Business Rules and iData: A Powerful Combination?

Bachelor thesis Computer Science

Jordy Voesten 0228362

Supervisors:
Dr. Peter Achten
Dr. Marko van Eekelen

June 11, 2007

CONTENTS

1. Introduction 5
1.1 Situation description 5
1.2 Research structure 7

2. Business Rules 9
2.1 What are Business Rules? 9
2.2 Why Business Rules? 10
2.3 Three kinds of Business Rules 11
2.3.1 Structural assertions 11
2.3.2 Action assertions or Constraints 12
2.3.3 Derivations 12
2.3.4 A different view 12
2.4 Case: Insurance Business Rules 12
2.5 Related work 14

3. ADL 16
3.1 Relational algebra 16
3.2 ADL explained 18
3.2.1 Defining a relation 18
3.2.2 Business Rules in ADL 19
3.2.3 Case: Insurance in ADL 20
3.3 What does ADL do? 22
3.4 Looking back 25

4. iData 26
4.1 The iData Toolkit 26
4.2 Generic programming 26
4.3 What are iData? 28
4.4 Using iData 29
4.5 Similarities and differences with ADL 30

5. Combination posibilities 32
5.1 ADL on its own 32
5.2 From ADL to iData 33
5.2.1 Cartesian Product 35
5.2.2 Converse 37

5.2.3 Sequential Composition 38
5.2.4 Condition 39
5.3 Example 39
5.4 Outcome 41

6. Conclusion 42
6.1 Summary 42
6.2 Future work 42
6.3 Possible consequences 43

Bibliography 44

Appendix 46

A. Business Rules Manifest 47

B. Insurance in iData 50

FOREWORD

This foreword is a more personal note on why I wrote my thesis on this subject.

As the completion of my Bachelor was nearing, I was trying to find a subject for my thesis that explored the part of computer science that has captured my interest most: Software Engineering. I wanted to see if this was indeed a direction that I liked. Still strugling to find my path in the world of computer science, I now feel that I have at least come a big step closer to the right path and will continu to do so in my Master studies.

I would like to thank the people that have made it possible for me to complete this thesis: First of all Professor Stef Joosten from the Open University, the Netherlands for taking time out of his very busy schedule to help me with understanding Business Rules and ADL in particular, without getting anything in return. Secondly my supervisors Dr. Peter Achten and Dr. Marko van Eekelen from ICIS at the Radboud University, Nijmegen for guiding me on the right path and being patient with me, for it took some time for me to complete this thesis. Last but certainly not least my family and friends for supporting me in everything that I do. Without you I would never have succeeded in doing this.

I hope you enjoy reading it.

1. INTRODUCTION

> We know why projects fail, we know how to prevent failure – So why do they still fail?
>
> Martin Cobb

The quote above is called Cobb's Paradox [12]. Martin Cobb is a member of the Standish Group[24]. They specialize in research about project failure in IT and have published a number of important papers about how many IT projects fail and why. Their research will be discussed in the next section. The concept of Business Rules might be a solution to the problems encountered in projects during this research. In short: the concept of Business Rules is a way to formalise the rules a business has to comply to. This is a relatively new concept and has gained popularity in recent years. This thesis will look at a way of expanding the functionality of one of the systems that is now used to check these rules, by building that same functionality and more in a different system. Why and how this is done is explained a bit further in section 1.2, followed by the actual questions this thesis will answer.

1.1 Situation description

The results of the Standish Group's research on project failure were staggering: In their initial CHAOS report[23] in 1994, it was researched that 80% of all projects failed! This failure means that they were 20% or more over budget, 20% or more late, and failed to meet 20% or more of the business requirements for the system. You may think that this was a long time ago, and things are far better now, but you would be wrong. In one of the follow-ups in 2003 the failure rate was still at 66% . This is of course a big step forward, but there is still a very long way to go. As you can see in Figure 1.1, still 15% of the projects fail completely or are cancelled prior to completion, 52% fail to meet objectives and 43% have a substantial cost overrun.

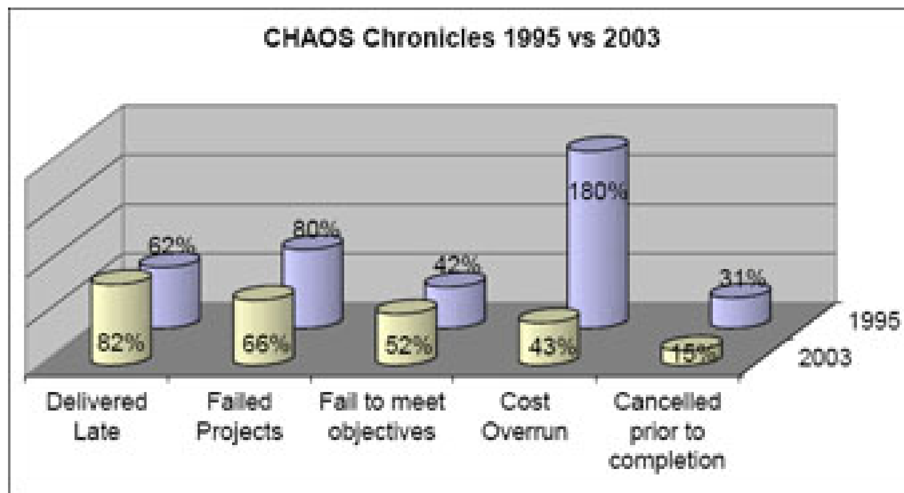

Figure 1.1 [24]

So overall less projects fail, and cost overrun has been greatly reduced, but even more projects are delivered late or fail to meet objectives, so Columbus´ egg has hardly been found. At best, 30% of projects are concidered really succesful. This is also acknowledged by Lewis[8] and King[7]. Next to researching the numbers, the reasons for failure have also been looked into. The outcome of the CHAOS research[23] featured a list of the top reasons why these projects fail:

R1 : Lack of User Input and Involvement
If the people who will use the system are not involved in the development, obvious faults in the system will not be detected until it is too late.

R2 : Incomplete Requirements & Specifications
If the requirements are incomplete, then the system will not comply to the users demands.

R3 : Changing Requirements & Specifications
If the requirements change in the course of the project and the system is still being built by the first requirements, then it will not comply to the users demands.

R4 : Lack of Executive Support
If management does not support the project, then delays can arise through bureaucracy and managers having the wrong priorities.

R5 : Unrealistic Expectations
If the expectations are unrealistic, it will be impossible to build a system that is done on time and within budget because it simply can't be done.

1.2 Research structure

What the concept of Business Rules is and why they are a possible solution to the problems stated above, will be discussed in detail in chapter 2. For now: Business Rules are a way of formally describing the rules a business has to comply to. The why part will only be briefly looked into, because the main goal of this thesis is to investigate the combination of two existing systems in order to expand the use of Business Rules. The systems are:

* ADL
  While a lot of people have written about Business Rules, only few have operationalized these rules. Professor Stef Joosten (Open University, the Netherlands and Partner of ICT consultant Ordina) is one of the few that have. His approach aims at formally checking whether a certain dataset complies with the predefined Business Rules. He has built a system called ADL which does exactly that. ADL stands for: A Description Language, so ADL is the name of the language *and* the name of the program. *ADL* will therefore be used if we talk about the language and *ADL-t* if we talk about the tool. Using relational algebra, a user can define invariant Business Rules. It is not (yet) possible to check calculating Business Rules. After running the program the system indicates which rules are violated and why. The user can then adapt the data accordingly and check again. It also generates a graphical overview of the relations in the dataset, which makes it easier to deduce new Business Rules. ADL will be discussed in chapter 3.

* iData
  iData is a dynamic web–system, built in the functional programming language Clean [19]. The choice of ADL is obvious: it is one of the only systems to work with Business Rules. The choice of iData as an alternative for using the rules may seem odd to some. The reason iData was chosen for this is twofold: First and foremost it was chosen because of the similarities between the two approaches. Both ADL and iData look at the data at hand at a certain time, and define everything by that dataset. So they both run, deliver an output, look at the changes the user makes, and run again. Next to that iData is written in a functional programming language and this makes it very suitable for trying to implement the calculating part of the Business Rules (section 2.3.4), the part ADL can not implement at the moment. Second it was chosen because of the available expertise on the system in this Faculty and my personal interest in functional programming methods. iData will be discussed in chapter 4.

The main question this thesis will answer is the following:

* Is the combination between ADL and iData a powerful, hence useful one?

This question can be broken down into two sub-questions:

(i) Is it possible to build the current Business Rules functionality from ADL-t in iData?

(ii) Is it possible to expand that functionality with the calculating part of Business Rules?

If the answer to both questions is yes, then the combination will have improved the Business Rules functionality.

Chapter 5 will examine the possibilities of combining the two systems, propose a choice on how the ADL–iData link would have to be made and will answer the main questions of this thesis. Conclusions will be summarized in chapter 6.

2. BUSINESS RULES

This chapter will explain what Business Rules are and why it would be a good idea to use them. Then it will present a case in which Business Rules can be used for the modeling of an information system and after that how they can be made operational. As a conclusion it will show an overview of other developments in the field.

2.1 What are Business Rules?

Business Rules are the result of the Business Rule Approach (BRA), which is meant to make implicit assumptions about a business explicit. This approach is the answer to a problem encountered by information analysts: While describing the nature of an organisation, a couple of important rules are taken for granted or neglected. They only surface when it is time to put the information about the business in code in the system. At that time it is probably to late to incorporate them and the programmer is left with the choices that are supposed to be made at a much higher level, by a business analyst. The programmer is not capable of making these choices, for this is not his job. The choices he then makes, can lead to big problems while using this new information system. So the neccesity for these choices to be made at a higher level is obvious.

The BRA was invented by *The Business Rules Group* [22]. In this group are a number of well-known information scientists, such as John A. Zachman (former IBM), Dr. Terry Halpin (well-known for his book on ORM modelling) and Ronald G. Ross, who is seen as the father of Business Rules. They regularly organize conferences on Business Rules and cooperate with the Object Management Group (OMG, `http://www.omg.org/`) and the World Wide Web Consortium (W3C, `http://www.w3.org/`) for the development of standards on the rules.

The concept of Business Rules is a simple one, but using them can be complicated. Business Rules can be defined as rules of your business, everything that you consider while running it. That makes it a very broad definition. It not only encompasses your outspoken rules, but also the rules embedded in your computer systems and the unspoken rules everyone in the company lives by. The rules simply tell an organisation what it can do. An example of such a rule for an insurance company could be:

Customers must have a known address, social security number and bank account.

This is a simple rule, which can easily be embedded in your computer systems. But what if the rule you're trying to implement is a rule almost none of your employees know, like:

If a customer has both home and car insurance with us, then he receives the following discount: 7% annually on the car insurance, 5% on the home insurance if the insurance is over 2,000.- and 10% on the home insurance if it is 2,000.- or less.

