Formal Model-Driven Engineering: Generating Data and Behavioural Components

Chen-Wei Wang
McMaster Centre for Software Certification,
McMaster University
Hamilton, Canada L8S 4K1
jackie@cse.yorku.ca

Jim Davies
Department of Computer Science,
University of Oxford
Oxford, United Kingdom OX1 3QD
jim.davies@cs.ox.ac.uk

Model-driven engineering is the automatic production of software artefacts from abstract models of structure and functionality. By targeting a specific class of system, it is possible to automate aspects of the development process, using model transformations and code generators that encode domain knowledge and implementation strategies. Using this approach, questions of correctness for a complex, software system may be answered through analysis of abstract models of lower complexity, under the assumption that the transformations and generators employed are themselves correct. This paper shows how formal techniques can be used to establish the correctness of model transformations used in the generation of software components from precise object models. The source language is based upon existing, formal techniques; the target language is the widely-used SQL notation for database programming. Correctness is established by giving comparable, relational semantics to both languages, and checking that the transformations are semantics-preserving.

1 Introduction

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 [15]. A software glitch in the anti-lock braking system caused Toyota to recall more than 400,000 vehicles in 2010 [25]; the total cost to the company of this and other software-related recalls in the same period is estimated at \$3bn. In October 2008, 103 people were injured, 12 of them seriously, when a Qantas airliner [3] dived repeatedly as the fly-by-wire software responded inappropriately to data from inertial reference sensors. A modern car contains the product of over 100 million lines of source code [4], and in the aerospace industry, it has been claimed that "the current development process is reaching the limit of affordability of building safe aircraft" [10].

The solution to the problems of increasing software complexity lies in the automatic generation of correct, lower-level software from higher-level descriptions: precise *models* of structure and functionality. The intention is that the same generation process should apply across a class of systems, or at least multiple versions of the same system. Once this process has been correctly implemented, we can be sure that the behaviour of the generated system will correspond to the descriptions given in the models. These models are strictly more abstract than the generated system, easier to understand and update, and more amenable to automatic analysis. This *model-driven* approach [11] makes it easier to achieve correct designs and correct implementations. Despite the obvious appeal of the approach, and that of related approaches such as domain-specific languages [8] and software product lines [18], much of the code that could be generated automatically is still written by hand; even where precise, abstract specifications exist, their implementation remains a time-consuming, error-prone, manual process.

The reason for the delay in uptake is simple: in any particular development project, the cost of producing a new language and matching generator, is likely to exceed that of producing the code by hand. As suitable languages and generators become available, this situation is changing, with significant implications for the development of complex, critical, software systems. In the past, developers would work to check the correctness of code written in a general-purpose programming language, such as C or Ada, against natural language descriptions of intended functionality, illuminated with diagrams and perhaps a precise, mathematical account of certain properties. In the future, they will check the correctness of more abstract models of structure and behaviour, written in a range of different, domain-specific languages; and rather than relying upon the correctness of a single, widely-used compiler, they will need to rely upon the correctness of many different code generators. The correctness of these generators, usually implemented as a sequence of model transformations, is thus a major, future concern.

In this paper, we present an approach to model-driven development that is based upon formal, mathematical languages and techniques. The objective is the correct design and implementation of components with complex state, perhaps comprising a large number of inter-related data objects. The approach is particularly applicable to the iterative design and deployment of systems in which data integrity is a primary concern. The modelling language employed has the familiar, structural features of object notations such as UML—classes, attributes, and associations—but uses logical predicates to characterise operations. An initial stage of transformation replaces these predicates with guarded commands that are guaranteed to satisfy the specified constraints: see, for example, [24]. The focus here is upon the subsequent generation of executable code, and the means by which we may prove that this generation process is correct.

The underlying thesis of the approach is that the increasing sophistication of software systems is often reflected more in the complexity of data models than in the algorithmic complexity of the operations themselves. The intended effect of a given event or action is often entirely straightforward. However, the intention may be only part of the story: there may be combinations of inputs and before-states where the operation, as described, would leave the system in an inconsistent after-state; there may be other attributes to be updated; there may be constraints upon the values of other attributes that need to be taken into account. Furthermore, even if the after-state is perfectly consistent, the change in state may have made some other operation, or sequence of operations, inapplicable.

Fortunately, where the intended effect of an operation upon the state of a system is straightforward, it should be possible to express this effect as a predicate relating before and after values *and* generate a candidate implementation. Using formal techniques, we may then calculate the domain of applicability of this operation, given the representational and integrity constraints of the data model. If this is smaller than required, then a further iteration of design is called for; if not, then the generated implementation is guaranteed to work as intended. In either case, code may be generated to throw an exception, or otherwise block execution, should the operation be called outside its domain. Further, by comparing the possible outcomes with the calculated domains of other operations, we can determine whether or not one operation can affect the availability of others.

The application of formal techniques at a modelling level—to predicates, and to candidate implementations described as abstract programs—has clear advantages. The formal semantics of a modern programming language, considered in the context of a particular hardware or virtual machine platform, is rich enough to make retrospective formal analysis impractical. If we are able to establish correctness at the modelling level, and rely upon the correctness of our generation process, then we may achieve the level of formal assurance envisaged in new standards for certification: in particular, DO-178C [21]. We show here how the correctness of the process can be established: in Section 3, we present the underlying semantic framework; in Section 4, the translation of expressions; in Section 5, the implementation of operations; in Section 6, the approach to verification.

2 Preliminaries

The BOOSTER language [6] is an object modelling notation in which model constraints and operations are described as first-order predicates upon attributes and input values. Operations may be composed using the logical combinators: conjunction, disjunction, implication, and both flavours of quantification. They may be composed also using relational composition, as sequential phases of a single operation or transaction. The constraints describing operations are translated automatically into programs written in an extended version of the Generalised Substitution Language (GSL), introduced as part of the B Method [1]. There may be circumstances under which a program would violate the model constraints, representing business rules, critical requirements, or semantic integrity properties. Accordingly, a guard is calculated for each operation, as the weakest precondition for the corresponding, generated program to maintain the model constraints. The result is an abstract program whose correctness is guaranteed, in a language defined by the following grammar:

$$
\begin{array}{lcll}
\textit{Substitution} & ::= & \textit{skip} & \mid\ \langle\langle \textit{PATH} \rangle\rangle := \langle\langle \textit{Expression} \rangle\rangle \\
 & \mid & \langle\langle \textit{Predicate} \rangle\rangle \longrightarrow \langle\langle \textit{Substitution} \rangle\rangle & \mid\ \langle\langle \textit{Substitution} \rangle\rangle \parallel \langle\langle \textit{Substitution} \rangle\rangle \\
 & \mid & \langle\langle \textit{Substitution} \rangle\rangle \ ; \ \langle\langle \textit{Substitution} \rangle\rangle & \mid\ \langle\langle \textit{Substitution} \rangle\rangle \ \square\ \langle\langle \textit{Substitution} \rangle\rangle \\
 & \mid & !\langle\langle \textit{Variable} \rangle\rangle : \langle\langle \textit{Expression} \rangle\rangle \bullet \langle\langle \textit{Substitution} \rangle\rangle & \\
 & \mid & @\langle\langle \textit{Variable} \rangle\rangle : \langle\langle \textit{Expression} \rangle\rangle \bullet \langle\langle \textit{Substitution} \rangle\rangle &
\end{array}
$$

Here, the usual notation of assignable variables is replaced with *paths*, each being a sequence of attribute names, using the familiar object-oriented 'dot' notation as a separator. *Predicate* and *Expression* represent, respectively, first-order predicates and relational and arithmetic expressions. *skip* denotes termination, $:=$ denotes assignment, and $\longrightarrow$ denotes a program guard: to be implemented as an assertion, a blocking condition, or as (the complement of) an exception. $\square$ denotes alternation, and @ denotes selection: the program should be executed for exactly one of the possible values of the bound variable. Similarly, $\parallel$ denotes parallel composition, with ! as its generalised form: all of the program instances should be performed, in parallel, as a single transaction. ; denotes relational or sequential composition. Inputs and outputs to operations need not be explicitly declared; instead, they are indicated using the decorations ? and ! at the end of the attribute name.

