Analysis of Business Processes Specified in Z against an E-R Data Model

by

Jun Ginbayashi

Technical Monograph PRG-103
ISBN 0-902928-81-3

December 1992

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD
England

Copyright © 1992 Jun Ginbayashi

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD
England

Electronic mail: Jun.Ginbayashi@comlab.ox.ac.uk

Analysis of Business Processes Specified in Z against an E-R Data Model

Jun Ginbayashi

St Cross College, University of Oxford

Submitted in partial fulfilment of the requirements for the degree of Master of Science in Computation at the University of Oxford, September 1992

Abstract

A specification method for business processes is presented, in which not only the processes but also the database integrity constraints are specified in Z based on the structnre of an Entity–Relationship data model. The formality of Z facilitates strict reasoning about the correctness of the processes with respect to the database integrity constraints.

In this method, as in VDM, one can proceed towards the correct specification of a process by checking a series of proof obligations. The precondition of a process is determined as one attempts to discharge the proof obligations.

During the specification activity computer support may be useful. The requirements for and a prototype of a support tool are also presented. Such a tool might be integrated into existing CASE (Computer Aided Software Engineering) tools, such as the Information Engineering Facility.

Acknowledgements

I would like to thank Dr. Mark Josephs, my project supervisor, for his careful guidance throughout this project. I would also like to thank Ms. Divya Prasad of the Indian Institute of Science, India, for her patient instruction on the zedB tool and her helpful advice.

Thanks also to Mr. Paul Sanders of James Martin Associates (JMA) for his precise explanation of JMA's project on *Declarative Analysis in Information Engineering* and for his demonstration of the IEF tool.

I acknowledge Fujitsu Limited in Japan for granting me time and supporting my studies at Oxford.

My special thanks go to my wife, Ikuyo, for her continuous support through every moment.

Contents

1 Introduction 1

1.1 Background 1

1.2 Motivation 2

1.3 Overview 3

2 Business Processes and ER Data Models 4

2.1 What are Business Processes? 4

2.2 ER Data Model 5

3 Formalizing Business Processes using Z 9

3.1 Translating ER Diagrams into Z 9

3.1.1 Attributes 10

3.1.2 Records 11

3.1.3 Tables 12

3.1.4 Relationships 13

3.2 Expressing DB Integrity Constraints in Z 14

3.3 Definition of DB 15

3.4 Expressing Business Processes in Z 16

3.4.1 Attribute Level 17

3.4.2 Record Level 18

3.4.3 Table Level 19

3.4.4 Relationships 21

3.4.5 Business Processes updating DB 22

4 Proof Obligations 23

4.1 Structural Checking / Precondition Calculation 23

4.2 Attribute Level 27

4.3 Record Level 28

4.4 Table Level 30

4.5 Relationships 33

4.6 Unchanged Relationships 35

4.7 DB Integrity Constraints 37
4.8 Example: Resident Registration System 39

5 Support Tool **52**
5.1 Requirements 52
5.2 Specification 54
5.3 Example of Execution 57

6 Conclusion **68**
6.1 Summary 68
6.2 Remaining Problems 69
6.2.1 HCI Consideration 69
6.2.2 Output Consideration 69
6.3 Future Works 70
6.3.1 Code Generation 70
6.3.2 Animation of Specification 71

A Syntax of Specification Files **74**

B Orwell Source Code **78**

Chapter 1

Introduction

1.1 Background

There have been several attempts to apply the Z specification method [Spivey 92] to the development of business application systems [Woodcock 92, Karal 91]. Compared with other formal methods, such as VDM [Jones 86], one of the benefits obtained through the Z method is that the precondition of business processes can be calculated from the overall specification by means of the schema calculus. This feature not only allows us to concentrate on the heart of the specification before dealing with error case analysis, but also provides an opportunity to employ logical analysis tools, such as theorem provers [Neilson & Prasad 91], in the specification activity.

On the other hand, there is a well-known and widely-used method, called Entity–Relationship (ER) data modelling [Chen 76], for data analysis and database design of business application systems. The ER model gives us a specification tool for describing several structures of data to be stored in a database. Furthermore, some methodologies with CASE tools, such as Information Engineering [Napier 91, Texas Instruments 88], regard the ER data model as a basis for process description as well as for data specification.

Meanwhile there have been some attempts to describe certain kinds of static business rules in terms of the database (DB) integrity constraints [Sanders & Short 92, Nijssen & Halpin 89, Ginbayashi & Hashimoto 91]. These trials seem reasonable since there are lots of business rules which must not be violated by any process and should be treated as attached to data rather than each process.

Formalizing the ER model using Z is a promising approach to making formal methods more applicable in industry. Within the Oxford University Programming Research Group (PRG) some research has been done

in this area [Josephs & Redmond-Pyle 91a, Josephs & Redmond-Pyle 91b, Karal 91]. Compared with procedural specification languages (e.g. SQL), one advantage of using a declarative specification language such as Z is that the specification written in it can be completely independent of its implementation. This separation of concerns is important because:

- Manipulation of specification, such as precondition calculation, should be done without considering any implementation details.
- There should be plenty of room for choice in implementation and optimization.

As the major result of this series of research in PRG, a library of generic Z schemas has been obtained, which can be used as standard schemas in ER modelling [Josephs & Redmond-Pyle 91b].

1.2 Motivation

By combining the research results stated in the preceding section, one can approach the activity of specifying business application systems as follows:

- Write a specification of a system in terms of ER data model, DB integrity constraints and processes.
- Translate the specification into Z schemas by using a certain set of standard schemas.
- Prove the correctness of the specification by means of Z schema calculation.

There are two approaches to proving correctness. One is to calculate precondition schemas from process schemas and simplify them by logical and mathematical reduction laws, such as in [Neilson & Prasad 91]. However, this approach may lead us to unnecessarily complicated calculations because a process schema must describe all its effects on the entire database even if it changes just a small part.

The other approach is to break down the correctness proof into a number of proof obligations according to the structure of the ER data model. Here we can proceed towards the correct specification of a process by discharging those proof obligations one by one and strengthening its precondition if necessary. Note that the use of proof obligations is also characteristic of VDM.

The purpose of our study is to investigate the validity of the second approach and propose a reasonable method for finding the correct specification of business processes.

1.3 Overview

Here is a brief summary of the remaining chapters and appendices:

Chapter 2 gives an introductory definition of business processes and an overview of ER data modelling.

Chapter 3 introduces a set of standard Z schemas which express ER model, DB integrity constraints and business processes. This is an extension of the results obtained in [Josephs & Redmond-Pyle 91a, Josephs & Redmond-Pyle 91b].

Chapter 4 explains the structured proof obligations for the correctness of business processes with some examples.

Chapter 5 considers the requirements of a support tool for the specification activity based on the results of our research.

Chapter 6 concludes the report with a summary of our research and some suggestions for future work.

Appendix A shows the syntax of a specification document which a support tool should read.

Appendix B contains the Orwell source code of the tool prototype. (Orwell is a functional programming language [Wadler & Miller 90, Bird & Wadler 88].)

Chapter 2

Business Processes and ER Data Models

2.1 What are Business Processes?

To begin with, let us consider an example of a banking system. A typical process which is meaningful for people in a bank is a transaction on a certain account, such as payment or withdrawal of a certain amount of money. The bankers must keep track of the balance of every account and are necessarily very anxious not to pay more money than permitted on each account. In other words, they are interested in the data to be stored in their database and how to control changes to the data in executing a transaction. They are not interested, however, in any implementation details such as a certain sequential access to required data in a network database or techniques for exclusive control on the database.

As an appropriate tool for specifying the structure of stored data from such a user's point of view, the Entity–Relationship(ER) data model [Chen 76] has been widely accepted because of both its ease of use and conceptual clarity.

On the other hand, it is only recently that the importance of process modelling from the business point of view has been recognized [Napier 91, Texas Instruments 88]. There might be several possible ways of defining a business process:

- a computing process just corresponding to a transaction in a business sense. (Business View)
- the smallest cycle which end users can recognize. (HCI View)

- a process which updates a database maintaining integrity constraints on the database. (Database View)
- a data processing component unit of a business function. (Information Engineering View)

among others. In this dissertation, we shall adopt the third one (i.e. Database View) and try to specify such a process in terms of entities and relationships.

2.2 ER Data Model

As the ER data model is broadly known, we do not inspect it in detail here but illustrate the model by an example adopting the notation in the Information Engineering Facility (IEF ™) [Texas Instruments 88]. Figure 2.1 shows an example of a simple banking system.

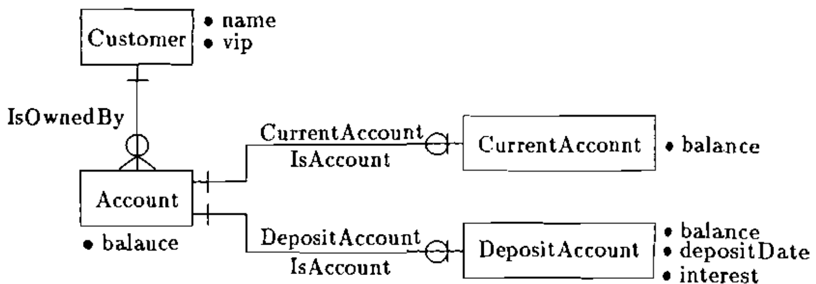


Figure 2.1: Example of ER diagram

In this diagram, the following objects are defined:

- Four entities: Customer, Account, CurrentAccount, DepositAccount. (Their attributes, in other words, data items, can be seen near them.)
- Three relationships: IsOwnedBy (between Account and Customer), CurrentAccountIsAccount (between CurrentAccount and Account), DepositAccountIsAccount (between DepositAccount and Account).

The diagram also implies that the following constraints always hold:

- The relationship IsOwnedBy is a one-to-many correspondence between the current set of all instances of Account and that of Customer.

- The relationship CurrentAccountIsAccount is a one-to-one correspondence between tbe current set of all instances of CurrentAccount and that of Account.

- The relationship DepositAccountIsAccount is a one-to-one correspondence between the current set of all instances of DepositAccount and that of Account.

The first constraint, for example, means that each customer may own one or more accounts while each account is always owned by exactly one customer.

Unfortunately, however, such kinds of constraints expressed by the ER diagram can cover only a small portion of business rules. For example, we can consider a business rule in the banking system sayiug that

> for every CurrentAccount whose owner is a VIP (represented by an attribute *vip* having the valne *yes*), its *balance* can go overdrawn by up to £100.

This and even more complicated rules ought to be expressed by formal (or informal) text accompanying the diagram. In general, these business rnles can be expressed by some conditions between several entities and relationships and therefore regarded as DB integrity constraints in a "subsystem" [Josephs & Redmond-Pyle 91a] or a "subject area" [Napier 91, Texas Instruments 88]. For example, the above rule can be captured within CustomerCurrentAccountSubsystem as illustrated in Figure 2.2. We will see a formal specification of this rule in Section 3.2.

Note that although IEF ™ provides the useful notion of 'subtype', we shall not use it in this dissertation so as to keep things as simple as possible. The constraints for a 'subtype' must instead be expressed as DB integrity constraints. Also note that we shall not consider many-to-many correspondences because a many-to-many correspondence between two entities can always be replaced with an additional entity (representing the Cartesian product of those two entities) and two one-to-many correspondences as illustrated in Figure 2.3.

Now, let us consider a business process which changes the state of a database. Since the database consists of a number of entities and relationships, the process is broken down into parallel subprocesses each of which acts on an entity or relationship.

Since each entity is essentially a file, i.e. a finite set of records, the subprocess on the entity can be implemented by a parallel execution of several basic operations, of which there are three kinds:

- Modify existing records in the file.

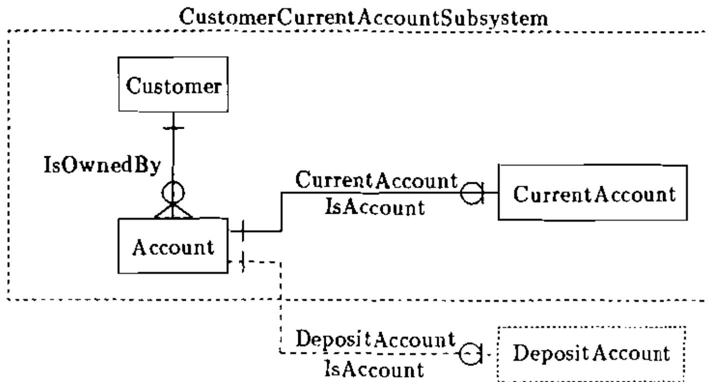


Figure 2.2: Example of Subsystem

- Insert new records in the file.
- Delete existing records from the file.

In these operations, two things must be specified: (1) how to identify the existing record(s) to modify or delete using input data and data stored in the database and (2) how to determine the new data of the record(s) to modify or insert using input data and data stored in the database.

For example, a process Withdraw in the banking system would modify a certain record in CurrentAccount snch that (1) the record is identified by inputting its ID number, CurrentAccountId, and (2) its *balance* is decreased by the inputted amount of money.

Similarly, subprocesses on relationships also consist of basic operations as follows:

- Modify existing links in the relationship.
- Insert new links in the relationship.
- Delete existing liuks from the relationship.

In summary, the total business process is specified by several snbprocesses, each of which is a combination of three kinds of basic operation on a certain entity or relationship.

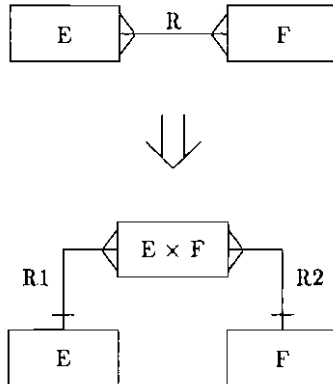


Figure 2.3: Resolution of Many-to-Many Correspondence

Chapter 3

Formalizing Business Processes using Z

In this chapter a set of standard Z schemas which can be used to specify business processes is introduced. As seen in the preceding chapter, a business process can be specified with its effects on each component of the ER data model. To begin with, therefore, we shall see several standard schemas which specify each component of the ER data model. Next, schema expressions for DB integrity constraints, the whole database and business processes are each considered.

Note that these standard schemas should be considered as patterns or forms used in the activity of specification rather than some sort of fixed, complete parts (such as abstract data types). For example, even when two different processes affect the same entity, we write two different schemas to specify their effects on the entity, although they have the same pattern.

3.1 Translating ER Diagrams into Z

An ER model consists of many entities and relationships. Each entity has several attributes and each occurrence of the entity has a record (i.e. a tuple of its attribute values). Therefore, it is reasonable to specify each entity with two schemas:

- a record schema which specifies the structure of attributes and constraints on each record.
- a table schema which specifies the set of all existing occurrences (i.e. the table) and constraints on them (such as volumetric constraints).

as in [Josephs & Redmond-Pyle 91a].

Here, however, we shall break down these two schemas into six classes of schemas for the purpose of using them in constructing proof obligations in a uniform way which will be stated in Chapter 4. These six classes are:

- for each attribute, an attribute schema which specifies the attribute and its type.
- for each attribute, an attribute constraint schema which specifies all constraints on the attribute, if any.
- a record schema which specifies all the attributes possessed by an entity.
- a record constraint schema which specifies all constraints on the record, if any.
- a table schema which specifies all existing occurrences.
- a table constraint schema which specifies all constraints on the table, if any.

Similarly, we shall describe a relationship with two separate schemas:

- a relationship schema which specifies a partial function from its detail entity to its master entity.
- a relationship constraint schema which specifies "cardinality constraints" [Nijssen & Halpin 89], such as in one-to-one correspondences, and all other constraints on the partial function.

3.1.1 Attributes

For each entity E and its attribute x of type T, we set up an attribute schema and an attribute constraint schema, if there are any constraints on x, of the following forms:

$[T]$

```
┌─ xofE ──────────────────────────
│ x : T
└─────────────────────────────────
```

```
┌─ xofEConstraints ──────────────────────
│ xofE
├──────────
│ (constraints)
└────────────────────────────────────────
```

For example, if entity DepositAccount has an attribute *balance*, the value of which must be a positive integer, then

```
┌─ balanceofDepositAccount ──────────────
│ balance : ℤ
└────────────────────────────────────────
```

```
┌─ balanceofDepositAccountConstraints ───
│ balanceofDepositAccount
├──────────
│ balance > 0
└────────────────────────────────────────
```

If DepositAccount has also other attributes *depositDate* of type *Date* and *interest* of type *InterestType* with no constraints this time, we have only one schema for each:

[*Date*, *InterestType*]

```
┌─ depositDateofDepositAccount ──────────
│ depositDate : Date
└────────────────────────────────────────
```

```
┌─ interestofDepositAccount ─────────────
│ interest : InterestType
└────────────────────────────────────────
```

3.1.2 Records

For each entity E, we set up a record schema and a record constraint schema, if there are any constraints on each record, of the following forms:

```
┌─ ERecord ──────────────────────────────
│ xofEConstraints   (or xofE if there are no constraints on x)
│ yofEConstraints   (or yofE if there are no constraints on y)
│ ...
└────────────────────────────────────────
```

$$
\begin{array}{|l}
\hline
\textit{ERecordConstraints} \\
\textit{ERecord} \\
\hline
\text{(constraints)} \\
\hline
\end{array}
$$

where $x, y, \ldots$ are attributes of E.

For example, if the entity DepositAccount has just three attributes *balance*, *depositDate* and *interest*, and if *depositDate* and *interest* are interdependent, then

$$
\begin{array}{|l}
\hline
\textit{DepositAccountRecord} \\
\textit{balanceofDepositAccountConstraints} \\
\textit{depositDateofDepositAccount} \\
\textit{interestofDepositAccount} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{DepositAccountRecordConstraints} \\
\textit{DepositAccountRecord} \\
\hline
P(\textit{depositDate}, \textit{interest}) \\
\hline
\end{array}
$$

where $P(\textit{depositDate}, \textit{interest})$ is a predicate describing the interdependency between these two attributes.

3.1.3 Tables

For each entity E, we set up a table schema and a table constraint schema, if there are any constraints on the table, of the following forms:

$[\textit{EId}]$

$$
\begin{array}{|l}
\hline
\textit{ETable} \\
\textit{knownE} : \mathbb{F}\ \textit{EId} \\
\textit{tableE} : \textit{EId} \mathrel{+\!\!\!\!\!\twoheadrightarrow} \textit{ERecord} \\
\hline
\textit{knownE} = \operatorname{dom} \textit{tableE} \\
[\forall\ \textit{ERecord} \mid \theta \textit{ERecord} \in \operatorname{ran} \textit{tableE} \bullet \textit{ERecordConstraints}, \\
\qquad\qquad\qquad \text{if there are any constraints on the record}\ ] \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{ETableConstraints} \\
\textit{ETable} \\
\hline
\text{(constraints)} \\
\hline
\end{array}
$$

For example, if the number of all occurrences of DepositAccount is limited to *MaxDepositAccounts*, then

$[DepositAccountId]$

$$
\begin{array}{l}
\textbf{DepositAccountTable} \\
\hline
knownDepositAccount : \mathbb{F}\ DepositAccountId \\
tableDepositAccount : DepositAccountId \twoheadrightarrow DepositAccountRecord \\
\hline
knownDepositAccount = \mathrm{dom}\ tableDepositAccount \\
\forall\ DepositAccountRecord \mid \\
\quad \theta DepositAccountRecord \in \mathrm{ran}\ tableDepositAccount \bullet \\
\quad DepositAccountRecordConstraints \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textbf{DepositAccountTableConstraints} \\
\hline
DepositAccountTable \\
\hline
\#knownDepositAccount \leq MaxDepositAccounts \\
\hline
\end{array}
$$

3.1.4 Relationships

For each relationship R between the detail entity E and the master entity F, we set up a relationship schema and a relationship constraint schema of the following forms:

$$
\begin{array}{l}
\textbf{RRelationship} \\
\hline
R : EId \twoheadrightarrow FId \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textbf{RConstraints} \\
\hline
RRelationship \\
knownE : \mathbb{F}\ EId \\
knownF : \mathbb{F}\ FId \\
\hline
\mathrm{dom}\ R \subseteq knownE \\
\mathrm{ran}\ R \subseteq knownF \\
\text{(constraints)} \\
\hline
\end{array}
$$

Note that R can always be defined as a partial function since we omit the many-to-many correspondence case as explained in Section 2.2.

For example, the relationship IsOwnedBy in Section 2.2 can be specified as follows:

IsOwnedByRelationship

$$IsOwnedBy : AccountId \nrightarrow CustomerId$$

IsOwnedByConstraints

$$\begin{array}{l} IsOwnedByRelationship \\ knownAccount : \mathbb{F}\ AccountId \\ knownCustomer : \mathbb{F}\ CustomerId \\ \hline \operatorname{dom} IsOwnedBy = knownAccount \\ \operatorname{ran} IsOwnedBy \subseteq knownCustomer \end{array}$$

We should also note that a cardinality constraint: "each Customer may own one or more Accounts" is guaranteed by both the functionality of *IsOwnedBy* and the first predicate in the constraint schema while the other cardinality constraint: "each Account is always owned by exactly one Customer" is guaranteed by both the functionality of *IsOwnedBy* and the two predicates.

3.2 Expressing DB Integrity Constraints in Z

As suggested in Section 2.2, DB integrity constraints can be expressed within subsystems or "subject areas". For example, constraints for the 'subtype' CurrentAccount of Account can be expressed as follows:

CurrentAccountIsSubtypeofAccount

$$\begin{array}{l} AccountTable \\ CurrentAccountTable \\ CurrentAccountIsAccountRelationship \\ \hline tableCurrentAccount \mathbin{;} (\lambda\ CurrentAccountRecord \bullet \theta AccountRecord) \\ \qquad \subseteq CurrentAccountIsAccount \mathbin{;} tableAccount \end{array}$$

where it is assumed that *CurrentAccountRecord* contains all components of *AccountRecord*.

