Highlights

- Model-driven tools can reduce the cost of development and verification.
- Information systems can be produced automatically from object oriented designs.
- A formal, model-driven approach is proposed for use in safety critical systems.
- A framework is provided for the correctness of model transformations.
Formal Model-Driven Engineering of Critical Information Systems

Jim Davies, David Milward, Chen-Wei Wang, James Welch

Abstract

Model-driven engineering is the generation of software artefacts from abstract models. This is achieved through transformations that encode domain knowledge and implementation strategies. The same transformations can be used to produce quite different systems, or to produce successive versions of the same system. A model-driven approach can thus reduce the cost of development. It can also reduce the cost of verification: if the transformations are shown or assumed to be correct, each new system or version can be verified in terms of its model, rather than its implementation. This paper introduces an approach to model-driven engineering that is particularly suited to the development of critical information systems. The language of the models, and the language of the transformations, are amenable to formal analysis. The transformation strategy, and the associated development methodology, are designed to preserve systems integrity and availability.

Keywords: model-driven engineering, formal methods, critical systems, information systems, data migration

1. Introduction

1.1. Failures of critical systems

Our society is increasingly dependent upon the behaviour of complex software systems. Errors in the design and implementation of these systems can have significant consequences. In August 2012, a ‘fairly major bug’ in the trading software used by Knight Capital Group lost that firm $461m in 45 minutes [1]. A software glitch in the anti-lock braking system caused Toyota to recall more than 400,000 vehicles in 2010 [2]. The total cost to the company of this and other software-related recalls in the same period was estimated at $3bn. In October 2008, 103 people were injured, 12 of them seriously, when a Qantas airliner dived repeatedly as the fly-by-wire software responded inappropriately to data from inertial reference sensors [3].

In critical systems development, processes are put in place to detect errors and to mitigate their effects. Incidents such as those listed above have many contributing causes, many of which will be identified as failures of process. For example, only one of the eight causes given for the loss of the $125m Mars Climate Orbiter satellite [4] was directly related to development: “verification and validation process did not adequately address ground software”; the others were failures of communication and procedure. There is nevertheless considerable advantage to be gained from the adoption of automatic tools and techniques that promote correctness in development; if these can be used to eliminate certain kinds of error, then the dependency upon other processes is reduced; expensive, manual effort can be focussed upon surrounding issues of management and validation.
1.2. Critical information systems

Information systems typically contain large amounts of valuable data subject to complex constraints. The value of the data will usually exceed that of the system: the consequences of data loss, data corruption, or inappropriate access may be unthinkable. This is not simply a matter of deletion. Consider, for example, a clinical system that holds information about doctors and patients. Such a system might hold, as part of a staff record, a list of patients allocated to a particular doctor. The allocation of a doctor may appear also as part of a patient record. If the system reaches a state in which patient A appears on the staff record for doctor B, but doctor B is not listed on the patient record for patient A, then information has been lost. If that information were required for a subsequent decision, then the consequences could be unfortunate.

The constraints upon the data in the system may be representation or type invariants: if these fail to hold, then the behaviour of the system may be unpredictable. Alternatively, they may be semantic constraints or business rules. The symmetric relationship between allocation data on patient and doctor records is a simple example of a semantic constraint: if it fails to hold, then the meaning of the data within the system is unclear. A constraint that no patient is allocated to more than one doctor might be seen as a business rule: if the allocations are properly, symmetrically recorded, then the meaning may be clear, but the rule has been broken. This may mean that some function on the system fails to work as expected. These categories are not exclusive: depending upon the implementation architecture, some business rules may correspond also to representation invariants.

1.3. Formal, model-driven engineering

We might hope to establish, by way of formal specification and proof, that each operation on a system could be guaranteed to satisfy the data constraints. We might proceed by: writing a formal specification \( \text{Op} \) for each operation; writing a formal specification \( \text{C} \) for each constraint; and proving that \( \text{Op} \) preserves the conjunction of all \( \text{Cs} \). However, this approach has three, significant shortcomings: the specification of an operation may not be accurately reflected in its implementation; the system requirements, and hence the data constraints, may change, requiring that any existing proof be revisited; the proof may be difficult and expensive. It is also quite likely, on any particular iteration, that \( \text{Op} \) does not satisfy \( \text{C} \), and one or both specifications will need to be revised.

A scalable model-driven approach to development, in which software artefacts are generated automatically from precise, abstract models, offers a potential solution. If the implementation of each operation is generated automatically from its specification, and if the generation process is correct, then we can be sure that the specifications are correctly implemented. If the data constraints can be translated and incorporated, automatically, as part of the specifications, then we can be sure that these constraints will be satisfied in operation.

The initial expression of the constraints, and the initial specification of each operation, would still need to be validated. Furthermore, some formal proof may be required to determine whether the results of the translation process—extended versions of the operation specifications that are guaranteed to satisfy the data constraints—correspond to expectations. However, this validation and proof can be conducted at a higher level of abstraction, with many details delegated to a once-only proof of the generation and translation processes. The cost of developing critical information systems, where data integrity can be guaranteed and functionality is predictable, is thus greatly reduced.

1.4. This paper

In this paper, we introduce a formal language, \textit{Booster}, for the specification of data constraints and operations upon information systems. This language is inspired by earlier formal techniques, and set in the context of object-oriented design. We use the Z notation [5] to show how abstract models of information systems can be translated into more concrete designs, and then compiled into complete, working implementations, and how the processes of translation and compilation can be formally verified. We present a simple methodology for the iterative development of critical information systems. We discuss the application of the methodology in the context of critical systems development, exemplified by the development of a system for patient monitoring and self-management in long-term conditions.
This paper builds upon a contribution to the Formal Techniques for Safety Critical Systems workshop (FTSCS 2012) [6]: a methodology for establishing the correctness of the compilation process, from concrete design to working implementation, in the context of critical systems. That paper contains additional details of the generation of SQL code; this paper presents a substantially updated version of the methodology, extended to address transformation from abstract design to implementation. That paper contains a worked example based upon a hotel booking system; this paper contains an extract from the design of an actual critical information system developed using Booster: fully implemented, and relied upon by thousands of users.

This paper complements a contribution on Success Stories in Model-Driven Engineering [7]: a report upon the application of the approach to the development of three different information systems. In that paper, we outline the original version of the Booster language and methodology and discuss lessons learned in its application to three case studies. The version of the language presented here is quite different from that of [7] and other, earlier papers [8, 9]: framing information and other ‘compiler directives’ are no longer required; the constraint language is now the full first-order predicate calculus, rather than three classes of ‘programming constraint’; operations are now characterised as a single predicate, and may be composed using operation references. The transformations are now implemented in a declarative, functional language, more amenable to verification, and can be targeted at different platforms.

2. Language

Booster is an object modelling notation: the structure of a system is described as a collection of associated classes, in the style of the Unified Modeling Language (UML) [10]. The semantics of Booster are entirely consistent with the semantics of class diagrams in UML, and it is perfectly possible to use UML as an alternative means of presenting Booster models. The key difference lies in the way that operations are described. In most applications of UML, this is achieved using an imperative programming notation. In Booster, each operation is characterised by a single, logical predicate: describing the relationship between the values of attributes before and after the operation is performed.

UML has its own language of predicates: the Object Constraint Language [11]. This is an object-oriented language, with operation and property calls that can be used to specify structural constraints, pre- and postconditions, and guards upon the transitions in state machines. An OCL postcondition can contain references to the values of attributes before an operation is performed, and can thus be used—for primitive operations—in the same way as a Booster operation predicate. For compound operations, it would be perfectly feasible to extend the OCL language specification [12] to allow references to the pre- and postconditions of other operations, treating each OCL ‘context’ as a property of the operation. Existing UML tools, however, do not support this usage.

The syntax of OCL is considerably more verbose than that used within conventional predicate logic, reflecting in part a decision to make every aspect of the context of definition explicit. More importantly, the semantics of OCL is only partially defined when it comes to the specification of operations, reflecting the fact that UML sets out to support a range of different paradigms for interaction. In Booster, we are able to settle on a single paradigm, one that admits both abstraction and compositionality in design: every operation is implemented as an atomic transaction upon the state of the system—and not merely the current object. This greatly simplifies the interpretation of the predicate language.

We may observe that the behavioural semantics for UML is not compositional with regard to abstraction and concurrency: that is, if we allow concurrent execution of operations, then it is not possible to derive a behavioural specification of a compound model from the behavioural specifications of its components—unless that specification contains every detail of the implementation, and thus has no abstraction at all. In particular, if we allow concurrent execution, then a characterisation of operations in terms of pre- and postconditions is not compositional. This is a particular problem in automated model-driven engineering, where we need to derive the implementation of an object from the specifications of operations, and the implementation of a system from the specifications of its component classes and associations.

For these reasons, we have chosen to give Booster its own syntax and semantics, rather than adapt and extend those of UML. The syntax is based closely upon that of the B Method [13]: specifically, the Abstract
Machine Notation (AMN). The semantics, and the means of distinguishing between values of attributes before and after an operation, is based upon that of the Z notation [5]. Object-oriented extensions have been proposed for each of these formal methods—most notably, UML-B [14] and Object-Z [15]—but again these have taken quite a different approach to behavioural semantics, supporting manual operation-by-operation design but not the automatic derivation of an implementation for a system from specifications of its component classes.

An essential feature of the Booster approach is the decision that every operation should be implemented as a transaction upon the system, rather than the local object. This makes Booster unsuitable for the development of object-oriented programs in general, where concurrent execution of operations—even upon a single object—is standard practice. Booster is intended instead for the development of information systems, where concurrent access is limited to preserve semantic consistency, expressed in terms of essential relationships between different items of data. Information systems are often designed using UML: the classification and association of entities, the stratification of models and meta-models, and the description of the intended effect of operations upon them, offering significant advantages over earlier entity-relationship (ER) approaches.