These abstract programs are interpreted as operations at a component applications programming interface (API), with the data model of the component given by a collection of class and association declarations in the usual object-oriented style. The integrity constraints and business rules for the data model can be given as predicates in the same notation, or using the object constraint language (OCL) of the Unified Modelling Language (UML) [11].

As a simple, running example, consider the following description of (a fragment of) the data model for a hotel reservations system

```
class Hotel {                                    class Reservation {
  attributes                                       attributes
    reservations : seq(Reservation.host) [*] }       host : Hotel.reservations }
```

A single hotel may be the **host** for any number of reservations. It may also be the **host** of a number of rooms and allocations: see the class association graph [9] of Figure 1. The action of creating a new reservation may be specified using a simple operation predicate in the context of the `Hotel` class:

```
reserve { # allocations < limit & reservations' = reservations ^ <r!> & r!.room = m? }
```

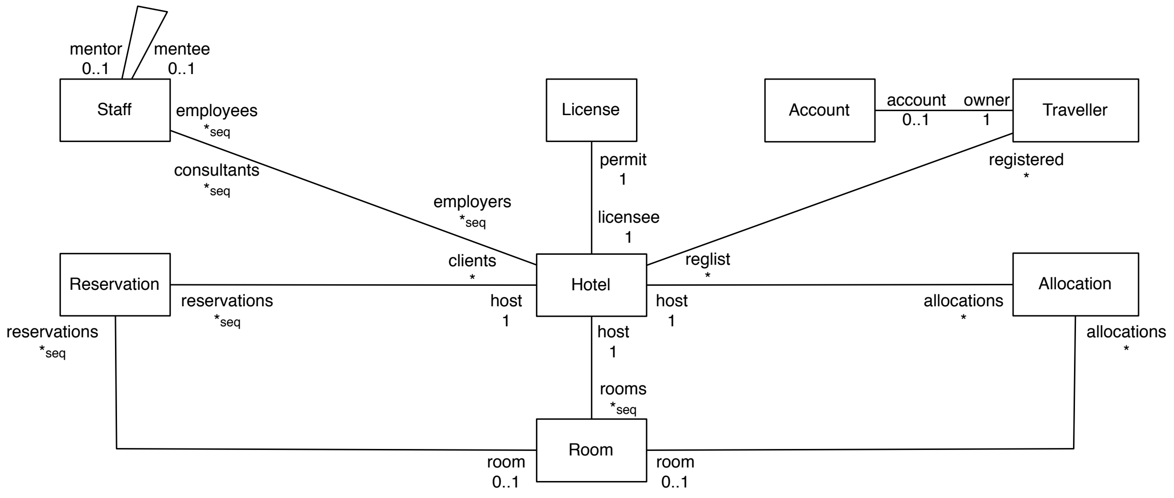

Figure 1: Hotel Reservation System (HRS)—Graph of Class Associations

This requires that a new reservation be created and appended to the existing list, modelled as an ordered association from `Hotel` to `Room`, and that the room involved is given by input `m?`. The operation should not be allowed if the number of reservations in the system has already reached a specified `limit`.

If the constructor operation predicate on `Reservation` mentions a set of dates `dates?`, then this will be added as a further input parameter. We might expect to find also a constraint insisting that any two different reservations associated with the same room should have disjoint sets of dates, and perhaps constraints upon the number of reservations that can held by a particular traveller for the same date. For the purposes of this paper, however, we will focus simply upon the required, consequential actions and the description of the operation as an abstract program.

```
reserve {
     r! : extent(Reservation) & dates? : set(Date) & m? : extent(Room)
       & card(allocations) < limit
  ==>
     r!.dates := r!.dates \/ dates? || r!.status := "unconfirmed"
  || r!.host := this || reservations := ins(reservations, #reservations + 1, r!)
  || r!.room := m?   || m?.reservations := ins(m?.reservations, #m?.reservations + 1, r!)}
```

In this abstract program, the two reservations attributes, in the hotel and room objects, are updated with a reference to the new reservation, the dates attribute of the new reservation is updated to include the supplied dates, and the status attribute is set to `"unconfirmed"`, presumably as a consequence of the constructor predicate for the `Reservation` class.

3 A Unified Implementation and Semantic Framework

To illustrate our formal, model-driven approach, we will consider the case in which the target is a relational database platform. The above program would then be translated into a SQL query, acting on a relational equivalent of our original object model. The transformations can be described using the Haskell [2] functional programming language: in the diagram of Figure 2, thin-lined, unshaded boxes represent to denote Haskell program data types, and thin arrows the executable transformations between them. These constitute an *implementation* framework. The thick-lined, shaded boxes denote the relational semantics of corresponding data types, thick lines with circles at one end the process of assigning a formal meaning, and arrows with circles at each end the relationship between formalised concepts. These constitute a corresponding *semantic* framework for establishing correctness.

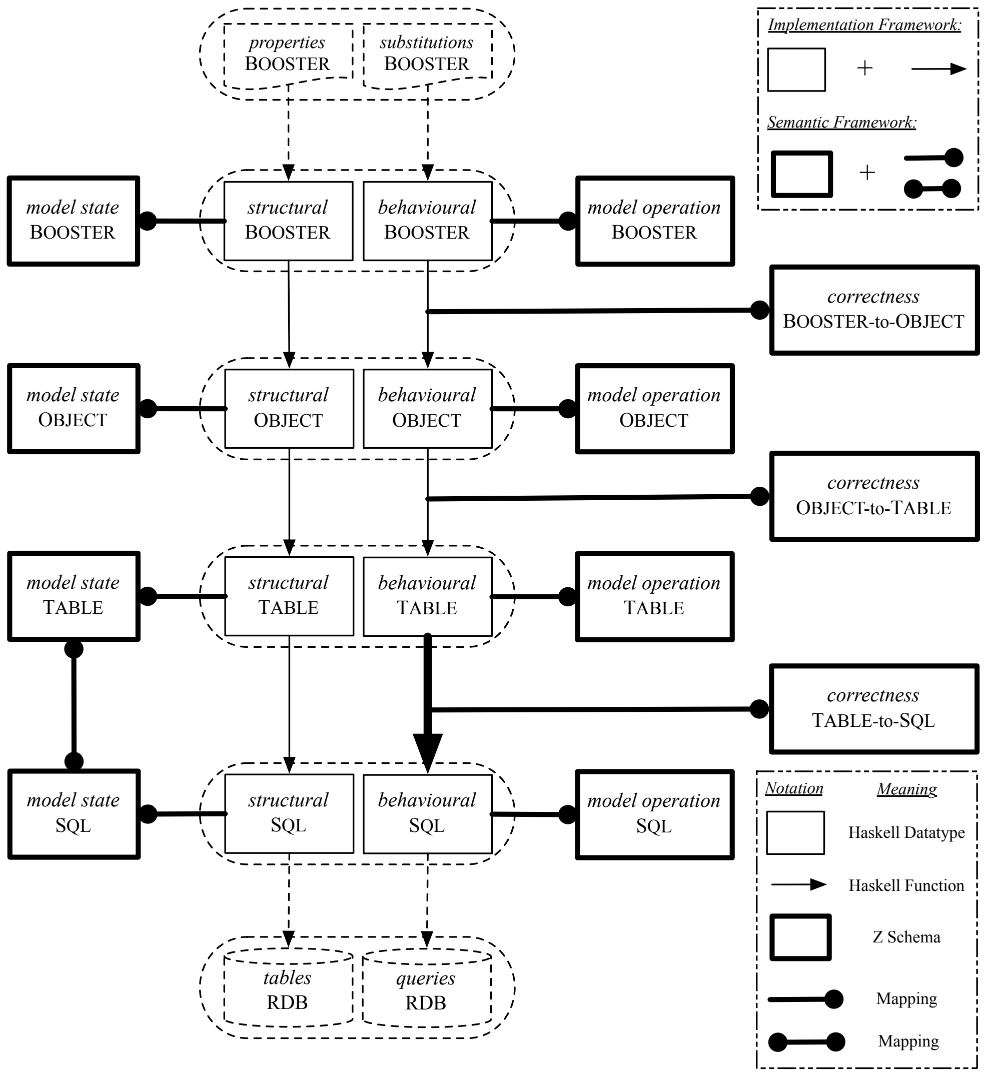

Figure 2: BOOSTER Model to SQL Database: Implementation & Semantic Framework

Four kinds of models are involved in our transformation pipeline: 1) a BOOSTER model, in extended GSL notation, generated from the original predicates; 2) an OBJECT model representing an object-oriented relational semantics for that model; 3) an intermediate TABLE model reflecting our implementation strategy; 4) a SQL model expressed in terms of tables, queries, and key constraints. A final model-to-text transformation will be applied to generate a well-formed SQL database schema.