In general, an integrity constraint on a subject area which consists of entities $E_1, \ldots, E_i$ and relationships $R_1, \ldots, R_j$ can be expressed in the following form:

```
IC
E1 Table
...
Ei Table
R1 Relationship
...
Rj Relationship
----------
(constraints)
```

The other example of an integrity constraint about VIP stated in Section 2.2 can be expressed as follows:

```
RuleofVipHasLargeOverdraftLimit
CustomerTable
CurrentAccountTable
IsOwnedByRelationship
CurrentAccountIsAccountRelationship
----------
∀ c : CustomerId; a : CurrentAccountId |
    c ∈ knownCustomer ∧ a ∈ knownCurrentAccount ∧
    a ↦ c ∈ CurrentAccountIsAccount ; IsOwnedBy ∧
    (tableCustomer c).vip = yes •
    (tableCurrentAccount a).balance ≥ −100
```

3.3 Definition of DB

Since the database of the system reflects the ER data model, we can define the whole database consisting of entities $E_1, \ldots, E_n$, relationships $R_1, \ldots, R_m$ and integrity constraints $IC_1, \ldots, IC_k$ with the following form:

```
DB
E1 TableConstraints   (or E1 Table if there are no constraints on the table)
...
En TableConstraints   (or En Table if there are no constraints on the table)
R1 Constraints
...
Rm Constraints
IC1
...
ICk
```

For example, the database of the banking example is as follows:

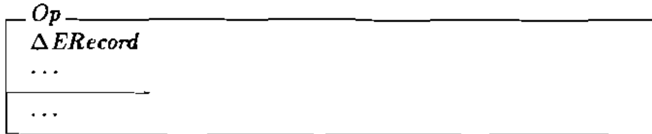
```
┌─ BankingDB ─────────────────────────────────
│ CustomerTable
│ AccountTable
│ CurrentAccountTable
│ DepositAccountTableConstraints
│ IsOwnedByConstraints
│ CurrentAccountIsAccountConstraints
│ DepositAccountIsAccountConstraints
│ CurrentAccountIsSubtypeofAccount
│ DepositAccountIsSubtypeofAccount
│ RuleofVipHasLargeOverdraftLimit
│ RuleofNonVipHasSmallOverdraftLimit
└─────────────────────────────────────────────
```

3.4 Expressing Business Processes in Z

As we have seen in Section 2.2, an entire process can be broken down into parallel subprocesses, each of which updates a certain entity E or a certain relationship R. Next, each subprocess is broken down into parallel basic operations (*Op*'s) each of which in turn modifies, inserts, or deletes some records (or links) in entity E (or relationship R). Furthermore, each basic operation on an entity of type 'modify' or 'insert' can be broken down into atomic operations, each of which modifies or creates just one value of a certain attribute.

It is worthwhile noticing that the constraints which must be preserved by each process and operation vary according to the layer of data and processes. Each atomic operation on a certain attribute has to preserve the constraints of the attribute, but may possibly violate other constraints at a higher level than the attribute level. Each basic operation *Op* on some records in a certain entity has to preserve the constraints of the records which it tries to modify or insert but may possibly violate the constraints of the entity table (such as volumetric constraints) and the DB integrity constraints. Each subprocess acting on an entity has to maintain the constraints of the entity table, and only the entire process has to care about the DB integrity constraints.

Consequently, at the record level for instance, we would write a schema

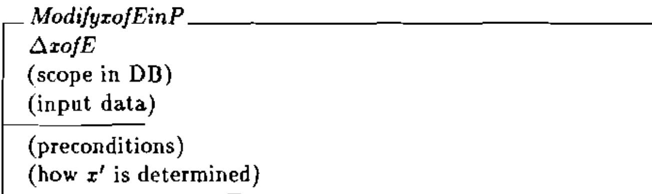


which would be promoted up first to a subprocess on *ETable* and then to a process on *DB*, rather than a schema

or

3.4.1 Attribute Level

For each atomic operation of any process P, we set up one attributeoperation schema as follows:
If the operation modifies the value of attribute x of entity E, then



If the operation creates the value of attribute x of entity E, then

```
┌─ CreatexofEinP ─────────────────────────────
│ xofE'
│ (scope in DB)
│ (input data)
├──────────
│ (preconditions)
│ (how x' is determined)
└─────────────────────────────────────────────
```

where *xofE* is an attribute schema defined as in Section 3.1.1. (scope in DB) is a set of schema references to some schemas in DB of before-state, which are needed to describe (preconditions) and (how x' is determined). (input data) is a set of declarations of some global input data which are needed to describe (preconditions) and (how x' is determined).

Note that here we use *xofE* instead of *xofEConstraints*. It means that these operations have not yet been guaranteed to maintain the constraints on x. The next chapter will examine how these operations are proved to preserve the constraints (See Section 4.2).

For example, the process *Withdraw* in the banking system modifies the attribute *balance* of CurrentAccount by subtracting the inputted amount of money from the present value of this attribute, so we have:

```
┌─ ModifybalanceofCurrentAccountinWithdraw ───
│ ΔbalanceofCurrentAccount
│ amount? : ℕ
├──────────
│ balance' = balance − amount?
└─────────────────────────────────────────────
```

3.4.2 Record Level

For each basic operation of any process P, we set up one record operation schema as follows:
If the operation modifies the record(s) of entity E, then

```
┌─ ModifyERecordinP ──────────────────────────
│ ΔERecord
│ (scope in DB)
│ (input data)
│ ModifyxofEinP
│ ModifyyofEinP
│ ...
│ ModifyzofEinP
├──────────
│ (preconditions)
└─────────────────────────────────────────────
```

If the operation inserts the record(s) of entity E, then

```
┌─ CreateERecordinP ─────────────────────────
│ ERecord'
│ (scope in DB)
│ (input data)
│ CreatexofEinP
│ CreateyofEinP
│ ...
│ CreatezofEinP
├──────────
│ (preconditions)
└────────────────────────────────────────────
```

where *ERecord* is a record schema defined as in Section 3.1.2. $x, y, \ldots, z$ are attributes of E and *ModifyxofEinP*, ..., *ModifyzofEinP*, *CreatexofEinP*, ..., *CreatezofEinP* are attribute operation schemas as defined in the preceding section. (scope in DB) is a set of schema references to some schemas in DB of before-state, which are needed to describe (preconditions). (input data) contains some global input data needed to describe (preconditions).

Also note that here we use *ERecord* instead of *ERecordConstraints*. It means that these operations have not yet been guaranteed to maintain the constraints on the record (See Section 4.3).

For example, the process *Withdraw* in the banking system modifies a certain record of CurrentAccount which consists of just one attribute *balanee*, so we have:

```
┌─ ModifyCurrentAccountRecordinWithdraw ─────
│ ΔCurrentAccountRecord
│ ModifybalanceofCurrentAccountinWithdraw
└────────────────────────────────────────────
```

3.4.3 Table Level

For each subprocess on entity E of any process P, we set np one subprocess schema as follows:

$$
\begin{array}{l}
\textit{SubprocessonEinP} \\
\hline
\Delta \textit{ETable} \\
\text{(scope in DB)} \\
a_1?, \ldots, a_n? : \textit{EId} \\
b_1?, \ldots, b_m? : \textit{EId} \\
c_1?, \ldots, c_k? : \textit{EId} \\
\text{(other input data)} \\
\hline
\text{(preconditions)} \\
\{a_1?, \ldots, a_n?\} \subseteq \textit{knownE} \\
\text{disjoint } \langle \{b_1?, \ldots, b_m?\}, \textit{knownE} \rangle \\
\{c_1?, \ldots, c_k?\} \subseteq \textit{knownE} \\
\text{disjoint } \langle \{a_1?, \ldots, a_n?\}, \{c_1?, \ldots, c_k?\} \rangle \\
\textit{knownE}' = (\textit{knownE} \setminus \{c_1?, \ldots, c_k?\}) \cup \{b_1?, \ldots, b_m?\} \\
\exists \Delta \textit{ERecord} \mid a_1? \mapsto \theta \textit{ERecord} \in \textit{tableE} \land \\
\qquad\qquad a_1? \mapsto \theta \textit{ERecord}' \in \textit{tableE}' \bullet \\
\qquad\qquad\qquad \textit{ModifyERecord}_1 \textit{inP} \\
\ldots \\
\exists \Delta \textit{ERecord} \mid a_n? \mapsto \theta \textit{ERecord} \in \textit{tableE} \land \\
\qquad\qquad a_n? \mapsto \theta \textit{ERecord}' \in \textit{tableE}' \bullet \\
\qquad\qquad\qquad \textit{ModifyERecord}_n \textit{inP} \\
\exists \textit{ERecord}' \mid b_1? \mapsto \theta \textit{ERecord}' \in \textit{tableE}' \bullet \\
\qquad\qquad\qquad \textit{CreateERecord}_1 \textit{inP} \\
\ldots \\
\exists \textit{ERecord}' \mid b_m? \mapsto \theta \textit{ERecord}' \in \textit{tableE}' \bullet \\
\qquad\qquad\qquad \textit{CreateERecord}_m \textit{inP} \\
\{a_1?, \ldots, a_n?, b_1?, \ldots, b_m?\} \lhd \textit{tableE}' = \\
\qquad \{a_1?, \ldots, a_n?, c_1?, \ldots, c_k?\} \lhd \textit{tableE} \\
\hline
\end{array}
$$

where *ETable* is a table schema as defined in Section 3.1.3. $\textit{ModifyERecord}_1 \textit{inP}$, ..., $\textit{ModifyERecord}_n \textit{inP}$ are the record operation schemas (as defined in the preceding section) which describe how the records associated with the ID's, $a_1?$, ..., $a_n?$, are modified respectively. $\textit{CreateERecord}_1 \textit{inP}$, ..., $\textit{CreateERecord}_m \textit{inP}$ are the record operation schemas (as defined in the preceding section) which describe how the records associated with the ID's, $b_1?$, ..., $b_m?$, are created respectively. $c_1?$, ..., $c_k?$ are the ID's associated with the records to be deleted. (preconditions) is a set of preconditions includiug all preconditions obtained in the analysis of its basic operations and atomic operations. (scope in DB) is a set of schema references to some schemas in DB of before-state, which are needed to describe (preconditions). (other input data) contains all input data for all the record operation schemas and

all the attribute operation schemas, and other global input data needed to describe (preconditions).

Also note that here we use *ETable* instead of *ETableConstraints*. It means that this subprocess has not yet been guaranteed to maintain the constraints on the table (See Section 4.4).

For example, the process *Withdraw* in the banking system modifies a certain record of CurrentAccount of which ID is inputted, so we have:

$$
\begin{array}{l}
\textit{SubprocessonCurrentAccountinWithdraw} \\
\hline
\Delta \textit{CurrentAccountTable} \\
\textit{eid?} : \textit{CurrentAccountId} \\
\textit{amount?} : \mathbb{N} \\
\hline
\textit{eid?} \in \textit{knownCurrentAccount} \\
\textit{knownE}' = \textit{knownE} \\
\exists \Delta \textit{CurrentAccountRecord} \mid \\
\quad \textit{eid?} \mapsto \theta \textit{CurrentAccountRecord} \in \textit{tableCurrentAccount} \wedge \\
\quad \textit{eid?} \mapsto \theta \textit{CurrentAccountRecord}' \in \textit{tableCurrentAccount}' \bullet \\
\qquad \textit{ModifyCurrentAccountRecordinWithdraw} \\
\{\textit{eid?}\} \ndres \textit{tableCurrentAccount}' = \\
\qquad \{\textit{eid?}\} \ndres \textit{tableCurrentAccount}
\end{array}
$$

3.4.4 Relationships

For each subprocess on relationship R of any process P, we set up one subprocess schema as follows:

$$
\begin{array}{l}
\textit{SubprocessonRinP} \\
\hline
\Delta \textit{RRelationship} \\
\text{(scope in DB)} \\
\textit{InputDataForModifyRinP} : \textit{EId} \pfun \textit{FId} \\
\textit{InputDataForInsertRinP} : \textit{EId} \pfun \textit{FId} \\
\textit{InputDataForDeleteRinP} : \mathbb{F}\ \textit{EId} \\
\hline
\text{(preconditions)} \\
\operatorname{dom} \textit{InputDataForModifyRinP} \subseteq \operatorname{dom} R \\
\text{disjoint}\ \langle \operatorname{dom} \textit{InputDataForInsertRinP}, \operatorname{dom} R \rangle \\
\textit{InputDataForDeleteRinP} \subseteq \operatorname{dom} R \\
\text{disjoint}\ \langle \operatorname{dom} \textit{InputDataForInsertRinP}, \textit{InputDataForDeleteRinP} \rangle \\
R' = ((\textit{InputDataForDeleteRinP} \ndres R) \cup \\
\qquad \textit{InputDataForInsertRinP}) \oplus \\
\qquad \textit{InputDataForModifyRinP}
\end{array}
$$

where E is the detail entity and F is the master entity of R. *RRelationship* is a relationship schema defined as in Section 3.1.4. (scope in DB) is a set of schema references to some schemas in DB of before-state, which are needed to describe (preconditions).

Also note that here we use *RRelationship* instead of *RRelationshipConstraints*. It means that this subprocess has not yet been guaranteed to maintain the constraints on the relationship (See Section 4.5).

For example, the process *CreateNewAccount* in the banking system would insert a new link into the relationship IsOwnedBy, so we have:

SubprocessonIsOwnedByinCreateNewAccount

$\Delta IsOwnedByRelationship$
$cid? : CustomerId$
$newaid? : AccountId$

$newaid? \notin \mathrm{dom}\, IsOwnedBy$
$IsOwnedBy' = IsOwnedBy \cup \{newaid? \mapsto cid?\}$

3.4.5 Business Processes updating DB

Assume that a business process P affects only entities $E_1, \ldots, E_i$, and relationships $R_1, \ldots, R_j$. Then we can define the substantial part of P by specifying its effects on $E_1, \ldots, E_i, R_1, \ldots, R_j$ as follows:

$$\begin{aligned} T \mathrel{\widehat=} & SubprocessonE_1inP \wedge \cdots \wedge SubprocessonE_iinP \wedge \\ & SubprocessonR_1inP \wedge \cdots \wedge SubprocessonR_jinP \wedge \\ & GlobalPrecondition_1inP \wedge \cdots \wedge GlobalPrecondition_kinP \end{aligned}$$

where $SubprocessonE_1inP, \ldots, SubprocessonE_iinP, SubprocessonR_1inP, \ldots,$ and $SubprocessonR_jinP$ are subprocess schemas as defined in Section 3.4.3, 3.4.4. $GlobalPrecondition_1inP, \ldots,$ and $GlobalPrecondition_kinP$ describe the preconditions which arise not from a particular entity or relationship affected by P but from some constraints of relationships unchanged or DB integrity constraints (See Section 4.6, 4.7).

Meanwhile the entire process P can also be defined as follows:

$$P \mathrel{\widehat=} \Delta DB \wedge T \wedge (\bigwedge_{otherE} \Xi ETable) \wedge (\bigwedge_{otherR} \Xi RRelationship)$$

In principle, once the entire process is defined, the precondition of the process can be obtained by Z schema calculation. However, we do not take this approach, the reason for which will be stated in the next chapter.

Chapter 4

Proof Obligations

4.1 Structural Checking / Precondition Calculation

As overviewed in Section 1.2, there may be two approaches to ensure the correctness of the specification of business processes:

- Once the Z specification for the entire process P is obtained. its precondition can be calculated by Z schema calculation and simplified by logical and mathematical laws. Then we can compare it with the desired precondition which P should have and can strengthen the specification if necessary (e.g. by adding error handling [Woodcock 92]).

- As we have seen in the preceding chapter, the constraints which the process P has to preserve are structured and layered corresponding to the structure of the ER data model. Therefore we can construct a series of proof obligations through which the specification is corrected and strengthened by adding some preconditions (in its meaning in VDM [Jones 86]) if necessary. These proof obligations are all of the same form:

  $$I \land P \land \Xi(I \setminus P) \vdash I'$$

  where I and P are a constraint schema and an operation schema respectively at the appropriate level. $\Xi(I \setminus P)$ means that all components of I which are not contained in P remain unchanged.

In the first approach, since the entire process can be defined as follows (See Section 3.4.5):

$$P \mathrel{\widehat=} \Delta DB \wedge T \wedge (\bigwedge_{otherE} \Xi ETable) \wedge (\bigwedge_{otherR} \Xi RRelationship)$$

$$\begin{aligned} T \mathrel{\widehat=} & \; SubprocessonE_1inP \wedge \cdots \wedge SubprocessonE_iinP \wedge \\ & SubprocessonR_1inP \wedge \cdots \wedge SubprocessonR_jinP \wedge \\ & GlobalPrecondition_1inP \wedge \cdots \wedge GlobalPrecondition_kinP, \end{aligned}$$

we need to calculate pre P.

However, this calculation cannot be broken down since pre $(P \wedge Q)$ is not equal to (pre P) $\wedge$ (pre Q) in general. In consequence, we are forced to manipulate the huge schema P, which seems unduly complicated when only a small number of entities and relationships are affected by the process.

On the contrary, one of the benefits we can enjoy from the second approach, i.e. structural checking, is that this form of proof obligations can be broken down in a structural way.

The final proof obligation is:

$$DB \wedge T \wedge \Xi(DB \setminus T) \vdash DB'$$

which means that if a state of DB satisfies all constraints, so does its state after the process operates upon part of it (and the rest is left unchanged). Since DB is defined as follows (See Section 3.3):

$$\begin{aligned} DB \mathrel{\widehat=} & \; E_1\,TableConstraints \wedge \ldots \wedge E_n\,TableConstraints \wedge \\ & R_1\,Constraints \wedge \ldots \wedge R_m\,Constraint \wedge \\ & IC_1 \wedge \ldots \wedge IC_k, \end{aligned}$$

by **Break-down Lemma 1** below, the final proof obligation can be broken down into:

$$E_1\,TableConstraints \wedge T \wedge \Xi(E_1\,TableConstraints \setminus T) \vdash E_1\,TableConstraints'$$

$$\ldots$$

$$E_n\,TableConstraints \wedge T \wedge \Xi(E_n\,TableConstraints \setminus T) \vdash E_n\,TableConstraints'$$

$$R_1\,Constraints \wedge T \wedge \Xi(R_1\,Constraints \setminus T) \vdash R_1\,Constraints'$$

$$\ldots$$

$$R_m\,Constraints \wedge T \wedge \Xi(R_m\,Constraints \setminus T) \vdash R_m\,Constraints'$$

$$IC_1 \wedge T \wedge \Xi(IC_1 \setminus T) \vdash IC_1'$$

$$\ldots$$

$$IC_k \wedge T \wedge \Xi(IC_k \setminus T) \vdash IC_k'$$

Furthermore, by **Break-down Lemma2** below, these obligations result in:

For all E affcted by P,

$$ETableConstraints \wedge SubprocessonEinP \vdash ETableConstraints'$$

For all R between E and F such that one or more among these three are affected by

$$RConstraints \wedge EffectonRofP \wedge EffectonEofP \wedge EffectonFofP \vdash RConstraints'$$

$$\text{where} \quad EffectonRofP = \begin{cases} SubprocessonRinP, & \text{if P affects R} \\ \Xi RRelationship, & \text{otherwise} \end{cases}$$

$$EffectonEofP = \begin{cases} SubprocessonEinP, & \text{if P affects E} \\ \Xi ETable, & \text{otherwise} \end{cases}$$

$$EffectonFofP = \begin{cases} SubprocessonFinP, & \text{if P affects F} \\ \Xi FTable, & \text{otherwise} \end{cases}$$

For all IC containing some entity or relationship which is affected by P,

$$IC \wedge (\bigwedge_{EinIC} EffectonEofP) \wedge (\bigwedge_{RinIC} EffectonRofP) \vdash IC'$$

$$\text{where} \quad EffectonEofP = \begin{cases} SubprocessonEinP, & \text{if P affects E} \\ \Xi ETable, & \text{otherwise} \end{cases}$$

$$EffectonRofP = \begin{cases} SubprocessonRinP, & \text{if P affects R} \\ \Xi RRelationship, & \text{otherwise} \end{cases}$$

In the following sections, we further break down the obligations regarding to eutity tables according to the layer structure of attributes, records and tables.

Break-down Lemma 1 *Let I_1 and I_2 be constraint schemas both of which an operation schema P must preserve. Then, if we can prove*

$$I_1 \wedge P \wedge \Xi(I_1 \setminus P) \vdash I_1' \quad \textit{and}$$
$$I_2 \wedge P \wedge \Xi(I_2 \setminus P) \vdash I_2',$$

then we can prove

$$(I_1 \wedge I_2) \wedge P \wedge \Xi((I_1 \wedge I_2) \setminus P) \vdash I_1' \wedge I_2'.$$

Break-down Lemma 2 *Let I be a constraint schema which an operation schema P must preserve. Let P_1 be the part of P which satisfies $P \vdash P_1$ and $I \setminus P = I \setminus P_1$ (i.e. the part within I). Then, if we can prove*

$$I \wedge P_1 \wedge \Xi(I \setminus P_1) \vdash I',$$

then we can prove

$$I \wedge P \wedge \Xi(I \setminus P) \vdash I'.$$

The proof of each lemma is obvious.

Another benefit from structural checking compared with precondition calculation is its ability to deal with nondeterminism properly. For example, assume that I and P are the following:

I

$x : \mathbb{Z}$
$x > 0$

P

$x' : \mathbb{Z}$
$x' = 1 \lor x' = -1$

In this case, on the one hand, the proof obligation degenerates into

$$P \vdash I'$$

since P does not refer to any component of I and neither does I'. Obviously this proof obligation is not satisfied.

On the other hand, precondition calculation in Z leads us to

$$\text{pre}\,(P \land \Delta I)$$

$=$

$x : \mathbb{Z}$
$x > 0$ $\exists x' : \mathbb{Z} \mid x' > 0 \bullet x' = 1 \lor x' = -1$

$=$

$x : \mathbb{Z}$
$x > 0$

and it might be concluded that P is correct.

Consequently, as far as the correctness of a specification is concerned, constraints have to be checked (as in VDM), rather than assuming a sufficiently strong precondition is intended (as in Z).