2.1. Classes, associations, and attributes

A Booster model is comprised of a series of class declarations, each of which introduces a number of attributes, associations, operations, and constraints. Attributes may be of four primitive types: `String`, `Int`, `Float`, or `DateTime`; they may also take values from user-defined enumerations. Associations are declared in the usual, textual way: by declaring association ends of appropriate type. The multiplicity of an association is given by the declarations of the association ends: types may be introduced as mandatory, optional, or set-valued; the last of these has an optional, additional multiplicity constraint. Mandatory associations are declared without decoration; optional associations are declared using square brackets; set-valued associations are declared using the keyword `set`.

In Booster, all associations are paired. Without the ability to navigate in both directions, we may need to consider all of the objects of the source class when implementing an operation of the target class: we would not be able to refer directly to the set of currently-linked objects. As the normal mode of editing is textual rather than visual, the name of the mirror association is included as part of the declaration as an aide-memoire.

As an example, consider the following pair of class declarations, illustrated in Figure 1:

```java
class A {
    attributes
    m : Int
    b : [B : a]
}
class B {
    attributes
    n : Int
    a : [A . b]
}
```

The first of these introduces a class `A` with a single, integer-valued attribute `m` and an optional association `b`. An instance of this association will be a link to an object of class `B`. The mirror association `a` is identified as part of the declaration of `b`. 
2.2. Operations and constraints

An operation is declared as a single predicate on the values of expressions involving attributes, inputs, and outputs. Simple predicates are built from expression relational operators including equality (=), inequality (/=), set membership (\in), and set non-membership (/\in). Complex predicates may be constructed using the usual Boolean combinators of conjunction, disjunction, implication, and negation: \&, or, =>, and not. Universal and existential quantifiers may be used over finite sets of values, \forall x : X @ p and \exists x : X @ p, with bound variables x being introduced for the scope p. As in the Z notation, attribute names may be decorated with a prime (') to indicate that we are referring to the value of that attribute after the operation has been performed.

The name of another operation, qualified if necessary by a path to the object on which the operation is defined, can be included within an operation predicate. The effect is to include all of the constraint information of that operation, including the specified transformation of any attributes. This constraint information may be conjoined, disjoined, qualified, quantified, or negated, depending upon the way in which the other operation is being used as part of the operation being defined. It is possible also to combine predicates, and hence operations, using relational composition: p ; q denotes an operation that behaves as if there were an intermediate state, related to the initial state by p and to the final state by q.

Value expressions may be constructed using arithmetic, set, and sequence operators. The expression syntax is that of the Abstract Machine Notation. For example, \cup denotes set union, \cap denotes intersection, and card denotes the cardinality function. The same collection of expressions may be used also in constraints: invariant properties introduced within class declarations to represent integrity constraints or business rules. All associations are navigable at the model level and a constraint may refer to attributes declared in associated classes. Access control is enforced through interface specifications.

For example, the earlier class declarations may be extended to produce the complete model shown in Figure 2. Here, class A has a single operation Inc, whose intended effect is described by the predicate m’ = m + 1. Whenever this operation is performed, the value of attribute m afterwards should be one greater than it was before. A similar operation is declared in class B. Class A contains an invariant insisting that, in any object of class A, the value of attribute m should always be strictly less than 10. The invariant in class B insists, in any object of that class, that the value of n should be strictly greater than the value of m in a linked object a of class A. As a is declared as an optional attribute of B, this constraint will apply if and only if such a linked object exists.

The predicate presented as part of an operation declaration is only a partial characterisation: each operation is constrained also by the constraints of the model, and further constraints may be added by the transformation rules—reflecting heuristics about the interpretation of the original, user-supplied predicates in the context of a particular application domain. In this example, the invariant within A constrains the action of A.Inc. Whenever the operation is performed, it is not only the case that m must be incremented; it must also be true that the new value of m is strictly less than 10. It would be problematic to perform
A. Inc when the current value of m is 9: we cannot both meet the constraint of the operation and satisfy the model invariant.

In general, we expect the availability of an operation to be determined in part by constraints declared elsewhere in the model. For example, the availability of a. Inc for an instance a of class A, is determined partly by the constraint a. m < n declared in the associated class B: if there is a linked object b of class B for which m is one less than the value of attribute b. n, then to increment m without also incrementing b. n would violate that constraint. The transformation rules outlined in the following section ensure that operations cannot be performed when the resulting state would violate integrity constraints or business rules. In an implementation, this can be achieved through an exception mechanism, or by blocking an operation invocation, depending upon the target platform technology. In systems where a matching user interface is generated, any operation that is inapplicable in the current state of the system can be ‘greyed out’ or otherwise made unavailable to the user.

2.3. Translation

A Booster model might be characterised as a computation independent model, in the sense of the Model-Driven Architecture [16, §2.2]: it sets out what the system is expected to do, but does not explain how this is to be achieved. For example, the operation specification

\[ \text{Op} \{ \ x : \ s' \ \} \]

asserts that, after the operation is performed, the value or object reference x is present in the set s. This could be achieved by the assignment \( s := s \cup \{ x \} \). It could also be achieved by \( s := \{ x \} \), or by any assignment of the form \( s := \{ x \} \cup t \). In the particular case where x is already an element of s, it could be achieved by the trivial command skip.

A set of model transformations are used to translate a Booster model into a model that is platform independent but no longer computation independent. The target language is a variant of the language of guarded commands proposed by Dijkstra [17], and generalised by Nelson [18]. The core concrete syntax of this language is shown below:

\[
\text{Command} ::= \skip \mid \text{Path} := \text{Expression} \mid \text{new Id} : \text{Id} \mid \text{Constraint} \Rightarrow \text{Command} \\
\| \text{Command} \mid \text{Command} \| \text{Command} \\
\| \text{all Id} : \text{Expression} . \text{Command} \mid \text{any Id} : \text{Expression} . \text{Command}
\]

\skip is a command that is always available, and has no effect upon the state of the system. The assignment \( a := e \) assigns the value of expression e to attribute a, which may be qualified by a navigation path in the usual way. \text{new} is used to denote the creation of a new, named object reference in the course of an operation: the new reference (first identifier, of type Id) points to a data structure representing a newly-initiated object of a particular class (second identifier). These and any other commands may be guarded (\Rightarrow) by a constraint, with the result that they are available only when the constraint is satisfied.

The | operator represents a prioritised choice between two guarded commands. In the command \( p \mid q \), if the guard for p is satisfied, then the command will behave as p. If not, the program behaves as q, and if the guard for q is not satisfied, the command as a whole will not proceed. The command thus proceeds if the disjunction of the two guards is satisfied. In contrast, || represents parallel composition. In \( p || q \), each of the commands p and q will proceed, subject to the satisfaction of the conjunction of their guards. The | and || operators have generalised equivalents in any and all: the first operand declares a bound variable name; the second operand is an expression which denotes a finite set of object references; the third operand is a command which may be applied to any or all of the objects—in any order—referred to in the expression. Finally, the ; operator allows the sequential composition of commands within a single transaction: the effect of \( p ; q \) is that of executing p and then q.

The first stage of the translation process is the extension of each operation specification to include any other constraints upon the inputs and attributes involved. This involves the computation of a ‘transitive closure’ of related constraints: an attribute mentioned in the specification may be constrained relative to
another that is not; other constraints upon that other attribute may need to be considered; those other
constraints may involve further attributes, and so on. The applicability of an operation may depend upon
the value of an attribute appearing in a different part of the object model: the transformation rules make any
such dependency explicit, so that the operation can be correctly implemented. It may be that the dependency
is unintended and unwanted, in which case we might expect the model to be revised following inspection
of the extended specification and/or analysis of the implementation.

The second stage is the generation of a command for each operation. Each conjunct of the operation
specification is mapped to a statement in the guarded command language. For example, a conjunct stating
simply that \( a' = e \) will be mapped to the assignment \( a := e \). If there is no obvious translation, then the
conjunct is mapped to \( \text{skip} \): the condition must then be established by some other part of the command;
either that, or it must hold when the operation is invoked, and be preserved by the generated implementation;
if this is not the case, then the operation will be unavailable.

The third stage of the translation process is the generation of initial guards: for every choice command,
and for every ‘completed’ guarded command. In each case, this is the weakest precondition for the command
to achieve the extended operation specification, viewed as a constraint upon attribute values and outputs
after the command is performed. For the subset of the guarded command language used, these preconditions
may be calculated automatically: in the current version of Booster, we do not permit recursive definitions.
The generated guards are strong enough to ensure that whenever a sequential command \( p ; q \) is executed,
the result of executing \( p \) will be a state in which the guard of \( q \) is satisfied.

Similarly, the guards are strong enough to ensure that whenever a parallel command \( p | q \) is executed,
the specified result is guaranteed despite the potential for interference between \( p \) and \( q \). These two commands
may access the same variables, they may even assign to the same variables, but the effect of doing so within
the same transaction can be calculated, and the necessary constraint upon attribute values and inputs
imposed. For example, the specification \( x' = y \land y' = y' = 3 \) will be translated to the command
\[
y = 3 \rightarrow (x := y \land x := 3)
\]
in which the initial guard \( y = 3 \) has been added automatically to ensure that any execution of the operation
is guaranteed to satisfy the original specification.

In the current version of Booster, assignments to expressions referring to the values of attributes after an
operation is performed are mapped to \( \text{skip} \). It would be possible to perform a topological sort of conjuncts,
to determine whether there is an order in which the corresponding assignments or substitutions might
sensibly be performed. However, the advantage of doing so is quite marginal, and would seem to conflict
with the overall approach or philosophy. The translation process is intended to make the consequences
of decision intentions explicit, and to automate routine aspects of development, in a way that helps the
developer to discover errors in design, or to safely implement a design that is already correct. The fact that
the specification
\[
x' = g' \land g' = 3
\]
is mapped to the guarded command
\[
x = 3 \rightarrow (\text{skip} | y := 3)
\]
reflects this. An approach relying on topological sort—or, more generally, upon the solution of multiple,
simultaneous equations—would be less transparent, less compositional, and more fragile in the context of
continuing development.

