

**Response To
Business Rules In Models
OMG Request For Information**

**from
David Bevington
(bevingd@lineone.net)**

rev date 21/06/01

PREFACE

This document provides information about the IBM IAA Product Builder project. It is based solely on the personal recollections and judgements of the author. It has not been reviewed or authorised by IBM in any way.

1 Introduction

The objectives of this document are:

- to make the OMG Business Rules Working Group aware of an IBM project that built IAA Product Builder (IPB), a business rules engine developed for use by the insurance industry
- to assist it to make its own assessment of what can be learnt from that project

The amount of information provided has been constrained by the BRWG's request to limit submissions to 25 pages. The author is happy to provide further information, either orally or in written form, and to support the work of the BRWG in any way that they see fit.

The first section provides a brief description of IPB. The next sections then follow the format specified in the RFI. As requested in the RFI the author presents his own conclusions of what can be learnt from the IPB project. These can be summarised as follows:

- model driven development has had limited success because the System Model layer of the Zachman Framework has largely been ignored
- useful amounts of code can only be generated from a System Model not a Business Model
- rules formally captured in a Business Model need to be further transformed and incorporated into a complete System Model which covers additional aspects of a business system design
- the OMG should sponsor the development of a formal language for the specification of System Models
- the OMG should encourage the development of generators that have as their inputs a System Model, a Technology model, and details of the particular target platforms

In the Appendix, ASL (Application Specification Language) is briefly described. ASL is a major development of the IPB business rules language, and is the author's design of a System Model specification language which enables the capture of a complete System Model in a precise, but technology independent, form.

2 The IPB Project

2.1 Objectives

IPB was designed to assist insurance companies to build insurance policy management systems which enabled them to introduce new insurance products rapidly, and without the need for additional programming by IT staff. It did this by allowing actuaries to specify a new product as a structure in a 'Bill Of Material' form, and to specify new product behaviour in the form of business rules. The rules covered things such as product instance validation and premium calculation.

Within the generic policy management system (to be written by the customer) product methods were defined which encapsulated all the product specific behaviour. When any product specific behaviour was required, the policy management system would invoke a product method, supplying the product identifier as the first parameter, plus any other parameters. This would invoke the run time component of IPB which would retrieve the particular product rule, execute it, and return the results to the calling environment.

2.2 IPB Architecture

IPB consisted of a product and rule development environment targeted at actuaries, and a run time component which provided a rules execution engine. Version controlled product specifications and rules created by actuaries in the development environment were compiled and exported to the run time rules engine dynamically without the need for IT intervention, or halting of the policy management system.

The lead designer of IPB was Dr Trevor Hopkins, formerly of Manchester University, and if the BRWG wishes to understand the design of IPB in detail then they are advised to contact Trevor Hopkins directly, and to make a formal request to IBM for more information about IPB.

2.3 IPB Components

2.3.1 Product Specification GUI

IPB provided a GUI to enable the easy specification of product structure. In the IPB approach to products, all insurance products consist fundamentally of a number of paired benefits and payments. The customer makes the payments, and the insurance company provides the benefits if the conditions of the policy are met. The GUI allowed actuaries to build a multi-level hierarchical product structure by dragging and dropping already defined components onto a product structure tree.

2.3.2 Logical Data Model

IPB provided a textual language for the specification of the logical data model. The logical data model was normally specified in third normal form, so that it was as simple as possible, and made writing of business rules as simple as possible. It consisted of the definition of a number of Types, their Attributes, and the relationships between them. The initial version of the language, for political and historical reasons, had a relational basis and was not capable of describing every static object model that can be described in UML. Later versions were planned which would have been fully equivalent to UML.

2.3.3 Rule Specification GUI

IPB allowed actuaries to define business rules and algorithms either via a GUI or through the use of a text editor. Although the GUI allowed most, but not all, rules to be specified by a point and click interface, it proved slow and clumsy in practice. Actuaries preferred to specify the rules directly in the rules language, using a slightly enhanced text editor.

2.3.4 Rules Language

IPB had its own unique rules language. It was a combination of a functional programming language based on SML (Standard Meta language), and a database retrieval language based on the notion of a classification.

A classification was a function which took an input set of Types (i.e. Entities or Business Objects) defined in the logical data model and returned a set of Types which were those Types from the input set that met a business rule. The general form of a classification was:

```
inputSet businessRule
```

The business rule was required to be a Boolean valued function. It was applied against each member of the input set and determined whether it became part of the output set.