4.2 Attribute Level

As we have seen in Section 3.4.1, an operation schema at the lowest level is of the following forms:

```
┌─ ModifyxofEinP ─────────────────────────
│ ΔxofE
│ (scope in DB)
│ (input data)
├──────────
│ (preconditions)
│ (how x' is determined)
└─────────────────────────────────────────
```

```
┌─ CreatexofEinP ─────────────────────────
│ xofE'
│ (scope in DB)
│ (input data)
├──────────
│ (preconditions)
│ (how x' is determined)
└─────────────────────────────────────────
```

The proof obligation for each of these operations is:

$$xofEConstraints \wedge ModifyxofEinP \vdash xofEConstraints'$$

$$CreatexofEinP \vdash xofEConstraints'$$

only when *xofEConstraints* exists.

If the proof obligation cannot be discharged, it is necessary to add some preconditions (and possibly extend the scope in DB) to the operation schema so that the proof obligation becomes provable this time.

For example, the process *CreateNewDepositAccount* in the banking system inserts a new record in DepositAccount. The value of the attribute *balance* of the record is determined to be equal to the inputted data *amount*?, so we have:

```
┌─ CreatebalanceofDepositAccountinCreateNewDepositAccount ──
│ balanceofDepositAccount'
│ amount? : ℕ
├──────────
│ balance' = amount?
└─────────────────────────────────────────
```

Since the attribute *balance* of DepositAccount has a constraint schema *balanceofDepositAccountConstraints* which describes "$balance > 0$" as in Section 3.1.1, we have a proof obligation:

$$CreatebalanceofDepositAccountinCreateNewDepositAccount \vdash balanceofDepositAccountConstraints'$$

Comparing the schemas on both sides, it becomes clear that the proof obligation cannot be proved unless the precondition: "$amount? > 0$" is added to the left hand side. Consequently, we must strengthen the operation schema as follows:

```
── CreatebalanceofDepositAccountinCreateNewDepositAccount ──
│ balanceofDepositAccount'
│ amount? : ℕ
├──────
│ amount? > 0
│ balance' = amount?
└──────────────────
```

4.3 Record Level

As we have seen in Section 3.4.2, an operation schema at the record level takes one of the following forms:

```
── ModifyEReeordinP ──────────────
│ ΔERecord
│ (scope in DB)
│ (input data)
│ ModifyxofEinP
│ ModifyyofEinP
│ ...
│ ModifyzofEinP
├──────
│ (preconditions)
└──────────────────
```

```
— CreateERecordinP ——————————————————
| ERecord'
| (scope in DB)
| (input data)
| CreatexofEinP
| CreateyofEinP
| ...
| CreatezofEinP
|——————————
| (preconditions)
——————————————————————————————
```

Note that once all the proof obligations for its atomic operations (i.e. *ModifyxofEinP*, ..., *ModifyzofEinP*, *CreatexofEinP*, ..., *CreatezofEinP*) have beeu checked as in the preceding section, *ERecord'* is guaranteed since *ERecord* contains all constraints just at attribute level.

The proof obligation for each of these operations is:

$$ERecordConstraints \wedge ModifyERecordinP \vdash ERecordConstraints'$$

$$CreateERecordinP \vdash ERecordConstraints'$$

only when *EReeordConstraints* exists.

We may have to add some preconditions to the operation schema if the proof obligation cannot be proved.

For example, the process *CreateNewDepositAccount* in the banking system inserts a new record in DepositAccount. We assume that all its atomic operations have already been guaranteed to preserve their constraints. Then the operation at the record level is as follows:

```
— CreateDepositAccountRecordinCreateNewDepositAccount ——
| DepositAccountRecord'
| CreatebalanceofDepositAccountinCreateNewDepositAccount
| CreatedepositDateofDepositAccountinCreateNewDepositAccount
| CreateinterestofDepositAccountinCreateNewDepositAccount
——————————————————————————————
```

Since *DepositAccountRecord* has a constraint schema *DepositAccountRecordConstraints* which describes "$P(depositDate, interest)$" as in Section 3.1.2, we have a proof obligation:

$$CreateDepositAccountRecordinCreateNewDepositAccount \\ \vdash DepositAccountRecordConstraints'$$

Comparing the schemas in both sides, it may be concluded that the precondition: "*P*(*today*?, *interest*?)" is necessary in the left hand side. Consequently, we may strengthen the operation schema as follows:

```
┌─ CreateDepositAccountRecordinCreateNewDepositAccount ────
│ DepositAccountRecord'
│ CreatebalanceofDepositAccountinCreateNewDepositAccount
│ CreatedepositDateofDepositAccountinCreateNewDepositAccount
│ CreateinterestofDepositAccountinCreateNewDepositAccount
├──────────
│ P(today?, interest?)
└──────────────────────────────────────────
```

4.4 Table Level

As we have seen in Section 3.4.3, a subprocess schema at the table level is of the following forms:

SubprocessonEinP

$\Delta ETable$
(scope in DB)
$a_1?, \ldots, a_n? : EId$
$b_1?, \ldots, b_m? : EId$
$c_1?, \ldots, c_k? : EId$
(other input data)

(preconditions)
$\{a_1?, \ldots, a_n?\} \subseteq knownE$
$\text{disjoint} \langle \{b_1?, \ldots, b_m?\}, knownE \rangle$
$\{c_1?, \ldots, c_k?\} \subseteq knownE$
$\text{disjoint} \langle \{a_1?, \ldots, a_n?\}, \{c_1?, \ldots, c_k?\} \rangle$
$knownE' = (knownE \setminus \{c_1?, \ldots, c_k?\}) \cup \{b_1?, \ldots, b_m?\}$
$\exists \Delta ERecord \mid a_1? \mapsto \theta ERecord \in tableE \wedge$
$\quad a_1? \mapsto \theta ERecord' \in tableE' \bullet$
$\quad\quad ModifyERecord_1 inP$
...
$\exists \Delta ERecord \mid a_n? \mapsto \theta ERecord \in tableE \wedge$
$\quad a_n? \mapsto \theta ERecord' \in tableE' \bullet$
$\quad\quad ModifyERecord_n inP$
$\exists ERecord' \mid b_1? \mapsto \theta ERecord' \in tableE' \bullet$
$\quad\quad CreateERecord_1 inP$
...
$\exists ERecord' \mid b_m? \mapsto \theta ERecord' \in tableE' \bullet$
$\quad\quad CreateERecord_m inP$
$\{a_1?, \ldots, a_n?, b_1?, \ldots, b_m?\} \lhd tableE' =$
$\quad \{a_1?, \ldots, a_n?, c_1?, \ldots, c_k?\} \lhd tableE$

Note that once all the proof obligations for its atomic operations and basic operations (i.e. $ModifyERecord_1 inP$, ..., $ModifyERecord_n inP$, $CreateERecord_1 inP$, ..., $CreateERecord_m inP$) have been checked as in Section 4.2, 4.3, $ETable'$ is guaranteed since $ETable$ contains all constraints just at attribute level and record level.

The proof obligation for this subprocess is:

$$ETableConstraints \wedge SubprocessonEinP \vdash ETableConstraints'$$

only when *ETableConstraints* exists.

We may have to add some preconditions to the subprocess schema if the proof obligation cannot be proved.

For example, the process *CreateNewDepositAccount* in the banking system inserts a new record in DepositAccount. We assume that all its atomic operations and the basic operation have already been guaranteed to preserve their constraints. Then the subprocess at the table level is as follows:

SubprocessonDepositAccountinCreateNewDepositAccount

$\Delta DepositAccountTable$
$eid? : DepositAccountId$
$amount? : \mathbb{N}$
$today? : Date$
$interest? : InterestType$

$amount? > 0$
$P(today?, interest?)$
$eid? \notin knownDepositAccount$
$knownDepositAccount' = knownDepositAccount \cup \{eid?\}$
$\exists DepositAccountRecord' \mid$
$\quad eid? \mapsto \theta DepositAccountRecord' \in tableDepositAccount' \bullet$
$\qquad CreateDepositAccountRecordinCreateNewDepositAccount$
$\{eid?\} \ntriangleleft tableDepositAccount' = tableDepositAccount$

Note that here the preconditions found so far (i.e. $amount? > 0$ and $P(today?, interest?)$) appear in the predicate part.

Since *DepositAccountTable* has a constraint schema *DepositAccountTableConstraints* which describes "$\#knownDepositAccount \leq MaxDepositAccounts$" as in Section 3.1.3, we have a proof obligation:

$$\begin{aligned}&DepositAccountTableConstraints \land \\ &\qquad SubprocessonDepositAccountinCreateNewDepositAccount \\ &\qquad\quad \vdash DepositAccountTableConstraints'\end{aligned}$$

Comparing the schemas in both sides, it may be concluded that the precondition: "$\#knownDepositAccount \leq MaxDepositAccounts - 1$" is necessary in the left hand side. Consequently, we may strengthen the subprocess schema as follows:

$$
\begin{array}{|l}
\hline
\textit{SubprocessonDepositAccountinCreateNewDepositAccount} \\
\Delta \textit{DepositAccountTable} \\
\textit{eid?} : \textit{DepositAccountId} \\
\textit{amount?} : \mathbb{N} \\
\textit{today?} : \textit{Date} \\
\textit{interest?} : \textit{InterestType} \\
\hline
\textit{amount?} > 0 \\
P(\textit{today?}, \textit{interest?}) \\
\#\textit{knownDepositAccount} \leq \textit{MaxDepositAccounts} - 1 \\
\textit{eid?} \notin \textit{knownDepositAccount} \\
\textit{knownDepositAccount}' = \textit{knownDepositAccount} \cup \{\textit{eid?}\} \\
\exists\, \textit{DepositAccountRecord}' \mid \\
\quad \textit{eid?} \mapsto \theta \textit{DepositAccountRecord}' \in \textit{tableDepositAccount}' \bullet \\
\qquad \textit{CreateDepositAccountRecordinCreateNewDepositAccount} \\
\{\textit{eid?}\} \ntriangleleft \textit{tableDepositAccount}' = \textit{tableDepositAccount} \\
\hline
\end{array}
$$

4.5 Relationships

As we have seen in Section 3.4.4, a subprocess schema on a relationship R between entities E and F is of the following forms:

$$
\begin{array}{|l}
\hline
\textit{SubprocessonRinP} \\
\Delta \textit{RRelationship} \\
\text{(scope in DB)} \\
\textit{InputDataForModifyRinP} : \textit{EId} \twoheadrightarrow \textit{FId} \\
\textit{InputDataForInsertRinP} : \textit{EId} \twoheadrightarrow \textit{FId} \\
\textit{InputDataForDeleteRinP} : \mathbb{F}\ \textit{EId} \\
\hline
\text{(preconditions)} \\
\operatorname{dom} \textit{InputDataForModifyRinP} \subseteq \operatorname{dom} R \\
\text{disjoint}\ \langle \operatorname{dom} \textit{InputDataForInsertRinP}, \operatorname{dom} R \rangle \\
\textit{InputDataForDeleteRinP} \subseteq \operatorname{dom} R \\
\text{disjoint}\ \langle \operatorname{dom} \textit{InputDataForInsertRinP}, \textit{InputDataForDeleteRinP} \rangle \\
R' = ((\textit{InputDataForDeleteRinP} \ntriangleleft R) \cup \\
\qquad \textit{InputDataForInsertRinP}) \oplus \\
\qquad \textit{InputDataForModifyRinP} \\
\hline
\end{array}
$$

Note that *RRelationship'* is always guaranteed since *RRelationship* contains just (partial) functionality of R. Instead there is always the constraint schema *RConstraints*.

The proof obligation for this subprocess is:

$$RConstraints \land SubprocessonRinP \land EffectonEofP \land EffectonFofP$$
$$\vdash RConstraints'$$

$$\text{where}\quad EffectonEofP = \begin{cases} SubprocessonEinP, & \text{if P affects E} \\ \Xi ETable, & \text{otherwise} \end{cases}$$

$$EffectonFofP = \begin{cases} SubprocessonFinP, & \text{if P affects F} \\ \Xi FTable, & \text{otherwise} \end{cases}$$

Here the effects on E and F of the process have to be taken into account since *RConstraints* contains the referential integrity and other constraints, which can be affected by inserting or deleting some occurrences of E or F.

If the proof obligation cannot be discharged, we may have to add some preconditions to the subprocess schema.

For example, the process *CreateNewAccount* in the banking system inserts a new link in the relationship IsOwnedBy as in Section 3.4.4:

$$\begin{array}{l}
\textit{SubprocessonIsOwnedByinCreateNewAccount} \\
\hline
\Delta \textit{IsOwnedByRelationship} \\
cid? : \textit{CustomerId} \\
newaid? : \textit{AccountId} \\
\hline
newaid? \notin \operatorname{dom} \textit{IsOwnedBy} \\
\textit{IsOwnedBy}' = \textit{IsOwnedBy} \cup \{newaid? \mapsto cid?\} \\
\hline
\end{array}$$

At the same time, the process inserts a new record in Account as follows:

$$\begin{array}{l}
\textit{SubprocessonAccountinCreateNewAccount} \\
\hline
\Delta \textit{AccountTable} \\
newaid? : \textit{AccountId} \\
amount? : \mathbb{N} \\
\hline
newaid? \notin knownAccount \\
knownAccount' = knownAccount \cup \{newaid?\} \\
\exists \textit{AccountRecord}' \mid \\
\quad newaid? \mapsto \theta \textit{AccountRecord}' \in tableAccount' \bullet \\
\quad\quad \textit{CreateAccountRecordinCreateNewAccount} \\
\{newaid?\} \lhd tableAccount' = tableAccount \\
\hline
\end{array}$$

Here we assume that *CreateAccountRecordinCreateNewAccount* is defined somewhere. We also assume that the entity Customer remains unchanged under this process. Then the proof obligation is:

$$\begin{array}{l} \mathit{IsOwnedByConstraints} \land \mathit{SubprocessonIsOwnedByinCreateNewAccount} \land \\ \quad \mathit{SubprocessonAccountinCreateNewAccount} \land \Xi \mathit{CustomerTable} \\ \qquad \vdash \mathit{IsOwnedByConstraints}' \end{array}$$

Comparing the schemas in both sides, it may be concluded that the precondition: "$cid? \in knownCustomer$" is necessary in the left hand side. Consequently, we must add not only this precondition but also a schema reference *CustomerTable* as a part of (scope in DB) to the subprocess schema as follows:

$$\begin{array}{l} \textit{SubprocessonIsOwnedByinCreateNewAccount} \\ \hline \Delta \mathit{IsOwnedByRelationship} \\ \mathit{CustomerTable} \\ cid? : \mathit{CustomerId} \\ newaid? : \mathit{AccountId} \\ \hline cid? \in \mathit{knownCustomer} \\ newaid? \notin \operatorname{dom} \mathit{IsOwnedBy} \\ \mathit{IsOwnedBy}' = \mathit{IsOwnedBy} \cup \{newaid? \mapsto cid?\} \\ \hline \end{array}$$

4.6 Unchanged Relationships

As mentioned in the preceding section, the referential integrity and some other constraints in a relationship R between E and F can be affected when the process P changes either *ETable* or *FTable*. Since they can be affected even when R itself remains unchanged, we must have proof obligations for such cases as follows:

$$\begin{array}{l} \mathit{RConstraints} \land \mathit{EffectonEofP} \land \mathit{EffectonFofP} \land \Xi \mathit{RRelationship} \\ \qquad \vdash \mathit{RConstraints}' \end{array}$$

$$\text{where} \quad \mathit{EffectonEofP} = \begin{cases} \mathit{SubprocessonEinP}, & \text{if P affects E} \\ \Xi \mathit{ETable}, & \text{otherwise} \end{cases}$$

$$\mathit{EffectonFofP} = \begin{cases} \mathit{SubprocessonFinP}, & \text{if P affects F} \\ \Xi \mathit{FTable}, & \text{otherwise} \end{cases}$$

If the proof obligation cannot be discharged, we may have to set up some global precondition schema *GlobalPrecondition* as in Section 3.4.5 rather than add some predicates to either *SubprocessonEinP* or *SubprocessonFinP* since the precondition should be attributed to neither E nor F.

Once such a precondition schema has been established, the proof obligation becomes of form:

$$RConstraints \land EffectonEofP \land EffectonFofP \land \Xi RRelationship \land GlobalPrecondition \vdash RConstraints'$$

Otherwise we may find that we have missed some subprocesses of the process. In such a case, we must specify those subprocesses and check the proof obligations again which could be spoiled by this modification.

For example, the process *CreateNewCustomer* in the banking system inserts a record in the entity Customer:

$$
\begin{array}{l}
\textit{SubprocessonCustomerinCreateNewCustomer} \\
\hline
\Delta CustomerTable \\
newcid? : CustomerId \\
name? : Name \\
vip? : YesNo \\
\hline
newcid? \notin knownCustomer \\
knownCustomer' = knownCustomer \cup \{newcid?\} \\
\exists\ CustomerRecord' \mid \\
\quad newcid? \mapsto \theta CustomerRecord' \in tableCustomer' \bullet \\
\qquad CreateCustomerRecordinCreateNewCustomer \\
\{newcid?\} \lhd tableCustomer' = tableCustomer \\
\hline
\end{array}
$$

Here we assume that *CreateCustomerRecordinCreateNewCustomer* is defined somewhere.

Also assume that the relationship IsOwnedBy has the constraint: "the number of customers who owns no accounts is limited to 100", i.e.

$$
\begin{array}{l}
\textit{IsOwnedByConstraints} \\
\hline
IsOwnedByRelationship \\
knownAccount : \mathbb{F}\ AccountId \\
knownCustomer : \mathbb{F}\ CustomerId \\
\hline
\mathrm{dom}\ IsOwnedBy = knownAccount \\
\mathrm{ran}\ IsOwnedBy \subseteq knownCustomer \\
\#(knownCustomer \setminus \mathrm{ran}\ IsOwnedBy) \leq 100 \\
\hline
\end{array}
$$

We also assume that the entity Account and the relationship IsOwnedBy remain unchanged under this process. Then the proof obligation is:

$$IsOwnedByConstraints \land SubprocessonCustomerinCreateNewCustomer \land \Xi AccountTable \land \Xi IsOwnedByRelationship \vdash IsOwnedByConstraints'$$

Comparing both sides, it may be concluded that the precondition: "$\#(knownCustomer \setminus \mathrm{ran}\ IsOwnedBy) \leq 99$" is necessary in the left hand side. Consequently, we may set up a global precondition schema, such as:

$$\begin{array}{|l}
\hline GlobalPrecondition_1 \\
CustomerTable \\
IsOwnedByRelationship \\
\hline
\#(knownCustomer \setminus \mathrm{ran}\ IsOwnedBy) \leq 99 \\
\hline
\end{array}$$

Now the new proof obligation is:

$$\begin{array}{l}
IsOwnedByConstraints \wedge SubprocessonCustomerinCreateNewCustomer \wedge \\
\quad \Xi AccountTable \wedge \Xi IsOwnedByRelationship \wedge GlobalPrecondition_1 \\
\qquad \vdash IsOwnedByConstraints'
\end{array}$$

which can be proved.

4.7 DB Integrity Constraints

If the process P affects some entity or relationship within a subject area on which some DB integrity constraint IC is specified, it could be violated by the process. The proof obligation is as follows:

$$\begin{array}{l}
IC \wedge (\bigwedge_{EinIC} SubprocessonEinP) \wedge (\bigwedge_{RinIC} SubprocessonRinP) \wedge \\
\quad (\bigwedge_{otherEinIC} \Xi ETable) \wedge (\bigwedge_{otherRinIC} \Xi RRelationship) \\
\qquad \vdash IC'
\end{array}$$

If the proof obligation cannot be proved, we may have to set up some global precondition schema *GlobalPrecondition* or find some mistake in the specification.

Once such a precondition schema has been established, the proof obligation becomes of form:

$$\begin{array}{l}
IC \wedge (\bigwedge_{EinIC} SubprocessonEinP) \wedge (\bigwedge_{RinIC} SubprocessonRinP) \wedge \\
\quad (\bigwedge_{otherEinIC} \Xi ETable) \wedge (\bigwedge_{otherRinIC} \Xi RRelationship) \wedge \\
\qquad GrobalPrecondition \\
\qquad \quad \vdash IC'
\end{array}$$

For example, the process *Withdraw* in the banking system modifies a record in the entity CurrentAccount as defined in Section 3.4.3:

$$
\begin{array}{|l}
\textit{SubprocessonCurrentAccountinWithdraw} \\
\hline
\Delta CurrentAccountTable \\
eid? : CurrentAccountId \\
amount? : \mathbb{N} \\
\hline
eid? \in knownCurrentAccount \\
knownE' = knownE \\
\exists \Delta CurrentAccountRecord \mid \\
\quad eid? \mapsto \theta CurrentAccountRecord \in tableCurrentAccount \land \\
\quad eid? \mapsto \theta CurrentAccountRecord' \in tableCurrentAccount' \bullet \\
\qquad ModifyCurrentAccountRecordinWithdraw \\
\{eid?\} \lhd tableCurrentAccount' = \\
\qquad \{eid?\} \lhd tableCurrentAccount \\
\hline
\end{array}
$$

The process may violate the DB integrity constraint *RuleofVipHasLargeOverdraftLimit* since it specifies a constraint among Customer, CurrentAccount, IsOwnedBy and CurrentAccountIsAccount as in Section 3.2:

$$
\begin{array}{|l}
\textit{RuleofVipHasLargeOverdraftLimit} \\
\hline
CustomerTable \\
CurrentAccountTable \\
IsOwnedByRelationship \\
CurrentAccountIsAccountRelationship \\
\hline
\forall c : CustomerId; \ a : CurrentAccountId \mid \\
\quad c \in knownCustomer \land a \in knownCurrentAccount \land \\
\quad a \mapsto c \in CurrentAccountIsAccount \, ; IsOwnedBy \land \\
\quad (tableCustomer\ c).vip = yes \bullet \\
\quad (tableCurrentAccount\ a).balance \geq -100 \\
\hline
\end{array}
$$

We assume that the other entity Customer and the two relationships remain unchanged under this process. Then the proof obligation is:

$$
\begin{array}{l}
RuleofVipHasLargeOverdraftLimit \land \\
\quad SubprocessonCurrentAccountinWithdraw \land \\
\quad \Xi CustomerTable \land \Xi IsOwnedByRelationship \land \\
\quad \Xi CurrentAccountIsAccountRelationship \\
\qquad\qquad \vdash RuleofVipHasLargeOverdraftLimit'
\end{array}
$$

Comparing both sides, we may set up a global precondition schema, such as:

$$
\begin{array}{|l}
\hline
\textit{GlobalPrecondition}_2 \\
\textit{CustomerTable} \\
\textit{CurrentAccountTable} \\
\textit{IsOwnedByRelationship} \\
\textit{CurrentAccountIsAccountRelationship} \\
eid? : \textit{CurrentAccountId} \\
amount? : \mathbb{N} \\
\hline
eid? \in \textit{knownCurrentAccount} \\
\forall c : \textit{CustomerId} \mid \\
\quad c \in \textit{knownCustomer} \land \\
\quad eid? \mapsto c \in \textit{CurrentAccountIsAccount} \mathbin{;} \textit{IsOwnedBy} \land \\
\quad (\textit{tableCustomer}\ c).vip = yes \bullet \\
\quad amount? \leq (\textit{tableCurrentAccount}\ eid?).balance + 100 \\
\hline
\end{array}
$$

Now the new proof obligation is:

$$
\begin{array}{l}
\textit{RuleofVipHasLargeOverdraftLimit} \land \\
\quad \textit{SubprocessonCurrentAccountinWithdraw} \land \\
\quad \Xi \textit{CustomerTable} \land \Xi \textit{IsOwnedByRelationship} \land \\
\quad \Xi \textit{CurrentAccountIsAccountRelationship} \land \textit{GlobalPrecondition}_2 \\
\qquad \vdash \textit{RuleofVipHasLargeOverdraftLimit}'
\end{array}
$$

which can be proved.