Then it becomes clear why it would be nice to pre-define all these rules, so not everyone has to know all the different rules by heart, as long as the system does. The official definition from an information system point of view is:

> A business rule is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behavior of the business. The business rules which concern the project are atomic: that is, they cannot be broken down further. [20]

The point, of course, is to identify the rules that are required for a business to function, document them, make them operational and automate their management. For Business Rules to be useful in the field, it was neccesary to make strict regulations on how they should look like. This is called *The Business Rules Manifesto* [21] which is included in appendix A of this thesis.

2.2 Why Business Rules?

The greatest benefit here is the separation from the Information System itself. Separating the two and putting Business Rules on a higher level makes it easier to change the rules and lets you see what part of the information system you need to change in order to comply with the rules again. This promotes reuse of the systems, because it is much clearer what exact rules the system has to live by, so it's much easier to view the whole system as a black box. As you could read in chapter 1, the reason so many projects fail is the fact that requirements and specifications are incomplete and change during the course of the project, users are not involved in the development and there is a lack of executive support. These problems can probably be greatly reduced using Business Rules. $R1$: If it is clear to the users what the Business Rules are and they agree with them, then they don't need to be involved in the implementation choices. $R2$: If the entire business is contained in the Business Rules, then the requirements and specifications are clear: The system has to comply with these rules in order to realise all the

business goals set for it. *R*3: If requirements change, then it's a lot easier to change the Business Rules, then it would be to change the specification and code, because the rules are set at such a high level it doesn't take much adaptation. *R*4: Finally, a lack of executive support comes from the fact that the executives don't have the technical knowledge to understand what the project is really about. By using Business Rules, they don't need to have that technical knowledge, because they understand the business and that is exactly what the rules supply: a business point of view the system is being built by.

Ronald Ross explains the key factor: [2] "The primary goal today is agility. Rules need to change as quickly as your business changes. Your business analysts want to be able to get their hands on rules without digging through mountains of code to do it. They need to evaluate the impact of rule changes quickly so you can quickly determine the route you want to take. Its literally impossible to operate a business at scale, across the globe, without automation."

2.3 Three kinds of Business Rules

Now that we've seen what Business Rules are, and why we want to use them, let's see what kinds of Business Rules there are. If you search for information on the different kinds of Business Rules, there is no general agreement on what kinds there are. While most people talk about three kinds, not everyone divides these three in the same way. After looking at a number of different options and opinions [10, 4, 5] the most complete and thorough division is the following one: Structural assertions, Action assertions or Constraints and Derivations as the three different kinds. Some see four, but that is because they divide structural assertions into two sorts. They will each be explained separately.

2.3.1 Structural assertions

Structural assertions are the base of the business. They can be divided into *Terms* and *Facts* and are called structural assertions because they express some aspect of the structure of an enterprise. Terms are the basic building entities of your business, for example in an insurance company, a term could be Customer, Insurance, Package or Car (for car insurances). Facts relate terms to each other by defining a relationship between two or more terms. In that same example a fact could be: "Customer has Car". Facts are also divided into two kinds: *Base Facts* and *Derived Facts*. Base facts are those facts that are basic like the example above, but derived facs are the result of a Derivation Rule, which will be discussed later in this chapter (2.3.3). A derived fact could be: "Car is Oldtimer".

2.3.2 Action assertions or Constraints

Action assertions or constraints are those rules which limit or describe behaviour. They state a constraint or condition that limits or controls the action of the enterprise and thus concern a dynamic aspect of the business. They are mainly "If...then..." statements that set a condition or authorization. Action assertions could be: "If a customer wants to buy car insurance, then he or she must have a drivers license" (condition) or "A customer may only have car insurance for a maximum of 5 cars" (authorization)

2.3.3 Derivations

The last kind of Business Rules are the Derivations. Derivations are either a *Mathematical Calculation* or *Inference*. The first one produces a derived fact according to the given calculation, for example: "Car insurance costs = standard insurance costs + added package costs". The second one also produces a derived fact, but now it is inferred from other data: "Car is Oldtimer" if it is older than 25 years, so we have to use the current date and the build date of the car.

2.3.4 A different view

As was said before, this is the most complete and thorough division made in Business Rules, but for Operationalizing Business Rules in ADL it is required that we also make an orthogonal division. This means grouping Business Rules as either *Invariant* or *Calculating*. This is neccesary because Invariants are always true, and this can be checked by the system. Calculating Business Rules need more steps in time for the system to comply to all the rules again. These two types feature in all the different kinds mentioned above. For example, the structural assertion "Customer has Car" is an invariant in the system that can be checked, but the derived fact "Car is Oldtimer" needs the system to calculate the car's age in order to check if this is true. The action assertion "Customer has Drivers License" is also invariant, but the authorization rule for insuring a maximum of five cars is calculating. As an exception, derivations are always calculating rules, because even in the *Inference* mentioned above, the age of the car has to be calculated.

2.4 Case: Insurance Business Rules

To give an example of how business rules can be applied, inspiration was found in a column written by Ronald G. Ross about personal insurance [11]. This case will be used as a running example for the entire thesis, pieces will be used to explain ADL and iData and in the combination between them.

We will not cover all the bases for a complete analysis, but try to give some basic business rules for an insurance company. What would the business rules for a personal insurance look like?

Lets say the insurance company offers the following kinds of insurance:

1. Car insurance
2. Health insurance
3. Life insurance

These are divided in subcategories, for example: car insurance is divided into regular cars and old-timers. Also, there are packages to choose from, such as:

1. New car replacement: If you total your new car, it will be replaced by a car of the same make and model.
2. Accident forgiveness: If you crash your car, the insurance rate will not go up.
3. Safe driving bonus: For every six-month period you don't have an accident, you can earn a discount on your insurance rate.

Furthermore, if you choose more than one insurance from this company, you get a discount. If you buy life insurance and car insurance, you get a 7% discount annually on car insurance and up to 10% on your home insurance. If you also buy life insurance, then the home discount can get up to 20% and the life insurance discount up to 10%.

Lets look at the business rules to go with this. First some rules to define our customers:

1. A customer must be at least 18 years old
2. Customers must have a known address, birth date, social security number and bank account.

Now, lets look at the car insurance:

1. A customer who wants to buy a car insurance must have a valid drivers license.
2. A customer has to own a car of the make and model the insurance is for.
3. Old-timer insurance is only possible for cars older than 25 years.
4. An insured car must have a make, model, serial number, list price, current value and build date.

5. The New car replacement package is only possible when a customer buys a new car and is only valid for the first year after the build date of the car.

6. The Accident forgiveness package is only possible if a customer is at least 24 years old.

A business rule for health insurance might be:

1. New customers who want to buy health insurance must be checked by an approved doctor, before the insurance is granted.

Discount business rules could be:

1. If a customer has both home and car insurance with us, then he receives the following discount: 7% annually on the car insurance, 5% on the home insurance if the insurance is over 2,000.- and 10% on the home insurance if it is 2,000.- or less.

2. If a customer has home, car and life insurance with us, then he receives the following discounts: 7% annually on the car insurance, 10% on the home insurance if the insurance is over 2,000.- annually and 20% on the home insurance if it is 2,000.- or less annually, 5% on life insurance if it is over 3,000.- annually and 10% if it is under 3,000.- annually.

This concludes the example.

2.5 Related work

As you can see in the picture on the next page, Ordina is not the only organisation working with Business Rules. Oracle also sees great benefits in working with the rules and have developed The Oracle Business Rules Components [18], which consist of a Rules engine, a Rules SDK, and a Rule Author Tool. Then there is JESS [16], the Rule engine for Java, and ILOG JRules, also for Java [15]. IBM is working on a Business Rules for E-commerce projects and this includes a new Business Rule Markup Language (BRML). It is used in connection with IBM's CommonRules [14]. Another name often encountered when searching for Business Rules is Fair Isaac's Blaze Advisor, which is part of their Enterprise Decision Management suite of software and services [13]. As you can see Business Rules are a hot item, because they might be the able to solve some of the major problems Software development is facing today.

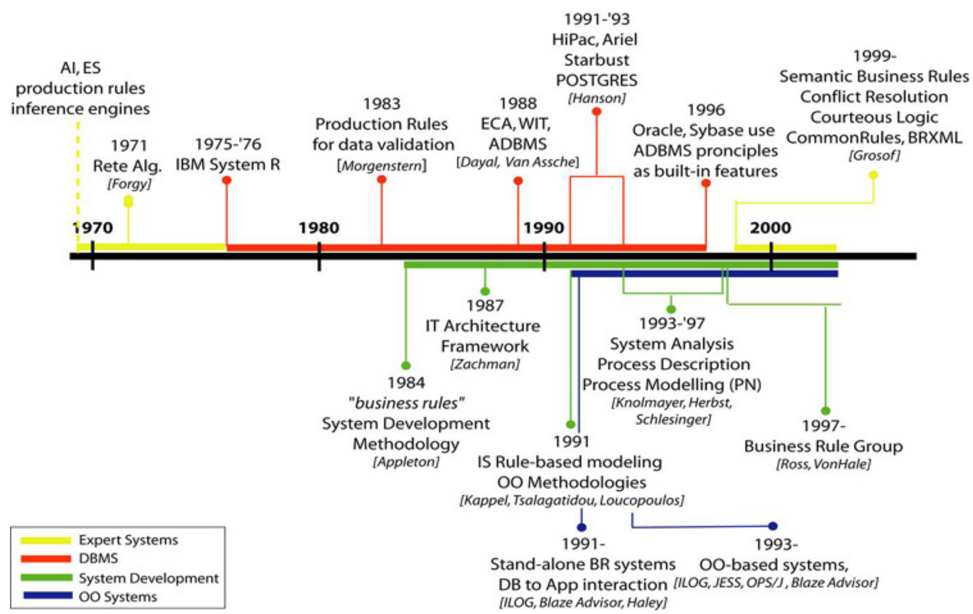

Figure 2.1 An overview of Business Rules development[5].

3. ADL

As the introduction stated, ADL (A Description Language) is a language for defining Business Rules and a program for specifying and checking Business Rules, by using relational algebra. So before continuing with ADL, the next section will explain some of the details of this algebra. This will be limited to the most basic parts, just enough to be able to read and understand ADL. If you are not familiar with the theory, it is advised to read one of the many books on relational theory, for instance [3].