We use Haskell to define metamodels of model structures and operations as data types. Our transformations are then defined as Haskell functions: from BOOSTER to OBJECT, then to TABLE, and finally to SQL. Our relational semantics is most easily described using the Z notation [26]. Other formal languages with a transformational semantics would suffice for the characterisation of model and operation constraints, but Z has the distinct advantage that each operation, and each relation, may be described as a single predicate: rather than, for example, a combination of separate pre- and post-conditions; this facilitates operation composition, and hence a compositional approach to verification.

4 Path & Expression Transformation

The descriptions of operations in the BOOSTER, OBJECT, and TABLE models are all written in the GSL language; the difference between them lies in the representation of attribute and association references. Instead of creating three versions of a language type `Substitution`, one for each of the reference notations, we employ a type `PATH` as a generic solution: see Figure 3.

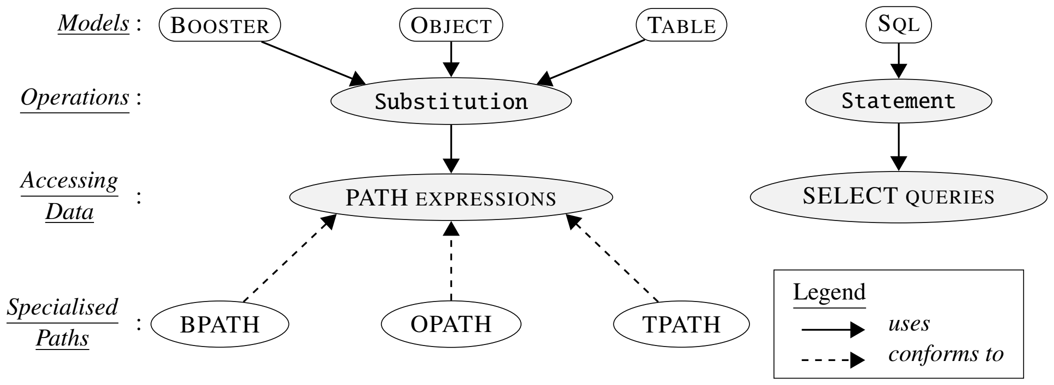

Figure 3: Datatypes of Behavioural Models

We define

```
data PATH = BPath BPATH | OPath OPATH | TPath TPATH
```

where **BPath**, **OPath**, and **TPath** are type constructors. A BOOSTER model path (of type `BPATH`) is represented as a sequence $\langle a_1, \ldots, a_n \rangle$ of name references to attributes/properties. We will refer to this range $1\,..\,n$ of indices for explaining the corresponding OBJECT and TABLE model paths.

We consider structures of the types `OPATH` and `TPATH` in detail. Paths of type `OPATH` are used to indicate explicitly which properties/classes are accessed, along with its chain of navigation starting from the current class.

```
data OPATH         = BaseOPath REF_START        | RecOPath OPATH TARGET
data REF_START     = ThisRef BASE               | SCRef    IDEN_PROPERTY EXPRESSION BASE
data TARGET        = EntityTarget IDEN_PROPERTY | SCTarget IDEN_PROPERTY EXPRESSION
data BASE          = ClassBase N_CLASS          | SetBase N_SET | IntBase | StrBase
type IDEN_PROPERTY = (N_CLASS, N_ATTRIBUTE)
```

An object path is a left-heavy binary tree, where the left-most child refers to its starting reference and all right children represent target classes/properties that are accessed. The starting reference of an object path—which denotes access to, e.g the current object, an element of a sequence-valued property through indexing, etc.—provides explicit information about the base type of that reference. All intermediate and the ending targets of an object path contextualise the properties with their enclosing classes (i.e. `IDEN_PROPERTY`).

For each context path $\langle a_1, \ldots, a_i \rangle$, where $(1 \leq i \leq n-1)$, an OBJECT model path (of type `OPATH`) identifies a target class C; if the source BOOSTER path is valid, then attribute a_{i+1} must have been declared in C.

Example object path. As an example of how the transformation on paths works in practice, consider the `Account` class (Figure 1 shown on page 103). The path `this.owner.reglist` denotes a list of registered hotels and has its `OPATH` counterpart:

```
RecOPath (RecOPath (BaseOPath (ThisRef (ClassBase Account)))
                   (EntityTarget (Account, owner)))
         (EntityTarget (Traveller reglist))
```

where **`RecOPath`** and **`BaseOPath`** are constructors for, respectively, recursive and base OBJECT paths. **`EntityTarget`** and **`ClassBase`** construct type information about the three context paths: `(Account)` for `this`, `(Account, owner)` for `this.owner`, and `Traveller, reglist` for `this.owner.reglist`.

On the other hand, we use a path of type `TPATH` to indicate, for each navigation to a property in the OBJECT model, the corresponding access to a table which stores that property.

```
data TPATH    = BaseTPath                 REF_START
              | RecTPath     TPATH        T_ACCESS
data T_ACCESS = ClassTAccess IDEN_PROPERTY
              | AssocTAccess IDEN_PROPERTY
              | SetTAccess   IDEN_PROPERTY
              | SeqTAccess   IDEN_PROPERTY            -- retrieve all indexed components
              | SeqTCAccess  IDEN_PROPERTY EXPRESSION  -- retrieve an  indexed component
```

A table path is left-heavy (as is an `OPATH`), where the left-most child refers to its starting reference and all right children represent target tables that are accessed. The starting reference of a table path provides exactly the same information as its `OPATH` counterpart (i.e. `REF_START`). All intermediate and the ending targets of a table path denote accesses to a variety of tables, predicated upon our implementation strategy. When the target property is sequence-valued, we distinguish between the two cases where one of its indexed components is to be accessed (`SeqTCAccess`) and where all indexed components are to be accessed (`SeqTAccess`).

For each attribute a_i, where $(1 \leq i \leq n)$, a TABLE model path (of type `TPATH`) recursively records which sort of table (e.g. class tables, association tables, or set tables) it is stored, based on the target class of its context path.

Example table path. The above OBJECT path has its `TPATH` counterpart:

```
RecTPath (RecTPath (BaseTPath (ThisRef (ClassBase Account)))
                   (AssocTAccess (Account, owner)))
         (AssocTAccess (Traveller reglist))
```

where **`RecTPath`** and **`BaseTPath`** construct, respectively, recursive and base TABLE paths. Properties `owner` and `reglist` are accessed in the two corresponding association tables.

