Database Development of a Work-Flow Planning and Tracking System Using VDM-SL

Rudolf Schlatte[1,2] and Bernhard K. Aichernig[1]

[1] Technical University of Graz, Institute for Software Technology (IST), Münzgrabenstr. 11/II, A-8010 Graz, Austria.
[2] Ferk Informatik, Kasernstr. 82, A-8010 Graz, Austria.
Emails: {rschlatt | aichernig}@ist.tu-graz.ac.at

Abstract. This paper presents the techniques and experiences of an industrial project, applying the formal method VDM in order to develop a work-flow planning and tracking system. A method for creating formal models of relational databases has been developed — specifically, formal representations of SQL datatypes, a model of the structure of a database including primary and foreign keys and the formulation of queries in the model. With this approach, familiar informal design methods from the database world can be used without giving up the expressive power of formal methods. The transition from the informal to the formal world can be automated, keeping the relevant parts of the formal model synchronized with minimal effort. The formal method used is VDM, but the results are easily transferred into similar model oriented methods.

1 Introduction

During the last few years the interest in so called light-weight formal methods has been growing rapidly [7, 9, 20, 17, 15, 10]. One of the main reasons for this is the realization that the advantages of specifying a system formally should not be missed, even if formal proofs are out of a project's scope. Since language determines our thinking, a precise formal specification technique facilitates the consistent design of computer based system at an appropriate level of abstraction — and abstraction is the key to mastering a system's complexity.

Consequently, a formal method has been chosen in order to develop the complex database of a work-flow planning and tracking system for the company Ferk Informatik. The chosen method has been the Vienna Development Method (VDM).

VDM is one of the most widely used formal methods, and it can be applied to the construction of a large variety of systems. It is a model-oriented method, i.e. its formal descriptions (specifications) consist of an explicit model of the system being constructed [16, 11]. Furthermore, its specification language VDM-SL is ISO-standardized [19]. During the course of the design, the model will evolve from abstract to concrete, as algorithm specifications change from implicit to explicit and data types are refined to match the implementation more closely. For example, the first model of a dictionary will likely use a set structure to

hold the set of known words. A more refined model could use a tree structure for the same purpose. The finished model can serve as a definitive reference for the implementer or as the input for a code generator. The IFAD VDM-SL Toolbox [13] has been chosen as the development tool, since it supports our light-weight approach, by providing specification animation checking the internal consistency of the formal model.

However, well-established design methods and tools exist for data base development. Database design in general consists of identifying the entities that are to be represented in the database and their attributes that relate to the problem domain. A telephone database will store persons with their names, address, telephone number(s), but perhaps not their shoe size. An overview of the IDEF1X design method can be found in [3] or online at [14].

This work reports about the needed unification of both formal and informal methods. To minimize the effort of keeping the different models synchronized the approach was designed in a way that the conversion from database design to the formal world can be automated. Then, a tool was written that automatically generates VDM-SL code describing the database model. The method used is VDM, but the approach can be easily transferred into similar model oriented methods such as B [1], Z [21], or RAISE [12].

After this introduction, Section 2 presents the project's domain and an overview of the new design approach. In Section 3 the techniques are described how entity-relationship diagrams (ERD) are translated into formal VDM-SL structures. After that Section 4 shows how VDM supports the enrichment of the initial database model with additional integrity constraints that cannot be expressed by ERDs. Section 5 explores the specification of database queries inside the formal model and how to translate them back into SQL. Finally, Section 6 identifies possible areas of further work, Section 7 gives an overview of related research activities, and Section 8 summarizes experiences gained during the project.

2 Project Description

The scenario for the workflow planning and tracking system was an engine repair facility. The workflow there has the following properties that a planning system has to deal with:

- Planning in advance is difficult; urgent orders that have to be dealt with immediately will invalidate any fixed schedule.
- There are many different kinds and sizes of engines, more than can be entered into the system in advance.
- The time needed to complete the repairing of an engine is difficult to estimate—often, additional work to do is discovered during the repair process.