There were two types of rules, functional rules and relationship rules. Functional rules operated against the properties of each business object in the set, for example:

```
Person ((today - birthDate) > years(50))
```

selects people over fifty where birthDate is a property of the Type Person. Relationship rules select a Type if a specified relationship is present, for example:

```
Person hasReservationFor 1 orMore Hotel
```

selects people who have a reservation for a hotel. Classification specifications could be nested, so for example:

```
(Person ((today - birthdate) > years(50))) hasReservationFor 1 orMore Hotel
```

selects people over fifty who have a hotel reservation.

Typically a classification implementing a business rule would be applied against a single Type selected by its identifier, and if the resultant set was empty then the Type instance had failed the business rule.

IPB provided built-in accessor functions to access the properties of business objects retrieved via classifications. Therefore any retrieval of data possible in SQL was also possible in the IPB rules language. Language extensions to include updating of business objects were also designed but not implemented.

The functional aspects of the IPB rules language were similar to SML. Functions could be nested to any level. It was strongly typed, with an extensive set of logical datatypes unrelated to any particular hardware. They were organised into a hierarchy, so for example a function parameter could be defined as number which meant that the parameter could be integer, real, decimal or ratio.

A key aspect of the IPB rules language was that it was completely side-effect free. In combination with the strong datotyping this enabled a very high degree of checking by the compiler, and many common programming errors were simply not possible in the IPB rules language. Once a rule or algorithm had successfully compiled it would always run successfully to completion, though there could still be logical errors. However experience showed that the logical errors remaining after successful compilation were few and easy to find. This was assisted by the use of functional decomposition which allowed lower level functions to be fully tested before they were used in higher level functions.

A unique feature of the IPB rules language was that though classifications were specified against the logical data model, not the physical database design, the C and SQL generated operated against the physical database design. This meant that business rules could be specified against a simple business view of the database, and business people specifying rules did not need to be aware of the details of the physical database design. It also allowed the physical database to be retuned without the need to change the source code of any business rule.

As a result of the above features any rule or function written in the IPB rules language was a pure business rule and completely independent of any physical implementation. IPB rules and algorithms were purely at the System Model (logical) level of the Zach-

man framework, and were uninfluenced by the Technology Model layer and lower layers.

Although the IPB rules language was primarily a functional programming language, it did have OO features. Partly this was because IPB itself was written in Smalltalk which meant the development team were very well aware of the benefits of OO programming. The polymorphic datatype system has already been mentioned. In addition Derived Attributes could be defined for Types which were functions taking a particular Type instance as a starting point and deriving an attribute via any combination of computation and database retrieval. A Derived Attribute could have input parameters, so they were read only, side-effect free, methods of the Type. A Derived Attribute could be used anywhere a normal Attribute could be used.

2.3.5 Physical Database Design

IPB also provided a related language which allowed for the complete specification of the physical database design, and the mapping from the logical model to the physical database design. The physical database design could be highly denormalised, and any denormalisation that could be expressed through relational algebra was supported.

2.3.6 Rules Compiler

IPB included a compiler which compiled the business rules and algorithms to a mixture of C and SQL. One important feature of this compiler was the extensive checking made possible by the strong datatyping and side-effect free functional programming language.

However the most important and unusual aspect of this compiler was that it took in business rules written against a logical data model in third normal form, and generated C and SQL against a physical database design that could be highly denormalised. This was possible because the compiler had as inputs the business rules, the logical data model, the physical database design, and a built-in knowledge of SQL and C.

This key design feature made it relatively easy for business users to write business rules, while still allowing efficient production strength code to be generated.

2.3.7 Run Time Environment

Version controlled products and rules could be exported from the development environment to the run time environment. This did not require the policy management system to be halted. Product definitions were exported as additional rows to a DB2 product table which was defined in both the logical model and the physical database design. Rules and algorithms were exported to an MVS program library as C modules with embedded SQL.

The run time environment provided a data server rules engine. Rules were invoked via the product methods described earlier. There was no mechanism for directly invoking the rules engine from a client or a middle tier business logic layer.

2.4 Project History

The project ran from mid 1995 to early 1998 with an average number of staff involved of around fifty people. Version 1 was delivered in 1997 but had a number of significant limitations. Version 1.1 was delivered in 1998 and was a reliable, usable, product which produced code which was quite efficient given the newness of the language and the early point in the life cycle of the compiler. Efficiency of the generated code in terms of CPU consumption ranged from 8 to 0.8 times that of carefully optimised hand coding, with a median value of around 2.5. The development team saw no fundamental reasons why the generated code should not eventually match or even exceed the efficiency of hand written code.

