
$t\_un' = \emptyset$
$t\_s' = TANKER$
$using' = \emptyset$

When we initialize the system, the queue is emptied. There is no tanker at any berth. That's why all tankers are sailing.

Arrival of a tanker

When a tanker asks for permission to dock, three situations can occur: either a berth is free or there's none or the tanker's identification is unknown. So, we have

$$ARRIVAL \triangleq ARRIVAL\_IF\_ROOM \lor ARRIVAL\_IF\_NO\_ROOM \lor IS\_NOT\_SAILING$$

As one can see, we have chosen to develop a robust Oil Terminal Control System. Does the customer agree with this solution?

First case: A berth is free

We can allocate this free berth to the approaching tanker. The system gives the controller informations about the new allocated berth.

The tanker is transfered from t_s to t_un.

ARRIVAL_IF_ROOM

$\Delta OIL\_TERMINAL$
$tanker? : TANKER$
$mes! : MESSAGE$
$xberth! : BERTHS$

$\mathbf{ran}\ using \subset berths \land tanker? \in t\_s$
$t\_un' = t\_un \cup \{ tanker? \}$
$xberth! \in berths \setminus \mathbf{ran}\ using$
$using' = using \oplus \{ tanker? \mapsto xberth! \}$
$mes! = ok\_for\_dock\_and\_unload$
$t\_q' = t\_q$
$t\_s' = t\_s \setminus \{ tanker? \}$

Second case: no berth is free

All berths being occupied and the tanker being sailing, it is added to the queue. A message is sent to the controller warning him about the situation.

ARRIVAL_IF_NO_ROOM

$\Delta OIL\_TERMINAL$
$tanker? : TANKER$
$mes! : MESSAGE$

---

ran $using = berths \wedge tanker? \in t\_s$
$t\_s' = t\_s \setminus \{ tanker? \}$
$t\_q' = t\_q \frown \langle tanker? \rangle$
$t\_un' = t\_un$
$mes! = sorry\_queue\_up$
$using' = using$

Third case: a wrong tanker identification

The given identification doesn't correspond to a sailing tanker. The controller is issued with a warning.

IS_NOT_SAILING

$\Xi OIL\_TERMINAL$
$tanker? : TANKER$
$mes! : MESSAGE$

$tanker? \notin t\_s$
$mes! = sorry\_this\_thanker\_is\_not\_sailing$

Departure of a tanker

When a tanker is unloaded, it leaves its berth. As for the ARRIVALs, we have three different cases. There may be a queue when the tanker is leaving or there may be none. It could also be possible for the controller to give a wrong tanker identification to the system.

$$DEPARTURE \triangleq DEPART\_IF\_QUEUE \lor DEPART\_IF\_NO\_QUEUE \lor UNKNOWN\_TANKER$$

Does the customer agree with this robust formalization of the departure of a tanker?

First case: there is no queue

In this case we only have to transfer the tanker from t_un to t_s because the tanker is now sailing. The system generates a message here too. This one informs the controller of the tanker departure.

DEPART_IF_NO_QUEUE

$\Delta OIL\_TERMINAL$
$tanker? : TANKER$
$mes! : MESSAGE$

$tanker? \in t\_un$
$t\_q = \emptyset$
$t\_un' = t\_un \setminus \{ tanker? \}$
$t\_s' = t\_s \cup \{ tanker? \}$
$t\_q' = t\_q$
$using' = \{ tanker? \} \lhd using$
$msg! = ok\_sail\_off$

Second case: there is queue

In this case the unloaded tanker is tranfered from t_un to t_s and the first queuing tanker can use the freed berth. The system generates a message which contains informations about the freed berth and the unqueued tanker (*.newtanker!*).

DEPART_IF_QUEUE

$\Delta OIL\_TERMINAL$
$tanker? : TANKER$
$msg! : MESSAGE$
$newtanker! : TANKER$
$berth! : BERTHS$

$tanker? \in t\_un$
$t\_q \neq \emptyset$
$berth! = using(tanker!)$
$newtanker! =$ **head** (t_q)
$t\_un' = (t\_un \setminus \{ tanker? \}) \cup \{newtanker!\}$
$t\_q' =$ **tail** (t_q)
$t\_s' = t\_s \cup \{ tanker? \}$
$using' = (\{tanker?\} \lhd using) \oplus$
$\{newtanker! \mapsto berth!\}$
$msg! = ok\_ask\_for\_a\_new\_tanker$

Third case: Invalid informations

The controller is trying to transfer a tanker which is not docked.

UNKNOWN_TANKER_AT_DOCKS

$\Xi OIL\_TERMINAL$
$tanker? : TANKER$
$msg! : MESSAGE$

$tanker? \notin t\_un$
$msg! = sorry\_Unknown\_Tanker$

Enquiries

There are three possible query operations. First we can question the system about which tankers are in the queue. It could also be interesting to have a request at our disposal which would give us infomations about the free berths. Another request could be: "Give me all the occupied berths and for each one its tanker!."

$$ENQUIRY \triangleq ENQUIRY\_QUEUE \lor ENQUIRY\_BERTHS\_FREE \lor ENQUIRY\_BERTHS\_AND\_TANKER$$

First request

This request returns the tankers which are in the queue.

ENQUIRY_QUEUE

$$\Xi OIL\_TERMINAL$$
$$list\_tanker\_queuing! : \mathbb{P}\ TANKER$$
$$list\_tanker\_queuing! = \textbf{ran}\ t\_q$$

It could be interesting to know if the customer agrees with this representation of the result, i. e. a set, or if he prefers a sequence as result.

Second request

With this request the controller will be able to know which berths are free.

ENQUIRY_BERTHS_FREE

$$\Xi OIL\_TERMINAL$$
$$list\_free\_berths! : \mathbb{P}\ BERTHS$$
$$list\_free\_berths! = berths \setminus \textbf{ran}\ using$$

Third request

This request enables the controller to have informations about the occupied berths.

ENQUIRY_BERTHS_AND_TANKERS

Ξ*OIL_TERMINAL*

list_berths_tanker! : *TANKER* $\rightarrowtail$ *BERTHS*

---

list_berths_tanker! = *using*