3.1 Relational algebra

Relational algebra is used to manipulate relations effectively, quickly and without mistakes. In order to define these relations, the reader has to be familiar with *Sets* and *Set theory*. These are the four basic relations and operators on sets that are useful when looking at relational algebra:

- equality: $X = Y$ if X and Y contain the same elements
- inclusion: $X \subseteq Y$ if every element of X is also an element of Y
- union: $X \cup Y$ is the set of elements in X or Y
- intersection: $X \cap Y$ is the set of elements in X and Y

Relations are the basic entity, used to build specifications. They allow us to talk about the contents of a set, without actually quoting those contents. Any relation has a *name*, *source* and a *target*. A relation is a Cartesian Product. This means it is a set of all possible ordered pairs whose first component is a member of the *source* of the relation and the second component is a member of the *target* of the relation. In the case of these relations, it is not all the possible combinations, but a subset defined by the instances of these relations. For instance the relation *hasLicense* (relations start with lower case letters) has *Customer* as its source (source and target start with upper case letters) and *DriversLicense* as its target. Together, name, source and target are called the signature of the relation, written as:

$hasLicense : Customer \times DriversLicense$

A possible instance of this relation is ("Voesten", "NL313360") where Voesten

is the name of a *Customer* and NL313360 is the number of the customer's *DriversLicense*. This can also be written as:

Voesten hasLicense NL313360

Operators are used to manipulate these relations. We already saw the $\subseteq, \cap$ and $\cup$ operators from set theory, but those alone are not enough. The two most important operators we also need are the *converse* of a relation and the *(sequential) composition* of two relations. The converse of a relation simply swaps the source and target of a relation and is denoted as $\tilde{r}$, where r is the relation we want to converse. The converse of *hasLicense* would then be:

$hasLicense^{\sim} : DriversLicense \times Customer$

Composition of two relations is denoted as ; and makes one relation of its two arguments. So it is a way to glue two relations together as one. So if there are two relations: $r : A \times B$ and $s : B \times C$ then the composition of those two would be: $(r; s) : A \times C$. As you can see, for composition to work, the target of the first relation (B) has to be the same as the source of the second relation. In the following example, both converse and composition are used to build a new relation called *carInsurance*. This relation is a composition of $hasLicense^{\sim} : DriversLicense \times Customer$ and $hasCar : Customer \times Car$, and is denoted as:

$carInsurance$ -: $hasLicense^{\sim}$;$hasCar$

and thus has the following signature:

$carInsurance : DriversLicense \times Car$

Another part of relational algebra that ADL uses are the *multiplicities*. These are:

- univalent: A relation r : $A \times B$ is *univalent* if each element of A corresponds to *at most one* element of B.
- total: A relation r : $A \times B$ is *total* if each element of A corresponds to *at least one* element of B.
- function: A relation is a *function* if it is both univalent and total. So every element of A corresponds to *exactly one* element of B.
- injective: A relation r : $A \times B$ is *injective* if each element of B corresponds to *at most one* element of A.

- surjective: A relation r : $A \times B$ is *surjective* if each element of B corresponds to *at least one* element of A.
- bijective: A relation is *bijective* if it is a function that is both injective and surjective. So every element of B corresponds to *exactly one* element of A.

Now that we know the basics of relational algebra, let's look at the structure of ADL in the next section.

3.2 ADL explained

So how does ADL work? It's fairly simple: a user takes his Business Rules in one hand, and the company's data in the other. In our example, the user works at an Insurance company as we saw in our case (2.4). He then devides his data and rules into smaller pieces, let's say everything about a *Customer*: all rules about customers and all data about them. He then separates the *Invariant Business Rules* from the *Calculating Business Rules* (2.3.4), because ADL can only handle the first group. He has now narrowed it down enough to put this information in an ADL *Pattern*.

A pattern in ADL is an entity within the program. Patterns are stored within a *Context*. A context is the biggest building block in an ADL program. It has a name and contains one or more patterns. In our case, the name of the context would be *Insurance* and the pattern would be named *Customer*. If for example the insurance company would also start doing some banking, this could still be kept separately from the insurance rules by defining a new context called *Banking*, but still check all the rules using a single program. This makes it very adaptable to changes in the business.

3.2.1 Defining a relation

Back to our user: since our case only contains a handful of Business Rules, it is enough to define a single context containing a single pattern. He now has to define all relations and at the same time fill them with the data about those relations. For *hasLicense* this would look like this:

```
hasLicense  :: Customer * DriversLicense    [INJ,SUR]
PRAGMA "Customer " " has drivers license " " ,
    according to the legal authority"
 = [ ("Voesten", "NL313360")
   ; ("Pepels", "NL377748")
   ; ("Simon", "NL395504")
   ].
```

The [INJ,SUR] part means that the relation is injective and surjective. The PRAGMA keyword is followed by an explanation about this relation. After that the user defines a number of instances according to the data he

has, and the first relation is done. We need these simple relations to define parts of our Rules. After repeating this for all simple relations, the more complicated Business Rules are next.

What we end up with is a list of rules and a population for these rules that needs to be checked, defined in the ADL file. We run the program and the outcome is a model of these rules and a check whether all defined instances comply to these rules. What this lookes like can be seen in section 3.3. So ADL is a check at a given moment, not a system that runs while you change the rules.

3.2.2 Business Rules in ADL

If we now take another look at the Business Rules from our case (2.4), we will see that the rules are all compositions of simple relations. For instance the rule:

A customer who wants to buy car insurance must have a valid drivers license.

uses the relation *hasLicense* we saw earlier and the rule:

A customer has to own a car of the make and model the insurance is for.

uses the relation *hasCar*. To be sure the data complies with both rules before a customer can buy the insurance, we use the composed relation we saw earlier:

$carInsurance$ -: $hasLicense$~ ;$hasCar$

In ADL this would look like:

```
carInsurance -: hasLicense~;hasCar
 EXPLANATION "A Customer has to own the car he
 wants to insure, and has to have a drivers license."
```

The EXPLANATION keyword lets us explain what this rule is about, which makes it easier for others to read. For this composed relation, it is also necessary to define the instances of the data, so they can be checked for compliance:

```
carInsurance :: DriversLicense * Car    [INJ]
 PRAGMA "Customer with License " " bought insurance for car "
  = [ ("NL377748", "BMW325")
    ; ("NL313360", "AMDB9")
    ; ("NL313360", "Porsche911Turbo")
    ].
```

So now all Invariant Business Rules can be built using the simple relations as pieces in a puzzle. The next section shows us what the entire case would now look like, using ADL to check as many rules as possible.

3.2.3 Case: Insurance in ADL

The Business Rules case from 2.4 in ADL:

```
CONTEXT Insurance
PATTERN Customer

 carInsurance -: hasLicense~;hasCar
  EXPLANATION "A Customer has to own the car he
  wants to insure, and has to have a drivers license."

 carPackage -: hasCar;typeOfCar;typeForPackage
  EXPLANATION "A Customer can only have the
  carPackage for the type of car he owns"

 healthInsurance -: checkedBy;approvedDoc
  EXPLANATION "A customer must be checked by
  an approved doctor to buy health insurance"

 healthInsurance :: Customer * Verdict
  = [ ("Voesten", "approved")
    ].

 checkedBy :: Customer * Doctor
  = [ ("Voesten", "Potgens")
    ].

 approvedDoc :: Doctor * Verdict
  = [ ("Potgens", "approved")
    ].

 carInsurance :: DriversLicense * Car   [INJ]
 PRAGMA "Customer with License " " bought insurance for car "
  = [ ("NL377748", "BMW325")
    ; ("NL313360", "AMDB9")
    ; ("NL313360", "Porsche911Turbo")
    ].

 carPackage :: Customer * PackageType
 PRAGMA "Customer " " bought package "
  = [ ("Hammond", "Oldtimerdiscount")
    ; ("Voesten", "NewCarReplacement")
    ].

 typeForPackage :: TypeOC * PackageType [INJ]
```

```
PRAGMA "The Type of Car " " is required for package "
 = [ ("New", "NewCarReplacement")
   ; ("Oldtimer", "Oldtimerdiscount")
   ].

hasLicense  :: Customer * DriversLicense    [INJ,SUR]
PRAGMA "Customer " " has drivers license " " ,
    according to the legal authority"
 = [ ("Voesten", "NL313360")
   ; ("Pepels", "NL377748")
   ; ("Simon", "NL395504")
   ].

hasCar :: Customer * Car   [INJ,SUR]
PRAGMA "Customer " " is owner of car "
 = [ ("Pepels", "BMW325")
   ; ("Simon", "Mazda323F")
   ; ("Voesten", "AMDB9")
   ; ("Hammond", "FordMustang")
   ; ("Voesten", "Porsche911Turbo")
   ].

builddate :: Car -> Date
PRAGMA "Builddate of " " is "
 = [ ("BMW325", "27101988")
   ; ("BMW525tds", "06071996")
   ; ("Mazda323F", "05012001")
   ; ("FordMustang", "19031976")
   ; ("AMDB9", "09112006")
   ; ("Porsche911Turbo", "07101995")
   ].

typeOfCar :: Car * TypeOC
PRAGMA "Car " " is of type "
 = [ ("FordMustang","Oldtimer")
   ; ("AMDB9", "New")
   ].

birthdate  :: Customer -> Date
PRAGMA "Birthdate of " " is "
 = [ ("Voesten", "30051983")
   ; ("Pepels", "04121982")
   ; ("Simon", "03071952")
   ; ("Blokpoel", "03031984")
   ; ("Hammond", "19121969")
   ].

ENDPATTERN
ENDCONTEXT
```

3.3 What does ADL do?

The program makes a visual graph and a datamodel of all the relations in the pattern. It checks whether the data complies to all the rules. If this is not the case it tells you which rule was violated and how. The user can then alter the data and run the program again. This may lead to new violations, which require new actions by the user. If no new violations occur, the data complies to all Business Rules. The program looks like this:

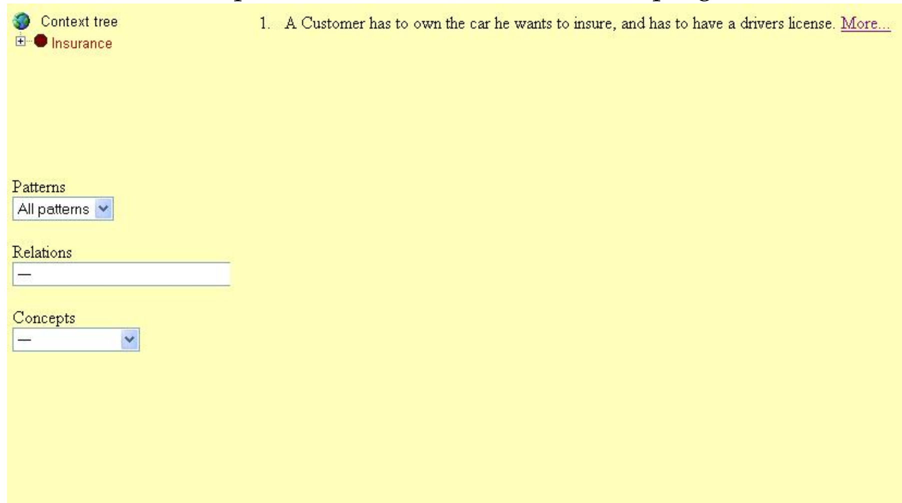

Figure 3.1 The main window
The main window shows the user an overview of all patterns in the ADL file.