The system deals with these issues by breaking down the task of repairing an engine into its processes—dismantling the machine, cleaning, re-wiring rotor or stator, etc. For each of the processes a time estimate is calculated based on the type and size of the engine. The system keeps a list of pending processes,

suggesting an agenda and warning when a process is delayed. When work on a process is begun or suspended or when a process is finished, the system adjusts its state accordingly.

The system as planned maintains

- standard tasks and standard processes that describe which types of activity can be performed,
- tasks and processes that are currently active,
- parameters associated with standard tasks and standard processes (engine descriptions, results of measurements etc.),
- values associated with active tasks and processes, and
- resource types associated with standard processes and the corresponding resources needed by a process (e.g. machines, workers).

All in all, the database model consists of about 30 tables at the moment.

Currently, the system is in the early prototype stage. All data structures are kept in an ORACLE database server. Write access is allowed only via stored procedures, so that the database has control over data integrity all of the time. A middle layer communicates with the server via ODBC (open database connectivity) and provides functionality for the client programs that implement a user interface.

2.1 Process Overview

Figure 1 gives an overview over the development process used. From the requirements, an entity-relationship diagram is constructed; this process is described in [14]. The next step is to translate this diagram into VDM-SL. This is covered in Section 3. Once this is completed, additional integrity constraints going beyond the basic relational model can be identified from the requirements and integrated with the basic database model from the previous step—examples are given in Section 4. The constraints identified in this refinement step can be used to check for valid input data, either in the client program(s) or in the database itself using triggers. A trigger is a small piece of code running on the database server that is executed before or after database operations and checks various constraints.

Finally, a formal model is created using the database definition that was derived in the previous steps. Section 5 gives examples how to use the VDM-SL constructs to formulate database queries and how these queries are expressed as SQL statements in program code.

3 SQL Data-Models in VDM-SL

3.1 A Project Example

This section describes a way of transferring entity-relationship diagrams [4, 23] into corresponding VDM-SL constructs. A small, reduced subset of the data model of the workflow planning system shall serve as an example.

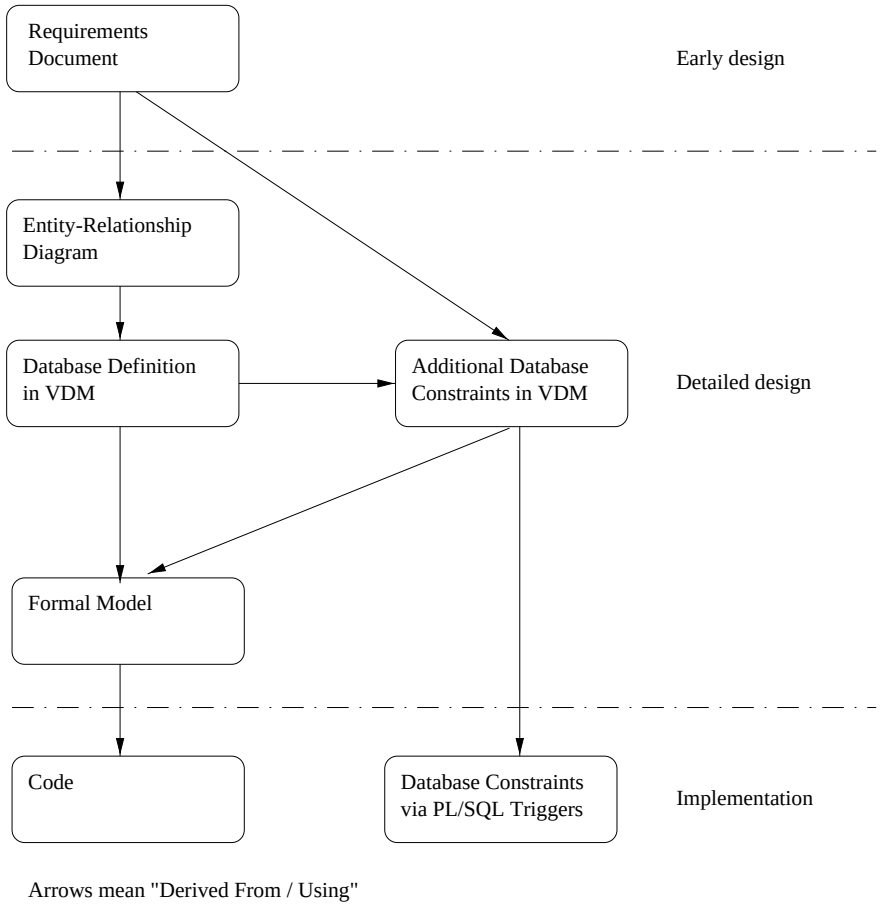
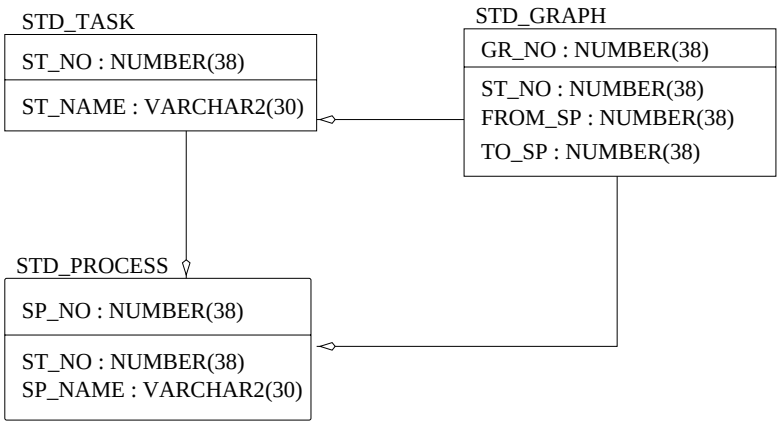