The project was moderately successful from a sales point of view. Several licences were sold to major insurance companies at a price which covered a significant part of the development cost. However in two major markets, namely the USA and Japan, customers, encouraged by some IBM staff who were against the project for political reasons, adopted a 'wait and see' attitude. This lack of sales in the USA and Japan meant that the project was not immediately profitable.

The project was funded directly by the IBM insurance business, and was not part of IBM product development or IBM Research. At the time the IBM insurance business was under pressure to maximise return on capital over a two year plan horizon, and so it decided to cancel the project. The project team argued strongly that over a five year period the project would break even, and would be strongly cash flow positive thereafter, but to no avail.

3 Role

The role of the author was to work with potential customers of IPB at a technical level. The first part of the role involved understanding customer requirements for the rules engine, understanding how customers would integrate it with their own applications, and

influencing the external design of IPB accordingly. The second part of the role involved pre-sales technical support, particularly demonstrating to customers that the IPB rules language was sufficiently powerful to express the complex rules and algorithms found in the insurance world. This involved coding up a number of very large and complex customer actuarial calculations in the IPB rules language.

4 Use Of Frameworks

IPB provided a framework for the server side execution of business rules, and was not directly a user of other frameworks. It did assume a single logical database for the application using IPB. Although it did not have any support for a physically distributed implementation of the single logical application database, it would have been perfectly possible (though expensive) to extend the logical to physical mapping so that the rules were compiled to execute against a distributed physical database design.

IPB did match very precisely the levels of the Zachman framework. Specifically it provided a language for expressing the Data and Function columns of the System Model (logical) level of the Zachman Framework. The language was completely unaffected by the content of lower levels of the Zachman framework. The IPB physical database design specification was an implementation of the Technology Model Data column, and the source code generated corresponded to the Detailed Representation Process column.

5 Use Of Modelling

IPB assumed that the customer business analysts using IPB would produce a static object model or an entity relationship model for the application database. This could then be easily captured for IPB purposes using the IPB logical data model specification language. One of the planned IPB enhancements was to generate the IPB logical data model specification from a UML based modelling tool such as Rational Rose.

There was no expectation that customers using IPB would produce any particular kind of process model, though obviously where customers had produced and documented business rules as a precursor to application development this would be useful in precisely capturing these rules in the IPB language. Traceability back to earlier capture of business rules could easily be supported by the use of standard comments.

Within the development of IPB itself a complete UML static object model was produced which covered all persistent classes. This was produced using Rational Rose and was the key piece of high level internal system documentation that was maintained throughout the project. IPB Smalltalk classes were not generated from this, but there were formal manual procedures within the project to ensure that the IPB classes exactly matched the UML static object model.

Use cases were used to capture the design of the user interaction with IPB. These were fairly detailed but did not attempt to cover every abnormal situation that could arise.

Other OO design tools such as object interaction diagrams were not used. However internal interfaces between the various parts of IPB were very thoroughly and completely documented.

6 Handling Of Rules

These questions do not really apply to IPB itself but rather to the developers of applications who wished to use IPB so it will be answered in that context.

b) The potential customers of IPB were not potential customers because they wanted an implementation tool for business rules that they had already captured. They were potential customers because they were looking for more flexible applications which allowed them to introduce new products more rapidly. Market pressures were forcing them to introduce innovative new products quickly, and the major bottleneck was the updating of their existing, often very old, IT systems to manage these new products. Building an application which was a product independent policy management framework, and using a rules engine was seen as the best way to provide this flexibility.

c) The aspects of business rules covered by IPB were strictly limited to business behaviour. Other aspects such as software requirements, system requirements, architecture, etc. were not covered by IPB business rules.

d) e) In the case of IPB used for its original purpose in the insurance industry, the prime owners and managers of the business rules were actuaries. They were responsible both for writing and managing the rules. Actuaries are a rather atypical bunch of users who are very aware of their responsibilities, and do unusual things like providing a specification of a premium calculation in ALGOL. However the expectation of the development team was that IPB would eventually become a fully generalised rules engine with normal business users where responsibility for business rules would be much less well defined. IPB already provided a degree of user security and the expectation was that the ownership of rules and the tracking of changes would need to be enhanced for a generalised version.

f) A brief description of the IPB rules language was given earlier. The author suggests the BRWG request a copy of the IPB Rules Language manual from IBM. This provides a complete reference manual of the language and also includes many examples.