Path transformation. We now specify the above `OPATH`-to-`TPATH` transformation in Haskell:

```
objToTabPath :: OBJECT_MODEL -> PATH -> PATH
objToTabPath om (OPath opath) = TPath (objToTabPath' om opath)
```

where the first line declares a function `objToTabPath`, and the second line gives its definition: matching an input object model as `om` and an input path as `(OPath opath)`, whereas the RHS constructs a new `PATH` via `TPath`. The transformation of object paths is given by

```
objToTabPath' om (RecOPath op tar) =
  case tar of
    EntityTarget (c, p) | (c, p) `elem` biAssoc'    om c -> RecTPath tp (AssocTAccess (c, p))
                        | (c, p) `elem` classTables tm c -> RecTPath tp (ClassTAccess (c, p))
                        | (c, p) `elem` setTables   tm   -> RecTPath tp (SetTAccess   (c, p))
                        | (c, p) `elem` seqTables   tm   -> RecTPath tp (SeqTAccess   (c, p))
    SCTarget (c, p) oe  -> let te = objToTabExpr om oe   in RecTPath tp (SeqTCAccess (c, p) te)
  where tm = objToTab      om
        tp = objToTabPath' om op
```

where each condition specified between `|` and `->` denotes a special case of the matched entity target, consisting of class `c` and property `p`. For example, the condition `(c, p) 'elem' classTables tm c` denotes properties that are stored in the table for class `c`.

Each recursive object path is structured as (`RecOPath op tar`), where `op` is its prefix (i.e. context) of type `OPATH`, which we recursively transform into a table path equivalent `tp`; and `tar` is its target property. For each given `tar`, table access is determined by checking membership against various domains: a bidirectional association will be accessed by means of an association table. If the target property is sequence-valued (i.e. the case of `SCTarget`), it cannot be accessed for its entirety, but only one of its members through indexing. The function `objToTabExpr` transforms the index expression `oe` that contains paths of type `OPATH` to `te` that contains paths of type `TPATH`. The function `objToTab` transforms an object model `om` to an equivalent table model `tm`.

SQL database statements express paths via (nested) `SELECT` queries. For example, the above TABLE path has its SQL statement counterpart:

```
SELECT (VAR   'reglist')
       (TABLE 'Hotel_registered_Traveller_reglist')
       (VAR   'oid' = (SELECT (VAR   'owner')
                              (TABLE 'Account_owner_Traveller_account')
                              (VAR    oid = VAR this)))
```

where `oid` is the default column (declared as the primary key) for each table that implements an association. We can show [23] by structural induction that the transformation from *BPATH* to *OPATH*, from *OPATH* to *TPATH*, and from *TPATH* to `SELECT` statements are correct.

Expression transformation. We transform both predicates and expressions on TABLE model into SQL expressions:

```
toSqlExpr  :: TABLE_MODEL -> Predicate  -> SQL_EXPR
toSqlExpr' :: TABLE_MODEL -> Expression -> SQL_EXPR
```

Some transformations are direct

```
toSqlExpr tm (And p q) = toSqlExpr tm p 'AND' toSqlExpr tm q
```

whereas others require an equivalent construction:

```
toSqlExpr' tm (Card e) | isPathExpr e = SELECT [COUNT (VAR oid)] (toSqlExpr' tm e) TRUE
```

5 Assignment Transformation

The most important aspect of the model transformation is the handling of attribute assignments and collection updates. There are 36 distinct cases to consider, given the different combinations of attributes and bidirectional (opposite) associations. We present a single, representative case in Table 1, for an association between an optional attribute (multiplicity 0..1) and a sequence-valued attribute (ordered with multiplicity *) .

Bi-Assoc. Decl.	#	GSL Substitution	SQL Queries

seq-*to*-opt
`class A` `class B`
`bs: seq(B.ao)` `ao: [A.bs]`
	23	$bs := \text{ins}(bs, i, that)$
		$\parallel$
		$that.ao := this$
			`UPDATE` t `SET` $index = index + 1$
			`WHERE` $ao = this$ `AND` $index \geq i$;
			`INSERT INTO` $t\ (bs, ao, index)$ `VALUE` $(that, this, i)$;

Table 1: Assignment Transformation Pattern for *sequence-to-optional* Bi-Association

From left to right, the columns of the table present declarations of properties, numerical identifiers of patterns, their abstract implementation in the substitution program, and their concrete implementation in database queries. The dummy variables *this* and *that* are used to denote instances of, respectively, the current class and the other end of the association.

For each case (for each row of the completed table), we define a transformation function `toSqlProc` that turns a substitution into a list of SQL query statements.

```
toSqlProc tm _ s@(Assign _ _) = transAssign tm s
transAssign :: TABLE_MODEL -> Substitution -> [STATEMENT]
```

The function `toSqlProc` delegates the task of transforming base cases, i.e. assignments, to another auxiliary function `transAssign` that implements the 36 patterns. The recursive cases of `toSqlProc` are straightforward. For example, to implement a guarded substitution, we transform it into an `IfThenElse` pattern that is directly supported in the SQL domain; and to implement iterators (`ALL`, `ANY`), we instantiate a loop pattern, declared with an explicit variant, that is guaranteed to terminate.

6 Correctness Proofs

The correctness of both BOOSTER-to-OBJECT and OBJECT-to-TABLE transformations can be established by constructing a relational model mapping identifiers and paths to references and primitive values, and then showing that the different reference mechanisms identify the same values in each case. To prove the correctness of the TABLE-to-SQL transformation (shown as the vertical, thick arrow in Figure 2 on page 104), we need also to introduce *linking invariants* between model states. We first formalise states and operations for each model domain. In the Z notation, sets and relations may be described using a schema notation, with separate declaration and constraint components and an optional name:

name
declaration
---
constraint

Either component may include schema references, with the special reference Δ denoting two copies of another schema, typically denoting before- and after-versions, the attributes of the latter being decorated with a prime ($'$). The remainder of the mathematical notation is that of standard, typed, set theory.

We map the state OBJECT model into a relational semantics $\mathscr{S}_{obj}$, characterised by:

$\mathscr{S}_{obj}$
$OBJECT\_MODEL$
$extent : N\_CLASS \pfun \mathbb{P}\, ObjectId$
$value : ObjectId \pfun N\_PROPERTY \pfun Value$

$\operatorname{dom} extent = \operatorname{dom} class$
$\forall c : N\_CLASS;\ o : ObjectId \mid$
$\quad c \in \operatorname{dom} extent \wedge o \in extent(c) \bullet \operatorname{dom}(value(o)) = \operatorname{dom}((class\, c).property)$

The inclusion of *OBJECT_MODEL* (whose details are omitted here) enables us to constrain the two mappings according to the type system of the object model in question. *Value* denotes a structured type that encompasses the possibilities of undefined value (for optional properties), primitive value, and set and sequence of values.

The state of a table model will be composed of: 1) the type system of the object model in context; and 2) functions for querying the state of such a context object model. More precisely,

$$\begin{array}{|l}
\hline \textit{TABLE\_MODEL} \\
\textit{OBJECT\_MODEL} \\
\textit{nTableModel} : \textit{N\_MODEL} \\
\textit{assocTables}, \textit{setTables} : \mathbb{P}(\textit{N\_CLASS} \times \textit{N\_PROPERTY}) \\
\hline
\end{array}$$

where *assocTables*, *setTables* and *seqTables* are reflective queries: for example, *assocTables* returns the set of attributes/properties (and their context classes) that are stored in the association tables. We formalise the TABLE model state as:

$$\begin{array}{|l}
\hline \mathscr{S}_{tab} \\
\mathscr{S}_{obj} \\
\textit{TABLE\_MODEL} \\
\hline
\end{array}$$

For each instance of $\mathscr{S}_{obj}$, there should be a corresponding configuration of *TABLE_MODEL*. A SQL database corresponds to a set of named tables, each containing a set of column-value mappings:

$$\begin{array}{|l}
\hline \mathscr{S}_{sql} \\
\textit{tuples} : \textit{N\_TABLE} \rightarrow\!\!\!\!\!+ \mathbb{P}\,\textit{Tuple} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline \textit{Tuple} \\
\textit{values} : \textit{N\_COLUMN} \rightarrow\!\!\!\!\!+ \textit{ScalarValue} \\
\hline
\end{array}$$