Fig. 1. Overview of the development process.

The workflow system manages pending activities, calculates timing information and resource utilization for tasks that are derived from a set of pre-defined templates. Figure 2 shows the entity-relationship diagram and SQL DDL (data definition language) statements of the subset of the database model that stores these templates. Most of the columns of the tables have been omitted for clarity. The example model consists of three tables: the table STD_PROCESS stores *standard processes*, each describing an atomic activity to be performed. These process templates are grouped into *standard tasks* (table STD_TASK). The processes in a task are ordered as a directed acyclic *graph* that is stored in table STD_GRAPH in the form of pairs of process numbers that form a before-after-relationship (see Figure 3 on page 10 for an example of this table).




```
CREATE TABLE STD_TASK (
  ST_NO NUMBER(38) PRIMARY KEY,
  ST_NAME VARCHAR2(30) NOT NULL);

CREATE TABLE STD_PROCESS (
  SP_NO NUMBER(38) PRIMARY KEY,
  ST_NO NUMBER(38) NOT NULL REFERENCES STD_TASK(ST_NO),
  SP_NAME VARCHAR2(30));

CREATE TABLE STD_GRAPH (
  GR_NO NUMBER(38) PRIMARY KEY,
  ST_NO NUMBER(38) NOT NULL REFERENCES STD_TASK(ST_NO),
  FROM_SP NUMBER(38) NOT NULL REFERENCES STD_PROCESS(SP_NO),
  TO_SP NUMBER(38) NOT NULL REFERENCES STD_PROCESS(SP_NO));
```

Fig. 2. ER diagram and SQL DDL statements for the sample database.

3.2 Modeling SQL Datatypes

The first step is to find representations of the SQL datatypes in VDM-SL. Every datatype in the database is modeled by a corresponding VDM-SL datatype—see Table 1 for a summary. The table lists the datatypes from the Oracle database product; other databases have slightly varying names for some of these datatypes. The VDM part is given in the ASCII notation that is used by IFAD's toolbox products and also when using mathematical notation would be impractical, for example when using VDM expressions as comments in source code.

Character Datatypes The datatypes CHAR and VARCHAR2 store character information. They differ in the way storage is allocated: in a column of type CHAR(n), storage for n characters is allocated for every row of data, shorter values are blank-padded to the length of the column. Columns of type VARCHAR2(n) can store up to n characters, but the space is not allocated until

Table 1. SQL datatypes and their VDM-SL counterparts (in ASCII notation).