The fact that the IPB rules language was a functional programming language like SML was a culture shock for experienced programmers used to procedural languages like Fortran, Cobol, C, or even JAVA. It required a different way of breaking down a problem into code. The author with a background in procedural and OO languages found a functional language difficult at first. However customer personnel with no programming background did not have any significant difficulties. The main reason for this is that a

calculation in a functional programming language looks very like the arithmetic we all did at school. After working with the IPB rules language the author has become a huge fan of functional programming. Writing a large complex programme and having it run through to completion on its first execution was a totally new and amazing experience for the author. The combination of strong datatyping and freedom from side effects is very important in designing a rules language which is sufficiently powerful but keeps to the absolute minimum the possibility of programming errors. The hard truth is that any precise expression of business rules is programming, and unfortunately many business rules are complex. So minimising the opportunity for error is a vital part of the design of a business rules language.

The author believes that the best language for expressing business function is an OO language where the methods are specified using strongly typed functional code, not procedural code.

g) IPB did not provide the ability to specify directly any constraints on the structure or content of the database. However rules could be executed as part of some transaction and if the rule result indicated that the business data was invalid the transaction logic could take appropriate action.

The author believes that there is value in directly specifying business rules which are database constraints, and this facility is included in ASL (Application Specification Language) which is described later. At the Business Model level of the Zachman framework it is legitimate just to specify a business rule as an invariant. However when one moves down to the System Model level it is essential to specify when the constraint is tested and what action to take if the constraint fails. ASL defines that a business rule is tested prior to the commitment of any relevant database update by a transaction. If the business rule is violated the transaction is failed, and a business error message returned. Therefore in ASL a business rule specification must include a business error message, and requires compliance with a standard for transaction return signatures so that the error message can always be included in any transaction return signature.

h) As described earlier the IPB rules were C code and static SQL generated from the IPB rules language, and executed at run time. Their outcome would vary dependent on the input parameters and the data in the database.

i) j) IPB specifically allowed business users to change rules, but what became very clear was that technology which allows a business user to specify a business rule, and put it into operation within a few minutes, in a running on-line system, is potentially very dangerous. Even though the technology allows a business user to spend a few minutes updating a rule, hit a button, and have it in operation a few minutes later, the following precursor steps are still absolutely essential:

- discussion with colleagues
- agreement and documentation of planned rule changes
- preparation of test cases
- full testing of changes in the development environment

- review of test results
- testing of changes in a system test environment
- review of test results
- formal authorisation to go live
- pilot live use
- review of pilot use

IPB customers envisioned taking at least a week to go through these steps and in most cases considerably longer.

k) IPB rules were seen as stand alone specifications and were not part of any other model.

l) IPB rules referred to the types and attributes specified in the logical data model.

m) The consistency of the business rules with the logical data model was checked by the IPB compiler and any errors flagged to the user.

n) o) The source of the business rules in IPB varied across the different IPB customers, but as the authors of the rules were generally all in a single small department this was not a major issue for IPB.

p) Control of business rules in IPB was relatively straightforward as all business rules were identified as a particular point in a 2D matrix defined by the list of all products and the list of all product methods. In addition there was a third dimension of a version control number. One of the lessons learnt was that version control, and the release and control mechanisms, were a much bigger and more important part of IPB than first realised.

7 Benefits Of Using Business Rules

Unfortunately due to the cancellation of IPB, a policy management system using the IPB rules engine was never completed, so it is not possible to make any statement about the benefits.

8 Solutions For Major Limitations/Issues

8.1 IPB Limitations/Issues

When IPB was conceived, the belief was that building a rules engine was the difficult high technology task that IBM should undertake, and that building a product independent policy management system which utilised IPB was a relatively straightforward task that IBM customers could undertake. IPB was indeed a highly sophisticated and complex product, but there was enough skill within IBM to design it, build it, and deliver it more or less to schedule, and as originally conceived. Getting business users to write and manage business rules was not a major problem either. The major problems all occurred in the attempts to build a product independent policy management system to utilise IPB. It was this failure to be able to offer at least one successful example of a policy management system that used IPB which limited its sales.

Writing a product independent insurance policy management system is a lot harder than it seems at first. Typically such systems are absolutely riddled with assumptions about product structure and behaviour. Everything from screen layouts, to workflow, to calculations has subtle product assumptions and dependencies built in. It requires a very high degree of generalisation and abstraction to remove all this product specific code into product methods. It is surprisingly difficult, and not the way most application developers think. Also the system that results tends to be poorly optimised for most products, and the performance tends to be a lot worse.

A particular instance of this general problem is the difficulty in defining static parameter lists for product methods. For the idea of product methods to work, each invocation of a product method has to have a parameter list structure and return value structure that is the same for all products. Otherwise the code in the policy management system to build the parameter list becomes product specific which is what the designers are trying to avoid in the first place. But often the parameters needed to calculate say an insurance premium will vary hugely between products. So one potentially ends up creating a huge parameter list that contains all possible parameters which is very tedious and clumsy. The other alternative is to use a minimal parameter list, and allow the product method to retrieve the data it needs from the database. This is undoubtedly the correct logical answer, but it has a big performance hit, as it will involve duplicating a lot of expensive database I/O that in a product specific system would not be required.