4.8 Example: Resident Registration System

In this section, the typical usage of the proof obligations introduced so far is illustrated with an example of a city office system. This system maintains some information on residents who live in the city. The ER data model for this system is shown in Figure 4.1.
Translation of the ER model into Z is as follows:

Basic Types:

$[\textit{Name}, \textit{Address}]$

$\textit{RelType} ::= \textit{Spouse} \mid \textit{Sibling} \mid \textit{Child} \mid \cdots$

Attributes:

$$
\begin{array}{|l}
\hline
\textit{addressofFamily} \\
address : \textit{Address} \\
\hline
\end{array}
$$

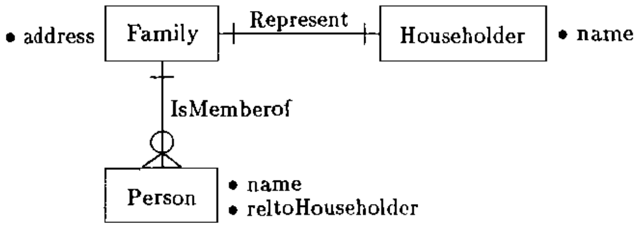


Figure 4.1: ER diagram for Resident Registration System

nameofHouseholder
name : *Name*

nameofPerson
name : *Name*

reltoHouseholderofPerson
reltoHouseholder : *RelType*

Records:

FamilyRecord
addressofFamily

HouseholderRecord
nameofHouseholder

PersonRecord
nameofPerson *reltoHouseholderofPerson*

Entity Tables:

$[FamilyId, HouseholderId, PersonId]$

FamilyTable

$$
\begin{array}{l}
knownFamily : \mathbb{F}\ FamilyId \\
tableFamily : FamilyId \nrightarrow FamilyRecord \\
\hline
knownFamily = \mathrm{dom}\ tableFamily
\end{array}
$$

HouseholderTable

$$
\begin{array}{l}
knownHouseholder : \mathbb{F}\ HouseholderId \\
tableHouseholder : HouseholderId \nrightarrow HouseholderRecord \\
\hline
knownHouseholder = \mathrm{dom}\ tableHouseholder
\end{array}
$$

PersonTable

$$
\begin{array}{l}
knownPerson : \mathbb{F}\ PersonId \\
tablePerson : PersonId \nrightarrow PersonRecord \\
\hline
knownPerson = \mathrm{dom}\ tablePerson
\end{array}
$$

Relationships:

RepresentRelationship

$$
Represent : HouseholderId \nrightarrow FamilyId
$$

RepresentConstraints

$$
\begin{array}{l}
RepresentRelationship \\
knownHouseholder : \mathbb{F}\ HouseholderId \\
knownFamily : \mathbb{F}\ FamilyId \\
\hline
\mathrm{dom}\ Represent = knownHouseholder \\
\mathrm{ran}\ Represent = knownFamily \\
Represent \in HouseholderId \rightarrowtail FamilyId
\end{array}
$$

Note that this constraints implies that the relationship is a one-to-one correspondence.

IsMemberofRelationship

$$
IsMemberof : PersonId \nrightarrow FamilyId
$$

IsMemberofConstraints

$IsMemberofRelationship$
$knownPerson : \mathbb{F}\ PersonId$
$knownFamily : \mathbb{F}\ FamilyId$

$\mathrm{dom}\ IsMemberof = knownPerson$
$\mathrm{ran}\ IsMemberof \subseteq knownFamily$

Note that this constraints implies that the relationship is a one-to-many correspondence.

Besides the ER data model, we consider one integrity constraint as follows:

DB integrity constraint:

RuleofMonogamy

$PersonTable$
$FamilyTable$
$IsMemberofRelationship$

$$\forall fid : FamilyId \mid fid \in knownFamily \bullet \\ \#\{pid : PersonId \mid \\ pid \in knownPerson \land \\ pid \mapsto fid \in IsMemberof \land \\ (tablePerson\ pid).reltoHouseholder = Spouse\} \leq 1$$

Note that this rule says that every householder has at most one spouse in his or her family.

DB:

CityOfficeDB

$FamilyTable$
$HouseholderTable$
$PersonTable$
$RepresentConstraints$
$IsMemberofConstraints$
$RuleofMonogamy$

Now we consider one process in this system as follows:

Process: HouseholderMoveOut

This process must update the database appropriately when a householder moves out of the city.

[First Try]

At first, we would think that this process is realized by just deleting the occurrence of Householder, i.e.

Input Data: $hid? : HouseholderId$

Subprocess on entity: *DeleteHouseholder*

$$
\begin{array}{|l}
\textit{DeleteHouseholder} \\
\hline
\Delta HouseholderTable \\
hid? : HouseholderId \\
\hline
hid? \in knownHouseholder \\
knownHouseholder' = knownHouseholder \setminus \{hid?\} \\
tableHouseholder' = \{hid?\} \lhd tableHouseholder \\
\hline
\end{array}
$$

Here comes the stage where we have to check some proof obligations. At first, we get a checklist showing which constraints we need to prove to be preserved by the process with a flag indicating whether the obligation has already been proved or not. At present, just one constraint among the three can be affected, so the checklist is as follows:

List of Proof Obligations: on *RepresentConstraints*, -not yet

The whole expression of this proof obligation and the (partial) expansion of each side are as follows:

$$
\begin{array}{l}
RepresentConstraints \land DeleteHouseholder \land \\
\qquad \Xi Family \land \Xi RepresentRelationship \\
\qquad\quad \vdash RepresentConstraints'
\end{array}
$$

LHS

=

$$\begin{array}{|l}
\Delta HouseholderTable \\
\Xi Family \\
\Xi RepresentRelationship \\
hid? : HouseholderId \\
\hline
\mathrm{dom}\ Represent = knownHouseholder \\
\mathrm{ran}\ Represent = knownFamily \\
Represent \in HouseholderId \twoheadrightarrow FamilyId \\
hid? \in knownHouseholder \\
knownHouseholder' = knownHouseholder \setminus \{hid?\} \\
tableHouseholder' = \{hid?\} \lhd tableHouseholder
\end{array}$$

RHS

$=$

$$\begin{array}{|l}
RepresentRelationship' \\
knownHouseholder' : \mathbb{F}\ HouseholderId \\
knownFamily' : \mathbb{F}\ FamilyId \\
\hline
\mathrm{dom}\ Represent' = knownHouseholder' \\
\mathrm{ran}\ Represent' = knownFamily' \\
Represent' \in HouseholderId \twoheadrightarrow FamilyId
\end{array}$$

Here we may find that "dom $Represent' = knownHouseholder'$" in RHS does not hold since *Represent* is left unchanged while *knownHouseholder* is changed. To make it correct, it is not sufficient to delete a link in Represent since "ran $Represent' = knownFamily'$" does not hold after that as *Represent* is injective.
We need the following case analysis:

Case 1 There remain one or more other persons in the family. In this case, one of them must become the new householder and other persons, if any, must change their *reltoHouseholder* values.

Case 2 There is no one else in the family. In this case, the family itself must be deleted.

Here we continue to think abont **Case 1**.

[Second Try]

Inpnt Data: $hid? : HouseholderId$
$pid? : PersonId$ (new householder)
$newrels? : PersonId \twoheadrightarrow RelType$

Subprocesses on entities: $Delete\&ModifyPerson$
$ModifyHouseholder$

```
┌─ ModifyreltoHouseholderofPerson ──────────────
│ ΔreltoHouseholderofPerson
│ rel : RelType
├──────────
│ reltoHouseholder' = rel
└───────────────────────────────────────────────
```

Note that the variable *rel* is used as a parameter here since we want to use this atomic operation for every person in the family.

```
┌─ nameofPersonUnchanged ───────────────────────
│ ΞnameofPerson
└───────────────────────────────────────────────
```

```
┌─ ModifyPersonRecord ──────────────────────────
│ ΔPersonRecord
│ ModifyreltoHouseholderofPerson
│ namcofPersonUnchanged
└───────────────────────────────────────────────
```

$$
\begin{array}{|l}
\hline
\textit{Delete\&ModifyPerson} \\
\Delta PersonTable \\
HouseholderTable \\
RepresentRelationship \\
IsMemberofRelationship \\
hid? : HouseholderId \\
pid? : PersonId \\
newrels? : PersonId \pfun RelType \\
\hline
hid? \in knownHouseholder \\
pid? \in knownPerson \\
\operatorname{dom} newrels? \subseteq knownPerson \\
pid? \notin \operatorname{dom} newrels? \\
\exists fid : FamilyId \mid hid? \mapsto fid \in Represent \bullet \\
\quad pid? \mapsto fid \in IsMemberof \land \\
\quad (\forall pid : PersonId \mid pid \in \operatorname{dom} newrels? \bullet \\
\qquad\qquad pid \mapsto fid \in IsMemberof) \\
knownPerson' = knownPerson \setminus \{pid?\} \\
\forall pid : PersonId; rel : RelType \mid pid \mapsto rel \in newrels? \bullet \\
\quad (\exists \Delta PersonRecord \mid \\
\qquad pid \mapsto \theta PersonRecord \in tablePerson \land \\
\qquad pid \mapsto \theta PersonRecord' \in tablePerson' \bullet \\
\qquad\quad ModifyPersonRecord) \\
\operatorname{dom} newrels? \ndres tablePerson' = \\
\qquad (\operatorname{dom} newrels? \cup \{pid?\}) \ndres tablePerson \\
\hline
\end{array}
$$

Here *rel* is used as a parameter to *ModifyPersonRecord*.

$$
\begin{array}{|l}
\hline
\textit{ModifynameofHouseholder} \\
\Delta nameofHouseholder \\
PersonTable \\
pid? : PersonId \\
\hline
pid? \in knownPerson \\
name' = (tablePerson\ pid?).name \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{ModifyHouseholderRecord} \\
\Delta HouseholderRecord \\
ModifynameofHouseholder \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{ModifyHouseholder} \\
\Delta \textit{HouseholderTable} \\
\textit{PersonTable} \\
\textit{hid?} : \textit{HouseholderId} \\
\textit{pid?} : \textit{PersonId} \\
\hline
\textit{hid?} \in \textit{knownHouseholder} \\
\textit{pid?} \in \textit{knownPerson} \\
\textit{knownHousehalder}' = \textit{knownHouseholder} \\
\exists \Delta \textit{HouseholderRecord} \mid \\
\quad \textit{hid?} \mapsto \theta \textit{HouseholderRecord} \in \textit{tableHouseholder} \land \\
\quad \textit{hid?} \mapsto \theta \textit{HouseholderRecord}' \in \textit{tableHouseholder}' \bullet \\
\qquad \textit{ModifyHouseholderRecord} \\
\{\textit{hid?}\} \lhd \textit{tableHouseholder}' = \{\textit{hid?}\} \lhd \textit{tableHouseholder} \\
\hline
\end{array}
$$

Here the precondition and the scope of the atomic operation are collected in this subprocess schema.

Since there is no constraint at attribute level nor record level, we have had no proof obligation so far.
However, here comes the checklist:

List of Proof Obligations:	on *RepresentConstraints*,	-not yet
	on *IsMemberofConstraints*,	-not yet
	on *RuleofMonogamy*,	-not yet

The first obligation is:

$$
\begin{array}{l}
\textit{RepresentConstraints} \land \textit{ModifyHouseholder} \land \\
\qquad \Xi \textit{Family} \land \Xi \textit{RepresentRelationship} \\
\qquad\quad \vdash \textit{RepresentConstraints}'
\end{array}
$$

and it can be proved obviously since *knownHouseholder* is unchanged.

The second obligation is:

$$
\begin{array}{l}
\textit{IsMemberofConstraints} \land \textit{Delete\&ModifyPerson} \land \\
\qquad \Xi \textit{Family} \land \Xi \textit{IsMemberofRelationship} \\
\qquad\quad \vdash \textit{IsMemberofConstraints}'
\end{array}
$$

but "dom $\textit{IsMemberof}' = \textit{knownPerson}'$" in RHS does not hold since *IsMemberof* is left unchanged while *knownPerson* is changed.

We may find that it is needed to delete a link in IsMemberof.

[Third Try]

We specify the additional subprocess with the rest remaining.

Input Data: $hid? : HouseholderId$
$pid? : PersonId$ (new householder)
$newrels? : PersonId \rightarrow\!\!\!\!\!\rightarrow RelType$

Subprocesses on entities: *Delete&ModifyPerson*
ModifyHouseholder

Subprocess on relationship: *DeleteIsMemberofLink*

DeleteIsMemberofLink

$\Delta\, IsMemberofRelationship$
$pid? : PersonId$

$pid? \in \mathrm{dom}\ IsMemberof$
$IsMemberof' = \{pid?\} \ntriangleleft IsMemberof$

Now the checklist becomes as follows:

List of Proof Obligations:	on *RepresentConstraints*,	-already OK
	on *IsMemberofConstraints*,	-not yet
	on *RuleofMonogamy*,	-not yet

Note that the first obligation is marked OK since nothing has been changed from the second try with respect to this constraint.

The second obligation is:

$$IsMemberofConstraints \wedge Delete\&ModifyPerson \wedge DeleteIsMemberofLink \wedge \Xi Family \vdash IsMemberofConstraints'$$

This time, it is OK.

The third obligation is:

$$RuleofMonogamy \wedge Delete\&ModifyPerson \wedge DeleteIsMemberofLink \wedge \Xi Family \vdash RuleofMonogamy'$$

It cannot be proved. We need to set up a global precodition schema here.

[Fourth Try]

We specify the additional precodition schema with the rest remaining.

Input Data: $hid? : HouseholderId$
$pid? : PersonId$ (new householder)
$newrels? : PersonId \rightarrowtail RelType$

Subprocesses on entities: *Delete&ModifyPerson*
ModifyHouseholder

Subprocess on relationship: *DeleteIsMemberofLink*

Global precondition: *LimitonNewSpouse*

$$
\begin{array}{|l}
\textit{LimitonNewSpouse} \\
\hline
PersonTable \\
RepresentRelationship \\
IsMemberofRelationship \\
hid? : HouseholderId \\
pid? : PersonId \\
newrels? : PersonId \rightarrowtail RelType \\
\hline
\exists fid : FamilyId \mid hid? \mapsto fid \in Represent \bullet \\
\quad \#\{pid : PersonId;\ rel : RelType \mid \\
\quad\quad pid \mapsto rel \in newrels? \land rel = Spouse \bullet pid\} + \\
\quad \#\{pid : PersonId \mid \\
\quad\quad pid \in knownPerson \land \\
\quad\quad pid \mapsto fid \in IsMemberof \land \\
\quad\quad pid \notin (\mathrm{dom}\ newrels?) \cup \{pid?\} \land \\
\quad\quad (tablePerson\ pid).reltoHouseholder = Spouse\} \leq 1 \\
\hline
\end{array}
$$

Note that this precondition is equivalent to saying that the number of the persons whose *reltoHouseholder* become *Spouse* is at most one.

The checklist becomes:

List of Proof Obligatious: on *RepresentConstraints*, -already OK
on *IsMemberofConstraints*, -already OK
on *RuleofMonogamy*, -not yet

Note that adding some global precondition schemas does not spoil any proof obligations already proved so far.

The last obligation is:

$$\begin{array}{l} \mathit{RuleofMonogamy} \wedge \mathit{Delete\&ModifyPerson} \wedge \\ \quad \mathit{DeleteIsMemberofLink} \wedge \Xi \mathit{Family} \wedge \mathit{LimitonNewSpouse} \\ \qquad \vdash \mathit{RuleofMonogamy}' \end{array}$$

and it becomes OK.

Finally, we have the whole process as follows:

$$\begin{array}{l} \mathit{HouseholderMoveOut} = \\ \qquad \mathit{Delete\&ModifyPerson} \wedge \mathit{ModifyHouseholder} \wedge \\ \qquad \mathit{DeleteIsMemberofLink} \wedge \mathit{LimitonNewSpouse} \end{array}$$

which has been proved to preserve all constraints in DB.

Furthermore, the total precondition of this process can be shown by collecting all the preconditions obtained so far as follows:

the precondition of *HouseholderMoveOut*

$=$

$$
\begin{array}{|l}
\hline
\mathit{HouseholderTable} \\
\mathit{PersonTable} \\
\mathit{RepresentRelationship} \\
\mathit{IsMemberofRelationship} \\
\mathit{hid?} : \mathit{HouseholderId} \\
\mathit{pid?} : \mathit{PersonId} \\
\mathit{newrels?} : \mathit{PersonId} \twoheadrightarrow \mathit{RelType} \\
\hline
\mathit{hid?} \in \mathit{knownHouseholder} \\
\mathit{pid?} \in \mathit{knownPerson} \\
\mathrm{dom}\ \mathit{newrels?} \subseteq \mathit{knownPerson} \\
\mathit{pid?} \notin \mathrm{dom}\ \mathit{newrels?} \\
\mathit{pid?} \in \mathrm{dom}\ \mathit{IsMemberof} \\
\exists\, \mathit{fid} : \mathit{FamilyId} \mid \mathit{hid?} \mapsto \mathit{fid} \in \mathit{Represent} \bullet \\
\quad \mathit{pid?} \mapsto \mathit{fid} \in \mathit{IsMemberof} \land \\
\quad (\forall\, \mathit{pid} : \mathit{PersonId} \mid \mathit{pid} \in \mathrm{dom}\ \mathit{newrels?} \bullet \\
\qquad\qquad \mathit{pid} \mapsto \mathit{fid} \in \mathit{IsMemberof}) \\
\exists\, \mathit{fid} : \mathit{FamilyId} \mid \mathit{hid?} \mapsto \mathit{fid} \in \mathit{Represent} \bullet \\
\quad \#\{\mathit{pid} : \mathit{PersonId};\ \mathit{rel} : \mathit{RelType} \mid \\
\qquad \mathit{pid} \mapsto \mathit{rel} \in \mathit{newrels?} \land \mathit{rel} = \mathit{Spouse} \bullet \mathit{pid}\} + \\
\quad \#\{\mathit{pid} : \mathit{PersonId} \mid \\
\qquad \mathit{pid} \in \mathit{knownPerson} \land \\
\qquad \mathit{pid} \mapsto \mathit{fid} \in \mathit{IsMemberof} \land \\
\qquad \mathit{pid} \notin (\mathrm{dom}\ \mathit{newrels?}) \cup \{\mathit{pid?}\} \land \\
\qquad (\mathit{tablePerson}\ \mathit{pid}).\mathit{reltoHouseholder} = \mathit{Spouse}\} \leq 1 \\
\hline
\end{array}
$$

We can proceed similarly also in **Case 2**.

Chapter 5

Support Tool

In this chapter, a computer tool supporting our activity of specifying business processes is introduced. Firstly, several requirements for such a tool are discussed, and secondly, an informal specification of its prototype is proposed. Finally, an example of execution of the prototype is introduced. The source code of the prototype in the functional programming language Orwell [Wadler & Miller 90, Bird & Wadler 88] is shown in Appendix B.

5.1 Requirements

Our specification activity consists of the following steps:

Step 1 We draw an ER diagram to illustrate the ER data model.

Step 2 Based on the ER diagram, we specify the following objects in the database:

- all entities (with attributes and constraints).
- all relationships (with constraints).
- all DB integrity constraints (with subject areas and predicates).

Step 3 For each business process to be considered, we specify:

- input data and output data.
- basic operations on entities (with update-type, target occurrences, preconditions and how to determine the new values of all the attributes).

- basic operations on relationships (with update-type, target occurrences, preconditions and new occurrences to which links must be made).
- global preconditions (with scopes and predicates).

Step 4 For each business process, we check the correctness of the specification according to the proof obligations explained in Chapter 4, i.e.

- checking each basic operation on an entity (with regard to the attribute constraints and record constraints, if any).
- checking each suhprocess on an entity, i.e. the collection of all the basic operations on the entity (with regard to the entity table constraints, if any).
- checking each subprocess on a relationship, i.e. the collection of all the basic operations on the relationship (with regard to the relationship constraints).

If some constraint is found not to be preserved, the operation can be modified by adding appropriate preconditions.
Furthermore,

- checking the other relationship constraints which could be violated.
- checking each DB integrity constraint which could be violated.