In most cases, the translation of each individual conjunct is quite straightforward. The value of the
Booster approach lies in this process of discovery, modification, and automatic implementation. The calculated
guards take full account of class invariants, symmetry and multiplicity restrictions upon associations,
attribute type constraints, and potential aliasing of attribute names, as well as the original specification of the operation—and the specifications of any other operations included in its definition. For example, the operation predicate

\[ \text{Inc} \{ m' = m + 1 \} \]

would be translated into the following command

\[ \text{Inc} \{ m < 9 & (b /= \text{null} \Rightarrow b.n > m + 1) \Rightarrow m := m + 1 \} \]

This can be invoked only if the current value of \( m \) is less than 9, and less than \( b.n - 1 \), if there is an associated object \( b \) of class \( B \). If these conditions are satisfied, then we may increment the value of \( m \)—achieving the desired effect of the operation—while maintaining the model invariants.

2.4. Compilation

A second set of model transformations are then used to produce a platform specific model, tailored to a particular implementation architecture. In our initial work with Booster [7], we generated bespoke, in-memory databases. The current version targets instead commonly-used relational database platforms; this has the benefit of reducing the “proof surface”: the extent of code that needs to be certified, verified, or validated; in the context of medical applications we are able to assume that the underlying database engine can be trusted, having been “proven” through global commercial use.

The object model is translated into a relational schema, and implemented as a series of SQL statements. Each operation program is translated into a stored procedure which has the corresponding effect upon the state. Different transformations have been implemented to target specific relational database platforms; however, the subset of the SQL language used is such that only minor variations are required—for the most part, concerning naming restrictions. The same approach can be used to target other technologies with quite different notions of transaction: such as document stores and distributed file systems.

The structural aspects of the translation are relatively straightforward: similar translations are performed by various object-relational bridging tools such as Hibernate [19]. In Booster, classes and associations are mapped to individual tables; attributes translated to columns, and the inheritance hierarchy—not discussed in this paper—is flattened. For example, our running example might produce the following MySQL script extract for creating tables and columns:

```sql
create table A (id int auto_increment primary key);
create table B (id int auto_increment primary key);
create table A_b (id int auto_increment primary key);
create table B_a (id int auto_increment primary key);
alter table B add column n int;
alter table A add column m int;
alter table A_b add column A int;
alter table A_b add column B int;
alter table B_a add column B int;
alter table B_a add column A int;
```

In the generated database, classes \( A \) and \( B \) are implemented as tables in which each row represents the state of an object instance, storing the values of primitive attributes for which the multiplicity is either 1 or 0..1. The first column of each table stores a unique integer-valued primary key, which will be used for object-references. Attributes that are set-valued, or which denote associations, are held in separate tables. The bidirectional association also gives rise to a pair of symmetric tables: the table \( A_b \) stores links corresponding to the references in attribute \( b \) of class \( A \); the table \( B_a \) stores links corresponding to the references in attribute \( a \) of class \( B \).

A greater challenge lies in the definition of appropriate procedures: the feature set available in a particular vendor’s implementation of SQL may not support all of the obvious translations. In such cases, part of the
required functionality may need to be implemented at the next layer of the design, as part of an applications programming interface (API) to the data store. A simple example is afforded by the treatment of input parameters in MySQL: it is not possible to verify that an input is of a particular, primitive type; this check must be performed by the API.

For example, the operation \texttt{A.Inc} specified above would be implemented as:

```sql
create procedure A_Inc (in this int )
begin
  declare exit handler for sqlwarning, sqlexception, not found
  begin rollback; end;
  start transaction;
  if (select m from A where id = this)<9 and ((select count(*) from A_b where A = this) = 1 and 
    ((select n from B where id = (select B from A_b where A = this)) > 
      (select m from A where id = this) + 1)
  or (select count(*) from A_b where A = this) = 0)
  then 
    set @m = (select m from A where this = id);
    update A
    set m = @m + 1 
    where this = id;
  end if ;
  commit ;
end //
```

Here, the generated procedure checks to see whether the guard given in the specification holds for the current state of the database: whether the value of \texttt{m} for the current object reference \texttt{this} is less than 9, and also more than one less than the value of \texttt{b.n} for any associated data corresponding to a linked object of class \texttt{B}. If this is the case, then the update can be performed.

The benefits of the approach could be quantified in terms of the number of statements needed at each level of description. In the above example, a single statement within the specification of an operation gave rise to four statements in a guarded command description, and twelve statements in a SQL implementation. The relative complexity of expression evaluation in SQL, where the original specifications involve object navigation paths, means that an increase of at least an order of magnitude should be expected. The increase will be greater for models including class and association invariants, including constraints upon multiplicity.

The current implementation of Booster includes more than 1200 rules to support the two levels of translation, 300 of these are used for SQL implementation. Most of these rules are trivial, and most can be considered independently of the others. A small number of rules involve complex pattern matching and calculation, and need careful consideration. Once proved correct, however, they will serve for the production of many different systems.

3. Methodology

3.1. Model-driven engineering

In model-driven engineering, the abstract models may be seen as the source code for different aspects or components of the system. In describing the Model-Driven Architecture (MDA), a particular approach proposed by the Object Management Group, Franke explains that “MDA is about using modelling languages as programming languages rather than merely as design languages” [20]. For each development iteration, the constraint information contained in the model is all that the compiler has to work with. In particular, the compiler will need to determine what is to happen if the operation is called in circumstances where the
constraint is not applicable: that is, for combinations of state and input values that lie outside the calculated precondition.

As we have suggested above, if the generated system holds data of any value, then it would not seem sensible to allow an arbitrary update to the state. In the absence of any default action, the effect of calling an operation outside its precondition should be to leave the state of the system unchanged. Further, if we wish to adopt a compositional approach, in the sense that a composite operation should be inapplicable whenever one or more of its components is inapplicable, then it is not enough for the operation to leave the state unchanged; instead, its inapplicability must be recorded or communicated.

Within the precondition, the specification is applicable, and the intended effect of the operation is known. However, it may be that the compiler does not know how to achieve this effect: that is, part of the constraint information may lie outside the domain of the transformation rules. Where this is the case, the intended effect of the operation may be known, but is not achievable. The inapplicability of the specification will be reflected by documented, blocking behaviour of the generated implementation: a choice made in this domain to protect data integrity. The developer may then choose to revisit the model, expanding upon the specification to bring it within the domain of the transformation rules.

We may also encounter constraints that admit two different interpretations and—although implementations for each are easily generated—our heuristics are not strong enough to tell us which we should choose. In this case, we assume that the user would prefer to be informed of the issue, through the generation of a guard that prevents the execution of the generated command, which may be based upon either interpretation, whenever the two different implementations would produce different results. Consider, for example, a model that includes the information shown in the diagram of Figure 3. Here, for every instance of class C, the two attributes bp and bq should point to different objects of class B. As a consequence, for any instance of class B, the two set-valued attributes cp and cq should be disjoint.

When the operation AddQ is called with an input c? that is already in cp, then the overall intention is unclear: should c? be removed from cq, or should the operation be blocked? While it could be the case that either alternative would be equally acceptable, it is more likely that the designer has failed to make their intentions clear. Deleting a link may have consequences for other data: it may even be that, to achieve a new state in which the model constraints are satisfied, deletions need to be propagated across the whole system. Is this what the designer intends? We could add heuristics to resolve such choices, but for critical information systems it may be better to generate an implementation that blocks when intentions are unclear, instead of making unexpected or unintended modifications.

It is important also to recognise that any critical information system may come with expectations of availability. An implementation that guarantees to maintain the integrity of the data, but that does not allow some key operation to be performed would clearly be unsatisfactory. The compiler alone cannot check for this, and we require a complementary development methodology to ensure that only satisfactory implementations are deployed. There are two aspects to this methodology, one addressing the development of individual operations, and another addressing the continuing development of system models.
3.2. Operation development

Within a particular model of the system, we can use the automatic transformations to develop a complete specification for each operation. As illustrated in Figure 4, we may proceed iteratively as follows:

1. We write down a specification $P \land E$ for the operation, where $P$ is a description of the intended guard—the complement of the intended ‘unavailability’—for the operation and $E$ is a description of its intended effect. We write down also an availability condition $A$: a description of circumstances in which the operation should be available. We may write down also a ‘frame’ $F$ for the operation, a list of attributes that we might expect to be updated.

2. We use the automatic transformations to generate a guarded command version of the operation, $G \rightarrow C$, in which the overall conjunction of model information $M$ will be automatically incorporated. We then compare the generated guard $G$ with the availability specification $A$, and the generated command $C$ with the intended effect $E$. These comparisons may be done manually, or with the assistance of a proof tool.

3. If $G$ is stronger than $A$, then the operation is not as available as intended. We may address this by weakening either $P$ or $M$, by modifying $E$, or by strengthening $A$. A typical modification to $E$ will involve the addition of a disjunct, specifying an alternative outcome for conditions outside $G$.

   (If the strength of the guard reflects some uncertainty in implementation, if some aspect of the specification could not be interpreted, then we could also introduce or enable an additional transformation rule, providing an additional heuristic for translation. In most cases, however, it will be enough to rewrite the specification so that existing rules will apply.)

4. If $G$ is weaker than or equivalent to $A$, then the operation will be available as required. Note that $P$ is taken account of in the generation process, and $G$ will be stronger than or equivalent to $P$ in any case. $G$ being strictly stronger than $P$, or strictly weaker than $A$ is unlikely to represent a problem in design; it indicates simply that the operation is available, or unavailable, in circumstances where its availability, or unavailability, is not a matter of particular concern.

5. Optionally, if $C$ is updating attributes not mentioned in $E$, as suggested by their absence from frame $F$, then we may extend $F$ to indicate that this is acceptable. Alternatively, we may weaken $E$, or strengthen $P$ or $M$, and repeat the transformation.

Note that the expected availability $A$ and the expected frame $F$ do not influence the model transformation process. They are not part of the design model, they represent more abstract formalisations of requirements.
upon the system. Within an iteration, we may decide to revise A and F, in just the same way as we might decide to modify P, E, or M; when they are consistent, we may conclude that our iterative process of operation development is complete.