We use *ScalarValue* to denote the collection of basic types: strings, integers, and Booleans. We require mapping functions to retrieve values from TABLE and SQL:

$$\mathscr{M} : \mathscr{S}_{tab} \times (\textit{NClass} \times \textit{NProperty}) \rightarrow\!\!\!\!\!+ \mathbb{P}(\textit{Value} \times \textit{Value})$$

These return reference–value pairs for each kind of property. For example, set-valued properties are returned by

$$\begin{array}{l}
\mathscr{M}_{set} == \lambda\, s : \mathscr{S}_{tab};\ p : \textit{NClass} \times \textit{NProperty} \bullet \\
\qquad \bigcup \left\{ \begin{array}{l} o : \textit{ObjectId};\ v : \textit{Value};\ vs : \mathbb{P}\,\textit{Primitive} \mid \\ \quad o \in s.\textit{extent}\,(\textit{fst}\,p) \wedge v = s.\textit{value}\,(o)\,(\textit{snd}\,p) \wedge v = \textit{setValue}\,(vs) \bullet \\ \qquad \{\, v' : vs \bullet [\![o]\!]^{SV} \mapsto [\![v']\!]^{SV} \,\} \end{array} \right\}
\end{array}$$

The set of mappings for a particular table is given by

$$\lambda\, s : \mathscr{S}_{sql};\ n : \textit{NTable};\ c_1, c_2 : \textit{NColumn} \bullet \{\ \textit{row} : s.\textit{tuples}\,(n) \bullet \textit{row}.\textit{values}\,(c_1) \mapsto \textit{row}.\textit{values}\,(c_2)\ \}$$

and the necessary linking invariant is:

$$\begin{array}{|l}
\hline \text{TABLE} \leftrightarrow \text{SQL} \\
\mathscr{S}_{tab} \\
\mathscr{S}_{sql} \\
\hline
C \\
\hline
\end{array}$$

where C comprises six conjuncts, one for each possible unordered combination of association end multiplicities.

Each operation is implemented as an atomic transaction. $\mathscr{R}_{obj}$ represents the formal context, with the effect upon the state being described as a binary relation ($\leftrightarrow$).

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathscr{R}_{obj}} \\
\hline
input : \mathbb{P}\, N\_VARIABLE \\
output : \mathbb{P}\, N\_VARIABLE \\
effect : (\mathscr{S}_{obj} \times IO_{obj}) \leftrightarrow (\mathscr{S}_{obj} \times IO_{obj}) \\
\hline
effect \in \mathscr{S}_{obj} \times (input \rightarrow Value) \leftrightarrow \mathscr{S}_{obj} \times (output \rightarrow Value) \\
\hline
\end{array}$$

Each element of $IO_{obj} == N\_VARIABLE \nrightarrow Value$ represents a collection of inputs and outputs.

Using $\mathscr{S}_{obj}$ and $\mathscr{R}_{obj}$, we may write $System_{obj}$ to denote the set of possible object system configurations, each characterised through its current state (of type $\mathscr{S}_{obj}$) and its set of indexed operations (of type $\mathscr{R}_{obj}$). More precisely,

$$\begin{array}{|l}
\multicolumn{1}{l}{System_{obj}} \\
\hline
state : \mathscr{S}_{obj} \\
relation : N\_CLASS \nrightarrow N\_OPERATION \nrightarrow \mathscr{R}_{obj} \\
\hline
\end{array}$$

We will describe the effect of a primitive assignment (:=), and use this as the basis for a recursive definition of effect, based on the grammar of the GSL notation. If *AssignInput* is the schema $[path? : PATH;\ e? : Expression]$, then we may define

$$\begin{array}{|l}
\multicolumn{1}{l}{AssignEffect} \\
\hline
s, s' : \mathscr{S}_{obj} \\
AssignInput \\
\hline
s.nObjModel = s'.nObjModel \\
s.sets = s'.sets \\
s.classes = s'.classes \\
s.extent = s'.extent \\
\mathbf{let}\ p == target\,[\![path?]\!]\ ;\ o == context\,[\![path?]\!] \bullet s'.value = s.value \oplus \{o \mapsto \\
\qquad\qquad\qquad\qquad\qquad\qquad s.value\,(o) \oplus \{p \mapsto eval\,(e?)\}\} \\
\hline
\end{array}$$

The input *path*? can be either `OPATH` and `TPATH`: for the former, the other input expression *e*? involves paths, if any, of type `OPATH`; for the latter, it is `TPATH`. The (**let** *es* • *p*) expression, where *es* consists of a list of expression-to-variable bindings, denotes a predicate *p* on the variables of *es*.

We start by relating domains of the OBJECT model and TABLE model, where assignment paths are specified in, respectively, `OPATH` and `TPATH` (Fig 3). In the OBJECT model domain, an assignment is parameterised by a path of type `BPATH` and an expression that consists of paths, if any, of the consistent type. We formalise each OBJECT model assignment under the formal context of $\mathscr{R}_{obj}$, by defining its *effect* mapping though the constraint of *AssignEffect* and by requiring that the sets of external inputs and outputs are empty.

$$\begin{array}{|l}
\multicolumn{1}{l}{Assign_{obj}} \\
\hline
\mathscr{R}_{obj} \\
op? : OPATH \\
oe? : Expression \\
\hline
\forall s, s' : \mathscr{S}_{obj};\ AssignInput \mid path? = OPath\,(op?) \land e? = oe? \bullet AssignEffect \Leftrightarrow (s, \{\}) \mapsto (s', \{\}) \in effect \\
\hline
\end{array}$$

The characterisation $Assign_{tab}$ of an assignment in the TABLE model domain is similar to that of $Assign_{obj}$, except that the target is now of type `TPATH`, and the source is now of type *Expression*. We may then map our extended GSL substitution into a relation:

$$[\![-]\!]_{obj} : Substitution \rightarrow ((\mathscr{S}_{obj} \times IO_{obj}) \leftrightarrow (\mathscr{S}_{obj} \times IO_{obj}))$$

of the same type as the *effect* component of $\mathscr{R}_{obj}$. Given a TABLE path $tp?$ and an expression $te?$, we represent the assignment substitution $tp? := te?$ by the effect relation of $Assign_{tab}$ that exists uniquely with respect to $tp?$ and $te?$. More precisely,

$$[\![tp? := te?]\!]_{obj} = (\mu\, Assign_{tab}).effect$$

where $\mu\, Assign_{tab}$ denotes the unique instance of $\mathscr{S}_{tab}$ such that the constraint as specified in $Assign_{tab}$ holds, and *.effect* retrieves its relational effect on the model state. The definition of $Assign_{tab}$ is very similar to that of $Assign_{obj}$, except that the input path is constrained as $path? = TPath(\ldots)$.

We interpret a guarded substitution $g \longrightarrow S$ as a relation that has the same effect as $[\![S]\!]_{obj}$ within the domain of satisfying states of guard g (denoted as $[\![g]\!]_{obj}^{states}$); otherwise, it just behaves like *skip* as it will be blocked and cannot achieve anything. More precisely, we have:

$$[\![g \longrightarrow S]\!]_{obj} = \mathrm{id}\,(\mathscr{S}_{obj} \times IO_{obj}) \oplus (\, [\![g]\!]_{obj}^{states} \lhd [\![S]\!]_{obj} \,)$$

Similar rules may be defined for other combinators.

Each transaction is composed of SQL queries, and similar to $\mathscr{R}_{obj}$, we collect and produce, respectively, its list of inputs and outputs upon its initiation and completion. We use $\mathscr{R}_{sql}$ to denote such formal context, under which the transformational effect on the state of database is defined accordingly as a function, reflecting the fact that the database implementation is deterministic in its effect.