SQL	*VDM-SL*

CHAR (n)	`Char_`n `= seq of char;`
	`Char_`n `= seq of char`
	`inv c_`n `== len c_`n `= `n`;`

VARCHAR2 (n)	`Varchar2_`n `= seq of char;`
	`Varchar2_`n `= seq of char`
	`inv vc_`n `== len vc_`n `<= `n`;`

NUMBER	`Number = real;`

NUMBER (p)	`Number_`p `= int;`
	`Number_`p `= int`
	`inv n_`p `== n_`p `< `10^p

NUMBER (p,s)	`Number_`p`_`s `= real;`
	`Number_`p`_`s `= real`
	`inv n_`p`_`s `== n_`p`_`s `< 10**`p

DATE	`Date = token;`
	`Date = int;`
	`Date = real;`

needed. This leads to slightly different comparison semantics when values have different length. Specifically, `'a␣'='a'` when both values are of type CHAR, `'a␣'>'a'` otherwise.

The datatypes CHAR and VARCHAR2 can be modelled as sequences of characters. When a greater level of detail is needed, the length-invariants can be added to the VDM-SL datatypes.

Numeric Datatypes The basic datatype NUMBER corresponds to the VDM-SL datatype `real`. Additionally, a *precision* and *scale* can be added to NUMBER, specifying the number of digits to the left and the right of the decimal point. For a high-level model, these will not need to be handled, except that numbers with a scale of 0 can be modelled as $\mathbb{Z}$ instead of $\mathbb{R}$. Of course, constraints can be added to catch invalid test data in the formal model. Table 1 shows the invariants for constraining the precision of values.

The DATE Type The modelling of the DATE type is not as straightforward as string and numerical types. The main difficulty is the complicated date arithmetic that is implemented in SQL, but not in VDM-SL. So we rely on mapping only some properties of dates.

One approach that borrows from the Unix type `time_t` is to store dates as integers, counting the seconds from some fixed point in time. The other one is to use floating-point values, the part before the comma describing the day (again counting from some fixed date) and the fractional part describing the time (0 meaning midnight, 0.5 midday). Both approaches allow the comparison of dates and some simple date arithmetics (time differences, adding times). There

exist well-known algorithms for more advanced calculations such as day-of-week, add-month, etc. that can be implemented as functions if needed in the abstract model.

3.3 Modelling the Table Structure

The structure of a table is defined by the name and datatype of each of its columns. The table structure is modelled as a VDM record type. The record has the name of the table with a suffix of *-Type*. The fields of the record are equivalent to data items in the column of a relational table and use the types defined in Section 3.2.

Thus, the first step of modelling the example database looks as follows:

types

$$Number\text{-}38 = \mathbb{Z};$$

$$Varchar2\text{-}30 = \mathsf{char}^*;$$

$$\begin{aligned} Std\text{-}Task\text{-}Type :: & \; ST\text{-}NO : Number\text{-}38 \\ & ST\text{-}NAME : Varchar2\text{-}30 \end{aligned}$$

$$\begin{aligned} Std\text{-}Process\text{-}Type :: & \; SP\text{-}NO : Number\text{-}38 \\ & ST\text{-}NO : Number\text{-}38 \\ & SP\text{-}NAME : Varchar2\text{-}30 \end{aligned}$$

$$\begin{aligned} Std\text{-}Graph\text{-}Type :: & \; GR\text{-}NO : Number\text{-}38 \\ & ST\text{-}NO : Number\text{-}38 \\ & FROM\text{-}SP : Number\text{-}38 \\ & TO\text{-}SP : Number\text{-}38 \end{aligned}$$

Modelling `NULL` Values The VDM specification language's `nil` value has the same purpose as the SQL NULL Value: both are used to represent unknown or undefined values. The next paragraph shows what the table definition would look like if unnamed processes were allowed in the data model, the square brackets ([ ]) denoting an optional type:

$$\begin{aligned} Std\text{-}Process\text{-}Type\text{-}2 :: & \; SP\text{-}NO : Number\text{-}38 \\ & ST\text{-}NO : Number\text{-}38 \\ & SP\text{-}NAME : [Varchar2\text{-}30] \end{aligned}$$

3.4 Modelling the Tables

A relational database table stores data as an unordered set of rows, each containing the same sequence of typed atomic data. A table can be modelled in VDM-SL as a set of records[1].