3.3. Model development

In developing a design, we begin with an initial set of classes, associations, and operations—typically, just enough to support user login to the system—and then produce successive versions of the model. In each new version, we may add or remove classes, associations, attributes, and operations, or modify class constraints, association multiplicities, or operation specifications. Some of these changes will correspond to refactoring patterns identified by Fowler [21]: in-lining a particular class; extracting a superclass; replacing an association with an intermediate, association class. However, as a new version may reflect a new understanding of requirements, there is no constraint upon the nature of the changes that may be made.

The completion of a design iteration depends upon the stage of development, upon the nature of the changes made, and upon whether or not the new version is to be released as an update to an existing, working system. In the initial stages of development, we might expect to iterate rapidly upon a design: inspecting the results of the model transformation, perhaps entering test data, but not maintaining expected availability and frame requirements, or taking the trouble to complete operation specifications. At the point where a version is deployed as a working system, specifications will be validated, and generated commands will be checked against availability requirements. More than this, the issue of data migration will need to be addressed.

In a typical data migration, data is extracted from the current version of an information system into a collection of flat files; in more specialised situations, it may be extracted into a custom-written, intermediate database, or into a generic data format such as Resource Description Framework (RDF). It is then transformed using a combination of hand-written functions and queries to prepare for loading into the new version of the system. It is quite possible to find that some of these functions and queries are incorrect, at least on the first iteration, and that the data will not load. Even if the data loads, it is quite possible that business rules and semantic properties have not been properly addressed in the transformation or loading process, and that the integrity of the new system has been compromised.

In our model-driven development of critical information systems, we would wish to ensure that our data migration functions are correct. In the Booster approach, we may do this by creating an intermediate model as a disjoint union of the models for the existing and new versions of the system. The migration function can be specified in the same way as any other operation, and implemented using the same set of model transformations. The result is a procedure that can be applied only if the data in the existing system, transformed according to the specification, will satisfy the business rules and semantic constraints of the new version. In a development environment such as Eclipse [22], it is possible to generate a specification for the migration function by tracking the changes made to the system model and, where necessary, annotating attributes in the new model with expressions explaining how initial values are to be obtained.

4. Correctness

4.1. Semantics

The model transformations for the translation and compilation of Booster models are written in the declarative programming language Stratego [23]. The Spoofax language workbench [24] is used to apply the transformations, which are compiled into Java code within Eclipse plugins. A key consideration in the choice of Spoofax is its declarative, functional, and compositional nature, providing a more convenient basis for analysis and verification of the transformations than would be afforded, for example, by an imperative language such as Java.

The result of translation is an object model within Eclipse; the result of compilation is a SQL install script, which corresponds to an instance of a SQL metamodel, and may be used to produce a database implementation in the usual way. A default Java API, and a JavaScript-based user interface, are generated
to match the current model, and can be linked to the database: in applications of model-driven development, it is important that the artefacts produced at each iteration can function as components; no manual customisation, integration, or extension should be necessary.

To prove that this generation process is correct, we give a formal semantics $S$ to the Booster notation: for models in which operations are described as predicates, and also for those in which operations are described as guarded commands. We give a semantics $R$ to the SQL notation, sufficient for the formal interpretation of the particular class of SQL scripts that will be produced. If we then give a comparable relational semantics to the language of model transformations, Stratego, focussed upon the transformation of the particular kinds of models involved, we may use this to show that the model transformations employed are correct with respect to appropriate notions of refinement upon the model semantics.

In this section, we outline the model and relational semantics, and explain the notions of refinement that are applicable here. The formal semantics of Stratego is not considered here; ideally, this semantics would itself be formalised within a proof tool, so that our outline, manual proofs of correctness for the transformations could be formally verified.

4.2. Refinement

In applications of formal techniques, refinement corresponds to the removal of nondeterminism or uncertainty in a description, or the presentation of a design at a lower level of abstraction, where more of the implementation mechanism is exposed. In this context, we are interested in the refinement of abstract data types, for the notion of an abstract data type, representing the structure and functionality of an information system, will serve as an appropriate basis for both object and relational model semantics.

The usual notion of refinement upon abstract data types is that of data refinement. An abstract data type is completely characterised by the observable effects of finite, sequential programs acting upon a defined interface: the internal state is unimportant; what is considered is what may result from the application of a series of operations, each of which may accept input and produce output. One abstract data type $C$ refines another abstract data type $A$ precisely when the possible outcomes of any program acting on $C$ are also possible outcomes for the same program acting on $A$. If this is the case, then $C$ is a suitable replacement, implementation, or substitution for $A$: no finite, sequential test could tell the difference.

The notions of refinement required here are slightly different. For the translation process, a weaker notion is appropriate. Any program that completes should produce the same observable effects. It is quite possible, however, that a generated, guarded command is strictly less applicable than the combination of user-supplied precondition and model constraints. This will occur when there is no implementation strategy for part of a specification, or where there is insufficient information to determine which implementation strategy would be appropriate. The model transformations are performing precisely as intended, but their action does not correspond to data refinement. Instead, as we set out in [25], it corresponds to the trace refinement of abstract data types viewed as communicating processes.

For the compilation process, in which we map guarded commands to platform-specific notations, a stronger notion of refinement is required. We require that the effect of the implementation is consistent with the specification and that the implementation should be available precisely when the specification allows. If an operation were made more available, then it could be called in circumstances where the specification does not apply, and we would have no guarantee of system and data integrity. If it were made less available, then the availability condition ($A$ in the development of Section 3.2) may no longer be satisfied, and we would have no guarantee of availability in critical situations.

The usual notion of data refinement allows operations to be more available in implementation than in specification; there may be programs that would complete successfully on the refined abstract data type $C$ that would not have done so on the original data type $A$. We require instead the notion of ‘blocking’ data refinement [26], in which the availability information contained in the guard specification is reflected exactly in the SQL implementation.

The criteria for the correctness of these transformations are summarised in Figure 5. Here, translate represents the transformation of an object model with predicates into an object model with guarded commands, and compile represents the production of a relational database implementation. To establish the
correctness of the transformations, and confirm that each generated implementation will be correct with respect to its object model specification, we need to prove that translate corresponds to a trace refinement between the abstract data types produced by the object semantics, and that compile produces a blocking data refinement between the second of these abstract data types and the abstract data type produced by the relational semantics. In the following sections, we examine these requirements in greater detail, and outline the structure of the transformations that would need to be formally verified.

4.3. Abstract syntax

A simple specification for the abstract syntax of the Booster language, covering both predicate and guarded command descriptions, and for the abstract syntax of the SQL notation is given in Figure 6. The language used for the specification is the Z notation [5], in which patterns of declaration and constraint are described using named schemas, and types are given sets or schema references. Schema references can be used as declarations, constraints, or as sets: the schema

<table>
<thead>
<tr>
<th>Name</th>
<th>declaration</th>
<th>constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

introduces a set of bindings matching the declaration and satisfying the constraint. For example, the schema

<table>
<thead>
<tr>
<th>$S$</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$a, b : \mathbb{N}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a &lt; b$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

introduces a set of bindings $S$ of attributes $a$ and $b$ to natural number ($\mathbb{N}$) values. Bindings may be seen as instances of the familiar record types, or data objects, as seen in most programming languages. The dot operator ‘.’ may be used to select the value of an individual attribute within a binding such that the value of $a$ is always strictly less than that of $b$.

Attribute names may be decorated with a prime symbol (′) to indicate a different attribute of the same type. Schema references may be decorated in the same way: this has the effect of decorating all of the
component attributes. A decorated binding represents a second instance of the same schema type. This approach may be used to introduce two instances of a schema type in the same scope. The constraint of an enclosing schema then describes the relationship between those two instances, and provides a concise way of formally specifying a mapping or transformation.

The remainder of the Z notation consists of a typed version of the first-order predicate calculus, together with a simple language of sets and relations. The \( \mathbb{P} \) symbol denotes the powerset constructor: \( \mathbb{P} S \) is the set of all subsets of \( S \). The \( \rightarrow \) symbol denotes the set of all partial functions from one set to another. The \( \rightarrow \rightarrow \) symbol (pronounced ‘maps to’) constructs a pair, an element of a Cartesian product, from the surrounding elements. The definite description operator \( \mu \) denotes the unique element of the stated type satisfying the subsequent property. ‘dom’ and ‘ran’ denote the domain and range of a binary relation, respectively. For any relation \( R \) and any set \( A \), we write \( R \mid A \) to denote the relational image of \( A \) under \( R \). ‘\&’ and ‘\|’ denote sequential and parallel composition of relations.

In Figure 6, the four schemas to the left characterise the abstract syntax of Booster as follows. An object model consists of a number of class declarations, each of which introduces a name and a constraint expression for the class together with a set of attributes and a set of operations. An attribute declaration introduces a name and a type, but also the name of an ‘opposite’ attribute: the mirror attribute in a bidirectional association. An operation declaration may contain both a constraint and a program: as the model is transformed towards implementation, the program is constructed to ensure that the constraint is satisfied, while also respecting the other constraints of the model.

The schemas to the right describe the abstract syntax of a relational model: a platform-independent model used as the basis for subsequent generation of a SQL implementation. A relational model is a set of table and procedure declarations, together with an initialisation statement. Each table has a set of columns, together with a primary key; columns have names and types; procedures have names and statements. Although SQL implementations support the description of integrity properties in terms of foreign keys and database constraints, we have no need of these language features here. The only access to the database is
We define functions to map object and relational models to different forms of abstract data types. We will describe the first of these in detail; the second is entirely similar, differing only in the structure of system state. To give a semantics to an object model, we first define a suitable notion of state for the corresponding abstract data type. This comprises a mapping from object identifiers to class names, giving the type of each object, together with mappings from attribute and association names to components of state: mappings from object identifiers to values, and mappings between object identifiers, respectively.