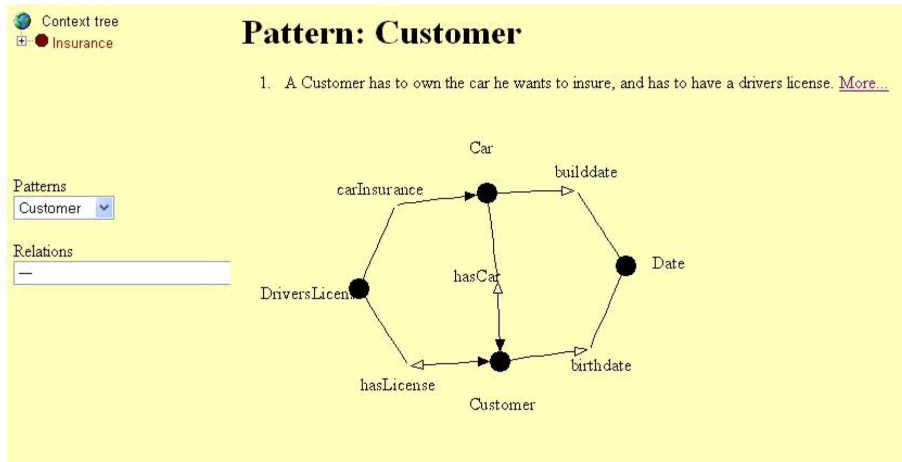

Figure 3.2 Customer pattern in ADL-t
In a pattern the user sees a graph of the rules in that pattern.

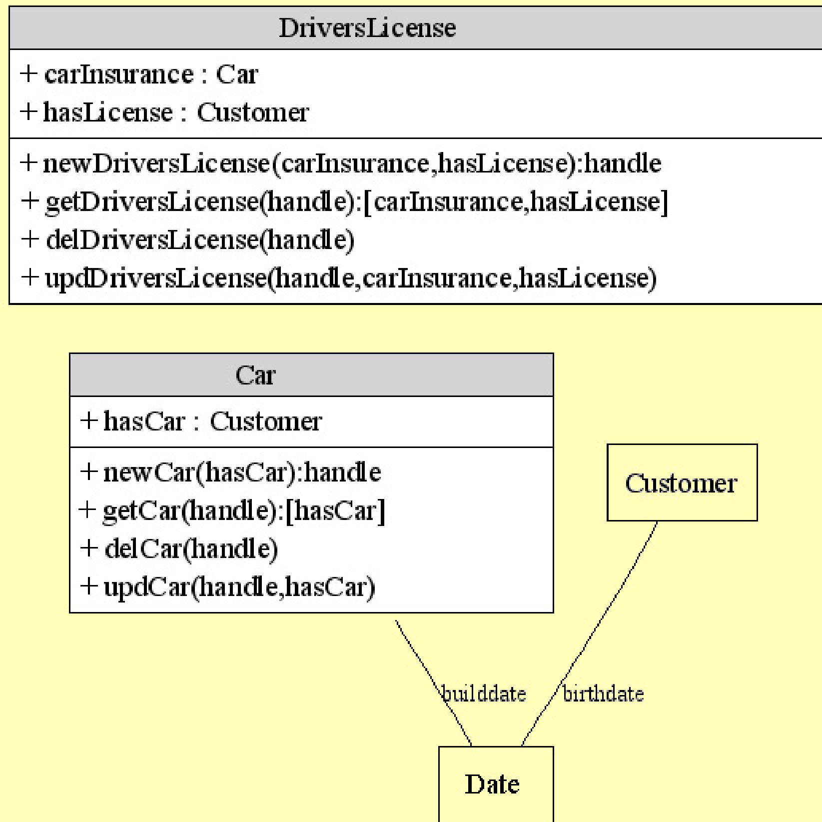

Figure 3.3 Customer data model in ADL-t

For each part of a relation (the black dots in the pattern graph) a data model is derived.

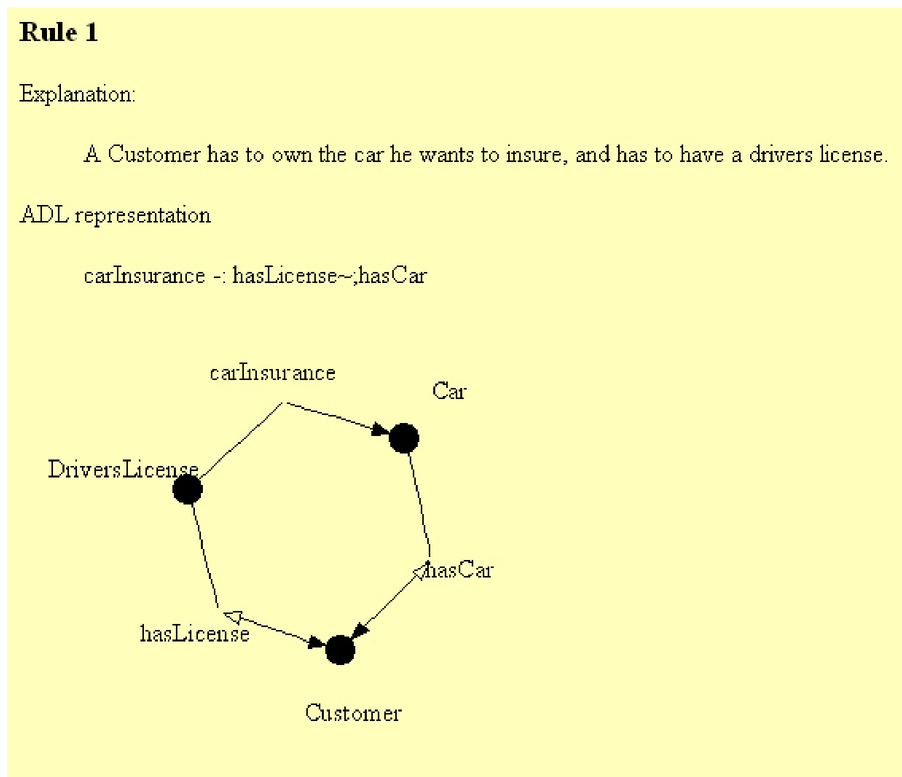

Figure 3.4 carInsurance business rule in ADL-t
Each rule can be separately viewed as a graph.

Context tree
Insurance

Patterns
Customer

Relations
builddate[Car*Date]

Relation: builddate :: Car * Date

A tuple, for example *(BMW325,27101988)* in the following table means:
Builddate of BMW325 is 27101988.

The following table displays the contents of this relation.

Car	Date
BMW325	27101988
BMW525tds	06071996
Mazda323F	05012001

Figure 3.5 builddate relation in ADL-t
Each relation can be looked at separately and this includes a table containing its population.

Context tree
Insurance

Patterns
Customer

Relations
carInsurance[DriversLicense*Car

Relation: carInsurance :: DriversLicense * Car

A tuple, for example *(NL377748,BMW325)* in the following table means:
Customer with License NL377748 bought insurance for car BMW325.

The following table displays the contents of this relation.

DriversLicense	Car
NL377748	BMW325

The following 2 instances do not satisfy the rule that there is a missing Car for each of the following DriversLicenses. (The relation *carInsurance* is total.)

DriversLicense	Car
NL313360	
NL395504	

Figure 3.6 carInsurance relation, including violations in ADL-t
If a violation has occurred, this is visible in a separate table.

3.4 Looking back

Let's look back at why we used Business Rules again: By separating them from the information system it made it easier to change them. We want to be able to quickly change our rules, because the business itself is rapidly changing most of the time and so do the requirements for the system. We can now see how we can accomplish that: If a rule changes or a new rule about a customer, for example, is added to the old rules, we simply look at how a relation changes and alter that part, or we make a new composition with the old rules and the new one. We can then easily check for any new violations caused by this change and adapt accordingly.

4. IDATA

As mentioned in the introduction, this chapter will be used to explain what iData is and how it is used. It will show some similarities between iData and ADL and this will help explain how the two could work together to incorporate Business Rules. Combination possiblities between the two will be discussed in chapter 5. For this chapter about iData it is assumed that the reader is familiar with functional programming in Clean [19].

4.1 The iData Toolkit

The iData Toolkit [9] is a toolkit which is writen in the functional programming language Clean [19], and is meant as a library for writing interactive, thin-client, dynamic, form based web applications. Such an application computes HTML pages that contain a set of interconnected iData elements. It is a server-side application, so all the work is done on the web server, not on the users computer. Basically a web application is nothing more than a load of HTML, combined with dynamic forms to change the data behind the application. iData takes care of the dynamic part: the forms, while giving the programmer the possibility to paste plain HTML in between those forms. There is an important reason this toolkit was written in a functional language: It uses generic programming techniques[6, 1], which make the programmers life a lot easier. The techniques will be explained in the next section, after which we will take a look at the iData elements (or in short just iData, for interactive data) and how they are used. Then we go back to the case of 2.4 and take a look at how iData can represent parts of this case.

4.2 Generic programming

In functional programming there are a lot of basic functions the programmer would like to use for any type. For example equality: He wants to be able to compare a couple of Integers to one another, but he also wants to do this for Doubles, Strings or maybe Lists. Instead of writing a slightly different function for every type, wouldn't it be a lot easier if he defined one general equality function and the system derived the function for every type he would need? Of course it would. Defining one function that works for every type is exactly what generic programming does. This is not

the same as overloading, although it uses overloading. Overloading uses the same name for different functions, while generic programming uses a generic type as a general unifier where all types can be derived from. So a programmer defines the general case for a function and automatically obtains a function for every concrete type. Specialized instances for types can also be defined, so if you want the system to do the work you can, but it is also possible to do it yourself, just the way you like it.

Let's look at some of the details: We need a type to represent *all possible types*. But this is not possible, for type correctness would become undecidable. Therefore we use a couple of simple types to represent any type. The following example was taken from [1]. If you look at, for instance, the equality function, it is necessary to write such a function for every type you want to use:

```
class eq t :: t t ! Bool
instance eq (List a) j eq a where
eq Nil Nil = True
eq (Cons x xs) (Cons y ys) = eq x y && eq xs ys
eq x y = False
instance eq (Tree a b) j eq a & eq b where
eq (Tip x) (Tip y) = eq x y
eq (Bin x lxs rxs) (Bin y lys rys) = eq x y && eq lxs lys && eq rxs rys
eq x y = False
```

This would mean a lot work has to be done every time you need something for a different type. This is what generic progrmamming does for you: We find what the generic part is for every type and build the functions we want for those types only. There are three most basic types:

```
:: UNIT         = UNIT
:: PAIR a b     = PAIR a b
:: EITHER a b   = LEFT a | RIGHT b
```

These basic types are all you need to define your function for. Every 'real' type can be expressed in terms of these basic types. If we look at the List type, in generic programming this would look like this:

```
:: List a   = Nil | Cons a (List a)
Generic type representation:
:: ListG a  :== EITHER UNIT (PAIR a (List a))
```

Now we need a way of going to and from those basic types and the real types we need, we need functions to go to and from the generic domain. These are typically called *fromList* to go from a List type to a ListG (for generic) type and *toList* to convert back to the real domain:

```
fromList :: (List a) → ListG a
fromList Nil            = LEFT UNIT
fromList (Cons a as)    = RIGHT (PAIR a as)
```

```
toList :: (ListG a) → List a
toList (LEFT UNIT)          = Nil
toList (RIGHT (PAIR a as))  = Cons a as
```

If this conversion is defined for every type you need in your program, then you only need to define your functions for the generic domain, and let the system derive the rest. Equality for the generic domain would then be:

```
instance == UNIT
where    (==) UNIT UNIT = True

instance == (EITHER a b)  | == a & == b
where    (==) (LEFT x)   (LEFT y)   = x == y
         (==) (RIGHT x)  (RIGHT y)  = x == y
         (==) _          _          = False

instance == (PAIR a b)  | == a & == b
where    (==) (PAIR x1 x2) (PAIR y1 y2) =
         x1 == y1 && x2 == y2
```

and for the concrete List type:

```
instance == (List a) | == a
where
    (==) x y = fromList x == fromList y
```

This looks like a lot more work, but consider this: You make the conversions to and from the generic domain for each type once, and you define the function you want only once for the generic domain. Now every time you build a new datastructure, you use the *derive* keyword to derive the function you need from the generic domain. So you invest some time in the beginning to get a lot of time back later on.