The other major limitation of IPB was that it was strictly a data server rules engine, allowing execution of rules only in the context of a database transaction or batch program. This meant that product specific logic or other business rules that needed to execute on a client or middle tier layer were not supported. This was particularly an issue when it came to validating a new insurance policy. IPB allowed a new policy that had been written to the database to be validated, which is of course necessary. However it is not very satisfactory for a user to spend say quarter of an hour with a customer capturing want

the customer wants and then to be told that the policy is invalid. Ideally the policy is validated on the user's machine as data is entered. This is actually a hard problem, which is not just about the availability of a rules engine to a client machine. It is easy to specify rules that operate against a complete policy and determine its validity. It is much harder to express rules which say a particular addition or value makes a partially defined policy definitely invalid, or once certain choices have been made to make further choices automatically which are now required.

8.2 Conclusions From The IPB Experience

8.2.1 Generate From A Complete Precise System Model

IPB successfully generated good production code from formally expressed business rules that were part of a System Model (note a System Model, NOT a Business Model). The difficulties all occurred in joining the generated rules to a separately compiled static system written in an old-fashioned lower level language such as Cobol. To the author the obvious answer to this problem was to specify a System Model of the whole application in an extended 'business rules' language and generate from that. There would then be no mismatch between two very different languages, and most importantly the generator could do a proper job of optimising across the business rule code and the rest of the system.

Extensions to the IPB rules language that were discussed and agreed informally by the team would have allowed business analysts to specify complete ACID (Atomic, Consistent, Isolated, Durable) transactions which included database writes, business rules and business algorithms. These extensions would have allowed the insurance product dependent business rules to have minimal parameter lists, because a generator implemented memory scratchpad area would allow efficient reference by business rules to database data already read into memory by the invoking transaction.

Therefore fairly simple and obvious extensions to the IPB rules language would have allowed the efficient generation of a complete database server, including all the transactions that encapsulated it to ensure its business integrity. To specify a complete application, further extensions are needed which enable the specification of:

- extended business processes that involve multiple ACID transactions
- business events
- the user interface

Note that all these additional specifications must also be at a System Model level, and be completely independent of implementation technology.

Before language extensions can be designed, the underlying semantics of what the language is specifying need to be defined. The author set out to do this and the result was a technical note published in the May 2000 issue of the IBM Systems Journal (see <http://www.research.ibm.com/journal/sj/392/bevington.html>). This technical note, plus subsequent work, has informally defined the following additional System Model constructs:

- **dialogue** - A process with a defined start and end in which a single user interacts with the system and invokes one or more transactions.
- **workflow procedure** - A process with a defined start and end in which multiple users interact with the system.
- **batch process** - A method of the database which may invoke multiple transactions.
- **event** - A change of database state which has business significance and which may initiate a workflow procedure.
- **desktop object** - An object whose data is visible to the user, and whose methods may be directly invoked by the user. It may contain other desktop objects as properties. The data of desktop objects is derived from data in the business objects contained in the database, and desktop object methods may invoke transactions, dialogues, workflow procedures, or batch processes.
- **desktop** - The specialised desktop object which is the first desktop object presented to a user after logon. Its methods allow the user to populate it with further desktop objects, and invoke transactions, dialogues, workflow procedures, or batch processes.
- **application** - A collection of desktops which are grouped together for management and control purposes.

With the constructs that are already well recognised, namely:

- **logical database**
- **business object**
- **transaction** (ACID)
- **business rule** (a rule which constrains the state of the database)
- **business function**

these new constructs enable the specification of a complete System Model from which in principle production code can be generated .

Note that there are two different kinds of 'business object'. The Business Objects which are persistent and encapsulated within the database, and the Desktop Objects which are temporary views of data provided for the user. Normally desktop objects will be signifi-