The corresponding abstract data type comprises a notion of initialisation, represented by an initial instance of the object state, together with a collection of named operations. The semantics of an operation may be characterised by a relation between input components, output components, the state of the abstract data type before, and the state after, the operation is performed. In Figure 7, this representation is captured as a schema *Operation* that includes two copies of the *ObjectState* schema, one of which is decorated with a prime to indicate that it represents the ‘after state’.

Our mapping from object models to object states is described by the schema *MapObjectState* in Figure 8. Here we use the \( \theta \) notation of Z: for any schema \( T \), the expression \( \theta T \) denotes a binding in which every attribute of \( T \) takes the value of that attribute in the current scope: for example, with the above definition through the generated API, and we know that each procedure in the API can be guaranteed to maintain data integrity.

### 4.4. Model semantics

Figure 7: Abstract data type semantics
of $S$, we have that $\theta.S.a = a$. The $\theta$ operator affords a means of referring to the current binding of a particular schema, in the same way that this affords a means of referring to the current instance of a class in object-oriented programming. It would be perfectly consistent to pronounce $\theta$ as ‘this’.

The $\text{link}$ component represents links corresponding to associations with a maximum multiplicity of 1: mandatory or optional relationships. The $\text{links}$ component represents links corresponding to associations with a maximum multiplicity greater than one: that is, one to many relationships. Similarly, $\text{value}$ records the values of attributes with a maximum multiplicity of 1, and $\text{values}$ records those of set-valued attributes. This separation will prove advantageous when we come to relate object model semantics to the semantics of the corresponding relational models. In this schema, the $\text{let} \ldots \equiv$ construct introduces two local abbreviations for use in the definition of these functions.

The abstract data type representation used for the semantics of relational models comprises an initial relational state and a set of named procedures. The notion of relational state is more complex, with the values of primitive attributes stored in class tables, links stored in association tables, and the values of set-valued attributes stored in attribute tables. This representation was also shown in Figure 7. The mapping from object models to abstract data type semantics is described by the schema $\text{ObjectSemantics}$ in Figure 8.

The initial object state is quite straightforward, with no instances of any class, and with $\text{link}$ and $\text{value}$ relations for each object being empty: the only item in the range of link $o$ for any $o$ is the empty set $\emptyset$, and the range of link is the set containing only the empty set.

As the Booster language permits the definition of dynamic invariants, constraints that refer to values of attributes both before and after any change to the state, the conjunction of class constraints corresponds to a relation between states. The constraint or guarded command expressions appearing in an operation...
declaration will also be interpreted as relations: however, these may have additional components representing input and output. The functions relation_M, relation_C, and relation_P represent the semantics of the predicate and guarded command notations; a similar function is defined for stored procedures. This is enough to characterise the semantics of an object model in the abstract syntax. The semantics of relational models may be characterised in the same way, using a schema RelationalSemantics. We do not need to give a semantics to the whole of SQL; we need only address features that could appear in a generated implementation.

4.5. Correctness

To show that an abstract data type \( C \) is a trace refinement of another abstract data type \( A \), it is enough to show that the relation corresponding to each operation in \( C \) is contained within the relation for the same operation in \( A \). To formalise this notion, we have only to define a mapping from our representation of operation semantics, described by the schema Operation, to an appropriate type of relations.

\[
\text{relation} : \text{Operation} \rightarrow (\text{ObjectState} \times \text{Input}) \leftrightarrow (\text{ObjectState} \times \text{Output})
\]

Our notion of trace refinement is then given by the following schema:

\[
\text{TraceRefinement}
\text{ObjectModelSemantics}; \text{ObjectModelSemantics}'
\forall m : \text{dom operation} \bullet \text{relation} (\text{operation}' m) \subseteq \text{relation} (\text{operation} m)
\]

Here, \( \theta_{\text{ObjectModelSemantics}'} \) is a refinement of \( \theta_{\text{ObjectModelSemantics}} \) precisely when the relation corresponding to the primed version of the operation is a subset of that corresponding to the unprimed version.

The correctness criterion for the combination of model transformations that perform the translation from predicate to guarded command forms of a Booster object model may then be characterised as follows. If Translate represents the semantics of this combination of transformations, as a function upon the abstract syntax of object models, then we need to establish that

\[
\text{Translate} \circ \theta_{\text{ObjectSemantics}} \Rightarrow \theta_{\text{ObjectSemantics}'} \circ \text{TraceRefinement}
\]

for all valid instances of ObjectModel.

To show that an abstract data type \( C \) is a blocking data refinement of another abstract data type \( A \), it is enough to exhibit a forwards simulation: a mapping \( f \) from the state space of \( A \) to the state space of \( C \) with the properties that

1. the initialisation, or initial state, of \( C \) is contained within the image of the initialisation of \( A \), under the mapping \( f \),
2. the image under \( f \) of the domain of each operation on \( A \) is contained within the domain of the same operation on \( C \), and
3. the effect of each operation on \( C \) is a possible effect of the same operation on \( A \), with the outcome mapped by \( f \).

If these conditions hold, then an inductive argument could be used to show that the effect of any program, any sequence of operations, acting upon \( C \) could also be observed of the same program acting upon \( A \). A complete explanation of the argument is presented by Bolton and Davies [27], as a development of earlier work on refinement and simulation [5].

The correctness criterion for the combination of model transformations that perform the compilation step, from the guarded command form of an object model to a relational database implementation, requires the identification of a suitable simulation, as a function from ObjectState to RelationalState. If we define a mapping from procedure semantics to relations,

\[
\text{relation} : \text{Procedure} \rightarrow (\text{RelationalState} \times \text{Input}) \leftrightarrow (\text{RelationalState} \times \text{Output})
\]
our notion of blocking data refinement is given by the following schema:

\[
\begin{align*}
\text{DataRefinement} & \quad \text{ObjectModelSemantics} \rightarrow \text{RelationalModelSemantics} \\
\text{mapState} & : \text{ObjectState} \rightarrow \text{RelationalState} \\
\end{align*}
\]

\[
\begin{align*}
\text{mapState} & (\{ \text{initObjectState} \}) = \{ \text{initRelationalState'} \} \\
\text{dom} & : \text{dom operation} = \text{dom procedure'} \\
\forall m : \text{dom operation} & \bullet \quad (\text{mapState} \parallel \text{id})(\text{dom}(\text{relation}(\text{operation } m))) \subseteq \text{dom}(\text{relation}(\text{procedure'} m)) \\
& \land \\
(m & \text{mapState} \parallel \text{id}) \textRelation (\text{procedure'} m) \subseteq \textRelation (\text{operation } m) \circ (\text{mapState} \parallel \text{id}) \\
\end{align*}
\]

The function mapState acts upon object states. It needs to be augmented with an additional component
acting upon input and output, before being applied to the relations corresponding to operations and procedures. As the input and output values are not mapped, the effect of this additional component is described
by the identity relation 'id'.

The correctness criterion for the combination of model transformations that perform the compilation from
object models to relational implementations may then be characterised as follows. If Compile represents the
semantics of this combination of transformations, as a function upon the abstract syntax of object models,
then we need to establish that

\[
\exists \text{mapState} : \text{ObjectState} \rightarrow \text{RelationalState} \bullet \\
\text{Compile} \circ \text{RelationalSemantics} \Rightarrow \text{ObjectSemantics} \circ \text{DataRefinement}
\]

for all valid instances of ObjectModel.

4.6. Transformations

The translation process is divided into a number of phases, each of which is implemented as a separate
Stratego function, and applied in sequence:

\[
\begin{align*}
\text{translate} & : m \rightarrow <\text{wp}> \\
& <\text{program}> \circ <\text{elaborate}> \circ <\text{parse}> \\
\end{align*}
\]

The first function, parse, takes a Booster model and creates a lookup table and graph of the corresponding
abstract syntax. The second, elaborate, qualifies and re-orients the model constraint information so that
it is ready for instantiation and insertion into individual operation specifications. The work of translation
is done by the two subsequent functions program and wp.

The program function considers the part of the operation specification that refers to ‘after values’: those
represented by primed versions of class and attribute names, or by output parameters, decorated with
exclamation marks. Each conjunct is considered in turn, and a candidate program is generated that would
be guaranteed to achieve the specified result. As an example, consider the operation AddQ, declared in the
context of the model of Figure 3:

\[
\begin{align*}
\text{class B} & \{ \\
\text{attributes} & \quad \text{cp} : \text{set}(\text{C.bp})[*] \\
\text{cq} & : \text{set}(\text{C.bq})[*] \\
\text{operations} & \quad \text{AddQ} \{ \text{c?} : \text{cq'} \} \\
\} \\
\text{class C} & \{ \\
\text{attributes} & \quad \text{bp} : [\text{B.cq}] \\
\text{bq} & : [\text{B.cq}] \\
\text{invariant} & \quad \text{bp} /\!=\! \text{bq} \\
\} \\
\end{align*}
\]
The two pairs of mirrored associations correspond to a collection of eight dynamic invariants at the model level, made explicit by the `elaborate` function:

```plaintext
default b : B @ default c : C @ default c : b.cp' => c.bp' = b
default b : B @ default c : C @ default c /= b
default b : B @ default c : C @ default c : b.cp' => c.bp' /= b
default b : B @ default c : C @ default c /= b
```

The first four invariants are concerned with the possible addition or removal of a reference from `cp` or `cq`, and the consequential requirements upon opposite attributes `bp` and `bq`. The second four are concerned with the setting or unsetting of `bp` and `bq`, and the consequential requirements upon `cp` and `cq`.

The `program` function begins by generating the assignment

```plaintext
this.cq := this.cq \ { c? }
```

which would have the effect of adding input reference `c?` to the one-to-many association `cq` for the current object `this`. The function will then examine the set of paths that would be updated, and include an additional conjunct based upon the third of the dynamic invariants above

```plaintext
c?.bp' = this
```

An additional assignment will then be added:

```plaintext
this.cq := this.cq \ { c? }
|| c?.bp := this
|| c?.bp /= null -> c?.bp := c?.bp - {c?}
```

This program would add `c?` to the attribute `cq`, update the opposite attribute in `c?`, and also remove `c?` from the opposite attribute of any previous partner.