Now back to programming web applications. The iData elements are types with a generic representation, so every function can be defined for the generic domain once and used for every record in your application: The programmer can derive a set of functions to display, update, serialize and deserialize arbitrary types, which is exactly what you need for the web.

4.3 What are iData?

iData are elements that can be used as modular building blocks. An iData element can store *values* (also called *states*) of a certain type and for such a value, a *form* (also called *rendering*) can be generated. This form is automatically derived from the value of an element and the type of that value, using the generic programming techniques from the previous section. The *mkEditForm* function is a generic function, so for every type, an instance can be derived. The look and feel of the form can be modified to the taste of the programmer, or a default design can be used. iData can have a number

of different forms, based on its type and the sort of form the programmer wants to use. For example, a programmer can choose an *EditForm* as a simple editor for a certain value, or he can use a *StoreForm*, which is an EditForm with a function that can alter the input for certain values. Other options are dropdownmenus, buttons and other HTML forms. A form in iData is a function that alters the state of the application: the HSt (for HTML State). This HSt is a global state and contains all the iData information. It is recalculated every time the user does something with the application. The main function in iData therefore is:

```
InsurancePage :: *HSt → (Html,*HSt)
```

The outcome of the function is a tuple containing the new HTML page and the altered HSt. This function is called every time the page is updated.

The user of the application can only use the forms to change the values of an iData element, and because the type of this value is determined before the form is generated, the only valid values he can give to an element is a value of the correct type. If a user tries to enter invalid values, the iData keeps it's old value or keeps the value which it was initialized with. Therefore the application is type-safe, which is important for its security. However, type constraints on the input are not always sufficient: iData elements can impose additional constraints on its values that can not be expressed with types (as we already saw with the function in a StoreForm), or they are interconnected and need to modify their value as a consequence of the modification of another iData change. Could these additional constraints be used to model our Business Rules?

4.4 Using iData

So let's see how this works. We're going to make editors for some information about a Customer in our Insurance case. First we have to define a record containing all customer fields:

```
:: Customer = { name              :: String
         ,   policyNr             :: Int
         ,   dateOfBirth          :: HtmlDate
         ,   address              :: String
         ,   socialSecurityNr     :: String
         ,   bankNr               :: String
         }
```

For this new record, we have to derive all the generic functions we need. These are gForm, gUpd, gPrint and gParse. This is the generic part in iData and will not be discussed further. Then we define which type of form we would like to use. In this case the record for a customer is a simple one, so

we take a simple EditForm. We initialize it by creating a default value:

```
customerForm :: *HSt → (Form Customer, *HSt)
customerForm hst = mkEditForm (Init,
    nFormId "customer" createDefault) hst

initCustomer :: String Int HtmlDate String String
             String → Customer
initCustomer name pnr dob address ssnr bnr
    =    {Customer
         | name = name
         , policyNr = pnr
         , dateOfBirth    = dob
         , address = address
         , socialSecurityNr = ssnr
         , bankNr = bnr
         }
```

In the *InsurancePage* function mentioned above, we now add an update function for the forms on that page, in this case the customerForm. We also make a *ButtonsForm*, to define a button with which the user can save the customer he just added in the CustomerForm. What the total would look like, if we repeat all these steps for other data we want to model in our application, you can read appendix B of this thesis. It contains a lot of detailed information, which is not very relevant for the implementation of Business Rules. For now it is enough to roughly know how iData is used so it is clear what is meant in the next chapter. In figure 4.1, you can see what the first part will look like:

Figure 4.1: CustomerForm in HTML by iData

4.5 Similarities and differences with ADL

After reviewing both systems it is obvious that they are both very different in use and purpose, but there are some important similarities between the

two. The biggest similarity is the way the systems look at their data. They both have certain datasets and when something changes (a new instance in ADL or an update in iData) the checks or functions belonging to that change are called upon to adapt to the change, after which the system is back in its original state of waiting on a change (but with different data). The biggest difference is that ADL does this check only once, while iData is a running system that will keep responding to changes over time. In the next chapter we will see how these similarities and differences will be used to examine and design a combination between the systems, that will lead to answers to the questions of this research.

5. COMBINATION POSIBILITIES

In this chapter we will look at the possibilities of combining ADL and iData, in order to expand the possiblities of implementing Business Rules. At first sight, there is more than one way to do this, so we first look at the options, then we discuss the transitions we want to make and finally some examples of the new system are given.

5.1 ADL on its own

As we saw in chapter 3, ADL-t only incorporates Invariant Business Rules. Since there is no element of time present in an ADL-t program (it's a one-time check on the rules and if something changes, you have to run it again) it is not possible to incorporate the Calculating Business Rules that depend on time in the system. ADL itself also doesn't have the syntax needed to make these rules. The language needs to be extended (ADL++) and the implementation therefore also needs an extension, or another system, like iData, to make this possible. If we want to implement this using iData we have to break down ADL's functionality into the basic blocks and define what iData has to do with these blocks, how to translate those blocks to iData. Since we keep using ADL as a starting point, we need some kind of transformation from one system to the other. Basically we are building an *ADL++ -to-iData* compiler.

This compiler can be built in three ways: We can extend ADL-t to deliver an iData specification, iData itself can be used to read ADL files and use these to make new iData, or a third possibility is to use an entirely different language to build this compiler. This is of course not an obvious choice, since the power of the combination lies in the fact that ADL and iData as concepts are so similar. That leaves us with the first two. Either one of these options is possible, and probably both equally difficult or easy to do, since ADL was made using Haskell, and iData using Clean, both functional languages.

So we either extend ADL-t or iData, but the effect is the same. An advantage of ADL-t extension would be that the basic ADL structure is already built into a functional program somewhere in ADL-t. Advantages of using iData would be that the transformation is quite small, because of the generics used and it is easier to use the power of the language your

working towards if the compiler is also written in that language. Another advantage is that you already have a fully functional (web-)userinterface at your disposal. A disadvantage would be that implementing it would be more work then with ADL-t, because you have to start from scratch. Both options are defendable, but in this thesis the iData option will be examined further, because of non-scientific reasons: more experience with programming in Clean and a closer relation to the experts in iData than the ones in ADL. Let's look at what should be done.

5.2 From ADL to iData

What we build exactly is this: A compiler that can read ADL files and than translate these files into a web interface in iData.

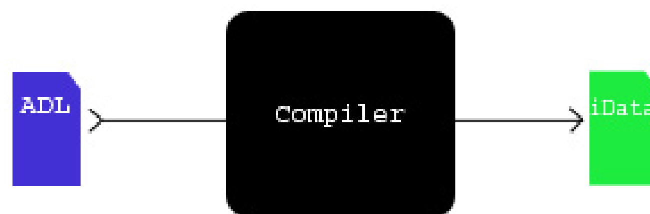

Figure 5.1: From ADL to iData

We will need a parser for ADL and funtions to transform the basic parts of ADL into iData implementations. These functions is what make in this section.

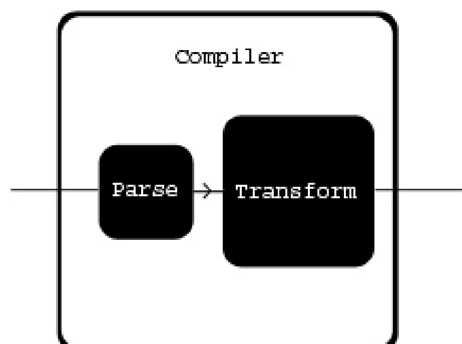

Figure 5.2: The compiler

First we have to extend ADL itself: In order to extend the use of Business Rules to the calculating part, we need to find a way to check if the system complies to them. ADL-t cannot do this because it can only work with static relations, but in iData it is possible to use functions for this purpose. We can still use the ADL syntax to define rules and relations in relational algebra, but have to make an addition to that syntax to make room for the calculating rules in the form of a boolean statement. For example:

$isOldtimer : Car \times BuildDate$

would have to become:

$isOldtimer : Car \times BuildDate(ifBuildDate > (DateToday + 25years))$

ADL's syntax has to be extended to allow this kind of definition. ADL-t does not have to do anything with this information, as long as it does not return error messages when this type of definition is used.

The basic blocks of ADL we talked about are: *Cartesian product*, *Sequential Composition* and *Converse* as we saw in chapter 3. This has to be extended with *Condition* for the boolean functions we want to use (ADL++). We now have to build the compiler from ADL++ to iData that can convert these four concepts into iData syntax. We have to take these concepts as a starting point to insure the correctness of the rules in iData and we want to compile it using ADL++ as a basis to insure the changable nature and high level of abstraction of the Business Rules. This compiler has to somehow make the switch from relational to functional programming. These basic concepts are used as combinators to build bigger functions. This resembles the use of parser combinators and combinatoric programming in general. Combinatoric programming is a way of programming that takes elementary program fragments (a combinator) and combines these to construct a bigger program using disciplined construction.

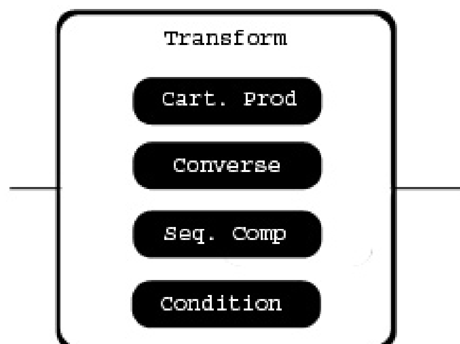

Figure 5.3: The four transformations

Because we are still in the design phase, the functions described below will stop at their definitions and a fairly detailed description of them in pseudocode. Fully implementing them, and thus building the entire compiler is outside the scope of this thesis.

So let's look at the four concepts in detail. Since the *Condition* is an extension not possible in ADL-t, for it is already functional, no conversion from relational to functional is neccesary here, but we still have to build functions for it. The other concepts are more complicated. We have to build functional editors that comply to the relational properties the user defined in ADL.

5.2.1 Cartesian Product

In ADL, the relations are cartesian products (chapter 3) and as we saw earlier, they have a *source* and a *target*. We want to build an iData that has editors for both of these, but we also want the iData to store all instances of that relation. This is done using a list of tuples. So the relations we fill using the iData have to be stored in that list. We'll need to build a *SetEditor* for that. This is the first combinator we will use. So the view of this form are two editor windows, forming one tuple, but the model is a list of tuples.