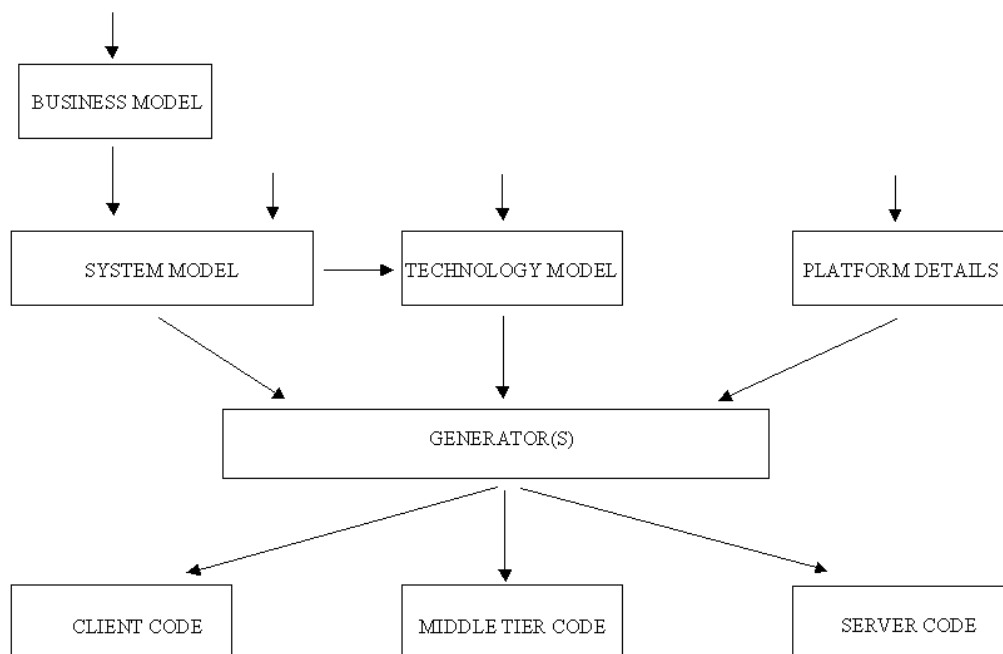
cantly denormalised, to provide different views of data for different users. For example imagine a Person business object, a Flight Instance business object, and a Reservation business object which relates them. An airline employee loading flights will want to see a Passenger List desktop object, showing for a particular flight all the passenger details, while customers will want an Itinerary desktop object listing all their flight reservations. These two different denormalisations to create two different desktop objects with significant duplication of data are correctly part of the System Model, while any denormalisation of the business objects in the database for performance reasons is part of the physical database design in the Technology Model. Many object modellers have no understanding of this key difference and build applications with a single set of denormalised business objects which have to serve the conflicting requirements of multiple users and physical database design.

Based upon this set of semantics the author has defined a formal language called ASL (Application Specification language). It is a major upgrade of the IPB rules language and allows the specification of a complete System Model in a precise form. The definition of ASL is currently contained in a document of around a hundred pages which is available from the author. An example application has been coded up in ASL and 'hand compiled' to the simplest possible single machine implementation in JAVA. This provides an executable version of the System Model and was done to demonstrate the completeness of ASL semantics.

8.2.2 Produce System Models By Transforming Business Models

Most people working in model driven development have tended to concentrate either on Business Models or on Technology Models. As a result the System Model layer of the Zachman framework has largely been ignored. People, including IBM Research, have been deceived by precisely expressed forms of Business Models into thinking that their precision will allow useful generation of code. Small fragments of code can be generated from a precise Business Model, but nothing extensive and really useful. The reason is that there is correctly too much missing from a Business Model to allow useful generation.

IPB successfully demonstrated that the correct way to generate useful amounts of code is as follows:



A Business Model has to be transformed by further design input into a System Model, and then combined with a Technology Model and platform details before useful integrated code can be generated.

Business Models are fundamentally about capturing an understanding of a business. A System Model on the other hand is a particular design intended to automate some part of the data processing described in the Business Model. A Business Model should capture the fundamental data and processes that the business must continue to perform on an ongoing basis. It should not be dependent on any particular organisation or geography of the business as these will change regularly. In designing an application, and capturing that design in a System Model, the following significant transformations and additions to the Business Model need to be performed:

- split the enterprise data model into a number of discreet logical database designs with some data duplication between them
- add further minor business objects and define every property of every business object
- split the overall business function into a number of separate applications and define the interfaces between them
- decompose business processes into workflow procedures, dialogues, batch processes, transactions, business rules, and business functions
- define the error messages that will be sent to a user when they attempt an action that would violate a business rule
- define the structure and content of the data that particular user groups can view
- define the processes that particular groups of users are allowed to invoke

These actions transform a description of the enduring data and processes of a business, i.e. a Business Model, into a particular design for an application, i.e. a System Model. During this transformation much of the content of the System Model will be derived more or less directly from the Business Model. Where this occurs there needs to be good traceability from Business Model to System Model.

Business Analysts need to stop thinking their job is done when they have produced a Business Model and carry on to produce System Models. Model-driven development has lost credibility because on too many occasions the only result is a Business Model gathering dust on a shelf.