The `wp` function then instantiates all of the relevant static invariants, including type and multiplicity constraints, and calculate the weakest precondition for the candidate program to achieve the conjunction of these properties with the original operation specification. The proposed assignments have no implications for type and multiplicity constraints; however, the elaborated form of the invariant declared within `C`,

```plaintext
forall c : C @ default c.bp /= null
```

does have a bearing upon applicability. The `wp` function produces the guarded program

```plaintext
forall c : C @ default cbp /= c.bq
```

which is applicable only if the current object is not referred to in the attribute `bp` of the input object `c?`.

The compilation process is also broken into a number of functions:
The structure function produces a relational schema to match the object model. Each guarded program is then translated into a stored procedure, composed using a small number of standard patterns:

1. Insert into <Table> (<PKColumn>) values (<PKValue>)
2. Delete from <Table> where <PKColumn> = <PKValue>
3. Update <Table> set <Column> = <Value> where <PKColumn> = <PKValue>
4. if ((select count(*) from <Table> where <ColumnName1> = <Value1>
    and <ColumnName2> = <Value2>) = 0 ) then Insert into <Table> (<ColumnName1>, <ColumnName2>)
    values (<Value1>, <Value2>) end

Together with conditional, sequential, and iterative constructs—the last of these implemented using cursors where necessary.

Objects and links are created and deleted using patterns 1 and 2. An assignment of the form a := e will map to a procedure using pattern 3, and an assignment of the form s := s \{ a \} will map to a procedure using pattern 4. Fully-qualified attribute names, or path expressions, are translated into nested select statements: for example, this.b.a is implemented as:

\[
\text{select } a \text{ from } B \text{ where id } = \text{(select b from A where id } = \text{this)}
\]

The bridges function creates tables that record the details of the mapping from the object model to the relational model; the metadata in these tables is used in the generation and configuration of the Java API and the Javascript user interface; it is also used to support the creation and execution of queries written in the language of the object model. The final function, script, generates an installation script to instantiate the tables and manage any data migration.

The argument for the correctness of the existing implementation has been based upon manual comparison of the Stratego functions, which act on the concrete syntax of Booster, with functions written in Z that act upon the abstract syntax specified above. For example, the intended effect of the structure function is described by:

\[
\text{CompileMapState}
\]

\[
\text{ObjectMapState: RelationalMapState}
\]

\[
\text{mapState : ObjectState } \leftrightarrow \text{ RelationalState}
\]

\[
(\forall c : \text{dom class } \cdot \text{value } = (\text{classTable (class c)}.\text{value})
(\forall a : \text{Name} \cdot \text{values a } = (\text{attribTable a}.\text{values})
(\forall a : \text{Name; o : ObjID } \cdot \text{link a o } = (\mu v : (\text{assocTable a}).\text{targets } \{ \{ o \} \}))
(\forall a : \text{Name; links a } = (\text{assocTable a}).\text{targets})
\]

which defines an appropriate, functional mapping from object model state to relational model state.

The simple, declarative nature of the Stratego language means that a mechanised, fully formal proof of correctness, involving the definition of a function from the concrete syntax of the language to relations upon abstract syntax of the modelling notations, would be perfectly feasible. The definition of such a function would provide additional reassurance as to the correctness of the transformation, as well as making their design more robust and accessible.

4.7. Model evolution

In critical applications, we may wish to establish in advance whether a proposed, new version of an information system would deliver the same service across existing interfaces. In our model-driven approach, the existence of a formal specification of system behaviour means that we have the opportunity to investigate specific properties in advance of implementation. It would be better still, however, if the change to the system corresponded to a data refinement: we could then be sure that, whatever use was being made of it, the new
version of the system would be at least as suitable as the old one. The notion of data refinement applicable here is the more familiar non-blocking version: we may produce a refinement by extending the domain of an operation, as behaviour outside this domain is seen as completely undefined.

As in the proof above, we could show that the new version was a data refinement by exhibiting a forwards simulation between the old and the new state spaces. Indeed, a series of simulations—fowards or backwards—would suffice. We could base our analysis of operation effects and preconditions on the translated version of the object model, in which operations are represented as guarded programs. These programs can be considerably more complex, and less amenable to analysis, than the operation specifications. It is thus sensible to ask whether we might base our analysis instead upon the object model.

An obvious problem with doing this is that, in general, the original computation-independent model does not contain enough information to determine the availability of operations. The transformations that produce the platform-independent model, with guarded commands, are defined upon the syntax of the modelling language: a new version of the model may contain a constraint that is semantically equivalent, but which is expressed in different terms—terms that may be produce different results under transformation. For example, if an operation \( M \) were described by \( x' = 4 \& y' = 5 \) in the old version of a model, and by

\[
x' = y' - 1 \& y' = 2 \times x' - 3
\]

in the new version, then—as suggested in Section 3.1—although these specifications are semantically equivalent the resulting implementations would be different: one would have an assignment that achieves the result, the other an assertion that it already holds.

There is, however, a circumstance in which a refinement of the object model can be guaranteed to produce a refinement of the generated system: when the domain or precondition of an operation is extended, but the applicable postconditions are left unchanged. Such a circumstance is quite likely to arise in the course of iterative development: it is a special case of Step 3 in the operation development methodology of Section 3.2.

If an operation is less applicable than expected, if there are cases that have not been considered, we may extend the specification to cover these cases while leaving the existing, applicable predicates in place. Provided that any new predicates involving after values are not also applicable with the existing domain, then the new system is guaranteed to be a data refinement of the old. The easiest way to achieve this is by extending the specification using the disjunction operator or.

As an example, consider the expanded model from Section 2, the textual form of which is shown in Figure 2. As explained in Section 2.3, the operation \( A.Inc \) would be translated into the program

\[
\text{Inc } \{ m < 9 \& (b /= \text{null} => b.n > m + 1) \rightarrow m := m + 1 \}
\]

If our expectation is that this operation should be available whenever \( m < 9 \), then the behaviour of the implementation in the case where

\[
b /= \text{null} \& b.n = m + 1
\]

is unsatisfactory, corresponding to undefined behaviour in the non-blocking version of data refinement. If we extend the specification to

\[
\{ m' = m + 1 \}
\]

or

\[
\{ b /= \text{null} \& b.n = m + 1 \& C \}
\]

where \( C \) is any constraint, then we are guaranteed a data refinement. If the translation of \( C \) does not produce any guard stronger than \( b /= \text{null} \& b.n = m + 1 \& m < 9 \), then we are guaranteed also to achieve the expected availability. For example, the extended specification

\[
\{ m' = m + 1 \}
\]

or

\[
\{ b /= \text{null} \& b.n = m + 1 \& m' = m + 1 \& b.Inc \}
\]

will produce the following guarded command

22
which, given the model constraints, will be available whenever \( m < 9 \).

5. Application

5.1. A safety critical information system

Two of the authors used the Booster technology in the development of a clinical information system called True Colours [7] used in the management of patients with long-term mental health conditions, including: severe depression, bipolar disorder, and psychosis. The system sends messages to patients, shows them a summary of their health record, and allows them to provide reports upon their condition, which are made available in real-time to clinicians and carers. It is currently in use by over 3,000 patients.

This may be seen as a safety-critical system under the intuitive notion put forward by Knight:

> If the failure of a system could lead to consequences that are determined to be unacceptable, then the system is safety critical. In essence, a system is safety critical when we depend on it for our well being. [28]

It would be unacceptable for the system to allow access to patient information to anyone other than the patient, or a designated carer, and the clinical staff responsible for their care. Given the sensitivities regarding mental health conditions, it would be unacceptable for the system even to reveal that it held information on a particular individual.

Many of the patients have come to depend upon the system, to a greater or lesser extent. If it were to send a message to the wrong patient, or to the wrong address, if it were to send duplicate messages, or if it were to fail to send a message, then this could cause distress and confusion. If it were to fail to accept a report, or to present a report to the clinician in a timely fashion, then the care of the patients involved might be adversely affected. Although patients in distress are advised to contact the healthcare provider also through other means, the system plays an important role in alerting both patient and clinician to a deteriorating condition: whether this is communicated through specific messages, or simply through a lack of response.

5.2. Patients and messages

Every message handled by the system contains attributes—phone numbers or email addresses—that identify the sender and the intended recipient, whether the message is generated automatically or received from a known contact. Every staff, carer, or patient record within the system is linked to the complete collection of messages sent to, received from, or otherwise associated with that individual. These links help to ensure that messages are properly managed: that every message reaches the correct recipient, and that all of the information obtained is properly included in patient records and reports.

A key aspect of management concerns the updating of address information: in particular, the phone numbers associated with each patient; these are used to determine where text messages are sent to, but also to authenticate or identify incoming reports. The system maintains a list of currentPhoneNumbers: one of these, identified as the primaryPhoneNumber, is used to send messages to the patient; messages received from any of the current numbers are added to the patient record. A list of expiredPhoneNumbers is also maintained, to take account of the fact that phones may be lost, stolen, or replaced. New messages from these numbers are not added to the record, although links to earlier messages will be preserved.

In the model for the system, the class Contact includes the following attributes and invariants:
class Contact {
    attributes
        ...
        currentPhoneNumbers : SET (PhoneNumber . currentPhoneNumberForContact) [+]
        primaryPhoneNumber : [ PhoneNumber . primaryPhoneNumberForContact ]
        expiredPhoneNumbers : SET (PhoneNumber . expiredPhoneNumberForContact) [+]

    invariants
        ...
        primaryPhoneNumber /= null => primaryPhoneNumber : currentPhoneNumbers
        currentPhoneNumbers /= {} => primaryPhoneNumber /= null
        forall p : PhoneNumber@p:expiredPhoneNumbers' => p /: currentPhoneNumbers'
        forall p : PhoneNumber@p:currentPhoneNumbers' => p /: expiredPhoneNumbers'
}

It is quite possible that no primary phone number has been identified for an individual: they might prefer to communicate only through email. The first invariant states that if a primary phone number exists, it must be one of the current numbers associated with the patient. The second invariant, a business rule for the system, states that if there is at least one current number, then a primary number should be identified.