An important part of those relations are the multiplicities. In ADL, these are written behind the relations between *[* and *]*. We have to find a way to include these multiplicities in the definition of a cartesian product in iData. They have to be checked every time an instance of the product is added or removed. This sounds a lot like the *SelfForm* used in iData, which applies a predefined function to the internal state only if the idata has been changed:

```
mkSelfForm :: !(InIDataId d) !(d → d) !*HSt → (Form d,!*HSt)
```

This function has three parameters: an ID, a function (used with every update) and of course the HSt.

We want to make a form that has two editor windows, one for the source and one for the target of the relation. This is also already present in iData, namely the *t2EditForm*, which combines two *EditForm*'s:

```
t2EditForm :: !(InIDataId (a,b)) !*HSt → ((Form a,Form b),!*HSt)
```

It has two parameters: an ID and the HSt.

The form we want to build must have similar properties to both the forms mentioned, so the definition looks like this:

```
CartProdForm :: ([(a,b)] → Maybe Mistake) (Form [(a,b)])
    *HSt → (Form [(a,b)],*HSt)  | == a & == b
```

This has three parameters: a function, which is used to check the multiplicities, an iData Form containing a list of tuples, and the HSt. It uses lists of tuples because the multiplicities are properties of sets, so we need the entire list to check them. A *Mistake* is a simple String, but it is used here for reasons of understandibility. The outcome of the *CartProdForm* is a form of a list of tuples with *a* and *b* and not a tuple of forms like with the *t2EditForm* above. This is done because the Form[(a,b)] is a better representation of the idea meant here: one form for one relation, which includes an editor for each part of the (binary) relation. As you can see, the *CartProdForm* doesn't use iDataID's but Form's. This was done because we want only one ID for each relation, but we want to be able to manipulate its Form several times. So before we build the *CartProdForm*, we first need a function that will deliver a Form from an iDataID. This is the (*mkSetEditor*):

```
mkSetEditor :: (InIDataId d) !*HSt → (Form d,!*HSt)
mkSetEditor inIDataId=:(_,{value}) hst
= mkViewForm inIDataId
    { toForm    = λ_ set _ → if (isEmpty set)
        createDefault (hd set)
    , updForm   = λ_ v → v
    , fromForm  = λ_ v → [v:value]
    , resetForm = Nothing
    } hst
```

So *CartProdForm* is the second combinator we build, and it uses the first (*mkSetEditor*) combinator. We also use mkViewForm, which is the standard form generating function in iData.

Which multplicities are required for the relation is only important when checking them, so it is probably easiest to make a list of the multiplicities defined in ADL a parameter of the function:

```
checkMult :: [Multiplicity] [(a,b)] → Maybe Mistake | == a & == b
```

As you can see, this is not the same definition of the function used in *CartProdForm*, because the list of multiplicities can be added later using currying. So where the function for checking is used in *CartProdForm*, there will be something like:

```
(checkMult [Uni,Tot,Inj,Sur])
```

The difference here with the *mkSelfForm* is that its function doesn't only change the internal state of the iData, it also has to show the user if and how the multiplicities are violated, because this shows which Business Rules are being violated. This is done because it is better to point the user to the violations then to prevent them without letting the user know what's wrong. So if the function encounters a violation, the instance has to be removed and

it has to flag that violation. This is done by the *Maybe Mistake*. This mistake string contains which instance of the relation violates its multiplicities. In pseudocode:

```
checkMult :: [Multiplicity] [(a,b)] → Maybe Mistake | == a & == b
checkMult [] _ = Nothing
checkMult [m:ms] list
| m == "Uni" = case checkUnivalent list of
        Nothing         = checkMult ms list
        Just mistake    = mistake
| m == "Tot" = ...
| ...for each multiplicity

checkUnivalent:: ![(a,b)]  →  Maybe Mistake | == a
checkUnivalent list
| removeDup (map fst list) == (map fst list) = Nothing
| otherwise = Just "The new instance violates the
    Univalent property of this relation and is therefore removed."
```

The whole form:

```
CartProdForm :: ([(a,b)] → Maybe Mistake) (Form [(a,b)])
    *HSt → (Form [(a,b)],*HSt)   | == a & == b
CartProdForm f setF hst
= case f setF.value of
    Nothing         = (setF,hst)
    Just mistake    = ({setF & value=tl' setF.value,
        form = setF.form ++ [toHtml mistake]},hst)

tl' :: [] → []
tl' [x:xs] = xs
tl' []     = []
```

The *mkSetEditor* enforces that every new instance of the relation is placed at the head of the list. This is useful to know, because we want to remove a violating instance in *CartProdForm*.

5.2.2 Converse

The converse in ADL is actually an inverse. "Converse merely swaps the left and right hand side" [17] An important property of the inverse is that if you inverse the inverse, you get the identity function. This is also true for converse in ADL, so it should have been called inverse. Since our cartesian product works on lists of tuples, making an inverse is very simple. To do this we use two well known functions, namely *map* and *swap*. *Swap* swaps two elements of a tuple and *map* maps the function behind it on every element of the list it is given. So:

```
map swap [(a0,b0),(a1,b1),...,(an,bn)] → [(b0,a0),(b1,a1),...,(bn,an)]
```

gives us a list of type [(b,a)] from a list of type [(a,b)], which is exactly what we want. Since we work with Forms, *Converse* simply returns a Form with an empty rendering and the inversed list with tuples. All we have to do now is inverse the multiplicities we have for a relation and we can inverse a relation. This is also very simple: univalent becomes injective and vice versa, surjective becomes total and vice versa.

```
Converse :: (Form [(a,b)]) *HSt → (Form [(b,a)],*HSt)
Converse form hst
= ({form.changed,(map (λ(a,b) → (b,a)) form.value),[]}, hst)

flipMult :: [Multiplicity] → [Multiplicity]
flipMult mult = map flipM mult

flipM :: Multiplicity → Multiplicity
flipM mult
|"uni" = "inj"
|"tot" = "sur"
|"inj" = "uni"
|"sur" = "tot"
```

5.2.3 Sequential Composition

Business Rules in ADL are compositions of the binary relations. These compositions can be built from two or more relations. We only discuss the first one (two relations) because compositions with more than two are similar. Let's say we want to make a composition of relations $A \times B$ and $B \times C$ into $A \times C$. What we want, is an editor that we can use to make new instances of this composition, but which also fills the two relations the compositions is based on. For instance: the composition *carInsurance*, based on the relations: *hasLicense* and *hasCar* will need editors for *Customer*, *DriversLicense* and *Car*. Every time the user enters new values into these editors, the link between the two underlying instances of *hasLicense* and *hasCar* has to be made, and relations instantiated with these values.

Since we want to use the previous functions as combinators, we use the forms of the cartesian products again. We combine the instances where the B's are similar, and put them in a new list of tuples [(a,c)]. Now, again, all we have to do is check the multiplicities for the combination. First, we need to combine the given multiplicities, but this turns out to be quite easy: if relation one is univalent and relation two is also univalent, its combination will also be univalent. If one is univalent and two is univalent as well as total (so it is a function), the combination will be only univalent. This works for all multiplicities, so all we have to do is find those that are given for both relations. Since we use the existing forms, the rendering of the iData will be empty. The only exception is when a violation takes place. The mistake string will be added to the rendering of the new Form and the last change

in the underlying Forms (which triggered the violation) will be reversed by removing that instance. We don't make a new iData for the composition, but simply combine the information we already have.

```
combineMult :: ([Multiplicity],[Multiplicity]) → [Multiplicity]
combineMult (m1,m2) = [s \\ s ← m1 , t ← m2 | s == t]

checkComposition :: ([(a,c)] → Maybe Mistake) (Form [(a,b)])
    (Form [(b,c)]) → (Form [(a,b)],Form [(b,c)],Form [(a,c)])
    | == a & == b & == c
checkComposition h setFa setFb
♯ combList = [(s,u) \\ (s,t)←setFa.value , (t2,u)←setFb.value | t == t2]
= case h combList of
    Nothing        = (setFa,setFb,(False,combList,[]))
    Just mistake   = (if setFa.changed {setFa & value=tl' setFa.value} setFa,
                      if setFb.changed {setFb & value=tl' setFb.value} setFb,
                      (False,combList,[toHtml mistake])
                     )

SeqCompForm :: [Multiplicity] [Multiplicity] (Form [(a,b)]) (Form [(b,c)])
    *HSt →  (Form [(a,c)],*HSt)  | == a & == b & == c
SeqCompForm mult1 mult2 forma formb hst
♯(forma,formb,formc) = checkComposition (checkMult(combineMult (mult1,mult2))
    forma formb)
= (formc,hst)
```

5.2.4 Condition

Adding a condition to the relation is the new part we can now add to ADL. As said before: since conditions are already functional, they will be easy to implement. We simply add a function containing the condition to the *CheckMultiplicities* function, which checks the condition for the first element of the list, since this is the new one. This all now seems a very small step, but that is the power of this ADL++ -to-iData combination. By building relations in a functional environment, conditions almost come for free. In the next section, you will find an example of how a condition can be checked.

5.3 Example

Now that we have succesfully captured ADL's functionality, let's go back to our insurance case. A webpage in iData for car insurance: Let's say we want a page for entering customers for a car insurance. So we want to fill the *carInsurance* relation is ADL. It is defined as:

$carInsurance$ -: $hasLicense^{\sim}$;$hasCar$

So it contains an inverse and a sequential composition. In our new system, this would look like this:

```
CarInsurancePage :: *HSt → (Html,*HSt)
CarInsurancePage hst
    # (hasCarF, hst) = CartProdForm (Init, mkFormId "hasCar" [])
    (checkMult []) hst
    # (hasLicenseF, hst) = CartProdForm (Init, mkFormId "hasLicense" [])
        (checkMult []) hst
    # (carInsuranceF, hst) = SeqCompForm (flipMult []) []
        (Converse hasLicense hst) hasCar hst
    = (mkHtml "Car Insurance demo page"
        [hasCarF.form,hasLicenseF.form,carInsuranceF.form],hst)
```

First, we define the *hasCar* and *hasLicense* relations as *SetEditors*. Then, we define the *carInsurance* relation as a *SeqCompForm* with the *converse* of *hasLicense* and *hasCar*. Finally, we use the three created forms in the HTML page.