$\mathscr{R}_{sql}$

$input, output : \mathbb{P}\, N\_VARIABLE$
$effect : (\mathscr{S}_{sql} \times IO_{sql}) \rightarrow (\mathscr{S}_{sql} \times IO_{sql})$

$this \in input$

The mechanism of referencing the current object (via *this*) is simulated through providing by default the value of *this* for each generated stored procedure or function. We model inputs and outputs in the same way as we do for IO_{obj}, except that the range of values is now of type *ScalarValue*.

For each SQL statement, we assign to it a relational semantics by mapping it to a relation on states (of type $\mathscr{S}_{sql}$). This is a similar process to that for $[\![-]\!]_{obj}$. More precisely, we define:

$$[\![-]\!]_{sql} : Statement \rightarrow ((\mathscr{S}_{sql} \times IO_{sql}) \leftrightarrow (\mathscr{S}_{sql} \times IO_{sql}))$$

And since a SQL stored procedure is defined as a sequential composition, we also define

$$[\![-]\!]_{\mathrm{seq}\,sql} : \mathrm{seq}\, Statement \rightarrow ((\mathscr{S}_{sql} \times IO_{sql}) \leftrightarrow (\mathscr{S}_{sql} \times IO_{sql}))$$

to derive its effect through combining those of its component statements via relational composition. For primitive query statements, we refer to their schema definitions. For example, we have:

$$[\![\texttt{UPDATE}\ t\ \texttt{SET}\ sets\ \texttt{WHERE}\ cond]\!]_{sql} = (\mu\, \texttt{UPDATE}).effect$$

where the state effect of query (`UPDATE` *table*? `SET` *sets*? `WHERE` *cond*?) is formally specified in a schema named `UPDATE`. The `UPDATE` query modifies in a table those tuples that satisfy a condition and takes as inputs *table*? a table name, *sets*? a mapping that specifies how relevant columns should be modified, and *cond*? a Boolean condition that chooses the range of tuples to be modified. The schema `UPDATE` is defined similarly as is $Assign_{tab}$, except that it imposes constraints on the model state $\mathscr{S}_{sql}$. We formalise an `IF`...`THEN`...`ELSE`... statement as the union of the semantic interpretations of the two sequences of statements in its body, each suitably restricted on its domain.

$$[\![\, \texttt{IF}\ b\ \texttt{THEN}\ stmts_1\ \texttt{ELSE}\ stmts_2 \,]\!]_{sql} = ([\![\, b \,]\!]_{sql}^{states} \lhd [\![\, stmts_1 \,]\!]_{\mathrm{seq}\,sql}) \cup ([\![\, \texttt{NOT}\, b \,]\!]_{sql}^{states} \lhd [\![\, stmts_2 \,]\!]_{\mathrm{seq}\,sql})$$

where $[\![\, b \,]\!]_{sql}^{states}$ denotes the set of satisfying state of a SQL expression b.

To define the semantics of a `WHILE` loop, we intend for the following equation to hold

$$[\![\, \texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE} \,]\!]_{sql} = [\![\, \texttt{IF}\ b\ \texttt{THEN}\ stmts \frown \langle \texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE} \rangle\ \texttt{ELSE}\ \langle\rangle \,]\!]_{sql}$$

where $\frown$ is the operator for sequence concatenation. By applying the definition of $[\![\, \_ \,]\!]_{sql}$ on `IF`...`THEN`...`ELSE`... and $[\![\, \_ \,]\!]_{\mathrm{seq}\,sql}$ on $\langle\rangle$, we have

$$\begin{aligned}\boxed{[\![\, \texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE} \,]\!]_{sql}} = {} & [\![\, b \,]\!]_{sql}^{states} \lhd (\, [\![\, stmts \,]\!]_{\mathrm{seq}\,sql} \mathbin{⨾} \boxed{[\![\, \texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE} \,]\!]_{sql}}\,) \\ & \cup \\ & [\![\, \texttt{NOT}\, b \,]\!]_{sql}^{states} \lhd \mathrm{id}\,(\mathscr{S}_{sql} \times IO_{sql})\end{aligned}$$

Let us define a function

$$F\,(X) = (\, [\![\, b \,]\!]_{sql}^{states} \lhd (\, [\![\, stmts \,]\!]_{\mathrm{seq}\,sql} \mathbin{⨾} X)\,) \cup (\, [\![\, \texttt{NOT}\ b \,]\!]_{sql}^{states} \lhd \mathrm{id}\,(\mathscr{S}_{sql} \times IO_{sql})\,)$$

When $X = [\![\, \texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE} \,]\!]_{sql}$, we obtain

$$[\![\, \texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE} \,]\!]_{sql} = F\,(\ [\![\, \texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE} \,]\!]_{sql}\ )$$

which means that $[\![\, \texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE} \,]\!]_{sql}$ should be a fixed-point of function F. The least fixed-point (LFP) of function F—i.e. $\bigcup_{n \in \mathbb{N}} F^n\,(\emptyset)$—exists by Kleene's fixed-point theorem, since F is easily provable to be continuous. We choose this LFP of F for the value of $[\![\, \texttt{WHILE}\ b\ \texttt{DO}\ stmts\ \texttt{END WHILE} \,]\!]_{sql}$.

We are now able to establish the correctness of the transformation with respect to the linking invariant. The commuting diagram of Figure 4 shows how a substitution program *prog* and its context TABLE model (i.e. $\theta\, TableModel$), are mapped by the transformation $toSqlProc\,(\theta\, TableModel)\,(prog)$ to produce an SQL implementation. The linking invariant holds for the before states TABLE $\leftrightarrow$ SQL and for the after states TABLE $\leftrightarrow$ SQL$'$. We then establish that for each state transformation, characterised by the relational effect of the generated SQL code from *prog*, there is at least a corresponding state transformation, characterised by the relational effect of the TABLE program, $[\![\, prog \,]\!]_{obj}$. This is an example of simulation between abstract data types [19].

We use a universal quantification $(\forall x \mid R(x) \bullet P(x))$ to state our correctness criterion: the x part declares variables, the $R(x)$ part constrains the range of state values, and the $P(x)$ part states our concern. Schemas defined above (i.e. $\mathscr{S}_{obj}$, $\mathscr{S}_{sql}$, and TABLE $\leftrightarrow$ SQL) are used as both declarations and predicates. If we declare

```
── TransInput ──────────────────────────
│ TableModel
│ prog : Substitution
└────────────────────────────────────────
```

$$\begin{tikzcd}
\textsc{Object Pre-state}\ \mathscr{S}_{obj} \arrow[r, "\llbracket prog \rrbracket_{obj}"] \arrow[d, dash, "\textsc{Table} \leftrightarrow \textsc{Sql}"'] & \textsc{Object Post-state}\ \mathscr{S}'_{obj} \arrow[d, dash, "(\textsc{Table} \leftrightarrow \textsc{Sql})'"] \\
\textsc{Sql Pre-state}\ \mathscr{S}_{sql} \arrow[r, "\llbracket toSqlProc\,(\theta\, TableModel)\,(prog) \rrbracket_{sql}"'] & \textsc{Sql Post-state}\ \mathscr{S}'_{sql}
\end{tikzcd}$$

Figure 4: Correctness of Model Transformation

to represent the inputs to the transformation, then

$$\begin{aligned}
&\forall\, TransInput;\ \Delta\mathscr{S}_{obj};\ \Delta\mathscr{S}_{sql}\ | \\
&\quad \textsc{Table} \leftrightarrow \textsc{Sql} \wedge \llbracket toSqlProc\,(\theta\, TableModel)\,(prog) \rrbracket_{\mathrm{seq}\,sql} \bullet \left(\ \exists\, \mathscr{S}'_{obj} \bullet (\textsc{Table} \leftrightarrow \textsc{Sql})' \wedge \llbracket prog \rrbracket_{obj}\ \right)
\end{aligned}$$

With the relational semantics outlined above, we may establish this result through a combination of case analysis and structural induction.