The third and fourth are dynamic invariants insisting that, for any operation, if it is the case that input \( p \) is included in the set of expired numbers after the operation, then \( p \) cannot also be included in the set of current numbers, and vice versa. The conjunction of these constraints is semantically equivalent to the disjointness condition

\[
\text{currentPhoneNumbers} \cap \text{expiredPhoneNumbers} = \{\}
\]

which could have been included. The form of the dynamic constraints, however, serves to disambiguate the model intentions: faced with an operation whose specification requires that \( p \) be added to one of the sets, the transformation rules will produce a command that will remove it from the other, if necessary.

The operation used to ‘expire’ a phone number has the following, simple specification:

\[
\text{ExpireNumber} \{ p? : \text{currentPhoneNumbers} & p? : \text{expiredPhoneNumbers}' \}
\]

The intended effect of the operation is that \( p? \) should be marked as expired, so that no further messages can be sent, and any further messages received are kept separate from the patient’s record. It is applicable only to current numbers: an attempt to mark any other number as ‘expired’ for this patient will be flagged as an error. The intended availability of the operation is simply \( p? : \text{currentPhoneNumbers} \): that is, it should be applicable to any current phone number.

The program phase of the transformation takes each conjunct in turn and generates a program for each: the first conjunct makes no assertions upon the state after the post-state of the method and so generates the program \( \text{skip} \); the second conjunct generates an assignment, and by pattern-matching upon the types of the attributes involved, generates the following parallel assignment, suitable for maintaining the bi-directionality property:

\[
\text{expiredPhoneNumbers} := \text{expiredPhoneNumbers} \lor \{ p? \} ||
\text{p? . expiredPhoneNumberForContact} := \text{this}
\]

This phase of the transformation continues by taking this substitution in the context of the dynamic invariants, and in accordance with the third invariant, described above, generates the additional substitution to be applied in parallel:

\[
\text{currentPhoneNumbers} := \text{currentPhoneNumbers} - \{ p? \} ||
\text{p? . currentPhoneNumberForContact} := \text{null}
\]
The weakest precondition (wp) phase of the transformation then considers these substitutions in the context of the first two (static) invariants, and considers the effects of aliasing. In the analysis of the first invariant, it is feasible that the input variable p? refers to the same object as the attribute primaryPhoneNumber, and so the following constraint is generated:

\[ p? = \text{primaryPhoneNumber} \implies \begin{cases} p? \neq \text{null} \implies p? : \text{currentPhoneNumbers} - \{p?\} \\ \end{cases} \]

This will be automatically simplified by the compiler to produce the following precondition:

\[ p? \neq \text{primaryPhoneNumber} \text{ or } p? = \text{null} \]

which may be further simplified when taken in conjunction with the original precondition: the second conjunct, \( p? = \text{null} \), may be assumed to be false.

The final generated program has a stronger guard than originally specified:

\[
\text{ExpireNumber} \{
\begin{align*}
\text{p?} : \text{currentPhoneNumbers} \& \text{p?} \neq \text{primaryPhoneNumber} \rightarrow \\
\text{currentPhoneNumbers} := \text{currentPhoneNumbers} - \{\text{p?}\} \quad \|
\text{p?} . \text{currentPhoneNumberForContact} := \text{null} \quad \|
\text{expiredPhoneNumbers} := \text{expiredPhoneNumbers} \lor \{\text{p?}\} \quad \|
\text{p?} . \text{expiredPhoneNumberForContact} := \text{this}
\end{align*}
\}
\]

To achieve the expected availability, we need to specify an alternative course of action for when \( p? \) is the primary phone number. The revised specification, chosen during the course of development, is

\[
\text{ExpireNumber} \{
\begin{align*}
\begin{cases} 
\text{p?} : \text{currentPhoneNumbers} \& \text{p?} : \text{expiredPhoneNumbers'} \\
or \\
\text{p?} = \text{primaryPhoneNumber} \& \text{currentPhoneNumbers} - \{\text{p}\} /\{ \} \&
\text{p?} : \text{expiredPhoneNumbers'} \& \text{q?} /= \text{p?} \& \text{primaryPhoneNumber'} = \text{q?} \\
\end{cases}
\end{align*}
\}
\]

This requires that a new, different number is supplied for use as the new primary number. In the program generated from this constraint, we find that this number must be drawn from the set of current numbers for that patient:

\[
\text{ExpireNumber} \{
\begin{align*}
\begin{cases} 
\text{p?} : \text{currentPhoneNumbers} \& \\
\text{p?} = \text{primaryPhoneNumber} \rightarrow \text{currentPhoneNumbers} - \{\text{p}\} = \{} \rightarrow \\
\text{currentPhoneNumbers} := \text{currentPhoneNumbers} - \{\text{p}\} \quad \|
\text{p?} . \text{currentPhoneNumberForContact} := \text{null} \quad \|
\text{expiredPhoneNumbers} := \text{expiredPhoneNumbers} \lor \{\text{p}\} \quad \|
\text{p?} . \text{expiredPhoneNumberForContact} := \text{this}
\end{cases}
\end{align*}
\}
\]

\[
\begin{align*}
\begin{cases} 
\text{p?} : \text{currentPhoneNumbers} \& \\
\text{p?} = \text{primaryPhoneNumber} \& \text{currentPhoneNumbers} - \{\text{p}\} /= \{} \&
\text{q?} /= \text{p?} \& \text{q?} : \text{currentPhoneNumbers} \rightarrow \\
\text{currentPhoneNumbers} := \text{currentPhoneNumbers} - \{\text{p}\} \quad \|
\text{p?} . \text{currentPhoneNumberForContact} := \text{null} \quad \|
\text{expiredPhoneNumbers} := \text{expiredPhoneNumbers} \lor \{\text{p}\} \quad \|
\text{p?} . \text{expiredPhoneNumberForContact} := \text{this} \quad \|
\text{primaryPhoneNumber} = \text{q?} \quad \|
\text{q?} . \text{primaryPhoneNumberForContact} := \text{this}
\end{cases}
\end{align*}
\]

25
If we wish to further weaken the guard, to eliminate the constraint that \( q? \) must be chosen from the list of current numbers, then we could do so by adding an additional dynamic invariant to the class:

\[
\text{forall } p : \text{PhoneNumber} \not\in \text{primaryPhoneNumbers}' \Rightarrow p : \text{currentPhoneNumbers}'
\]

The transformation rules would then be able to determine the course of action to take when the new primary number supplied is not already included in the list of current numbers: the number should be added to that set. The same effect could be achieved by adding the constraint \( q? : \text{currentPhoneNumbers}' \) to the specification.

5.3. Implementation

The generated SQL code for the system includes the statements shown below, which define the static structure for this part of the Contact record.

```sql
create table Contact (id int auto_increment primary key);
create table Contact_currentPhoneNumbers (id int auto_increment primary key);
create table PhoneNumber_currentPhoneNumberForContact
  (id int auto_increment primary key);
create table Contact_primaryPhoneNumber (id int auto_increment primary key);
create table PhoneNumber_primaryPhoneNumberForContact
  (id int auto_increment primary key);
create table Contact_expiredPhoneNumbers (id int auto_increment primary key);
create table PhoneNumber_expiredPhoneNumberForContact
  (id int auto_increment primary key);
```

It includes also the following SQL implementation for the Contact.ExpireNumber operation:

```sql
create procedure Contact_ExpireNumber (in this int, in p_in int, in q_in int)
begin
  declare exit handler ...
  start transaction;
  if (select count(*) from Contact_currentPhoneNumbers
      where Contact = this and PhoneNumber = p_in) > 0 and
  ((select count(*) from Contact_primaryPhoneNumber
      where Contact = this and PhoneNumber = p_in) = 1 and
  (select count(*) from Contact_currentPhoneNumbers
      where Contact = this and PhoneNumber != p_in) = 0) or
  (select count(*) from Contact_primaryPhoneNumber
      where Contact = this and PhoneNumber = p_in) = 0)
  then
```
begin
  delete from Contact_currentPhoneNumbers
    where Contact = this and PhoneNumber = p_in;
  delete from PhoneNumber_currentPhoneNumberForContact
    where PhoneNumber = p_in and Contact = this;
  insert into Contact_expiredPhoneNumbers
    (Contact, PhoneNumber) values (this, p_in);
  insert into PhoneNumber_expiredPhoneNumberForContact
    (PhoneNumber, Contact) values (p_in, this);
end;
elseif (select count(*) from Contact_currentPhoneNumbers
  where Contact = this and PhoneNumber = p_in) > 0 and
  (((select count(*) from Contact_primaryPhoneNumber
    where Contact = this and PhoneNumber = p_in) = 1 and
  (select count(*) from Contact_currentPhoneNumbers
    where Contact = this and PhoneNumber != p_in) > 0) and
  p_in != q_in and
  (select count(*) from Contact_currentPhoneNumbers
    where Contact = this and PhoneNumber = q_in) > 0)
  then
    begin
      delete from Contact_currentPhoneNumbers
        where Contact = this and PhoneNumber = p_in;
      delete from PhoneNumber_currentPhoneNumberForContact
        where PhoneNumber = p_in and Contact = this;
      insert into Contact_expiredPhoneNumbers
        (Contact, PhoneNumber) values (this, p_in);
      insert into PhoneNumber_expiredPhoneNumberForContact
        (PhoneNumber, Contact) values (p_in, this);
      update Contact_primaryPhoneNumber set PhoneNumber = q_in
        where Contact = this;
      update PhoneNumber_primaryPhoneNumberForContact set Contact = this
        where PhoneNumber = q_in;
    end;
end if;
commit;
end //

Where the object model represents the whole of a system, the transformations that implement the translation to SQL have access to a complete account of the availability and effect of every operation in the API. This offers significant opportunities for simplification and optimisation: generic aspects of the implementation, such as object-relational bridging, can be generated to match the current version of the model, instead of being implemented as a fixed, generic component. Of course, any simplifying or optimisation transformation brings with it a new proof obligation. In the existing version of Booster, additional functions are generated for the production of the user interfaces: for example, for determining the availability of an operation in order to determine whether a button to invoke that operation should be included in a generated web page, or for determining the contents of a drop-down box for input selection.