If some constraint is found not to be preserved, the process can be strengthened hy adding appropriate global precondition schemas or some missing subprocesses may be found.

Step 5 Once the correctness of the process is proved, the whole process and precondition of the process should be presented iu a suitable form.

As for **Step 1**, many kinds of CASE tools are available, such as IEF TM[Texas Instruments 88]. They provide us with a nice graphical interface not only for drawing the ER diagrams but also for many other data input and tool operations.

In **Step 2** and **Step 3**, our tool should support such a nice user interface as the CASE tools have, or provide some file interface to input a specification file prepared outside the tool (cf. Appendix A).

For **Step 4** and **Step 5**, our tool should have the following facilities:

- general Z schema manipulations: Display, Expand, etc.

- translation of the process specification based on ER model into Z schemas.
- assistance for checking the proof obligations, such as automatic reduction.
- interactive modification of the process specification.
- keeping track of both the list of proof obligations and the modification of the specification during **Step 4**.
- file interface to output a specification file in an appropriate form.

Note that it seems reasonable to allow the user to judge whether or not each proof obligation is OK. This is because we are often confronted with mistakes or flaws in our specification and it is difficult for the tool to amend them alone. Therefore, some kind of rigid theorem prover, such as zedB [Neilson & Prasad 91], is not actually required.

5.2 Specification

The prototype of our tool has the following functions:

1. reading a specification file
   - command line: `readspec` *filename*
   - parameters: *filename* is the name of a specification file to be read.
   - function: The information on a business process is inputted from a text file written in the form described in Appendix A.
2. listing names of all schemas
   - command line: **`schemalist`**
   - parameters: none
   - function: The list of names of all schemas the tool has created so far is displayed with sequential numbers in the following form:

0. (*name of schema*)
1. (*name of schema*)
...
(*number*). (*name of schema*)

For example,

0. CustomerRecord
1. AccountRecord
...

3. showing the definition of schemas

   - command line: `view` *par*
   - parameters: *par* must be an integer indicating a certain schema in the list shown by the command `schemalist`.
   - function: The definition of the schema designated by the number *par* is displayed in its vertical form as follows:

     ⟨*name of schema*⟩
     =
     ⟨*definition of schema in vertical form*⟩

4. showing the expanded form of schemas

   - command line: `view2` *par*
   - parameters: *par* must be an integer indicating a certain schema in the list shown by the command `schemalist`.
   - function: The expanded schema of the schema designated by the number *par* is displayed in its vertical form as follows:

     ⟨*name of schema*⟩
     =
     ⟨*expanded schema in vertical form*⟩

5. listing the proof obligations

   - command line: `checklist`
   - parameters: none
   - function: The list of the proof obligations is displayed with sequential numbers and flags indicating whether each obligation is already checked or not as follows:

     0. on ⟨*name of constraints*⟩ [by ⟨*name of operation*⟩] -⟨*okflag*⟩
     1. on ⟨*name of constraints*⟩ [by ⟨*name of operation*⟩] -⟨*okflag*⟩
     ...
     ⟨*number*⟩. on ⟨*name of constraints*⟩ [by ⟨*name of operation*⟩] -⟨*okflag*⟩

For example,

0. on AccountRecordConstraints by CreateAccountRecord -already OK
1. on DepositAccountIsSubtypeofAccount -not yet

6. expanding proof obligations

   - command line: `expand` *par*
   - parameters: *par* must be an integer indicating a certain proof obligation in the list shown by the command `checklist`.
   - function: The definition of the proof obligation designated by the number *par* and the expanded schemas of its both sides are displayed as follows:

     (definition of proof obligation)

     LHS
     =
     (expanded schema of left hand side in vertical form)

     RHS
     =
     (expanded schema of right hand side in vertical form)

7. reducing proof obligations

   - command line: `reduce` *par*
   - parameters: *par* must be an integer indicating a certain proof obligation in the list shown by the command `checklist`.
   - function: The list of the declarations and predicates in the expanded RHS which are not contained in the expanded LHS of the proof obligation designated by the number *par* is displayed with sequential numbers in the following form:

     Please check the following declarations and predicates in RHS:
     0. *(declaration)*
     ...
     (number). *(declaration)*
     (number). *(predicate)*
     ...
     (number). *(predicate)*

If the list is empty, only the following message will appear:

OK, this proof obligation is satisfied.

8. ticking proof obligations

   - command line: **checkok** *par*
   - parameters: *par* must be an integer indicating a certain proof obligation in the list shown by the command `checklist`.
   - function: The proof obligation designated by the number *par* is recorded as already checked.

5.3 Example of Execution

Here is a series of interaction of the prototype in the banking example. The specification of the system is written in the text file `banking.spec` as follows:

```
Banking System

DBName: BankDB

Entities:

Customer
        Attributes:
                name : Name
                vip  : YesNo

Account
        Attributes:
                balance : Z

CurrentAccount
        Attributes:
                balance : Z

DepositAccount
        Attributes:
                balance : Z
                        AttributeConstraints:
                        (0)     balance > 0
                depositDate : Date
```

```
                interest : InterestType
        RecordConstraints:
        (0)     P(depositDate, interest)
        TableConstraints:
        (0)     # knownDepositAccount <= MaxDepositAccount

Relationships:

IsOwnedBy : Account -+> Customer
        Type:   OptionalManytoOne
        Constraints:
        (0)     #(knownCustomer \ ran IsOwnedBy) <= 100

CurrentAccountIsAccount : CurrentAccount -+> Account
        Type:   OptionalOnetoOne

DepositAccountIsAccount : DepositAccount -+> Account
        Type:   OptionalOnetoOne

DBIntegrityConstraints:

CurrentAccountIsSubtypeofAccount :
        ScopeE: CurrentAccount, Account
        ScopeR: CurrentAccountIsAccount
        Constraints:
        (0)     tableCurrentAccount ;
                (lambda CurrentAccountRecord @ theta AccountRecord)
                        = CurrentAccountIsAccount ; tableAccount

DepositAccountIsSubtypeofAccount :
        ScopeE: DepositAccount, Account
        ScopeR: DepositAccountIsAccount
        Constraints:
        (0)     tableDepositAccount ;
                (lambda DepositAccountRecord @ theta AccountRecord)
                        = DepositAccountIsAccount ; tableAccount

RuleofVip :
        ScopeE: Customer, CurrentAccount
        ScopeR: IsOwnedBy, CurrentAccountIsAccount
        Constraints:
        (0)     forall c:CustomerId; a:CurrentAccountId |
                        c in knownCustomer /\
                        a in knownCurrentAccount /\
                        a |-> c in CurrentAccountIsAccount ; IsOwnedBy /\
                        (tableCustomer c).vip = yes @
                        (tableCurrentAccount a).balance >= -100

RuleofNonVip :
        ScopeE: Customer, CurrentAccount
```

```
        ScopeR: IsOwnedBy, CurrentAccountIsAccount
        Constraints:
        (0)     forall c:CustomerId; a:CurrentAccountId |
                        c in knownCustomer /\
                        a in knownCurrentAccount /\
                        a |-> c in CurrentAccountIsAccount ; IsOwnedBy /\
                        (tableCustomer c).vip = no @
                        (tableCurrentAccount a).balance >= -10

Process:

Withdraw

InputData:
        a? : CurrentAccountId
        amount? : Z

SubprocessesonEntities:

WithdrawonCurrentAccount on CurrentAccount

        InputData:
                a? : CurrentAccountId
                amount? : Z

        Preconditions:
        (0)     a? in knownCurrentAccount

        BasicOperations:

        Modify CurrentAccount
                Target: a?
                RecordOperation: WithdrawonCurrentAccountRecord
                InputData:
                        a? : CurrentAccountId
                Preconditions:
                (0)     a? in knownCurrentAccount
                AtomicOperations:
                balance by WithdrawonbalanceofCurrentAccount
                        InputData: amount? : Z
                        Operation:
                        (0)     balance' = balance - amount?
```

In this execution, the process Withdraw is going to be checked.

```
? run tool

readspec filename/schemalist/view num/view2 num/
```

```
checklist/expand num/reduce num/checkok num/end => readspec banking.spec

readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => schemalist

0. nameofCustomer
1. vipofCustomer
2. CustomerRecord
3. CustomerTable
4. balanceofAccount
5. AccountRecord
6. AccountTable
7. balanceofCurrentAccount
8. CurrentAccountRecord
9. CurrentAccountTable
10. balanceofDepositAccount
11. balanceofDepositAccountConstraints
12. depositDateofDepositAccount
13. interestofDepositAccount
14. DepositAccountRecord
15. DepositAccountRecordConstraints
16. DepositAccountTable
17. DepositAccountTableConstraints
18. IsOwnedByRelationship
19. IsOwnedByConstraints
20. CurrentAccountIsAccountRelstionship
21. CurrentAccountIsAccountConstraints
22. DepositAccountIsAccountRelationship
23. DepositAccountIsAccountConstraints
24. CurrentAccountIsSubtypeofAccount
25. DepositAccountIsSubtypeofAccount
26. RuleofVip
27. RuleofNonVip
28. BankDB
29. WithdrawonbalanceofCurrentAccount
30. WithdrawonCurrentAccountRecord
31. WithdrawonCurrentAccount

readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => view 31

WithdrawonCurrentAccount
=
 -----------------------------------------------------------
| Delta CurrentAccountTable
| a? : CurrentAccountId
| amount? : Z
|----------------
| a? in knownCurrentAccount
| knownCurrentAccount' = knownCurrentAccount
```

```
| exists Delta CurrentAccountRecord |
|       a? |-> theta CurrentAccountRecord in tableCurrentAccount /\
|       a? |-> theta CurrentAccountRecord' in tableCurrentAccount' @
|       WithdrawonCurrentAccountRecord
| { a? } <<| tableCurrentAccount' = { a? } <<| tableCurrentAccount
 -----------------------------------------------------------

readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => view2 31

WithdrawonCurrentAccount
=
 -----------------------------------------------------------
| knownCurrentAccount : F CurrentAccountId
| tableCurrentAccount : CurrentAccountId -+> CurrentAccountRecord
| knownCurrentAccount' : F CurrentAccountId
| tableCurrentAccount' : CurrentAccountId -+> CurrentAccountRecord
| a? : CurrentAccountId
| amount? : Z
|---------------
| knownCurrentAccount = dom tableCurrentAccount
| knownCurrentAccount' = dom tableCurrentAccount'
| a? in knownCurrentAccount
| knownCurrentAccount' = knownCurrentAccount
| exists Delta CurrentAccountRecord |
|       a? |-> theta CurrentAccountRecord in tableCurrentAccount /\
|       a? |-> theta CurrentAccountRecord' in tableCurrentAccount' @
|       WithdrawonCurrentAccountRecord
| { a? } <<| tableCurrentAccount' = { a? } <<| tableCurrentAccount
 -----------------------------------------------------------

readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => checklist

0. on CurrentAccountIsAccountConstraints        -not yet
1. on CurrentAccountIsSubtypeofAccount  -not yet
2. on RuleofVip -not yet
3. on RuleofNonVip      -not yet
```

Here we learn that these four are the constraints which could be violated by the process Withdraw. The first proof obligation is going to be expanded and checked.

```
readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => expand 0

CurrentAccountIsAccountConstraints /\
Xi CurrentAccountIsAccountRelationship /\
```

```
WithdrawonCurrentAccount /\
Xi AccountTable
        |- CurrentAccountIsAccountConstraints'

LHS
=
 -------------------------------------------------------
| CurrentAccountIsAccount : CurrentAccountId -+> AccountId
| knownCurrentAccount : F CurrentAccountId
| knownAccount : F AccountId
| CurrentAccountIsAccount' : CurrentAccountId -+> AccountId
| tableCurrentAccount : CurrentAccountId -+> CurrentAccountRecord
| knownCurrentAccount' : F CurrentAccountId
| tableCurrentAccount' : CurrentAccountId -+> CurrentAccountRecord
| a? : CurrentAccountId
| amount? : Z
| tableAccount : AccountId -+> AccountRecord
| knownAccount' : F AccountId
| tableAccount' : AccountId -+> AccountRecord
|---------------
| dom CurrentAccountIsAccount = knownCurrentAccount
| ran CurrentAccountIsAccount subseteq knownAccount
| CurrentAccountIsAccount in CurrentAccountId >+> AccountId
| knownCurrentAccount = dom tableCurrentAccount
| knownCurrentAccount' = dom tableCurrentAccount'
| a? in knownCurrentAccount
| knownCurrentAccount' = knownCurrentAccount
| exists Delta CurrentAccountRecord |
|       a? |-> theta CurrentAccountRecord in tableCurrentAccount /\
|       a? |-> theta CurrentAccountRecord' in tableCurrentAccount' @
|       WithdrawonCurrentAccountRecord
| { a? } <<| tableCurrentAccount' = { a? } <<| tableCurrentAccount
| knownAccount = dom tableAccount
| knownAccount' = dom tableAccount'
| CurrentAccountIsAccount' = CurrentAccountIsAccount
| knownAccount' = knownAccount
| tableAccount' = tableAccount
 -------------------------------------------------------

RHS
=
 -------------------------------------------------------
| CurrentAccountIsAccount' : CurrentAccountId -+> AccountId
| knownCurrentAccount' : F CurrentAccountId
| knownAccount' : F AccountId
|---------------
| dom CurrentAccountIsAccount' = knownCurrentAccount'
| ran CurrentAccountIsAccount' subseteq knownAccount'
| CurrentAccountIsAccount' in CurrentAccountId >+> AccountId
 -------------------------------------------------------
```

```
readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => reduce 0

Please check the following declarations and predicates in RHS:

0. dom CurrentAccountIsAccount' = knownCurrentAccount'
1. ran CurrentAccountIsAccount' subseteq knownAccount'
2. CurrentAccountIsAccount' in CurrentAccountId >+> AccountId
```

Since all of these three predicates can be deduced from LHS here, the obligation is OK.

```
readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => checkok 0

readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => checklist

0. on CurrentAccountIsAccountConstraints        -already OK
1. on CurrentAccountIsSubtypeofAccount  -not yet
2. on RuleofVip -not yet
3. on RuleofNonVip      -not yet
```

The next proof obligation is going to be checked.

```
readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => expand 1

CurrentAccountIsSubtypeofAccount /\
WithdrawonCurrentAccount /\
Xi AccountTable /\
Xi CurrentAccountIsAccountRelationship
        |- CurrentAccountIsSubtypeofAccount'

LHS
=
 ---------------------------------------------------------
| knownCurrentAccount : F CurrentAccountId
| tableCurrentAccount : CurrentAccountId -+> CurrentAccountRecord
| knownAccount : F AccountId
| tableAccount : AccountId -+> AccountRecord
| CurrentAccountIsAccount : CurrentAccountId -+> AccountId
| knownCurrentAccount' : F CurrentAccountId
| tableCurrentAccount' : CurrentAccountId -+> CurrentAccountRecord
| a? : CurrentAccountId
```

```
| amount? : Z
| knownAccount' : F AccountId
| tableAccount' : AccountId -+> AccountRecord
| CurrentAccountIsAccount' : CurrentAccountId -+> AccountId
|---------------
| knownCurrentAccount = dom tableCurrentAccount
| knownAccount = dom tableAccount
| tableCurrentAccount ; (lambda CurrentAccountRecord @
|        theta AccountRecord) = CurrentAccountIsAccount ; tableAccount
| knownCurrentAccount' = dom tableCurrentAccount'
| a? in knownCurrentAccount
| knownCurrentAccount' = knownCurrentAccount
| exists Delta CurrentAccountRecord |
|        a? |-> theta CurrentAccountRecord in tableCurrentAccount /\
|        a? |-> theta CurrentAccountRecord' in tableCurrentAccount' @
|        WithdrawonCurrentAccountRecord
| { a? } <<| tableCurrentAccount' = { a? } <<| tableCurrentAccount
| knownAccount' = dom tableAccount'
| knownAccount' = knownAccount
| tableAccount' = tableAccount
| CurrentAccountIsAccount' = CurrentAccountIsAccount
 ---------------------------------------------------------

RHS
=
 ---------------------------------------------------------
| knownCurrentAccount' : F CurrentAccountId
| tableCurrentAccount' : CurrentAccountId -+> CurrentAccountRecord
| knownAccount' : F AccountId
| tableAccount' : AccountId -+> AccountRecord
| CurrentAccountIsAccount' : CurrentAccountId -+> AccountId
|---------------
| knownCurrentAccount' = dom tableCurrentAccount'
| knownAccount' = dom tableAccount'
| tableCurrentAccount' ; (lambda CurrentAccountRecord @
|        theta AccountRecord) = CurrentAccountIsAccount' ; tableAccount'
 ---------------------------------------------------------

readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => reduce 1

Please check the following declarations and predicates in RHS:

0. tableCurrentAccount' ; (lambda CurrentAccountRecord @
        theta AccountRecord) = CurrentAccountIsAccount' ; tableAccount'
```

Note that the first two predicates in RHS have been eliminated by the command `reduce` since they appear in LHS just as they are.

In order to make the last one true, however, we find that another subpro-

cess (on Account) is needed. The following specification of the subprocess WithdrawonAccount is added to **banking.spec** with 'AlreadyChecked' data and we have the new file **banking2.spec**:

```
WithdrawonAccount on Account

        ScopeE: CurrentAccount
        ScopeR: CurrentAccountIsAccount
        InputData:
                a? : CurrentAccountId
                amount? : Z

        Preconditions:
        (0)     a? in knownCurrentAccount

        BasicOperations:

        Modify Account
                Target: (CurrentAccountIsAccount a?)
                RecordOperation: WithdrawonAccountRecord
                ScopeE: CurrentAccount
                ScopeR: CurrentAccountIsAccount
                InputData:
                        a? : CurrentAccountId
                Preconditions:
                (0)     a? in knownCurrentAccount
                AtomicOperations:
                balance by WithdrawonbalanceofAccount
                        InputData: amount? : Z
                        Operation:
                        (0)     balance' = balance - amount?

AlreadyChecked:
        on CurrentAccountIsAccountConstraints
```

The new specification is checked as follows:

```
readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => readspec banking2.spec

readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => checklist

0. on IsOwnedByConstraints      -not yet
1. on CurrentAccountIsAccountConstraints        -already OK
2. on DepositAccountIsAccountConstraints        -not yet
3. on CurrentAccountIsSubtypeofAccount  -not yet
```

```
4. on DepositAccountIsSubtypeofAccount  -not yet
5. on RuleofVip -not yet
6. on RuleofNonVip      -not yet
```

Note that two additional obligations have appeared because of the new subprocess on Account.

```
readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => expand 3

CurrentAccountIsSubtypeofAccount /\
WithdrawonAccount /\
WithdrawonCurrentAccount /\
Xi CurrentAccountIsAccountRelationship
        |- CurrentAccountIsSubtypeofAccount'

LHS
=
 ---------------------------------------------------------
| knownAccount : F AccountId
| tableAccount : AccountId -+> AccountRecord
| knownCurrentAccount : F CurrentAccountId
| tableCurrentAccount : CurrentAccountId -+> CurrentAccountRecord
| CurrentAccountIsAccount : CurrentAccountId -+> AccountId
| knownAccount' : F AccountId
| tableAccount' : AccountId -+> AccountRecord
| a? : CurrentAccountId
| amount? : Z
| knownCurrentAccount' : F CurrentAccountId
| tableCurrentAccount' : CurrentAccountId -+> CurrentAccountRecord
| CurrentAccountIsAccount' : CurrentAccountId -+> AccountId
|---------------
| knownAccount = dom tableAccount
| knownCurrentAccount = dom tableCurrentAccount
| tableCurrentAccount ; (lambda CurrentAccountRecord @
|       theta AccountRecord) = CurrentAccountIsAccount ; tableAccount
| knownAccount' = dom tableAccount'
| a? in knownCurrentAccount
| knownAccount' = knownAccount
| exists Delta AccountRecord |
|       (CurrentAccountIsAccount a?) |-> theta AccountRecord in tableAccount /\
|       (CurrentAccountIsAccount a?) |-> theta AccountRecord' in tableAccount' @
|       WithdrawonAccountRecord
| { (CurrentAccountIsAccount a?) } <<| tableAccount' = { (CurrentAccountIsAccoun
t a?) } <<| tableAccount
| knownCurrentAccount' = dom tableCurrentAccount'
| knownCurrentAccount' = knownCurrentAccount
| exists Delta CurrentAccountRecord |
```

```
|       a? |-> theta CurrentAccountRecord in tableCurrentAccount /\
|       a? |-> theta CurrentAccountRecord' in tableCurrentAccount' @
|       WithdrawonCurrentAccountRecord
| { a? } <<| tableCurrentAccount' = { a? } <<| tableCurrentAccount
| CurrentAccountIsAccount' = CurrentAccountIsAccount
 -----------------------------------------------------------

RHS
=
 -----------------------------------------------------------
| knownAccount' : F AccountId
| tableAccount' : AccountId -+> AccountRecord
| knownCurrentAccount' : F CurrentAccountId
| tableCurrentAccount' : CurrentAccountId -+> CurrentAccountRecord
| CurrentAccountIsAccount' : CurrentAccountId -+> AccountId
|---------------
| knownAccount' = dom tableAccount'
| knownCurrentAccount' = dom tableCurrentAccount'
| tableCurrentAccount' ; (lambda CurrentAccountRecord @
|       theta AccountRecord) = CurrentAccountIsAccount' ; tableAccount'
 -----------------------------------------------------------

readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => reduce 3

Please check the following declarations and predicates in RHS:

0. tableCurrentAccount' ; (lambda CurrentAccountRecord @
        theta AccountRecord) = CurrentAccountIsAccount' ; tableAccount'
```

Now we can deduce this from LHS and tick the obligation No.3.

```
readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => checkok 3
```

Similarly, we can proceed towards the correct specification of Withdraw by checking the remaining proof obligations, although we stop here.

```
readspec filename/schemalist/view num/view2 num/
checklist/expand num/reduce num/checkok num/end => end


(125.28 + 2.20 CPU seconds, 183876 reductions, 916444 cells)

?
```

Chapter 6

Conclusion

In this chapter, after summarizing the result of our work in the area of business process specification, some remaining problems and several directions for future work are proposed.