7 Example Implementation

Consider the implementation, on a relational database platform, of the operation `reserve` introduced in Section 2. Having translated the object model into a collection of database tables, the generation process will produce a stored procedure for each operation. The guard for `reserve` requires that the current number of allocations—characterised through the cardinality of the set-valued attribute `allocations`—is below a specific bound. We might include such a condition, for example, to ensure that the memory or storage requirements of the system remain within particular bounds; this may not be an issue for a hotel reservation system, but is a realistic concern in critical systems development. In the implementation, a stored function is generated that will establish whether or not the guard constraint holds for the current state, together with any input values. The remainder of the generated code will achieve the effect specified in the original operation constraint, translated into the representation, or orientation, of the database platform.

Class `Reservation` has `status` as an attribute, and this is stored in the corresponding class table. In the function, `AUTO_INCREMENT` allows the target SQL platform to generate a unique identifier for each inserted row. Set-valued properties, like attribute `dates` in class `Reservation` are stored in separate tables, with an `oid` column to identify the current object in a given method call. Associations such as `host` and `reservations` are stored in separate tables, with an `oid` column to identify the exact association instance. Since attribute `reservations` are also sequence-valued, an `index` column is required.

Schema of Tables Updated by 'reserve'

```
CREATE TABLE `Reservation`(`oid` INTEGER AUTO_INCREMENT, PRIMARY KEY (`oid`), `status` CHAR(30));
CREATE TABLE `Room_reservations_Reservation_room`(`oid` INTEGER AUTO_INCREMENT,
        PRIMARY KEY (`oid`), `reservations` INTEGER, `room` INTEGER, `index` INTEGER);
```

We generate also integrity constraints for association tables: although the generated procedures are guaranteed to preserve semantic integrity, this affords some protection against errors in the design of additional, manually-written procedures.

The value of the model-driven approach should be apparent following a comparison of the original specification for `reserve` with the fragments of the following SQL implementation. Manual production of queries that need to be take account of a large number of complex model constraints—as well as, for examples, constraints arising from data caching strategies—is time-consuming and error-prone. Furthermore, we may expect the design of a system to evolve during use: the challenge of maintaining correctness in the face of changing specifications (and platforms) adds another dimension of complexity to systems development; some degree of automation, in production and in proof, is essential.

In the following, variable names have been preserved from the BOOSTER domain, e.g. the input and output parameters `dates?` and `r!` at line 2, as well as caching variables `‘r!.status‘`, `‘r!.host‘`, and `‘r!.room‘` at line 4. Meta-variables are used to implement the `ALL` iterator in method `reserve`: Line 5 declares, respectively, the bound variable `‘x‘` and `‘x_variant‘` the variant of the loop, and Line 6 declares a cursor over the set-valued input `dates?`.

Queries Implementing 'reserve': Declarations

```
1 CREATE PROCEDURE ‘Hotel_reserve‘  (IN ‘this?‘ INTEGER,
2     IN ‘dates?‘ CHAR(30), IN ‘m?‘ INTEGER, OUT ‘r!‘ INTEGER)
3 BEGIN
4   DECLARE ‘r!.status‘ CHAR(30); DECLARE ‘r!.host‘ INTEGER; DECLARE ‘r!.room‘ INTEGER;
5   DECLARE ‘x‘ Date; DECLARE ‘x_variant‘ INTEGER;
6   DECLARE ‘x_cursor‘ CURSOR FOR (SELECT * FROM ‘dates?‘ WHERE TRUE);
```

Line 7 first creates a new instance of `Reservation` by inserting, for output `r!`, a row formatted as $\langle oid, \ldots \rangle$ into the appropriate class table, where *oid* is a unique value generated by the built-in function `last_insert_id()`, with the guarantee that each subsequent call to this functions returns a new value. It then assigns this unique identifier to `r!` for queries in later fragments to refer to.

Queries Implementing 'reserve': Creating an Empty Output

```
7     INSERT INTO ‘Reservation‘ () VALUE (); SET ‘r!‘ = last_insert_id ();
```

In Lines 8 to 10 the pair of `DROP TEMPORARY TABLE` and `CREATE TEMPORARY TABLE` queries update the value of a cache variable `‘m?.reservations‘` that denotes a multi-valued property: this kind of caching is useful in large database implementations. In Line 11 we update the caching variable `‘r!.host‘` of single-valued types of properties through a `SELECT INTO` query. We cache the value of attribute `host` possessed by the reservation `r!`. Any later paths with `‘r!.host‘` or `‘m?.reservations‘` as its prefix will be able to use its value directly without re-evaluation.

Queries Implementing 'reserve': Updating Caching Vars

```
8     DROP TEMPORARY TABLE IF EXISTS ‘m?.reservations‘;
9     CREATE TEMPORARY TABLE ‘m?.reservations‘ AS
10      SELECT ‘reservations‘ FROM ‘Room_reservations_Reservation_room‘ WHERE ‘room‘ = ‘m?‘;
11    SELECT ‘status‘ INTO ‘r!.status‘ FROM ‘Reservation‘ WHERE ‘oid‘ = ‘r!‘;
```

Lines 12 to 20 instantiate a finite loop pattern. In Line 12 we activate the declared cursor and and fetch its first available value. In Line 13 we also calculate the size of the data set that the cursor will iterate over and use it as the variant of the loop defined in Lines 14 to 20. The exit condition (Line 14) is characterised through decreasing—via the 2nd statement in Line 19—the value of `x_cursor`; the bound variable `x` is updated to the next data item at the end of each iteration (via the 1st statement in Line 19). In each iteration of the loop, from Lines 15 to 17 we re-cache the value of the set-valued path `r!.dates`, in case there are other paths which contain it as a prefix and are used later in the loop. In Line 18 we perform the first substitution in the specification of method `reserve`: we implement the substitution `r!.dates := r!.dates \/ dates?` via iterating through the input `dates?` with a bound variable `‘x‘`.

Queries Implementing 'reserve': Terminating Loop

```
12    OPEN ‘x_cursor‘; FETCH ‘x_cursor‘ INTO ‘x‘;
13    SELECT COUNT(*) INTO ‘x_variant‘ FROM ‘dates?‘ WHERE TRUE;
14    WHILE (‘x_variant‘) > (0) DO
15      DROP TEMPORARY TABLE IF EXISTS ‘r!.dates‘;
16      CREATE TEMPORARY TABLE ‘r!.dates‘ AS
17        SELECT ‘dates‘ FROM ‘Reservation_dates‘ WHERE ‘oid‘ = ‘r!‘;
18      INSERT INTO ‘Reservation_dates‘ (‘oid‘, ‘dates‘) VALUE (‘r!‘, ‘x‘);
19      FETCH ‘x_cursor‘ INTO ‘x‘; SET ‘x_variant‘ = ‘x_variant‘ - 1;
20    END WHILE; CLOSE ‘x_cursor‘;
```

Line 21 implements the update `r!.status := unconfirmed`. The two generated query statements—that are located in Lines 22 to 27 and Lines 28 to 31—implement the last two parallel assignments in `reserve` that update the *optional-to-sequence* association. They correspond exactly to the rules specified for pattern 23 in Section 5. The queries for the middle two parallel assignments in `reserve`, updating the *one-to-sequence* association, are entirely similar.

Queries Implementing 'reserve': Performing Updates

```
21  UPDATE ‘Reservation‘ SET ‘status‘ = 'unconfirmed' WHERE (‘oid‘) = (‘r!‘);
22  UPDATE ‘Room_reservations_Reservation_room‘
23  SET ‘index‘ = (‘index‘) + (1)
24  WHERE ‘room‘ = ‘m?‘ AND
25        ‘index‘>=(SELECT COUNT(‘oid‘)
26                  FROM (SELECT ‘reservations‘ FROM ‘m?.reservations‘ WHERE TRUE) AS reservations
27                  WHERE TRUE) + 1;
28  INSERT INTO ‘Room_reservations_Reservation_room‘ (‘reservations‘, ‘room‘, ‘index‘) VALUE
29    (‘r!‘,‘m?, (SELECT COUNT(‘oid‘)
30                FROM (SELECT ‘reservations‘ FROM ‘m?.reservations‘ WHERE TRUE) AS reservations
31                WHERE TRUE) + 1);
```

8 Discussion

The principal contribution of this paper is the presentation of a practical, formal, model-driven approach to the development of critical systems. Both the modelling notation and the target programming language are given a formal, relational semantics: the latter only for a specific subset of the language, sufficient for the patterns of implementation produced by the code generation process. The generation process is formalised as a functional program, easily related to the corresponding transformation on the relational semantics. It is perfectly possible to prove the generator correct; indeed, a degree of automatic proof could be applied here. The task of system verification is then reduced to the strictly simpler task of model verification or validation.