Where the object model represents part of a system, then we may wish to generate also foreign key constraints, and other checks upon data consistency at the level of the SQL code. While we can be sure that the operation implementations generated from the Booster model will maintain data integrity, it may be the case that other operations will be defined and used outside the specification, and we may wish to ensure that these, too, respect the invariant properties described in the model.
5.4. System evolution

In the development of any complex information system, we may expect our understanding of requirements to continue to evolve after the system is first deployed. This was certainly the case for True Colours: several revisions were required to address the needs of specific user groups, to incorporate new reporting standards, and to provide interoperability with other systems. 78 versions of the model were created during the initial period of development, in which the system was deployed only to clinicians and patients who had volunteered to help with testing. The first large-scale deployment took place in September 2011; a further 61 versions have been created and released since then, the majority of which required the definition of an operation for data migration.

The majority of the subsequent versions were intended as refinements to the deployed system: adding new features, but maintaining existing functionality. The users of the system—whether patients, carers, or clinicians—were quite averse to change: not only should existing functions continue to behave as expected, but the presentation of existing data should remain the same. A key feature of the system, illustrated in Figure 9, is the presentation of summary data, showing ways in which a patient’s condition has improved, or deteriorated, over a period of months or years. Although the display was not auto-generated, this data may be presented alongside information on medication, exercise, sleep, and anything else that the patient or clinician deems relevant.

Were a patient to find that some function no longer worked as expected, this could considerable confusion and distress. The risk of this happening can be reduced through detailed analysis of the generated guards for operations, checking that they meet the stated expectations of liveness, and checking also that these expectations are consistent with observed patterns of usage.

If a data point were lost from the summary data, or if some value were changed, then the appearance of a graph such as that shown in Figure 9 might change considerably. This may affect the patient’s understanding of their own condition, or the clinician’s recommendations regarding medication and other interventions. The risk of this happening can be reduced through careful design of the data migration function. For
example, where derived data appears in the model, such as a view or transformation of data used for a
web page or graph, we may add the constraint that the result of the derivation remains unchanged by the
data migration. The model transformations will then check that, even if changes have been made to the
underlying data representation, the usage of the data remains unaffected.

One aspect of the development demonstrated an additional advantage of the formal, model-driven ap-
proach to data migration. The deployment of the migration model is fully automatic. A secure web
service is used to: (1) put the existing version of the information system into read-only mode; (2) run the
automatically-generated query that characterises the guard for data migration; (3) if the guard is satisfied,
extract the data from the existing system and load it into the transformation system generated from the mi-
gration model; if not, abort and resume normal operation with the existing version of the system; (4) apply
the migration function within the transformation model; (5) initialise the new version of the information
system; (6) extract the data from the transformation model, load it into the new version of the system, and
switch over.

At no point does the developer need to have access to the operational data: new versions of the True
Colours system were deployed without the authors having access to patient information. Where the guard
was not satisfied, a modified migration was proposed, or—in some cases—the existing data was ‘cleaned’ by
an operational data manager in order to satisfy the constraints of the proposed migration.

6. Discussion

In critical systems engineering, there is considerable advantage to be gained from the adoption of auto-
matic tools and techniques that promote correctness in development: if these can be used to eliminate certain
kinds of error, then manual effort can be focussed upon other aspects of verification and validation. In this
paper, we have presented a formal technique for use in the development of information systems, aimed at
the elimination of errors related to the invocation of operations outside their domain of applicability: errors
that pertain to the violation of business rules, or the loss of data integrity.

The technique is supported by a development toolkit, implemented as a collection of model transfor-
mations written in the Stratego language and embedded in the Eclipse modelling environment. The tool
takes mathematical models of structure and functionality, and generates robust database implementations,
supported by automatically generated interfaces. The contribution of the paper has been: to set out a
methodology for application, explaining the iterative approach to development and deployment that the
technique is intended to facilitate; to present a framework for establishing the correctness of the underlying
transformations, in terms of existing notions of trace and data refinement; to report upon the application
of the technique in the development of a critical information system.

The limitations of the technique are characterised by the proposed domain of application. In the devel-
opment of critical information systems, we are concerned for the most part with simple transformations of
data that may have unforeseen consequences in the context of a complex data model, in which the values of
different attributes are related according to business rules and semantic integrity constraints. The technique
is not intended to support the correct development of complex algorithms, although it could be extended
with heuristics for particular classes of problem. Neither is it intended for the analysis of concurrent patterns
of behaviour: the weakest precondition calculations embedded in the transformations are based upon the
assumption that each operation is implemented as a transaction.

Despite these limitations, the technique has proved quite effective in practice. As reported in [7], it
has been used to develop a small number of information systems whose correctness is extremely important,
if not critical, to the organisations that they support; furthermore, the cost of developing those systems,
complete with a formal design-level specification, has been significantly less that would be associated with
conventional development techniques. In this sense, we may see model-driven technology as an enabling
factor in the successful application of formal methods. Certainly, the application of formal methods to
programming at any scale would seem to require either automatic proof of code-level properties, or the
automatic generation of code from formal specifications.

The closest related work, in terms of notation, is that of Khalafinejad and Mirian-Hosseinabadi [29] who
propose a framework for the automatic translation of Z operation schemas into SQL. Here, the authors make
the same observation that conditions of the form $x' = e$, asserting that the value of variable $x$ after the operation is equal to that of the expression $e$, can be achieved through assignment. They do not, however, consider whether or not one might wish to perform that assignment: without considering the operation in the wider context of a system model, all that may be achieved is the literal translation of simple predicates into programs.

The adoption of an object modelling approach, in which operations and constraints are defined in the context of relevant data, in which explicit reference may be made to attributes and operations declared within associated classes, and associations are themselves classifiers, supports a more modular, more concise treatment of structure and functionality than would be possible using the standard Z notation. Object-Z [30] allows the definition of object references, and constraints can mention attributes of other classes. Such object coupling limits our ability to refine the definitions of individual classes. Some have suggested conditions under which individual class refinement is possible [31]: requiring, essentially, that the operations do not depend upon the values of attributes in associated classes. We would argue, however, that this is unlikely to be the case in the design of any complex, critical information system, and it is in the analysis and implementation of complex systems that formal methods and tools are so badly needed.

Previous work by McComb and Smith [32], and separately by Smith [33], in which a semantics is developed for Object-Z, insist that read access to attributes should be through accessor operations only. A similar constraint is imposed upon the OhCircus notation [34]. In each case, the result is an approach to object modelling that addresses the communication models of object-oriented programming. This is quite different from the approach taken here, in which operations are composed to produce a single, atomic transaction upon the state, not a series of communications between objects. In Booster, as in Z, the semantics of a compound operation can be determined entirely from the relational semantics of its components: no consideration of sequences of communications is necessary.

The languages CSP-OZ [35] and TCOZ [36], both building upon earlier work on action systems [37], allow the definition of a separate guard, as well as a precondition, for each operation. In these operations, the guard and the precondition together define the operation. Our approach is quite different: a weakest precondition, derived from the operation specification, in the context of the model constraints, is used as a guard in the generated implementation. The availability constraint, if supplied, serves as an independent correctness criterion for the design.

A considerable amount of work has been done on the automatic transformation of object models written in UML. The Query/View/Transformation approach [38] focuses on design models, but in implementations such as ATL [39], transformations are described in an imperative style, and proofs of correctness would be more difficult. The higher-order strategies provided by Stratego remove the need for imperative code: the lack of such higher-order rules in other model transformation approaches has been noted in [40]. Additionally, the mechanism for creating dynamic rules provides a convenient method for creating a lookup table, and the concise syntax makes the transformation rules clearer and more amenable to inspection. However, whilst ATL integrates with the Eclipse Modeling Framework and existing UML tools, such support in Spoofax requires the development of additional tools or transformations.

More recent work on generating provably correct code, for example [41], is restricted to producing primitive getter and setter operations, as opposed to complex procedures. Mammar [42] adopts a formal approach to generating relational databases from UML models. However, this requires the manual definition of appropriate guards for predefined update operations: the automatic derivation of guarded programs from arbitrary specifications is not supported.

In the taxonomy of model transformations proposed by Mens and Gorp [43], our transformations are vertical, because the source and target models reside at different abstraction levels; they are exogenous, because the source and target models are instances of different metamodels; and they are syntactical, because only the syntax of the source model is considered, not its semantics. Our approach is completely automatic, with complex transformations, with a particular focus upon the preservation of behavioural semantics. This places our transformations in the same category as many language compilers, although we may argue that the accompanying, iterative methodology set out in Section 3.2 is rather more specific when it comes to development strategy.

We are continuing to develop the Booster technology through the application of information systems
in the healthcare domain. There is an expectation upon developers in this domain to produce evidence that their system meets published requirements, such as the 21 CFR 11 Code of Federal Regulations in the United States, or the Conformité Européenne marking requirements in the European Union. The provision of a formal specification of structure and functionality for each iteration of a design has significant benefits in this regard; it is important also, however, to have a convincing argument as to the correctness of the automated processes involved in system implementation and data migration. In this paper, we have set out the formal basis for such an argument; in the future, we hope to develop automated support for the proof of correctness of transformation rules.

Acknowledgements

We would like to thank the organisers of the Formal Techniques for Safety-Critical Systems workshop (FTSCS 2012) for the opportunity to present an earlier version of this work, and the referees for their comments upon that version. We would like to acknowledge the contributions of our colleagues Jeremy Gibbons and Edward Crichton. Finally, we would like to dedicate this paper to the memory of Ib Holm Sørensen (1949–2012), without whom none of this would have been possible.

References