6.1 Summary

Business processes can be specified in Z based on the structure of an Entity-Relationship (ER) data model. The purpose is to make the specification not only formal enough to reason rigorously about, but also declarative and so simpler to understand.

The interdependency among three things (the database integrity constraints, the business process and its precondition) has been explored. One approach to making the interdependency clear is to obtain the precondition of the process from the other two by schema calculation in Z. This seems quite simple and attractive in explaining the interdependency. Unfortunately, however, some weakness attached to the precondition calculation have been recognized (See Section 4.1). We take another approach in which the precondition is dealt with in a VDM-like manner, i.e. a precondition which has been found is written explicitly in the process specification so as to fulfil a proof obligation.

As a consequence, a much better way of doing the specification activity has been found and a support tool for the activity has been desigued.

More precisely, the results of our work are as follows:

- A specification method for business processes has been proposed, in which a business process is specified as a collection of basic operations of three types (i.e. Modify, Insert or Delete) on entity records or relationship links in an ER data model.

- A method for translating the process specification into Z has been proposed, which makes the specification more formal and uncovers some hidden consequences.
- A method for ensuring the correctness of the specification has been proposed, in which we can proceed towards the correct specification finding preconditions by checking a series of proof obligations in a bottom-up manner.
- A support tool for the specification activity has been proposed, a prototype of which has been developed.

6.2 Remaining Problems

6.2.1 HCI Consideration

So far we have ignored the existence of Human Computer Interaction (HCI) in the business processes. However, even in the business application domain, it is likely to be necessary for us to specify the HCI in an appropriate manner.

For example, the process HouseholderMoveOut in the city office system in Section 4.8 is specified to get a *PersonId* as input data because the new householder must be specified. However, we seldom input such an ID number directly partly because it is troublesome and partly because we do not (and need not) know the ID number in some cases. Instead, it is more likely that firstly the system would show the list of persons in his/her family, and then we would choose one person from the list by inputting the associated number or clicking on a mouse.

One idea in taking the HCI into account is to separate two concerns: the declarative specification and some specification for the dynamic behaviour of the system including HCI. We can regard the declarative specification as a total set of requirements and break down the set into some subsets, each of which specifies the effect of operation during each interaction cycle.

In order to describe the dynamic behaviour, we need some appropriate specification tool besides Z.

6.2.2 Output Consideration

We have also ignored the output processes of business systems. To specify the output processes, we must investigate the following problems:

- Periodical and/or statistical processing
- Triggers for such processes

- Requirements on some sequential order of output data

etc.

6.3 Future Works

6.3.1 Code Generation

If our specification method is both powerful enough to express most kinds of user's requirement and formal enough to manipulate mechanically, we might also wish for program code to be generated from its specification.

Moreover, since the precondition of a business process can easily be obtained by our method as explained in Chapter 4, it seems more feasible to generate a code, of the form:

IF (Precondition) THEN (AllOperations) ELSE (ErrorHandling)

In fact, however, there are lots of implementation details which we have to decide in implementing business systems. Therefore, it is important to identify which part of the system can be generated and which part cannot.

One interesting problem in code generation is how to serialize the subprocesses of a certain process.

As we have seen so far, a business process is specified as a set of several parallel subprocesses. When these subprocesses can be executed in parallel, there might be no problem. However, in the case when we try to implement them in some procedural language, the problem of serializing arises.

Actnally, there is a possibility that one subprocess refers to many records in a database while another subprocess happens to refer to almost the same set of records. In such a case, it is obviously better to mix these two subprocesses so that they can share the data, the procedure to search the data and the time in searching. In other words, there is a lot of room for optimization in serializing subprocesses.

Fortunately, an algorithm to compile nonprocedural specifications into procedural programs is known [Prywes & Pnueli 83]. We can hope that the same algorithm can be applied also to the present problem with slight modification.

One thing that we must notice is the fact that in our specification of subprocesses in Z the declaration part of schemas may contain sufficient information for a compiler or a human to analyze the optimization of the database accesses without considering the predicate part of the schemas.

6.3.2 Animation of Specification

Even if code generation for a real system might be too difficult a challenge, it is perfectly possible to generate the source code for an animation tool by which the user can check whether the specification is desirable or not if our specification method is powerful enough to express any requirement for such an animation tool. In other words, we may have a tool generator.

One interesting theme here is to find whether it is possible or not to specify the tool generator itself in our specification method. If it is possible, the tool generator can generate itself. Furthermore, we may expect the tool generator to generate even a better one than itself!

Bibliography

[Bird & Wadler 88] R. Bird & P. Wadler, *Introduction to Functional Programming*, Prentice-Hall, 1988.

[Chen 76] P. P. Chen, *The Entity-Relationship Model Toward a Unified View of Data*. ACM Transactions on Database Systems, Vol.1, No.1, pp.9-36, 1976.

[Codd 90] E. F. Codd, *The Relational Model for Database Management – Version 2*, Addison-Wesley, 1990.

[Ginbayashi & Hashimoto 91] J. Ginbayashi & K. Hashimoto, *Software Specification in Business Terminology*, Proceedings of the Fifteenth Annual International Compnter Software & Application Conference COMPSAC 91, pp.161-168, 1991.

[Jones 86] C. B. Jones, *Systematic Software Development Using VDM*, Prentice-Hall, 1986.

[Josephs & Redmond-Pyle 91a] M. B. Josephs & D. Redmond-Pyle, *Entity-Relationship Models Expressed in Z: A Synthesis of Structured and Formal Methods*, Oxford University Programming Research Group Technical Report PRG-TR-20-91, 1991.

[Josephs & Redmond-Pyle 91b] M. B. Josephs & D. Redmond-Pyle, *A Library of Z Schemas for use in Entity-Relationship Modelling*, Oxford University Programming Research Group Technical Report PRG-TR-21-91, 1991.

[Karal 91] B. Karal, *Integrating Struetured and Formal Methods: A Case Study*, Oxford University Programming Research Group MSc Dissertation, 1991.

[Napier 91] R. Napier, *Information Engineering & Application Development Using Knowledgeware ®'s Case Tool Set*, Prentice-Hall, 1991.

[Neilson & Prasad 91] D. Neilson & D. Prasad, *zedB: A Proof tool for Z built on B*, Proceedings of the Sixth Annual Z User Meeting, 1991.

[Nijssen & Halpin 89] G. M. Nijssen & T. A. Halpin, *Conceptual Schema and Relational Database Design: A fact oriented approach*, Prentice-Hall, 1989.

[Prywes & Pnueli 83] N. S. Prywes & A. Pnueli, *Compilation of Nonprocedural Specifications into Computer Programs*, IEEE Transactions on Software Engineering, Vol.SE-9, No.3, pp.267-279, 1983.

[Sanders & Short 92] P. Sanders & K. Short, *Declarative Analysis in Information Engineering*, CASE – Current Practice Future Prospects. K. Spurr & P. Layzell (Eds.), Wiley, 1992.

[Spivey 92] J. M. Spivey, *The Z Notation: A Reference Manual. Second Edition*, Prentice-Hall, 1992.

[Texas Instruments 88] Texas Instruments, *A Guide to Information Engineering Using the IEF* ™, *Second Edition*, Texas Instruments, 1988.

[Wadler & Miller 90] P. Wadler & Q. Miller, *An Introduction to Orwell 6*, Notes in Oxford University Computing Laboratory, 1990.

[Woodcock 92] J. Woodcock, *Chapter 2: Promotion, Chapter 3: Precondition, Chapter 4: Constructing Specifications*, MSc lecture notes on Systems Specification & Development, Oxford University Computing Laboratory, 1992.

Appendix A

Syntax of Specification Files

The notation used to describe the syntax is as follows:

nonterminal	nonterminal
"`literal`"	literal
[*pattern*]	optional
⟦*pattern*⟧	zero or more repetitions

Syntax

In the syntax below, any specific literal, *word*, *id*, *item*, or *comma_list* must be separated from others by one or more blank spaces. A blank space is a space, tab or newline character.

spec → *comment* "`DBName:`" *db_spec*
  "`Process:`" *pr_spec*

comment → ⟦*word*⟧

db_spec → *dbname* "`Entities:`" *entities*
  ["`Relationships:`" *relationships*]
  ["`DBIntegrityConstraints:`" *integrities*]

dbname → *id*

entities → *entity* ⟦*entity*⟧

entity → *entity_name* "`Attributes:`" *attributes*
  ["`RecordConstraints:`" *record_constraints*]
  ["`TableConstraints:`" *table_constraints*]

entity_name → *id*

attributes → *attribute* ⟦*attribute*⟧

```
attribute              → attribute_name ":" attribute_type
                         ["AttributeConstraints:" attribute_constraints]
attribute_name         → id
attribute_type         → expression
attribute_constraints  → predicates
record_constraints     → predicates
table_constraints      → predicates
predicates             → predicate [predicate]
predicate              → item expression
expression             → word [word]

relationships          → relationship [relationship]
relationship           → relationship_name ":" detail_name "-+>" master_name
                         "Type:" correspondence_type
                         ["Constraints:" constraints]
relationship_name      → id
detail_name            → id
master_name            → id
correspondence_type    → "OnetoOne" | "OnetoOptionalOne" |
                         "OptionalOnetoOne" | "OptionalOnetoOptionalOne" |
                         "ManytoOne" | "ManytoOptionalOne" |
                         "OptionalManytoOne" | "OptionalManytoOptionalOne"

integrities            → integrity [integrity]
integrity              → integrity_name ":"
                         ["ScopeE:" entity_name_list]
                         ["ScopeR:" relationship_name_list]
                         "Constraints:" constraints
integrity_name         → id
entity_name_list       → comma_list [comma_list]
relationship_name_list → comma_list [comma_list]

pr_spec         → prname
                  ["InputData:" input_data_list]
                  ["SubprocessesonEntities:" subprocessesonE]
                  ["SubprocessesonRelationships:" subprocessesonR]
                  ["GlobalPreconditions:" global_preconditions]
                  ["AlreadyChecked:" already_checked]
```

prname → *id*

input_data_list → *input_data* [*input_data*]

input_data → *variable_name* ":" *variable_type*

variable_name → *id*

variable_type → *expression*

subprocessesonE → *subprocessonE* [*subprocessonE*]

subprocessonE → *subprocess_name* "on" *entity_name*
["ScopeE:" *entity_name_list*]
["ScopeR:" *relationship_name_list*]
["InputData:" *input_data_list*]
["Preconditions:" *preconditions*]
"BasicOperations:" *basic_ops_onE*

subprocess_name → *id*

preconditions → *predicates*

basic_ops_onE → *basic_op_onE* [*basic_op_onE*]

basic_op_onE → *update_type* *entity_name*
[*interface*]
"Target:" *target_record*
["RecordOperation:" *record_op*]

update_type → "Modify" | "Insert" | "Delete"

interface → *expression*

target_record → *expression*

record_op → *record_op_name*
["ScopeE:" *entity_name_list*]
["ScopeR:" *relationship_name_list*]
["InputData:" *input_data_list*]
["Preconditions:" *preconditions*]
"AtomicOperations:" *atomic_ops_onE*

record_op_name → *id*

atomic_ops_onE → *atomic_op_onE* [*atomic_op_onE*]

atomic_op_onE → *attribute_name* "by" *atomic_op_name*
["ScopeE:" *entity_name_list*]
["ScopeR:" *relationship_name_list*]
["InputData:" *input_data_list*]
["Preconditions:" *preconditions*]
"Operation:" *predicates*

atomic_op_name → *id*

```
subprocessesonR → subprocessonR {subprocessonR}

subprocessonR   →  snbprocess_name "on" relationship_name
                   ["ScopeE:" entity_name_list]
                   ["ScopeR:" relationship_name_list]
                   ["InputData:" input_data_list]
                   ["Preconditions:" preconditions]
                   "BasicOperations:" basic_ops_onR

basic_ops_onR   → basic_op_onR {basic_op_onR}

basic_op_onR    →  update_type relationship_name
                   "DataforUpdate:" data_for_update

data_for_update → expression

global_preconditions → global_precondition {global_precondition}

global_precondition  →  precondition_name "::"
                        ["ScopeE:" entity_name_list]
                        ["ScopeR:" relationship_name_list]
                        ["InputData:" input_data_list]
                        "Preconditions:" preconditions

precondition_name  → id

already_checked    → checked_constraint {checked_constraint}

checked_constraint →  "on" constraint_name
                      ["by" operation_name]

constraint_name    → id

operation_name     → id
```

Lexical Grammar

Within *word*, *item*, *comma_list*, or *id* below, any literal, *letter*, *digit*, or *not_blank_space_char* must not be separated by any blank spaces.

```
word       → not_blank_space_char {not_blank_space_char}

item       → "(" digit {digit} ")"

comma_list → id {"," id} [","]

id         → letter { letter | digit | "_" }
```

Appendix B

Orwell Source Code

Symbols for Z notation (These can be changed if necessary.)

```
> symbolForfinite_set           = "F"
> symbolFormembership_op        = "in"
> symbolForset_union            = "cup"
> symbolForspot                 = "@"
> symbolFormaplet               = "|->"
> symbolForpartial_function     = "-+>"
> symbolForpartial_injection    = ">+>"
> symbolFordom_anti_restriction = "<<|"
```

Rule for naming Z schemas (These can be changed if necessary.)

```
> makeAttrSchemaName entityName attributeName
>        = attributeName ++ "of" ++ entityName

> makeAttrConstsSchemaName entityName attributeName
>        = makeAttrSchemaName entityName attributeName ++ "Constraints"

> makeRecordSchemaName entityName = entityName ++ "Record"

> makeRecordConstsSchemaName entityName
>       = makeRecordSchemaName entityName ++ "Constraints"

> makeIdName entityName = entityName ++ "Id"

> makeKnownName entityName = "known" ++ entityName

> makeTableName entityName = "table" ++ entityName

> makeTableSchemaName entityName = entityName ++ "Table"

> makeTableConstsSchemaName entityName
```

```
>         = makeTableSchemaName entityName ++ "Constraints"

> makeRelName relationshipName = relationshipName

> makeRelSchemaName relationshipName = relationshipName ++ "Relationship"

> makeRelConstsSchemaName relationshipName = relationshipName ++ "Constraints"

> makeIntegSchemaName integrityName = integrityName
```

Data structure

Common data structure

```
> word == [char]
> nameType == word
> expressionType == [word]
> typeType == expressionType
> predicateType == expressionType
```

Global sate

```
> state == (dbST, prST, scST, clST)

> initialstate = (db0, pr0, sc0, cl0)
```

Data on DB

```
> dbST == (dbnameST, entitiesST, relationshipsST, integritiesST)

> dbnameST       == nameType
> entitiesST     == [(nameType, entityData)]
> entityData     == (attributesType, recordConstsType, tableConstsType)
> attributesType == [(nameType, typeType, constraintsType)]
> recordConstsType == constraintsType
> tableConstsType == constraintsType
> constraintsType == [predicateType]

> relationshipsST == [(nameType, relationshipData)]
> relationshipData == (detailEntityNameType, masterEntityNameType,
>                      correspondenceType, constraintsType)
> detailEntityNameType == nameType
> masterEntityNameType == nameType
> correspondenceType ::= OnetoOne | OptionalOnetoOne | OnetoOptionalOne |
>                        OptionalOnetoOptionalOne | ManytoOne |
>                        OptionalManytoOne | ManytoOptionalOne |
>                        OptionalManytoOptionalOne

> integritiesST  == [(nameType, integrityData)]
```

```
> integrityData  == (scopeEType, scopeRType, constraintsType)
> scopeEType == [nameType]
> scopeRType == [nameType]

> db0 = ("", [], [], [])
```

Data on Process

```
> prST == (prnameType, inputDataType, subprocessesonEST, subprocessesonRST,
>           globalPreconditionsST, alreadyCheckedST)

> prnameType == nameType
> inputDataType   == [(nameType, typeType)]
> subprocessesonEST == [(nameType, subprocessonEData)]
> subprocessonEData == (entityNameType, scopeEType, scopeRType, inputDataType,
>                       preconditionsType, [basicOperationonEType])
> entityNameType == nameType
> preconditionsType == [predicateType]
> basicOperationonEType == (updateType, interfaceType, targetType,
>                           [recordOperationType])
> updateType ::= Modify | Insert | Delete
> interfaceType == expressionType
> targetType == expressionType
> recordOperationType == (nameType, scopeEType, scopeRType, inputDataType,
>                         preconditionsType, atomicOperationsST)
> atomicOperationsST == [(attributeNameType, nameType, atomicOperationType)]
> attributeNameType == nameType
> atomicOperationType == (scopeEType, scopeRType, inputDataType,
>                         preconditionsType, operationType)
> operationType == [expressionType]

> subprocessesonRST == [(nameType, subprocessonRData)]
> subprocessonRData == (relationshipNameType, scopeEType, scopeRType,
>                       inputDataType, preconditionsType,
>                       [basicOperationonRType])
> relationshipNameType == nameType
> basicOperationonRType == (updateType, dataForUpdateType)
> dataForUpdateType == expressionType

> globalPreconditionsST == [(nameType, globalPreconditionData)]
> globalPreconditionData == (scopeEType, scopeRType, inputDataType,
>                            preconditionsType)

> alreadyCheckedST == [(constraintNameType, [operationNameType])]
> constraintNameType == nameType
> operationNameType == nameType

> pr0 = ("", [], [], [], [], [])
```

Data on Schemas

```
> scST == (schemalistST, expandedSchemalistST)

> schemalistST   == [(nameType, schemaType)]
> schemaType == (inclType, declType, predType)
> inclType    == [schemaRefType]
> schemaRefType  == (deltaType, nameType, primeType)
> declType    == [(nameType, typeType)]
> predType    == [predicateType]

> expandedSchemalistST == [(nameType, schemaType)]

> deltaType  ::= Delta | Xi | NoDelta
> primeType  ::= Prime | NoPrime

> sc0 = ([], [])
```

Data on Checklist

```
> clST == [proofObligationData]

> proofObligationData == (constraintNameType, [operationNameType],
>                          [schemaRefType], okflagType)

> okflagType ::= Ok | Notyet

> cl0 = []
```

Function

ReadSpec

```
> readSpec filename
>       = (db, pr, sc, cl)
>    where (db, pr) = (parser . scanner) (filein filename)
>          sc = ([], [])
>          cl = []
```

Syntax Analyser

```
> scanner :: [char] -> [word]

> scanner = filter (~=[]) . foldr (breakonset [' ', '\n', '\t']) [[]]

> breakonset as x xss = [[]] ++ xss, if x $in as
>                     = [[x] ++ hd xss] ++ tl xss, otherwise

> parseonset as = filter (~=[]) . foldr (breakonset as) [[]]
```

```
> breakonseq (xss, []) x
>       = (init xss ++ [last xss ++ [x]], [])
> breakonseq (xss, (a:as)) x
>       = (xss ++ [[]], as), if x = a
>       = (init xss ++ [last xss ++ [x]], (a:as)), otherwise

> parseonseq as = fst . foldl breakonseq ([[]], as)

> parseoncomma_list = filter (~=[]) . foldr (breakon ',') [[]] .
>                         makelist ","

%> parseoncomma_list = filter (~=[]) . foldr (breakon ',') [[]] . concat

> breakonn n [] = []
> breakonn n (w:ws) = take n (w:ws) : breakonn n (drop n (w:ws))

> parsedef eq ws
>       = [(hd (wss!0), wss!1)
>           | wss <- breakonn 2 (trans (foldr (breakon eq) [[]] ws))]
>     where trans [id1, list1] = [id1, list1]
>           trans (id1:list1:xs:xss) = id1:init list1:trans ([last list1]:xs:xss)

> breakonitem x xss = [[]] ++ xss, if itemform x
>                   = [[x] ++ hd xss] ++ tl xss, otherwise

> itemform x = (#x >= 3 & hd x = '(' & last x = ')' & all isdigit (tl (init x)))

> isdigit p = ('0' <= p & p <= '9')

> parseonitem = filter (~=[]) . foldr breakonitem [[]]

> breakonseq2 (xss, as) x
>       = (xss ++ [[x]], tl (dropwhile (~= x) as)), if x $in as
>       = (init xss ++ [last xss ++ [x]], as), otherwise

> parseonlabel as = filter (~=[]) . fst . foldl breakonseq2 ([[]], as)

> parseafterlabel label postparser wss
>           = [], if ~(label $in (map hd wss))
>           = postparser (tl (hd [ws2 | ws2 <- wss; hd ws2 = label])), otherwise
```