Now for an example with the condition:

$$isOldtimer : Car \times BuildDate(if BuildDate > (DateToday + 25 years))$$

This translates into a normal *CartProdForm*, but now the function for checking multiplicities also checks whether the condition is satisfied. Since every condition for checking the age of something works with dates and a time interval, this function has te be made only once and can then be used with every age check. For the date to be available, we need to extract it from the HSt. This can be done at the beginning of the program, so the date is available to all condition functions, should they need it.

```
# (dateNow,hst) = accWorldHSt getCurrentDate hst
# (timeNow,hst) = accWorldHSt getCurrentTime hst
```

The extra parameter this function needs (the time interval demanded) is the first parameter, so this can also be curried. The function checks the condition and if it succeeds, the multiplicities are checked. If it fails, it removes the violating instance and returns a *Mistake*, just like the *checkMult* function does:

```
checkAge :: Interval [Multiplicity] [(a,b)] → Maybe Mistake | == a & == b
checkAge x mult list
= case ageCheck x (snd (hd list)) of
    Nothing      = checkMult mult list
    Just mistake = mistake

ageCheck :: Interval Date → Maybe Mistake
ageCheck x date
# difference        = diffDate dateNow date
| difference ≤ x    = Nothing
```

```
| otherwise         = Just "The new instance failed the
        age check and is therefore removed."

diffDate :: Date Date → Int
diffDate {d=d1,m=m1,y=y1} {d=d2,m=m2,y=y2}
= toInt (((y1-y2)*365 + (m1-m2)*(365/12) + (d1-d2)) / 365)
// difference in years, rounded down
```

For other conditions, similar functions will have to be written. In the *CarInsurancePage* it would look like this:

```
♯ (isOldtimerF, hst) = CartProdForm (Init, mkFormId "isOldtimer" [])
        (checkAge 25 []) hst
```

As you can see even better in the examples, the functions built are a combinator library for using relations. Since all major functions end with `!*HSt → (Form d,!*HSt)`, every combinator uses the HSt (and some other parameter(s)) and returns a form and the edited HSt.

5.4 Outcome

In this chapter we looked at the possibilities for combining ADL and iData, so the question now is: was it a success? We were able to capture all relational properties of ADL in the functional nature of iData, and this made it possible to use the functional properties of iData to expand the functionality of ADL (into ADL++). We are now able to use Calculating Business Rules. A big bonus of this combination is the graphical web interface the user now gets for free. This combination now gives us the use of all types of Business Rules, by implementing ADL++ into the iData interface.

6. CONCLUSION

6.1 Summary

Let's first summarize what we have looked at in this thesis: because of the need for agility and taking requirements to a higher level, in order to be more succesful in software projects, the Business Rules concept looks like it could certainly help in that area. We have looked at the problems in the software development process and how the rules can help improve that process. Since a good idea by itself is not enough, we have also looked at a way of implementing the rules. Since the current system for this (ADL-t) can only use a part of the rules, we looked for a way of expanding those possibilities. The iData Toolkit looked very useful here, so we tried combining the two systems, the results of which can be found in the previous chapter. The most important part of the combination was expanding the use of business rules. The two questions we wanted answered were:

(i) Is it possible to build the current Business Rules functionality from ADL-t in iData?
By converting the three main ADL relational properties (Carthesian product, Converse and Sequential compostion) to iData all Business Rules functionality from ADL-t was captured in iData.

(ii) Is it possible to expand that functionality with the calculating part of Business Rules?
By adding the Condition to the existing properties, the remainder of the Business Rules functionality missing in ADL-t (Calculating Business Rules) was also captured in iData.

The ADL–iData combination was succesfully designed, and in doing so the functionality of Business Rules was expanded. The expansion gives a user the possibility of checking even more rules than before. This could be a stepping stone in the development of Business Rules tools.

6.2 Future work

Since this thesis mainly examined and designed the combination, there remains a lot of work to be done. The use of Business Rules as thé solution

for all failing IT projects needs more examination and the combination designed here isn't ready for use either. The next step would be to fully implement it. This would mean building a parser for reading the ADL++ syntax and using the compiler to build the iData. If there is a need to change or expand the compiler, then it is important to stick to the combinatoric template of the functions, so it remains a combinatoric library. Finally, it would be wise to improve the resulting system aesthetically, because this will probably result in a more surveyable system.

6.3 Possible consequences

The main question in this thesis remains to be answered:

* Is the combination between ADL and iData a powerful, hence useful one?

Only part of this question can be answered at this point and that is the part answered above by answering the sub questions. If the combination is useful remains to be seen. Combining the systems has lead to a more complete use of Business Rules and this might be reason for more people to start using them. If this is the case then the combination is indeed a powerful one, since it will have enabled more projects to start working with the rules and if Business Rules are indeed the solution needed for projects to be more succesfull, then it will have contributed (if only slightly) to their success.

BIBLIOGRAPHY

[1] A. Alimarine and R. Plasmeijer. A generic programming extension for clean. *Arts, Th., Mohnen M., eds. Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001, Selected Papers, LNCS 2312, pages 168-185*, September 24-26, 2001.

[2] K. Blenkhorn Rodriguez. Business rules revolution - executives across all industries are using technology to rewrite the rules of business literally. *INSIGHT Magazine, The Magazine of the Illinois CPA society (August 2006).*, 2006.

[3] C. J. Date. *Database in Depth: Relational Theory for Practitioners*. O´ Reilly, 2005.

[4] B. Demuth. The Dresden ocl toolkit and the business rules approach. *European Business Rules Conference*, 2005.

[5] G. Dimitoglou. Business rule computation in distributed organisations. *Dissertation, George Washington University*, 2005.

[6] R. Hinze. A new approach to generic functional programming. *In The 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 119-132.*, January 2000.

[7] J. King. Survey shows common it woes. *Computerworld, June 23*, 2003.

[8] B. Lewis. The 70-percent failure. *InfoWorld*, 2003.

[9] R. Plasmeijer and P. Achten. iData for the World Wide Web - programming interconnected webforms. *Eighth International Symposium on Functional and Logic Programming, FLOPS 2006, April 24-26 2006, Fuji Susone, Japan*, 2006.

[10] R. G. Ross. *The Business Rule Book: Classifying, Defining and Modeling Rules*. Database Research Group, Boston, Massachusetts, 1997.

[11] R. G. Ross. A personal insurance saga - the economics of business rules. *Business Rules Journal, Vol. 7, No. 6 (June 2006)*, 2006.

GREY LITERATURE

[12] M. Cobb. Unfinished voyages: a follow-up to the chaos report. `http://www.standishgroup.com/sample_research/unfinished_voyages_1.php`, 1996.

[13] Fair Isaac. Products and services: Blaze advisor. `http://www.fairisaac.com/fic/en/product-service/product-index/blaze-advisor/`, 2007.

[14] IBM. Business rules for electronic commerce. `http://www.research.ibm.com/rules/home.html`, 1997.

[15] Javarules. The java business rules community. `http://www.javarules.org/`, 2006.

[16] JESS. Home website. `http://www.jessrules.com/`, 2006.

[17] S. Joosten. Rule based design. *unpublished draft*, 2005.

[18] B. Nainani, K. Gruenefeldt, and R. Mueller. Building flexible enterprise processes using oracle business rules and bpel process manager. *Oracle White Paper, Jan*, 2005.

[19] R. Plasmeijer and the Software Technology group. Clean. `http://clean.cs.ru.nl/`, 2006.

[20] The Business Rules Group. Defining business rules - what are they really? `http://rewerse.net/downloads/BRG-whatisBR_3ed.pdf`, 2000.

[21] The Business Rules Group. Business rules manifesto - the principles of rule independance. `http://www.businessrulesgroup.org/brmanifesto/BRManifesto.pdf`, 2003.

[22] The Business Rules Group. What is the business rules group. `http://www.thebusinessrulesgroup.org`, 2006.

[23] The Standish Group. The chaos report. `http://www.pm2go.com/sample_research/chaos_1994_1.asp`, 1994.

[24] The Standish Group. Home website. `http://www.standishgroup.com/`, 2006.

APPENDIX

A. BUSINESS RULES MANIFEST

Business Rules Manifesto

The Principles of Rule Independence

by Business Rules Group

Article 1. *Primary Requirements, Not Secondary*

1.1. Rules are a first-class citizen of the requirements world.

1.2. Rules are essential for, and a discrete part of, business models and technology models.

Article 2. *Separate From Processes, Not Contained In Them*

2.1. Rules are explicit constraints on behavior and/or provide support to behavior.

2.2. Rules are not process and not procedure. They should not be contained in either of these.

2.3. Rules apply *across* processes and procedures. There should be one cohesive body of rules, enforced consistently across all relevant areas of business activity.

Article 3. *Deliberate Knowledge, Not A By-Product*

3.1. Rules build on facts, and facts build on concepts as expressed by terms.

3.2. Terms express business concepts; facts make assertions about these concepts; rules constrain and support these facts.

3.3. Rules must be explicit. No rule is ever assumed about any concept or fact.

3.4. Rules are basic to what the business knows about itself — that is, to basic business knowledge.

3.5. Rules need to be nurtured, protected, and managed.

Article 4. *Declarative, Not Procedural*

4.1. Rules should be expressed declaratively in natural-language sentences for the business audience.

4.2. If something cannot be expressed, then it is not a rule.

4.3. A set of statements is declarative only if the set has no implicit sequencing.

4.4. Any statements of rules that require constructs other than terms and facts imply assumptions about a system implementation.

4.5. A rule is distinct from any enforcement defined for it. A rule and its enforcement are separate concerns.

4.6. Rules should be defined independently of responsibility for the *who*, *where*, *when*, or *how* of their enforcement.

4.7. Exceptions to rules are expressed by other rules.

Article 5. *Well-Formed Expression, Not Ad Hoc*

5.1. Business rules should be expressed in such a way that they can be validated for correctness by business people.

5.2. Business rules should be expressed in such a way that they can be verified against each other for consistency.

5.3. Formal logics, such as predicate logic, are fundamental to well-formed expression of rules in business terms, as well as to the technologies that implement business rules.

continued…

Copyright, 2003. Business Rules Group. *Version 2.0, November 1, 2003. Edited by Ronald G. Ross.* www.BusinessRulesGroup.org

Permission is granted for unlimited reproduction and distribution of this document under the following conditions: (a) The copyright and this permission notice are clearly included. (b) The work is clearly credited to the Business Rules Group. (c) No part of the document, including title, content, copyright, and permission notice, is altered, abridged or extended in any manner.

Article 6. *Rule-Based Architecture, Not Indirect Implementation*

6.1. A business rules application is intentionally built to accommodate continuous change in business rules. The platform on which the application runs should support such continuous change.

6.2. Executing rules directly – for example in a rules engine – is a better implementation strategy than transcribing the rules into some procedural form.

6.3. A business rule system must always be able to explain the reasoning by which it arrives at conclusions or takes action.

6.4. Rules are based on truth values. How a rule's truth value is determined or maintained is hidden from users.

6.5. The relationship between events and rules is generally many-to-many.

Article 7. *Rule-Guided Processes, Not Exception-Based Programming*

7.1. Rules define the boundary between acceptable and unacceptable business activity.

7.2. Rules often require special or selective handling of detected violations. Such rule violation activity is activity like any other activity.

7.3. To ensure maximum consistency and reusability, the handling of unacceptable business activity should be separable from the handling of acceptable business activity.

Article 8. *For the Sake of the Business, Not Technology*

8.1. Rules are about business practice and guidance; therefore, rules are motivated by business goals and objectives and are shaped by various influences.