8.2.3 Develop an OMG System Model Language

The OMG needs to sponsor work on a formal language for the capture of complete System Models, and support and encourage the development of generators that will further transform all of the System Model, not just the business rules, into useful code.

This language clearly needs to be part of the UML/OCL family but the requirements of a System Model language will drive some significant differences. For example there needs to be a database update facility which means the language will not be completely side ef-

fect free, though of course it should be as side effect free as possible. It also is not enough in a System Model language just to specify an invariant. Both when the invariant will be tested, and what will happen if the test fails, need to be made explicit. Probably more than one language will be required to cover all aspects of a System Model. For example the specification of workflow procedures has some unique demands. The language really has to be procedural with side effects. Also after each step of the procedure, the language needs to specify not only the next steps, but also who does them. A graphical concrete syntax probably does this best. Transactions on the other hand are much better specified in a textual functional programming language where the only side effects occur through updates to the database.

Whether the OMG formal System Model language is based on the IPB rules language and its extension into ASL is not important. What is important is that the language does the job required, is a good language design from a technical language design viewpoint, and becomes a widely adopted standard.

8.2.4 Summary

For model-driven development to be effective Business Models containing business rules need to be further transformed into System Models expressed in a precise formal language. The OMG should sponsor the development of a standard System Model language, and then should encourage the development of generators based upon it.

9 Appendix: An Overview Of ASL

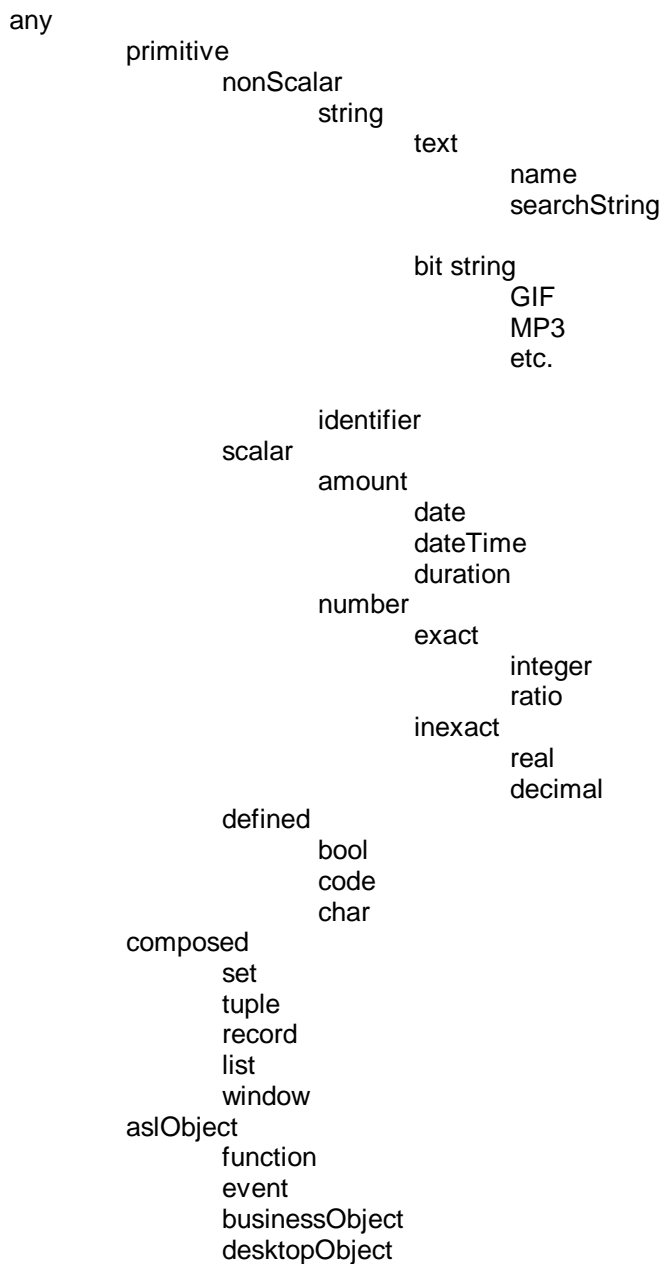
ASL is a language designed to allow the specification of a complete System Model free from any implementation dependencies. It has the following characteristics:

- it is an OO language where programmer-specified classes are instances of one of twelve specialised meta classes built into the language
- it is strongly typed with an extensive set of polymorphic datatypes
- it is as side effect free as possible
- it is a functional programming language in that object methods are specified in a functional programming style, and stand alone business functions may be specified
- it is a database query and update language
- it is not a general purpose programming language, but a specialised language only capable of specifying a business application

In designing ASL there was no intention to merge capabilities from several different kinds of programming language. The characteristics of ASL emerged naturally from the purpose of the language.