Parser

```
> parser ws = (parse_db (wss ! 1),
>              parse_pr (wss ! 2))
>     where wss = parseonseq ["DBName:", "Process:"] ws

> parse_db ws = (dbname, entities, relationships, integrities)
>     where wss = parseonlabel ["Entities:", "Relationships:",
>                               "DBIntegrityConstraints:"] ws
```

```
>            dbname = hd ws
>            entities = parseafterlabel "Entities:" parse_entities wss
>            relationships = parseafterlabel "Relationships:"
>                                              parse_relationships wss
>            integrities = parseafterlabel "DBIntegrityConstraints:"
>                                           parse_integrities wss

> parse_entities ws = [(name, parse_entityData ws2)|
>                      (name, ws2) <- parsedef "Attributes:" ws]

> parse_entityData ws = (parse_attributes (wss ! 0),
>                        recordConsts,
>                        tableConsts)
>       where wss = parseonlabel ["RecordConstraints:", "TableConstraints:"] ws
>             recordConsts = parseafterlabel "RecordConstraints:"
>                                             parse_predicates wss
>             tableConsts = parseafterlabel "TableConstraints:"
>                                             parse_predicates wss

> parse_attributes ws = [parse_attribute name ws2 |
>                        (name, ws2) <- parsedef ":" ws]

> parse_attribute name ws = (name, type, attrConsts)
>       where wss = parseonset ["AttributeConstraints:"] ws
>             type = wss ! 0
>             attrConsts = [], if # wss <= 1
>                        = parse_predicates (wss ! 1), otherwise

> parse_predicates = parseonitem

> parse_relationships ws = [(name, parse_relationshipData ws2)|
>                           (name, ws2) <- parsedef ":" ws]

> parse_relationshipData ws = (detail, master, corType, consts)
>       where wss = parseonseq [symbolForpartial_function,
>                               "Type:", "Constraints:"] ws
>             detail = hd (wss ! 0)
>             master = hd (wss ! 1)
>             corType = OnetoOne, if hd (wss ! 2) = "OnetoOne"
>                     = OptionalOnetoOne, if hd (wss ! 2) = "OptionalOnetoOne"
>                     = OnetoOptionalOne, if hd (wss ! 2) = "OnetoOptionalOne"
>                     = OptionalOnetoOptionalOne,
>                           if hd (wss ! 2) = "OptionalOnetoOptionalOne"
>                     = ManytoOne, if hd (wss ! 2) = "ManytoOne"
>                     = OptionalManytoOne, if hd (wss ! 2) = "OptionalManytoOne"
>                     = ManytoOptionalOne, if hd (wss ! 2) = "ManytoOptionalOne"
>                     = OptionalManytoOptionalOne,
>                           if hd (wss ! 2) = "OptionalManytoOptionalOne"
>             consts = [], if # wss <= 3
>                    = parse_predicates (wss ! 3), otherwise
```

```
> parse_integritiss ws = [(name, parse_integrityData ws2)|
>                         (name, ws2) <- parsedef ":" ws]

> parse_integrityData ws = (scopeE, scopeR, consts)
>       where wss = parseonlabel ["ScopeE:", "ScopeR:", "Constraints:"] ws
>             scopeE = parseafterlabel "ScopeE:" parseoncomma_list wss
>             scopeR = parseafterlabel "ScopeR:" parseoncomma_list wss
>             consts = parseafterlabel "Constraints:" parse_predicates wss

> parse_pr ws = (prname, inputData, subprsonE, subprsonR, globalPres, alreadyCh)
>       where wss = parseonlabel ["InputData:", "SubprocessesonEntities:",
>                                 "SubprocessesonRelationships:",
>                                 "GlobalPreconditions:", "AlreadyChecked:"] ws
>             prname = hd ws
>             inputData = parseafterlabel "InputData:" parse_inputData wss
>             subprsonE = parseafterlabel "SubprocessesonEntities:"
>                                         parse_subprocessesonE wss
>             subprsonR = parseafterlabel "SubprocessesonRelationships:"
>                                         parse_subprocessesonR wss
>             globalPres = parseafterlabel "GlobalPreconditions:"
>                                          parse_globalPreconditions wss
>             alreadyCh = parseafterlabel "AlreadyChecked:"
>                                          parse_alreadyChecked wss

> parse_inputData ws = parsedef ":" ws

> parse_subprocessesonE ws = [(name, parse_subprocessonEData ws2)|
>                             (name, ws2) <- parsedef "on" ws]

> parse_subprocessonEData ws = (entityName, scopeE, scopeR, inputD, pres,
>                                                                  basicOps)
>       where wss = parseonlabel ["ScopeE:", "ScopeR:", "InputData:",
>                                 "Preconditions:", "BasicOperations:"] ws
>             entityName = hd ws
>             scopeE = parseafterlabel "ScopeE:" parseoncomma_list wss
>             scopeR = parseafterlabel "ScopeR:" parseoncomma_list wss
>             inputD = parseafterlabel "InputData:" parse_inputData wss
>             pres = parseafterlabel "Preconditions:" parse_predicates wss
>             basicOps = parseafterlabel "BasicOperations:"
>                                        (parse_basicOpsonE entityName) wss

> parse_basicOpsonE entityName ws = [parse_basicOponE update ws2|
>                                    (update, ws2) <- parsedef entityName ws]

> parse_basicOponE update ws = (parse_update update,
>                               interface, target, recordOp)
>       where wss = parseonseq ["Target:", "RecordOperation:"] ws
>             interface = wss ! 0
>             target = wss ! 1
```

```
>               recordOp = [], if # wss <= 2
>                        = [parse_recordOp (wss ! 2)], otherwise

> parse_update update = Modify, if update = "Modify"
>                     = Insert, if update = "Insert"
>                     = Delete, if update = "Delete"

> parse_recordOp ws = (name, scopeE, scopeR, inputD, pres, atomicOps)
>       where wss = parseonlabel ["ScopeE:", "ScopeR:", "InputData:",
>                               "Preconditions:", "AtomicOperations:"] ws
>               name = hd ws
>               scopeE = parseafterlabel "ScopeE:" parseoncomma_list wss
>               scopeR = parseafterlabel "ScopeR:" parseoncomma_list wss
>               inputD = parseafterlabel "IuputData:" parse_inputData wss
>               pres = parseafterlabel "Preconditions:" parse_predicates wss
>               atomicOps = parseafterlabel "AtomicOperations:"
>                                         parse_atomicOps wss

> parse_atomicOps ws = [parse_atomicOp attrName ws2|
>                       (attrName, ws2) <- parsedef "by" ws]

> parse_atomicOp attrName ws = (attrName, name,
>                              (scopeE, scopeR, inputD, pres, ops))
>       where wss = parseonlabel ["ScopeE:", "ScopeR:", "InputData:",
>                               "Preconditions:", "Operation:"] ws
>               name = hd ws
>               scopeE = parseafterlabel "ScopeE:" parseoncomma_list wss
>               scopeR = parseafterlabel "ScopeR:" parseoncomma_list wss
>               inputD = parseafterlabel "InputData:" parse_inputData wss
>               pres = parseafterlabel "Preconditions:" parse_predicates wss
>               ops = parseafterlabel "Operation:" parse_predicates wss

> parse_subprocessesonR ws = [(name, parse_subprocessonRData ws2)|
>                           (name, ws2) <- parsedef "on" ws]

> parse_subprocessonRData ws = (relName, scopeE, scopeR, inputD, pres,
>                                                        basicOps)
>       where wss = parseonlabel ["ScopeE:", "ScopeR:", "InputData:",
>                               "Preconditions:", "BasicOperations:"] ws
>               relName = hd ws
>               scopeE = parseafterlabel "ScopeE:" parseoncomma_list wss
>               scopeR = parseafterlabel "ScopeR:" parseoncomma_list wss
>               inputD = parseafterlabel "InputData:" parse_inputData wss
>               pres = parseafterlabel "Preconditions:" parse_predicates wss
>               basicOps = parseafterlabel "BasicOperations:"
>                                      (parse_basicOpsonR relName) wss

> parse_basicOpsonR relName ws = [parse_basicOponR update ws2|
>                               (update, ws2) <- parsedef relName ws]
```

```
> parse_basicOponR update ws = (parse_update update, dataforUpdate)
>       where wss = parseonseq ["DataforUpdate:"] ws
>             dataforUpdate = wss ! 1

> parse_globalPreconditions ws = [(name, parse_globalPreconditionData ws2)|
>                                  (name, ws2) <- parsedef "::" ws]

> parse_globalPreconditionData ws = (scopeE, scopeR, inputD, pres)
>       where wss = parseonlabel ["ScopeE:", "ScopeR:", "InputData:",
>                                 "Preconditions:"] ws
>             scopeE = parseafterlabel "ScopeE:" parseoncomma_list wss
>             scopeR = parseafterlabel "ScopeR:" parseoncomma_list wss
>             inputD = parseafterlabel "InputData:" parse_inputData wss
>             pres = parseafterlabel "Preconditions:" parse_predicates wss

> parse_alreadyChecked ws
>             = [parse_ac ws2|
>                ws2 <- (filter (~=[]) . foldr (breakon "on") [[]]) ws]

> parse_ac ws = (constName, opName)
>       where wss = parseonseq ["by"] ws
>             constName = hd (wss ! 0)
>             opName = [], if # wss <= 1
>                    = [hd (wss ! 1)], otherwise


Schema generator

> generateSchemas :: state -> state

> generateSchemas = generatePr . generateDB . generateIntegrities .
>                   generateRels . generateEntities

> generateEntities :: state -> state

> generateEntities (db, pr, (schemalist, expandedSchemalist), cl)
>       = (db, pr, (schemalist', expandedSchemalist), cl)
>    where (dbname, entities, rels, integs) = db
>          schemalist' = concat (map generateEntity entities)

> generateEntity (entName, (attrs, recConsts, tableConsts))
>       = concat [generateAttribute entName attr | attr <- attrs] ++
>         generateRecord entName attrs recConsts ++
>         generateTable entName recConsts tableConsts

> generateAttribute entName (attrName, type, attrConsts)
>       = [makeAttributeSchema entName (attrName, type)] ++
>         constsSchema
>    where constsSchema = [], if attrConsts = []
>                       = [makeConstraintsSchema
```

```
>                             (makeAttrConstsSchemaName entName attrName)
>                             (makeAttrSchemaName entName attrName)
>                              attrConsts], otherwise

> makeAttributeSchema entName (attrName, type)
>         = (schemaName, (incl, decl, pred))
>     where schemaName = makeAttrSchemaName entName attrName
>           incl = []
>           decl = [(attrName, type)]
>           pred = []

> makeConstraintsSchema name baseName consts
>         = (schemaName, (incl, decl, pred))
>     where schemaName = name
>           incl = [(NoDelta, baseName, NoPrime)]
>           decl = []
>           pred = consts

> generateRecord entName attrs recConsts
>         = [makeRecordSchema entName attrs] ++
>           constsSchema
>     where constsSchema = [], if recConsts = []
>                        = [makeConstraintsSchema
>                             (makeRecordConstsSchemaName entName)
>                             (makeRecordSchemaName entName)
>                              recConsts], otherwise
>

> makeRecordSchema entName attrs
>         = (schemaName, (incl, decl, pred))
>     where schemaName = makeRecordSchemaName entName
>           incl = [(NoDelta,
>                    attrSchemaorConstsSchema entName attrName attrConsts,
>                    NoPrime) |
>                        (attrName, type, attrConsts) <- attrs]
>           decl = []
>           pred = []
>           attrSchemaorConstsSchema entName attrName attrConsts
>                = makeAttrSchemaName entName attrName, if attrConsts = []
>                = makeAttrConstsSchemaName entName attrName, otherwise

> generateTable entName recConsts tableConsts
>        = [makeTableSchema entName recConsts] ++
>          constsSchema
>     where constsSchema = [], if tableConsts = []
>                        = [makeConstraintsSchema
>                             (makeTableConstsSchemaName entName)
>                             (makeTableSchemaName entName)
>                              tableConsts], otherwise
```

```
> makeTableSchema entName recConsts
>         = (schemaName, (incl, decl, pred))
>      where schemaName = makeTableSchemaName entName
>            incl = []
>            decl = [(makeKnownName entName,
>                        [symbolForfinite_set,makeIdName entName]),
>                   (makeTableName entName,
>                        [makeIdName entName, symbolForpartial_function,
>                                                    recordSchemaName])]
>            pred = [[makeKnownName entName, "=", "dom",
>                        makeTableName entName]] ++
>                   predConsts
>            predConsts = [], if recConsts = []
>                       = [["forall", recordSchemaName, "|", "theta",
>                            recordSchemaName, symbolFormembership_op, "ran",
>                            makeTableName entName, symbolForspot,
>                            makeRecordConstsSchemaName entName]], otherwise
>            recordSchemaName = makeRecordSchemaName entName

> generateRels :: state -> state

> generateRels (db, pr, (schemalist, expandedSchemalist), cl)
>         = (db, pr, (schemalist', expandedSchemalist), cl)
>      where (dbname, entities, rels, integs) = db
>            schemalist' = schemalist ++
>                          concat (map generateRel rels)

> generateRel (relName, (detailName, masterName, corType, relConsts))
>       = [makeRelationshipSchema relName detailName masterName,
>         makeRelationshipConstsSchema
>              relName detailName masterName corType relConsts]
>

> makeRelationshipSchema relName detailName masterName
>       = (schemaName, (incl, decl, pred))
>    where schemaName = makeRelSchemaName relName
>          incl = []
>          decl = [(makeRelName relName,
>                   [makeIdName detailName, symbolForpartial_function,
>                    makeIdName masterName])]
>          pred = []

> makeRelationshipConstsSchema relName detailName masterName corType relConsts
>       = (schemaName, (incl, decl, pred))
>    where schemaName = makeRelConstsSchemaName relName
>          incl = [(NoDelta, makeRelSchemaName relName, NoPrime)]
>          decl = [(makeKnownName detailName,
>                       [symbolForfinite_set, makeIdName detailName]),
>                  (makeKnownName masterName,
>                       [symbolForfinite_set, makeIdName masterName])]
```

```
>          pred = [predDom, predRan] ++ predInjectivity ++ relConsts
>          predDom = ["dom", makeRelName relName, opDom,
>                            makeKnownName detailName]
>          opDom = "=", if corType $in [OnetoOne, OptionalOnetoOne,
>                                       ManytoOne, OptionalManytoOne]
>                = "subseteq", otherwise
>          predRan = ["ran", makeRelName relName, opRan,
>                            makeKnownName masterName]
>          opRan = "=", if corType $in [OnetoOne, OnetoOptionalOne,
>                                       ManytoOne, ManytoOptionalOne]
>                = "subseteq", otherwise
>          predInjectivity = [], if corType $in [ManytoOne, OptionalManytoOne,
>                                                ManytoOptionalOne,
>                                                OptionalManytoOptionalOne]
>                          = [[makeRelName relName, symbolFormembership_op,
>                              makeIdName detailName,
>                              symbolForpartial_injection,
>                              makeIdName masterName]], otherwise

> generateIntegrities :: state -> state

> generateIntegrities (db, pr, (schemalist, expandedSchemalist), cl)
>       = (db, pr, (schemalist', expandedSchemalist), cl)
>    where (dbname, entities, rels, integs) = db
>          schemalist' = schemalist ++
>                        map (makeIntegritySchema schemalist) integs

> makeIntegritySchema schemalist (integName, (scopeE, scopeR, integConsts))
>     = (schemaName, (incl, decl, pred))
>   where schemaName = makeIntegSchemaName integName
>          incl = inclE ++ inclR
>          inclE = [(NoDelta, tableorConsts entName, NoPrime) |
>                                                 entName <- scopeE]
>          tableorConsts entName
>              = makeTableSchemaName entName
>          inclR = [(NoDelta, makeRelSchemaName relName, NoPrime) |
>                                                 relName <- scopeR]
>          decl = []
>          pred = integConsts

> lookupSchema schemalist schemaName
>      = schemaName $in [name | (name, data) <- schemalist]

> generateDB :: state -> state

> generateDB (db, pr, (schemalist, expandedSchemalist), cl)
>       = (db, pr, (schemalist', expandedSchemaliet), cl)
>    where schemalist' = schemalist ++
>                        [makeDBSchema schemalist db]
```

```
> makeDBSchema schemalist (dbname, entities, rels, integs)
>       = (schemaName, (incl, decl, pred))
>    where schemaName = dbname
>          incl = inclE ++ inclR ++ inclIC
>          inclE = [(NoDelta, tableorConsts entName, NoPrime) |
>                                          (entName, data) <- entities]
>          tableorConsts entName
>              = makeTableConstsSchemaName entName,
>                                  if lookupSchema schemalist
>                                        (makeTableConstsSchemaName entName)
>              = makeTableSchemaName entName, otherwise
>          inclR = [(NoDelta, makeRelConstsSchemaName relName, NoPrime) |
>                                        (relName, data) <- rels]
>          inclIC = [(NoDelta, makeIntegSchemaName integName, NoPrime) |
>                                        (integName, data) <- integs]
>          decl = []
>          pred = []


> generatePr :: state -> state

> generatePr (db, pr, (schemalist, expandedSchemalist), cl)
>        = (db, pr, (schemalist', expandedSchemalist), cl)
>     where (prname, inputData, subprocessesonE, subprocessesonR,
>                  globalPreconditions, alreadyChecked) = pr
>           schemalist' = schemalist ++
>                         concat (map generateSubprocessonE subprocessesonE) ++
>                         map makeSubprocessonRSchema subprocessesonR ++
>                         map makeGlobalPreconditionSchema globalPreconditions

> generateSubprocessonE (subprocessName, subprocessData)
>        = concat [generateBasicOponE entName basicOp | basicOp <- basicOps] ++
>          [makeSubprocessonESchema subprocessName entName scopeE scopeR
>                                                   inputD pres basicOps]
>    where (entName, scopeE, scopeR, inputD, pres, basicOps) = subprocessData

> generateBasicOponE entName (update, interface, target, recordOpData)
>        = [], if update = Delete
>        = generateRecordOp entName update (hd recordOpData), otherwise

> generateRecordOp entName update (recordOpName, scopeE, scopeR, inputD,
>                                                  pres, atomicOps)
>        = [makeAtomicOpSchema entName update atomicOp |
>                                     atomicOp <- atomicOps] ++
>          [makeRecordOpSchema entName update recordOpName scopeE scopeR
>                                                   inputD pres atomicOps]

> makeAtomicOpSchema entName update (attrName, atomicOpName, atomicOpData)
>      = (schemaName, (incl, decl, pred))
>   where (scopeE, scopeR, inputD, pres, ops) = atomicOpData
```

```
>           schemaName = atomicOpName
>           incl = inclAttr ++ inclScope
>           inclAttr = makeInclDelta update (makeAttrSchemaName entName attrName)
>           inclScope = makeInclScope scopeE scopeR
>           decl = inputD
>           pred = pres ++ ops
>

> makeRecordOpSchema entName update recordOpName scopeE scopeR
>                                                        inputD pres atomicOps
>     = (schemaName, (incl, decl, pred))
>    where schemaName = recordOpName
>          incl = inclRec ++ inclScope ++ inclAtomicOps
>          inclRec = makeInclDelta update (makeRecordSchemaName entName)
>          inclScope = makeInclScope scopeE scopeR
>          inclAtomicOps = [(NoDelta, atomicOpName, NoPrime) |
>                               (attrName, atomicOpName, data) <- atomicOps]
>          decl = inputD
>          pred = pres

> makeInclDelta update schemaName
>     = [(delta, schemaName, prime)]
>    where delta = Delta, if update = Modify
>                = NoDelta, if update = Insert
>          prime = Prime, if update = Insert
>                = NoPrime, if update = Modify

> makeInclScope scopeE scopeR
>     = [(NoDelta, makeTableSchemaName ent, NoPrime) | ent <- scopeE] ++
>       [(NoDelta, makeRelSchemaName rel, NoPrime) | rel <- scopeR]

> makeSubprocessonESchema subprocessName entName scopeE scopeR
>                                                   inputD pres basicOps
>     = (schemaName, (incl, decl, pred))
>    where schemaName = subprocessName
>          incl = inclTable ++ inclScope
>          inclTable = [(Delta, makeTableSchemaName entName, NoPrime)]
>          inclScope = makeInclScope scopeE scopeR
>          decl = inputD
>          pred = pres ++ opForKnownE ++ opsForTableE ++ unchangedPart
>          listofModifyTargets
>               =  [target | (update, interface, target, data) <- basicOps;
>                            update = Modify]
>          listofInsertTargets
>               =  [target | (update, interface, target, data) <- basicOps;
>                            update = Insert]
>          listofDeleteTargets
>               =  [target | (update, interface, target, data) <- basicOps;
>                            update = Delete]
>          knownName = makeKnownName entName
```

```
>            tableName = makeTableName entName
>            recordSchemaName = makeRecordSchemaName entName
>            opForKnownE
>               = [[knownName++"'", "=", knownName]],
>                     if listofInsertTargets = [] & listofDeleteTargets = []
>               = [[knownName++"'", "=", "(", knownName, "\\"] ++
>                   makeSetExp listofDeleteTargets ++ [")", symbolForset_union] ++
>                   makeSetExp listofInsertTargets],
>                     if listofInsertTargets ~= [] & listofDeleteTargets ~= []
>               = [[knownName++"'", "=", knownName, "\\"] ++
>                   makeSetExp listofDeleteTargets],
>                     if listofInsertTargets = [] & listofDeleteTargets ~= []
>               = [[knownName++"'", "=", knownName, symbolForset_union] ++
>                   makeSetExp listofInsertTargets],
>                     if lietofInsertTargets ~= [] & listofDeleteTargets = []
>           opsForTableE = map (makeOpForTableE entName)
>                          [(update, interface, target, recordOpName) |
>                           (update, interface, target, recordOpData) <- basicOps;
>                              update ~= Delete;
>                           (recordOpName, scopeE, scopeR, inputD, pres, atomicOps)
>                                        <- recordOpData ]
>           makeOpForTableE entName (update, interface, target, recordOpName)
>                = interface ++
>                  ["exists", "Delta", recordSchemaName, "|"] ++
>                  target ++
>                  [symbolFormaplet, "theta", recordSchemaName,
>                   symbolFormembership_op, tableName, "/\\"] ++
>                  target ++
>                  [symbolFormaplet, "theta", recordSchemaName++"'",
>                   symbolFormembership_op, tableName++"'",
>                   symbolForspot, recordOpName],   if update = Modify
>                = interface ++
>                  ["exists", recordSchemaName++"'", "|"] ++
>                  target ++
>                  [symbolFormaplet, "theta", recordSchemaName++"'",
>                   symbolFormembership_op, tableName++"'",
>                   symbolForspot, recordOpName],   if update = Insert
>           unchangedPart = [lhs ++ ["="] ++ rhs]
>           lhs = [tableName ++ "'"], if listofModifyTargets = [] &
>                                        listofInsertTargets = []
>               = makeSetExp (listofModifyTargets ++ listofInsertTargets) ++
>                   [symbolFordom_anti_restriction, tableName++"'"], otherwise
>           rhs = [tableName],        if listofModifyTargets = [] &
>                                        listofDeleteTargets = []
>               = makeSetExp (listofModifyTargets ++ listofDeleteTargets) ++
>                   [symbolFordom_anti_restriction, tableName], otherwise

> makeSetExp list = ["{"] ++ hd list ++
>                   concat [[","] ++ e | e <- tl list] ++
>                   ["}"]
```

```
> makeSubprocessonRSchema (subprocessName, data)
>       = (schemaName, (incl, decl, pred))
>    where (relName, scopeE, scopeR, inputD, pres, basicOps) = data
>          schemaName = subprocessName
>          incl = inclRel ++ inclScope
>          inclRel = [(Delta, makeRelSchemaName relName, NoPrime)]
>          inclScope = makeInclScope scopeE scopeR
>          decl = inputD
>          pred = pres ++ [opForRel]
>          opForRel = [makeRelName relName++"'", "="] ++ deleteInsertModifyR
>          deleteInsertModifyR = deleteInsertR, if listofModifyData = []
>                              = ["("] ++ deleteInsertR ++ [")"] ++
>                                ["(+)"] ++ dataExp listofModifyData, otherwise
>          deleteInsertR = deleteR, if listofInsertData = []
>                        = ["("] ++ deleteR ++ [")", symbolForset_union] ++
>                          dataExp listofInsertData,  otherwise
>          deleteR = [makeRelName relName], if listofDeleteData = []
>                  = ["("] ++ dataExp listofDeleteData ++
>                    [symbolFordom_anti_restriction, makeRelName relName, ")"],
>                                                                     otherwise
>          listofModifyData = [dataForUpdate | (update, dataForUpdate) <- basicOps;
>                                              update = Modify]
>          listofInsertData = [dataForUpdate | (update, dataForUpdate) <- basicOps;
>                                              update = Insert]
>          listofDeleteData = [dataForUpdate | (update, dataForUpdate) <- basicOps;
>                                              update = Delete]
>          dataExp list = hd list, if tl list = []
>                       = ["("] ++ hd list ++
>                         concat [[symbolForset_union] ++ data | data <- tl list] ++
>                         [")"], otherwise

> makeGlobalPreconditionSchema (name, (scopeE, scopeR, inputD, pres))
>       = (schemaName, (incl, decl, pred))
>    where schemaName = name
>          incl = makeInclScope scopeE scopeR
>          decl = inputD
>          pred = pres
```