The implementation platform chosen to demonstrate the approach is a standard means of storing data, whether that data was originally described in a hierarchical, a relational, or an object-oriented schema. In particular, there are many products that offer a means of mapping [20] from object models (as used here) to a relational database implementation: Hibernate [5] is perhaps the best-known example. However, translating the data model to a data schema is relatively straightforward; the focus here is the generation of correct implementations for operations.

At the same time, much of the work on program transformation is focussed, unsurprisingly, upon code rewriting rather than the generation of complete software components with persistent data. The work on Vu-X [16], where modifications to a web interface are reflected back into the data model is an interesting exception, but has yet to be extended to a formal treatment of data integrity. The work on UnQL [12] supports the systematic development of model transformation through the composition of graph-based transformations: this is a powerful approach, but again no similar framework has been proposed.

Some work has been done in precise data modelling in UML, for example [7], but no formal account has been given for the proposed translation of operations. The Query/View/Transformation approach [17] focuses on design models, but the transformations [13] are described in an imperative, stateful, style, making proofs of correctness rather more difficult. Recent work on generating provably correct code, for example [22], is restricted to producing primitive getter and setter methods, as opposed to complex procedures. Mammar [14] adopts a formal approach to generating relational databases from UML models. However, this requires the manual definition of appropriate guards for predefined update methods: the automatic derivation of guards, and the automatic generation of methods from arbitrary constraint specifications, as demonstrated here, is not supported.

The unified implementation and semantic framework for transformation (Figure 2) presented here can be applied to any modelling and programming notation that admits such a relational semantics for the behaviour of components. It is important to note that the style of this semantics effectively limits the approach to the development of sequential data components: that is, components in which interactions with important data are managed as exclusive transactions; our semantic treatment does not allow us to consider the effects of two or more complex update operations executing concurrently.

In practice, this is not a significant limitation. Where data is encapsulated within a component, and is subject to complex business rules and integrity constraints, we may expect to find locking or caching protocols to enforce data consistency in the face of concurrent requests, by means of an appropriate sequentialisation. Where concurrency properties are important, they can be addressed using process semantics and model-checking techniques; a degree of automatic generation may even be possible, although this is likely to be at the level of workflows, rather than data-intensive programs.

Work is continuing on the development of the transformation and generation tools discussed here, with a particular emphasis upon the incremental development of operation specifications and models. It is most often the case that a precise model will prove too restrictive: when a property is written linking two or more attributes, it constrains their interpretation; if one of these attributes is used also elsewhere in the model, or within an operation, then that usage may not always be consistent with the now formalised interpretation. In our approach, such a problem manifests itself in the unavailability of one or more operations, in particularly circumstances.

As a guard is generated for each operation, sufficient to protect any data already acquired, each incremental version of the system can be deployed without risk of data loss. It can then be used in practice and in earnest, allowing users to determine whether or not the availability—or the overall design—of each operation and data view matches their requirements and expectations. Where an operation has a non-trivial guard, additional analysis may be required to demonstrate that the resulting availability matches requirements: in many cases, the necessary check or test can be automated. The work described here provides a sound foundation for this development process.

References

[1] J.-R. Abrial (1996): *The B-book: assigning programs to meanings*. Cambridge University Press.

[2] R. Bird (1998): *Introduction to Functional Programming using Haskell*. Prentice Hall.

[3] Australian Transport Safety Bureau (2011): *In-flight upset 154km West of Learmouth, WA, VH-QPA, Airbus A330-303*. Aviation Safety Investigations and Reports AO-2008-070.

[4] R. N. Charette (2009): *This car runs on code*. IEEE Spectrum. Available at `http://www.spectrum.ieee.org/feb09/7649`.

[5] JBoss Community: *Hibernate: Relational Persistence for Java and .NET*. `www.hibernate.org`.

[6] J. Davies, C. Crichton, E. Crichton, D. Neilson & I. H. Sørensen (2005): *Formality, evolution, and model-driven software engineering*. ENTCS 130, pp. 39–55, doi:10.1016/j.entcs.2005.03.004.

[7] B. Demuth & H. Hussmann (1999): *Using UML/OCL Constraints for Relational Database Design*. In: UML, LNCS 1723, pp. 598–613, doi:10.1007/3-540-46852-8_42.

[8] A. van Deursen, P. Klint & J. Visser (2000): *Domain-Specific Languages: An Annotated Bibliography*. SIGPLAN Notices 35(6), pp. 26–36, doi:10.1145/352029.352035.

[9] H.-E. Eriksson, M. Penker & D. Fado (2003): *UML 2 Toolkit*. Wiley.

[10] P.H. Feiler (2010): *Model-based validation of safety-critical embedded systems*. In: Aerospace Conference, IEEE, pp. 1 – 10, doi:10.1109/AERO.2010.5446809.

[11] D. S. Frankel (2003): *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley.

[12] S. Hidaka, Z. Hu, H. Kato & K. Nakano (2009): *Towards a compositional approach to model transformation for software development*. In: SAC, ACM, pp. 468–475, doi:10.1145/1529282.1529383.

[13] F. Jouault, F. Allilaire, J. Bézivin & I. Kurtev (2008): *ATL: A model transformation tool*. Science of Computer Programming 72(1–2), pp. 31–39, doi:10.1016/j.scico.2007.08.002.

[14] A. Mammar (2009): *A systematic approach to generate B preconditions: application to the database domain*. Software and Systems Modeling 8(3), pp. 385–401, doi:10.1007/s10270-008-0098-8.

[15] A. Massoudi (2012): *Knight Capital glitch loss hits $461m*. Financial Times.

[16] K. Nakano, Z. Hu & M. Takeichi (2009): *Consistent Web site updating based on bidirectional transformation*. Int. J. Softw. Tools Technol. Transf. 11(6), pp. 453–468, doi:10.1007/s10009-009-0124-3.

[17] OMG (2009): *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. OMG Document ptc/09-12-05, Object Management Group. `http://www.omg.org/spec/QVT/1.1/Beta2/PDF/`.

[18] K. Pohl, G. Böckle & F. J. van der Linden (2005): *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, doi:10.1007/3-540-28901-1.

[19] W. P. de Roever & K. Engelhardt (1999): *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press.

[20] C. Russell (2008): *Bridging the Object-Relational Divide*. ACM Queue 6(3), pp. 18–28, doi:10.1145/1394127.1394139.

[21] RTCA SC-205 (2011): *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*. Approved by Special Committee 205 of Radio Technical Commission for Aeronautic.

[22] K. Stenzel, N. Moebius & W. Reif (2011): *Formal Verification of QVT Transformations for Code Generation*. In: MoDELS, pp. 533–547, doi:10.1007/978-3-642-24485-8_39.

[23] C.-W. Wang (2012): *Model-Driven Development of Information Systems*. Ph.D. thesis, University of Oxford, Oxford University Research Archive.

[24] J. Welch, D. Faitelson & J. Davies (2008): *Automatic Maintenance of Association Invariants*. Software and Systems Modeling 7(3), pp. 287–301, doi:10.1007/s10270-008-0085-0.

[25] M. Williams (2010): *Toyota to recall Prius hybrids over ABS software*. Computerworld.

[26] J. Woodcock & J. Davies (1996): *Using Z*. Prentice Hall.