The tables of the example database are modelled in the following way, using the structure types defined in Section 3.3:

$$Std\text{-}Task\text{-}Table = Std\text{-}Task\text{-}Type\text{-set}$$

$$\begin{array}{l}\textsf{inv } tt \triangleq \\ \quad \forall\, t1 \in tt, t2 \in tt \cdot \\ \qquad t1 \neq t2 \;\Rightarrow\; t1.ST\text{-}NO \neq t2.ST\text{-}NO;\end{array}$$

$$Std\text{-}Process\text{-}Table = Std\text{-}Process\text{-}Type\text{-set}$$

$$\begin{array}{l}\textsf{inv } pt \triangleq \\ \quad \forall\, p1 \in pt, p2 \in pt \cdot \\ \qquad p1 \neq p2 \;\Rightarrow\; p1.SP\text{-}NO \neq p2.SP\text{-}NO;\end{array}$$

$$Std\text{-}Graph\text{-}Table = Std\text{-}Graph\text{-}Type\text{-set}$$

$$\begin{array}{l}\textsf{inv } gt \triangleq \\ \quad \forall\, g1 \in gt, g2 \in gt \cdot \\ \qquad g1 \neq g2 \;\Rightarrow\; g1.GR\text{-}NO \neq g2.GR\text{-}NO;\end{array}$$

Each element in the set represents a column of the table. The invariant guarantees the uniqueness of the primary key. In the case of a composite primary key consisting of more than one column, the invariant must ensure that at least one element is different for every pair of records in the set.

3.5 Modelling the Database

Now all tables in the database are modelled, including datatypes, NULL values and primary key uniqueness. Modelling the database means putting it all together and ensuring that foreign key references point to an existing record in another table.

The following VDM-SL code defines the sample database.

$$\begin{array}{l}DB\text{-}1 :: STD\text{-}TASK : Std\text{-}Task\text{-}Table \\ \qquad\quad STD\text{-}PROCESS : Std\text{-}Process\text{-}Table \\ \qquad\quad STD\text{-}GRAPH : Std\text{-}Graph\text{-}Table\end{array}$$

[1] Another representation that was tried but abandoned very soon was to use a mapping from key values to the non-key data. This approach proved to be impractical, because database queries can consist of arbitrary logical expressions over the values of any number of columns and can return more than one value, making a set-based approach the natural choice.

$$
\begin{array}{l}
\mathsf{inv}\ \mathsf{mk}\text{-}DB\text{-}1\,(tt, pt, gt) \triangleq \\
\quad \forall\, p \in pt \cdot \\
\qquad (\exists\, t \in tt \cdot p.ST\text{-}NO = t.ST\text{-}NO) \wedge \\
\qquad \forall\, g \in gt \cdot \\
\qquad\quad (\exists\, t \in tt \cdot g.ST\text{-}NO = t.ST\text{-}NO) \wedge \\
\qquad\quad (\exists\, p1 \in pt \cdot g.FROM\text{-}SP = p1.SP\text{-}NO) \wedge \\
\qquad\quad (\exists\, p2 \in pt \cdot g.TO\text{-}SP = p2.SP\text{-}NO)
\end{array}
$$

Most database engines ensure the validity of foreign key references at all times; so does the invariant of the record type storing the model of the database.