View

```
> view :: nameType -> state -> [char]

> view name (db, pr, (schema, exp), cl)
>       = "\n" ++ name ++ "\n=\n" ++ showSchema (assoc schema name)

> showSchema :: schemaType -> [char]

> showSchema (incl, decl, pred)
```

```
>          = " ----------------------------------------------------------\n" ++
>            concat ["| "++ showSchemaRef s ++ "\n"| s <- incl] ++
>            concat ["| "++name++" : "++showType type++"\n"|(name,type) <- decl] ++
>            centerline ++
>            concat ["| "++ showExpinSchema pr ++ "\n"| pr <- pred] ++
>            " ----------------------------------------------------------\n"
>      where centerline = "|---------------\n", if pred ~= []
>                       = "",                   otherwise

> showSchemaRef :: (deltaType, nameType, primeType) -> [char]

> showSchemaRef (NoDelta, n, d)  = n,         if d = NoPrime
>                                = n ++ "'", if d = Prime
> showSchemaRef (Delta, n, d) = "Delta "++ showSchemaRef (NoDelta, n, d)
> showSchemaRef (Xi, n, d)    = "Xi "++ showSchemaRef (NoDelta, n, d)

> showType :: typeType -> [char]

> showType type = makelist " " type

> showExpinSchema = showExp "|"

> showExp :: [char] -> expressionType -> [char]

> showExp hdstr [] = ""
> showExp hdstr (w:exp)
>          = w ++ "\n" ++ hdstr ++ "\t" ++ showExp hdstr exp,
>                                    if w $in [symbolForspot, "|", "/\\"]
>          = w ++ " " ++ showExp hdstr exp, otherwise
```

View2

```
> view2 :: nameType -> state -> ([char], state)

> view2 name (db, pr, (schema, exp), cl)
>          = ("\n" ++ name ++ "\n=\n" ++ showSchema sc,
>             (db, pr, (schema, exp'), cl))
>     where (sc, exp') = assocexpand schema exp name

> assocexpand :: schemalistST -> expandedSchemalistST -> nameType ->
>                                    (schemaType, expandedSchemalistST)

> assocexpand schema exp name
>          = (assoc exp name, exp),       if name $in [n|(n,d) <- exp]
>          = assocexpSchema schema exp name (assoc schema name), otherwise

> assocexpSchema schema exp name (incl, decl, pred)
>          = assocexpSchema2 schema exp name (incl', decl, pred')
>      where incl' = reverse incl2
```

```
>             (incl2, pred') = expDelta schema exp (incl, pred)

> expDelta schema exp ([], pred) = ([], pred)
> expDelta schema exp ((s:incl), pred)
>       = (setUnion incl1 incl2, setUnion pred1 pred2)
>    where (incl1, pred1) = f schema exp s
>          (incl2, pred2) = expDelta schema exp (incl, pred)
>          f schema exp (NoDelta, name, d) = ([(NoDelta, name, d)], [])
>          f schema exp (Delta,name,d)
>                = ([(NoDelta,name,NoPrime), (NoDelta,name,Prime)], [])
>          f schema exp (Xi, name, d)
>                = ([(NoDelta, name, NoPrime), (NoDelta, name, Prime)],
>                            [[var++"'", "=", var]|
>                             (schema1, exp1) <- [assocexpand schema exp name];
>                             (incl3, decl3, pred3) <- [schema1];
>                             (var,type) <- decl3])

> assocexpSchema2 schema exp name ([], decl, pred)
>       = (([], decl, pred), addpair (name, ([], decl, pred)) exp)

> assocexpSchema2 schema exp name (((d,n,p):incl), decl, pred)
>       = assocexpSchema2 schema exp' name (incl, setUnion decl2 decl,
>                                                  setUnion pred2 pred)
>    where ((incl', decl', pred'), exp') = assocexpand schema exp n
>          decl2 = decl',  if p = NoPrime
>                = [(name++"'", t)|(name, t) <- decl'], otherwise
>          pred2 = pred',  if p = NoPrime
>                = map (subst pairs) pred', otherwise
>          pairs = [(name, name++"'")|(name, t) <- decl']
```

Schemalist

```
> schemalist :: state -> [char]

> schemalist (db, pr, (schema, exp), cl)
>       = "\n" ++ concat (zipwith f ([0..], schema))
>    where f i (name, data) = show i ++ ". " ++ name ++ "\n"
```

Checklist

```
> checklist :: state -> [char]

> checklist (db, pr, sc, cl)
>       = "\n" ++ concat (zipwith f ([0..], cl))
>    where f i (constName, opName, schemaRefs, okflag)
>               = show i ++ ". on " ++ constName ++
>                 g opName ++ "\t" ++ h okflag ++ "\n"
>          g opName = "", if opName = []
```

```
>                  = "\tby " ++ hd opName, otherwise
>          h okflag = "-already OK", if okflag = Ok
>                  = "-not yet", if okflag = Notyet
```

Generate Checklist

```
> generateChecklist :: state -> state

> generateChecklist (db, pr, (schemalist, exp), cl)
>      = (db, pr, (schemalist, exp), cl')
>    where (prname, inputD, subE, subR, globalPres, ac) = pr
>          (dbname, entities, rels, integs) = db
>          cl' = concat (map genCheckonE subE) ++
>                concat (map makeCheckonR subR) ++
>                concat (map makeCheckonUnchangedR rels) ++
>                concat (map makeCheckonIC integs)
>          genCheckonE (subprocessName, subprocessData)
>              = genCheckonE2 subprocessName subprocessData
>          genCheckonE2 subprocessName (entName, scopeE, scopeR, inut, pres,
>                                                                basicOps)
>              = concat (map (genCheckonBasicOp entName) basicOps) ++
>                makeCheckonTable subprocessName entName
>          makeCheckonTable subprocessName entName
>               = [], if ~(lookupSchema schemalist
>                             (makeTableConstsSchemaName entName))
>               = [(makeTableConstsSchemaName entName,
>                  [subprocessName],
>                  [(NoDelta, makeTableConstsSchemaName entName, NoPrime),
>                   (NoDelta, subprocessName, NoPrime)],
>                  makeOkflag
>                    (makeTableConstsSchemaName entName, [subprocessName]))],
>                                                                 otherwise
>          makeOkflag pair = Ok, if pair $in ac
>                          = Notyet, otherwise
>          genCheckonBasicOp entName (update, interface, target, recordOp)
>                = [], if update = Delete
>                = genCheckonRecord entName update (hd recordOp), otherwise
>          genCheckonRecord entName update (recordOpName, scopeE, scopeR, input,
>                                                       pres, atomicOps)
>              = concat (map (makeCheckonAttribute entName update) atomicOps) ++
>                makeCheckonRecord recordOpName entName update
>          makeCheckonRecord recordOpName entName update
>              = [], if ~(lookupSchema schemalist
>                            (makeRecordConstsSchemaName entName))
>              = [(makeRecordConstsSchemaName entName,
>                  [recordOpName],
>                  makeSchemaRef update (makeRecordConstsSchemaName entName)
>                                       recordOpName,
>                  makeOkflag
>                    (makeRecordConstsSchemaName entName, [recordOpName]))],
```

```
>                                                                  otherwise
>         makeSchemaRef update constName opName
>              = [(NoDelta, constName, NoPrime),
>                 (NoDelta, opName, NoPrime)], if update = Modify
>              = [(NoDelta, opName, NoPrime)], if update = Insert
>         makeCheckonAttribute entName update (attrName, atomicOpName, data)
>             = [], if ~(lookupSchema schemalist
>                          (makeAttrConstsSchemaName entName attrName))
>             = [(makeAttrConstsSchemaName entName attrName,
>                 [atomicOpName],
>                 makeSchemaRef update (makeAttrConstsSchemaName entName attrName)
>                                 atomicOpName,
>                 makeOkflag (makeAttrConstsSchemaName entName attrName,
>                                   [atomicOpName]))], otherwise
>         makeCheckonR (subprocessName, subprocessData)
>             = makeCheckonR2 subprocessName subprocessData
>         makeCheckonR2 subprocessName (relName, scopeE, scopeR, input, pres,
>                                                                 basicOps)
>             = [(makeRelConstsSchemaName relName,
>                 [subprocessName],
>                 [(NoDelta, makeRelConstsSchemaName relName, NoPrime),
>                  (NoDelta, subprocessName, NoPrime),
>                  findEffectonE
>                    (hd [det | (det,mas,cor,consts) <- [assoc rels relName]]),
>                  findEffectonE
>                    (hd [mas | (det,mas,cor,consts) <- [assoc rels relName]])],
>                 makeOkflag (makeRelConstsSchemaName relName, [subprocessName]))]
>         makeCheckonUnchangedR (relName, (detName, masName, corType, consts))
>             = [], if relName $in (map fst affectedRs)
>             = [], if ~(relName $in (map fst affectedRs)) &
>                      ~(detName $in (map fst affectedEs)) &
>                      ~(masName $in (map fst affectedEs))
>             = [(makeRelConstsSchemaName relName,
>                 [],
>                 [(NoDelta, makeRelConstsSchemaName relName, NoPrime),
>                  (Xi, makeRelSchemaName relName, NoPrime),
>                  findEffectonE detName,
>                  findEffectonE masName],
>                 makeOkflag (makeRelConstsSchemaName relName, []))], otherwise
>         findEffectonE entName
>             = (NoDelta, assoc affectedEs entName, NoPrime),
>                             if entName $in (map fst affectedEs)
>             = (Xi, makeTableSchemaName entName, NoPrime), otherwise
>         findEffectonR relName
>             = (NoDelta, assoc affectedRs relName, NoPrime),
>                             if relName $in (map fst affectedRs)
>             = (Xi, makeRelSchemaName relName, NoPrime), otherwise
>         affectedEs = [(ent, name) |
>                          (name, data) <- subE;
>                          (ent, scopeE, scopeR, input, pres, basicOps) <- [data]]
```

```
>          affectedRs = [(rel, name) |
>                           (name, data) <- subR;
>                           (rel, scopeE, scopeR, input, pres, basicOps) <- [data]]
>          makeCheckonIC (integName, (scopeE, scopeR, consts))
>              = [(makeIntegSchemaName integName,
>                  [],
>                  [(NoDelta, makeIntegSchemaName integName, NoPrime)] ++
>                  map findEffectonE scopeE ++
>                  map findEffectonR scopeR,
>                  makeOkflag (makeIntegSchemaName integName, []))],
>                               if (notdisjoint scopeE (map fst affectedEs)) \/
>                                  (notdisjoint scopeR (map fst affectedRs))
>              = [], otherwise
```

Expand Proof Obligations

```
> expand :: num -> state -> ([char], state)

> expand i (db, pr, (schema, exp), cl)
>      = (output, (db, pr, (schema, exp), cl)),
>                  if ("LHSforProofObligation"++show i) $in (map fst schema)
>      = expand2 i (db, pr, (schema, exp), cl), otherwise
>      where output = "\n" ++ obligation ++
>                     "\nLHS\n=\n"++ showSchema lhs ++
>                     "\nRHS\n=\n"++ showSchema rhs
>            obligation = makelist " /\\\n" (map showSchemaRef schemaRefs) ++
>                         "\n\t|- " ++ const ++ "'\n"
>            (const, opName, schemaRefs, okflag) = cl ! i
>            lhs = assoc exp ("LHSforProofObligation"++show i)
>            rhs = assoc exp ("RHSforProofObligation"++show i)


> expand2 i (db, pr, (schema, exp), cl)
>      = (output, (db, pr, (schema2, exp3), cl))
>     where output = "\n" ++ obligation ++
>                    "\nLHS\n=\n"++ showSchema lhs2 ++
>                    "\nRHS\n=\n"++ showSchema rhs2
>          obligation = makelist " /\\\n" (map showSchemaRef schemaRefs) ++
>                       "\n\t|- " ++ const ++ "'\n"
>          (const, opName, schemaRefs, okflag) = cl ! i
>          lhs = (schemaRefs, [], [])
>          rhs = ([(NoDelta, const, Prime)], [], [])
>          schema2 = schema ++ [("LHSforProofObligation"++show i, lhs),
>                               ("RHSforProofObligation"++show i, rhs)]
>          (lhs2, exp2) = assocexpand schema2 exp
>                             ("LHSforProofObligation"++show i)
>          (rhs2, exp3) = assocexpand schema2 exp2
>                             ("RHSforProofObligation"++show i)
```

Reduce

```
> reduce :: num -> state -> ([char], state)

> reduce i  (db, pr, (schema, exp), cl)
>      = (msg, (db, pr, (schema, exp), cl))
>     where msg = "\nPlease expand it first.\n",
>                          if ~(("LHSforProofObligation"++show i) $in (map fst exp))
>                 = "\nOK, this proof obligation is satisfied.\n", if rest = []
>                 = "\nPlease check the following declarations " ++
>                   "and predicates in RHS:\n\n" ++
>                   concat (zipwith f ([0..], rest)), otherwise
>             rest = map show restdecl ++ map showExpoutSchema restpred
>             restdecl = setMinus decl2 decl1
>             restpred = setMinus pred2 pred1
>             (incl1, decl1, pred1) = assoc exp ("LHSforProofObligation"++show i)
>             (incl2, decl2, pred2) = assoc exp ("RHSforProofObligation"++show i)
>             f j pred = show j ++ ". " ++ pred ++ "\n"

> showExpoutSchema = showExp ""
```

CheckOK

```
> checkok :: num -> state -> state

> checkok i  (db, pr, sc, cl)
>      = (db, (prname, inputD, subE, subR, globalP, ac'), sc, cl')
>     where (prname, inputD, subE, subR, globalP, ac) = pr
>            cl' = take i cl ++
>                  [(const, opName, schemaRefs, Ok)] ++
>                  drop (i + 1) cl
>            (const, opName, schemaRefs, okflag) = cl ! i
>            ac' = ac, if (const, opName) $in ac
>                = ac ++ [(const, opName)], otherwise
```

Utility

```
> notdisjoint :: [a] -> [a] -> bool

> notdisjoint xs ys = or (map (member xs) ys)
>         where member xs y = y $in xs

> strtonum :: [char] -> num

> strtonum = foldl op 0
>    where op n ch = 10 * n + code ch - code '0'

> makelist :: [char] -> [[char]] -> [char]
```

```
> makelist c cs = "", if cs = []
>               = foldr1 (op c) cs, otherwise
>     where op c s1 s2 = s1 ++ c ++ s2

> addpair :: a -> [a] -> [a]

> addpair x xs = [x] ++ xs

> setUnion :: [a] -> [a] -> [a]

> setUnion xs [] = xs
> setUnion xs (y:ys) = setUnion xs ys,         if y $in xs
>                    = setUnion (xs++[y]) ys,  otherwise

> setMinus :: [a] -> [a] -> [a]

> setMinus [] ys = []
> setMinus (x:xs) ys = setMinus xs ys,         if x $in ys
>                    = [x] ++ setMinus xs ys,  otherwise

> subst :: [(a, a)] -> [a] -> [a]

> subst [] list = list
> subst ((x,y):ps) list = subst ps (subst1 (x,y) list)

> subst1 (x,y) [] = []
> subst1 (x,y) (c:cs)
>       = [y] ++ subst1 (x,y) cs, if x = c
>       = [c] ++ subst1 (x,y) cs, otherwise

> assoc :: [(a, b)] -> a -> b

> assoc xys x = hd [y|(x', y) <- xys; x' = x]

> delete :: a -> [a] -> [a]

> delete x [] = []
> delete x (y:ys) = delete x ys,         if y = x
>                 = [y] ++ delete x ys,  otherwise
>
```

Main

```
> command
>  ::= End | ReadSpec [char] | Schemalist | View nameType | View2 nameType |
>      Checklist | Expand num | Reduce num |
>      CheckOK num | Showdb | Showpr | Showsc | Showcl |
>      Nullcmd | Errorcmd [char]
```

Showdb, Showpr, Showsc and Showcl are commands for debugging.

```
> loop st = readedit prompt (execute st. readcmd st)

> prompt = "\nreadspec filename/schemalist/view num/view2 num/\n" ++
>          "checklist/expand num/reduce num/checkok num/end => "

> readcmd st = analyse st . words

> analyse st [] = Nullcmd
> analyse st [cmd] = End,          if cmd = "end"
>                  = Schemalist,   if cmd = "schemalist"
>                  = Checklist,    if cmd = "checklist"
>                  = Showdb,       if cmd = "showdb"
>                  = Showpr,       if cmd = "showpr"
>                  = Showsc,       if cmd = "showsc"
>                  = Showcl,       if cmd = "showcl"
>                  = Errorcmd cmd, otherwise
> analyse st [cmd, par] = ReadSpec par,                  if cmd = "readspec"
>                       = View (tobeviewed st par),      if cmd = "view"
>                       = View2 (tobeviewed st par),     if cmd = "view2"
>                       = Expand (tobeexpanded par),     if cmd = "expand"
>                       = Reduce (tobeexpanded par),     if cmd = "reduce"
>                       = CheckOK (tobeexpanded par),    if cmd = "checkok"
>                       = Errorcmd cmd,                  otherwise
> analyse st (cmd:par:pars:xs) = Errorcmd cmd

> tobeviewed (db, pr, (schema, exp), cl) x
>        = fst (schema ! (strtonum x))

> tobeexpanded x
>        = strtonum x

> execute st End              = end
> execute st (ReadSpec filename)
>           = write "" (loop (generateChecklist
>                              (generateSchemas (readSpec filename))))
> execute st Schemalist      = write (schemalist st) (loop st)
> execute st (View name)     = write (view name st) (loop st)
> execute st (View2 name)    = write (fst pair) (loop (snd pair))
>                                where pair = (view2 name st)
> execute st Checklist       = write (checklist st) (loop st)
> execute st (Expand i)      = write (fst pair) (loop (snd pair))
>                                where pair = (expand i st)
> execute st (Reduce i)      = write (fst pair) (loop (snd pair))
>                                where pair = (reduce i st)
> execute st (CheckOK i)     = write "" (loop (checkok i st))
> execute st Showdb          = write (show db) (loop st)
>                                where (db, pr, sc, cl) = st
```

```
> execute st Showpr          = write (show pr) (loop st)
>                                where (db, pr, sc, cl) = st
> execute st Showsc          = write (show sc) (loop st)
>                                where (db, pr, sc, cl) = st
> execute st Showcl          = write (show cl) (loop st)
>                                where (db, pr, sc, cl) = st
> execute st Nullcmd         = loop st
> execute st (Errorcmd cmd) = write ("The command '" ++ cmd ++ "' is unknown\n")
>                                  (loop st)

> run tool = loop initialstate (keyboard tool)

> tool = False
```

The followings are utility functions in [R.Bird & P.Wadler 88]:

```
> breakon a x xss = [[]] ++ xss, if x = a
>                 = [[x] ++ hd xss] ++ tl xss, otherwise

> words = filter (~= []). foldr (breakon ' ') [[]]

%> read msg g input = msg ++ line ++ "\n" ++ g line input2
%>                    where line   = before '\n' input
%>                          input2 = after '\n' input

> before x = takewhile (~= x)
> after x  = tl . dropwhile (~= x)

> write msg g input = msg ++ g input

> end input = ""
```

Readedit (for the use of Backspace in command line)

```
> readedit msg g input = msg ++ println ++ "\n" ++ g inputline input2
>                     where printline = printbs line 0
>                           inputline = fst (inputbs line)
>                           line      = before '\n' input
>                           input2 = after '\n' input

> printbs :: string -> num -> string

> printbs "" clmn = ""
> printbs (c:cs) clmn = c:printbs cs (clmn + 1),         if c ~= '\b'
>                     = "\b \b" ++ printbs cs (clmn - 1), if c = '\b' & clmn > 0
>                     = printbs cs 0,                     if c = '\b' & clmn = 0

> inputbs :: string -> (string, num)
```

```
> inputbs "" = ("", 0)
> inputbs (c:cs) = ((c:cs1), level), if c ~= '\b' & level = 0
>                = (cs1, level - 1), if c ~= '\b' & level > 0
>                = (cs1, level + 1), if c = '\b'
>       where (cs1, level) = inputbs cs
```