9.1 ASL Datatypes

ASL has a comprehensive hierarchy of datatypes as shown below.



A variable declared to be of a datatype which is not at the lowest level in the above hierarchy may be of any datatype which is lower in the hierarchy.

9.2 ASL Meta Classes

All specifications within ASL are instances of one of the following twelve meta classes:

- **database**
- **business object**
- **event**
- **business rule**
- **function**
- **transaction**
- **batch process**
- **dialogue**
- **workflow procedure**
- **desktop object**
- **desktop**
- **application**

The ASL meta classes free business analysts from having to specify explicitly all the behaviour that is common to each meta class instance. The common behaviour is built into the definition of each meta class. The result is that an ASL specification is much shorter and more compact than it would be in an OO language which has the single meta class Class.

All instances of these meta classes can be regarded as classes, however some of them have properties and no methods, some have properties and methods, and some have no properties and a single method. The general form of an ASL meta class instance specification is

```
nameOfMetaClassInstance metaClassName Keyword .... Keyword (  
    propertyName datatype;  
    .  
    .  
    propertyName datatype;  
    methodName (parameterList) returnedDatatype (.....method logic .....);  
    .  
    methodName (parameterList) returnedDatatype (.....method logic .....))
```

However some meta classes are basically just functions, and so are specified in a shortened form as in a normal functional programming language as follows.

```
nameOfMetaClassInstance metaClassName (parameterList) returnedDatatype Keyword .... Key-  
word  
    (.....function logic .....);
```

Also some meta classes, for instances database, have a shortened form which makes the specification quicker to enter, and enforces certain meta class constraints.

9.3 Functional Programming

The logic of methods and the logic of functions is specified in ASL using functional programming based on SML (Standard Meta Language). All logic in ASL consists of functions which always return a value. Functions may be nested to any depth.

For example the if then else statement of ASL is a function as follows

```
if expressionA  
then expressionX  
else expressionY  
end
```

where expressionA must evaluate to a Boolean. If expression A evaluates to true then the value of the if then else function is the value of expressionX . If expressionA evaluates to false the if then else evaluates to expressionY. So for example the statement

```
2 + if 4 > 3 then 6 else 5 end
```

always evaluates to 8.

Similarly loops are functions which return a value, though they do have an internal variable which is updated in place. So the standard do loop in ASL is as follows

```
loop i date from 1/1/2001 to 1/1/2002 by week(1)  
set acc [dec, int ] = [0.0, 0]  
do [acc.1 + 10.0, acc.2 + 1]  
end
```

where i is the loop control variable, with the upper and lower limits being two dates and the loop increment a duration. The local loop variable acc is a tuple which may be updated in place during each loop iteration specified in the do clause. The value returned by the whole loop function is the final value of acc after the last iteration.

ASL contains a comprehensive library of built in functions covering all the normal control statements, as well as all the normal functions on the various datatypes.

ASL also supports both recursion and high order functions. High order functions may seem over the top in a business specification language but there are some common business situations where the calculations require the retrieval and execution of a function.

For example building a flexible insurance policy management system. IPB was effectively providing an implementation of high order functions for a programming environment that does not normally support them,

9.4 ASL Database Definition, Query, And Update

ASL is also a database specification, query, and update language. With some minor extensions this part of ASL is currently the same as the IPB rules language described earlier in section 2.3.3. However it would be a straightforward task to modify ASL to be fully UML/OCL compatible, with UML style definition of the logical database and its relationships, and OCL style database navigation.

9.5 I/O

ASL provides a unique I/O function and a special composed datatype which provide an abstraction of human/computer interaction via a user interface. The logical interactions with a user, such as data presentation, data entry, selection of options from a list, requesting an action, etc. can be specified. Note that the user interface defined by ASL does not have to have a graphical implementation. An audio implementation is possible, or if the user was a robot a direct implementation by returning the appropriate I/O function return values.

The ASL I/O functions are only allowed within the context of a dialogue, and allow the dialogue to be a dynamic interaction with the user.

9.6 ASL Users

It is not anticipated that the users of ASL will be business personnel. Precise specification of business function in full detail is programming, and even with the removal of all technology considerations, can be complex. It is anticipated that business analysts with a sound IT background will write ASL specifications based on input from business personnel.

The best way for business users to verify that an ASL specification meets their requirements is to generate an executable prototype from the ASL specification. The prototype can be used both to verify the business function, and to generate a set of test cases for the final implementation.