In a different database scenario, foreign key columns in a table could be allowed the value NULL (`nil` in the model), meaning that there is no corresponding row of data in the referred table. In this case, the invariant has to be formulated a little different—if the foreign key column has a value different from NULL, there must exist a row of data in the other table that satisfies the foreign key condition. For example, if standard processes were allowed to exist without a containing standard task, the invariant would look this way:

$$
\begin{array}{l}
\mathsf{inv}\ \mathsf{mk}\text{-}DB\text{-}1\,(tt, pt, gt) \triangleq \\
\quad \forall\, p \in pt \cdot \\
\qquad (\exists\, t \in tt \cdot p \neq \mathsf{nil} \ \Rightarrow\ p.ST\text{-}NO = t.ST\text{-}NO) \wedge \\
\qquad \ldots
\end{array}
$$

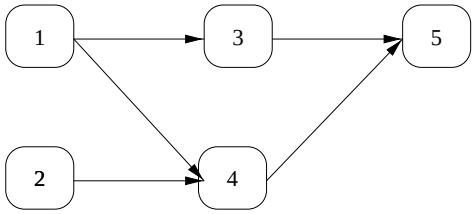
4 Enriching the Model

The invariant of the database model as formulated in Section 3 takes care of primary and foreign keys and of datatype constraints. Additional constraints deal with application-specific properties of data and are identified from the requirements. This section describes how these constraints are added to the formal model of the database.

Consider how the table STD_GRAPH stores a graph. Figure 3 shows a simple graph and corresponding entries in the table STD_GRAPH. The column for the tables primary key contains arbitrary unique values for each entry, the column ST_NO refers to standard task number 42 in table STD_TASK. The other two columns describe how standard processes 1 through 5 (presumably belonging to standard task 42) are related to each other.

For the database to be consistent, each standard process that is contained in the graph of a specific standard task has to actually belong to that standard task. This cannot be tested by simple foreign key constraints, since the condition involves not only the existence of data in the standard process table but also some properties of this data. The next paragraph shows how this invariant can be expressed in VDM-SL. The example uses the "iota" expression that extracts a value with some unique property from a set—this is safe because the type-invariant of the database guarantees the uniqueness of the primary key.

functions



STD_GRAPH			
GR_NO	ST_NO	FROM_SP	TO_SP
15	42	1	3
16	42	1	4
17	42	2	4
18	42	3	5
19	42	4	5

Fig. 3. An example graph and the corresponding database entries.

$$
\begin{array}{l}
\mathit{graph\text{-}references\text{-}consistent} : \mathit{DB}\text{-}1 \to \mathbb{B} \\
\mathit{graph\text{-}references\text{-}consistent}\ (\mathsf{mk}\text{-}\mathit{DB}\text{-}1\ (\text{-}, \mathit{processes}, \mathit{graph})) \triangleq \\
\quad \forall\, g \in \mathit{graph} \cdot \\
\qquad \mathsf{let}\ \mathit{p\text{-}pre} = \iota\, p \in \mathit{processes} \cdot p.\mathit{SP\text{-}NO} = g.\mathit{FROM\text{-}SP}, \\
\qquad\quad \mathit{p\text{-}post} = \iota\, p \in \mathit{processes} \cdot p.\mathit{SP\text{-}NO} = g.\mathit{TO\text{-}SP}\ \mathsf{in} \\
\qquad \mathit{p\text{-}pre}.\mathit{ST\text{-}NO} = g.\mathit{ST\text{-}NO} \wedge \\
\qquad \mathit{p\text{-}post}.\mathit{ST\text{-}NO} = g.\mathit{ST\text{-}NO};
\end{array}
$$

This condition can be enforced in the implementation with database triggers that check the validity of entries after each insert and update operation of table STD_GRAPH.

To continue with a more difficult example, we want to specify an additional invariant stating that the graph must not contain cycles.

$$
\begin{array}{l}
\mathit{cycles\text{-}in\text{-}graph} : \mathit{Std\text{-}Graph\text{-}Table} \times \mathit{Number}\text{-}38 \to \mathbb{B} \\
\mathit{cycles\text{-}in\text{-}graph}\ (\mathit{graph}, \mathit{task\text{-}no\text{-}in}) \triangleq \\
\quad \mathsf{let}\ \mathit{this\text{-}graph} = \{g \mid g \in \mathit{graph} \cdot g.\mathit{ST\text{-}NO} = \mathit{task\text{-}no\text{-}in}\}\ \mathsf{in} \\
\quad \exists\, \mathit{node\text{-}no} \in \mathit{all\text{-}nodes}\ (\mathit{this\text{-}graph}) \cdot \\
\qquad \mathit{node\text{-}no} \in \mathit{all\text{-}successors}\ (\mathit{this\text{-}graph}, \mathit{node\text{-}no}, \{\});
\end{array}
$$