8.2. Rules always cost the business something.

8.3. The cost of rule enforcement must be balanced against business risks, and against business opportunities that might otherwise be lost.

8.4. 'More rules' is not better. Usually fewer 'good rules' is better.

8.5. An effective system can be based on a small number of rules. Additional, more discriminating rules can be subsequently added, so that over time the system becomes smarter.

Article 9. *Of, By and For Business People, Not IT People*

9.1. Rules should arise from knowledgeable business people.

9.2. Business people should have tools available to help them formulate, validate and manage rules.

9.3. Business people should have tools available to help them verify business rules against each other for consistency.

Article 10. *Managing Business Logic, Not Hardware/Software Platforms*

10.1. Business rules are a vital business asset.

10.2. In the long run, rules are more important to the business than hardware/software platforms.

10.3. Business rules should be organized and stored in such a way that they can be readily redeployed to new hardware/software platforms.

10.4. Rules, and the ability to change them effectively, are fundamental to improving business adaptability.

B. INSURANCE IN IDATA

Ad-hoc solution in iData (Chapter 4)

```
module Insurance

import StdEnv, StdHtml

derive gForm   CustomerLink, Customer, Insurance,
    Package, InsuranceLink, PackageLink, []
derive gUpd    CustomerLink, Customer, Insurance,
    Package, InsuranceLink, PackageLink, []
derive gPrint  CustomerLink, Customer, Insurance,
    Package, InsuranceLink, PackageLink
derive gParse  CustomerLink, Customer, Insurance,
    Package, InsuranceLink, PackageLink

// main function:
Start world = doHtmlServer InsurancePage world

// building the records:
:: Customer ={name        :: String
    ,   policyNr    :: Int
    ,   dateOfBirth :: HtmlDate
    ,   address     :: String
    ,   socialSecurityNr:: String
    ,   bankNr      :: String
    }
:: CustomerLink ={customerInfo  :: Customer
    ,   insurance   :: [Insurance]
    }
:: Insurance    ={insuranceName :: String
    ,   price       :: Int
    ,   package     :: [Package]
    }
:: Package  ={packageName   :: String
    ,   rule        :: String
    ,   packagePrice    :: Int
    }
:: InsuranceLink={customer  :: CustomerList
    ,   insuranceName   :: String
    ,   price       :: Int
    }
```

```
:: PackageLink  ={customerId    :: CustomerList
    ,   insuranceId :: InsuranceList
    ,   packageName :: String
    ,   rule        :: String
    ,   packagePrice    :: Int
    }
:: CustomerList     :== PullDownMenu
:: InsuranceList    :== PullDownMenu

// Form creation/update functions:
adminForm :: ([CustomerLink] → [CustomerLink]) *HSt
    → (Form [CustomerLink], *HSt)
adminForm update hst = mkStoreForm (Init,
    pdFormId "admin" initCustomers) update hst

customerForm :: *HSt → (Form Customer, *HSt)
customerForm hst = mkEditForm (Init,
    nFormId "customer" (initCustomer "" 0
        initDate "" "" "")) hst

insuranceForm :: (InsuranceLink → InsuranceLink) *HSt
    → (Form InsuranceLink, *HSt)
insuranceForm update hst = mkStoreForm  (Init,
    nFormId "insurance" (initInsuranceLink "" 0 0
        initCustomers)) update hst

packageForm :: (PackageLink → PackageLink) *HSt
    → (Form PackageLink, *HSt)
packageForm update hst = mkStoreForm  (Init,
    nFormId "package" (initPackageLink "" "" 0 0 0
        initCustomers)) update hst

//==========================================================

buttonsForm :: PackageLink InsuranceLink Customer *HSt
    → (Form ([CustomerLink] → [CustomerLink]), *HSt)
buttonsForm packagel insurancel customer hst =
    ListFuncBut (Init, nFormId "buttons" myButtons) hst
where
    myButtons = [ (LButton (defpixel+20) "addCustomer",
        addNewCustomer customer )
    , (LButton (defpixel+40) "addInsuranceLink",addInsuranceLink insurancel)
    , (LButton (defpixel+40) "addPackageLink",addPackageLink packagel )
    ]

    addNewCustomer :: Customer → [CustomerLink] → [CustomerLink]
    addNewCustomer {Customer|name,policyNr,dateOfBirth,address
        ,socialSecurityNr,bankNr} =
    flip (:^) (initCustomerLink name policyNr dateOfBirth address
```

```
        socialSecurityNr bankNr)

    addInsuranceLink :: InsuranceLink → [CustomerLink] → [CustomerLink]
    addInsuranceLink insurancel=:{customer,insuranceName,price}
        = updateElt (λ{customerInfo={Customer|name,policyNr}} → name
            == toString customer)
         (λc → {c & insurance = [initInsurance insuranceName
            price:c.insurance]})

    addPackageLink :: PackageLink [CustomerLink] → [CustomerLink]
    addPackageLink packagel=:{PackageLink | packageName,rule,
        packagePrice,insuranceId,customerId} customers
        | toString packageName == "" || toString rule == "" ||
            toString packagePrice == "" || toString insuranceId == ""
            || isEmpty customers
                        = customers
        | otherwise     = updateAt (toInt /*packagel.*/customerId)
            updatedCustomerLink customers
    where
        {CustomerLink| customerInfo,insurance}
                        = customers!!(toInt /*packagel.*/customerId)
        updatedCustomerLink = { insurance = addPackage packagel insurance
                              , customerInfo = customerInfo
                              }

        addPackage :: PackageLink → [Insurance] → [Insurance]
        addPackage npackage=:{insuranceId,packageName,rule,packagePrice}
            = updateElt (λopackage=:{Insurance|insuranceName} →
                insuranceName == toString insuranceId)
                    (λopackage=:{Insurance|package,price}
                        → {opackage & package    = package ++
                          [{Package | packageName=packageName,
                          rule=rule,packagePrice=packagePrice}]
                                      , price = price + packagePrice
                                       })

//=============================================================
InsurancePage :: *HSt → (Html,*HSt)
InsurancePage hst
= updatePage (updateForms hst)
where
    updateForms :: *HSt → ((Form [CustomerLink],Form Customer,
        Form InsuranceLink,Form PackageLink,
        Form ([CustomerLink] → [CustomerLink])),*HSt)
    updateForms hst
        ♯ (customerF,hst) = customerForm     hst
        ♯ (insuranceF, hst) = insuranceForm  id hst
        ♯ (packageF,   hst) = packageForm    id hst
        ♯ (buttonsF,hst) = buttonsForm packageF.value insuranceF.value
```

```
        customerF.value hst
    ♯ (adminF,   hst) = adminForm   buttonsF.value hst
    ♯ (packageF,   hst) = packageForm  (adjPackages adminF.value) hst
    ♯ (insuranceF, hst) = insuranceForm (adjInsurances adminF.value) hst
    = ((adminF,customerF,insuranceF,packageF,buttonsF),hst)
where
    adjPackages :: [CustomerLink] PackageLink → PackageLink
    adjPackages customers package=:{customerId}
        = { package & customerId = addCustomerList customers customerId
                    , insuranceId    = initInsuranceList
                   (toInt package.insuranceId) (toInt customerId) customers
          }

    adjInsurances :: [CustomerLink] InsuranceLink → InsuranceLink
    adjInsurances customers insurance = {insurance & customer =
        addCustomerList customers insurance.customer}

    addCustomerList :: [CustomerLink] PullDownMenu → PullDownMenu
    addCustomerList customers (PullDown dim (i,_)) = PullDown dim
        (i,[name \\ {customerInfo ={Customer|name}} ← customers])

updatePage :: ((Form [CustomerLink],Form Customer,Form InsuranceLink
    ,Form PackageLink,Form ([CustomerLink] → [CustomerLink])),*HSt)
    → (Html,*HSt)
updatePage ((adminF,customerF,insuranceF,packageF,buttonsF),hst)
    = mkHtml "table test"
        [ H1 [] "Insurance Customers"
        , STable []
            [ [ STable []
                [ [lTxt "Add New Customer:"], customerF.form,[customerButton]
                , [lTxt "Add New Insurance:"], insuranceF.form, [insuranceButton]
                : if no_customers []
                [ [lTxt "Add New Package:"],packageF.form,  [packageButton]]
                ]
              : if no_customers []
              [ STable []
                [ [ lTxt "Customer Insurances and Packages:" ]
                , [ toHtml (adminF.value!!(toInt packageF.value.customerId)) ]
                ]
              ]]
            ]
        ] hst
where
    no_customers    = isEmpty adminF.value
    lTxt s      = B [] s
    [customerButton,insuranceButton,packageButton:_] = buttonsF.form

// initial values:
initDate :: HtmlDate
```

```
initDate = Date 1 1 1960

initCustomers :: [CustomerLink]
initCustomers = []

initCustomer :: String Int HtmlDate String String String → Customer
initCustomer name pnr dob address ssnr bnr
    =   {Customer
        | name = name
        , policyNr = pnr
        , dateOfBirth   = dob
        , address = address
        , socialSecurityNr = ssnr
        , bankNr = bnr }

initCustomerLink :: String Int HtmlDate String String String → CustomerLink
initCustomerLink name pnr dob address ssnr bnr
    =   { customerInfo = initCustomer name pnr dob address ssnr bnr
        , insurance = []}

initInsuranceLink :: String Int Int [CustomerLink] → InsuranceLink
initInsuranceLink iname price i customers
    =   { customer      = initCustomerList i customers
        , insuranceName = iname
        , price         = price }

initInsurance :: String Int → Insurance
initInsurance iname price
    =   { insuranceName         = iname
        , price         = price
        , package       = [] }

initPackageLink :: String String Int Int Int [CustomerLink] → PackageLink
initPackageLink pname rule packagePrice i j customers
    =   { customerId    = initCustomerList i customers
        , insuranceId   = initInsuranceList i j customers
        , packageName           = pname
        , rule          = rule
        , packagePrice      = packagePrice }

initInsuranceList :: Int Int [CustomerLink] → PullDownMenu
initInsuranceList i j []        = PullDown (1,defpixel) (0,[])
initInsuranceList i j customers = PullDown (1,defpixel)
    (i,[insuranceName \\ {Insurance|insuranceName} ← (customers!!j).insurance])

initCustomerList :: Int [CustomerLink] → PullDownMenu
initCustomerList i customers    = PullDown (1,defpixel)
    (i,[name \\ {customerInfo={Customer|name}} ← customers])
```

```
//=============================================================
// Useful list operations:
updateElt :: (a → Bool) (a → a) [a] → [a]
updateElt c f [] = []
updateElt c f [a:as]
    | c a       = [f a:as]
    | otherwise = [a:updateElt c f as]

updateElts :: (a → Bool) (a → a) [a] → [a]
updateElts c f [] = []
updateElts c f [a:as]
    | c a       = [f a:updateElts c f as]
    | otherwise = [  a:updateElts c f as]

(^:) infixr 5 :: a [a] → [a]
(^:) a as = [a:as]

(:^) infixl 5 :: [a] a → [a]
(:^) as a = as ++ [a]
```