$$
\begin{array}{l}
\mathit{all\text{-}nodes} : \mathit{Std\text{-}Graph\text{-}Type}\text{-}\mathsf{set} \to \mathit{Number}\text{-}38\text{-}\mathsf{set} \\
\mathit{all\text{-}nodes}\ (\mathit{graph}) \triangleq \\
\quad \{g.\mathit{FROM\text{-}SP} \mid g \in \mathit{graph}\} \cup \{g.\mathit{TO\text{-}SP} \mid g \in \mathit{graph}\};
\end{array}
$$

$$
\begin{array}{l}
all\text{-}successors : Std\text{-}Graph\text{-}Type\text{-set} \times Number\text{-}38 \times Number\text{-}38\text{-set} \to \\
\qquad Number\text{-}38\text{-set} \\
all\text{-}successors\ (graph, node\text{-}no\text{-}in, visited\text{-}nodes) \triangleq \\
\ \mathsf{let}\ direct\text{-}successor\text{-}set = direct\text{-}successors\ (graph, node\text{-}no\text{-}in)\ \mathsf{in} \\
\ -\ \text{base cases} \\
\ \mathsf{if}\ direct\text{-}successor\text{-}set = \{\} \lor \\
\quad node\text{-}no\text{-}in \in visited\text{-}nodes \\
\ \mathsf{then}\ \{\} \\
\qquad -\ \text{recursive case: merge direct successors with their successors} \\
\ \mathsf{else}\ \mathsf{let}\ successor\text{-}successor\text{-}set = \\
\qquad \bigcup \{ all\text{-}successors\ (graph, node\text{-}no, \\
\qquad\qquad visited\text{-}nodes \cup \\
\qquad\qquad \{node\text{-}no\text{-}in\}) \mid \\
\qquad\quad node\text{-}no \in direct\text{-}successor\text{-}set \}\ \mathsf{in} \\
\quad direct\text{-}successor\text{-}set \cup successor\text{-}successor\text{-}set;
\end{array}
$$

$$
\begin{array}{l}
direct\text{-}successors : Std\text{-}Graph\text{-}Type\text{-set} \times Number\text{-}38 \to \\
\qquad Number\text{-}38\text{-set} \\
direct\text{-}successors\ (graph, node\text{-}no\text{-}in) \triangleq \\
\ \{g.TO\text{-}SP \mid g \in graph \cdot g.FROM\text{-}SP = node\text{-}no\text{-}in\}
\end{array}
$$

The function *cycles-in-graph* takes a graph table and a task number and checks if at least one of the nodes of the graph is contained in the set of its successors—that is, whether it can be reached by a path originating from itself, which is a property of a directed graph with cycles. The auxiliary function *all-successors* recursively collects all nodes that are reachable from a given node.

With these functions in place, we can extend the database definition with stronger data-invariants.

types

$$
\begin{array}{l}
DB = DB\text{-}1 \\
\\
\mathsf{inv}\ db \triangleq \\
\quad \mathsf{inv}\text{-}DB\text{-}1\ (db) \land wf\text{-}db\ (db)
\end{array}
$$

functions

$$
\begin{array}{l}
wf\text{-}db : DB \to \mathbb{B} \\
wf\text{-}db\ (db) \triangleq \\
\ graph\text{-}references\text{-}consistent\ (db) \land \\
\ \forall\, task \in db.STD\text{-}TASK \cdot \\
\quad \neg\ cycles\text{-}in\text{-}graph\ (db.STD\text{-}GRAPH, task.ST\text{-}NO)
\end{array}
$$

The type DB can now be used to model the behavior of the database.

5 Modelling Database Operations

This section is different from Section 3 insofar as the transformation process from SQL to VDM described there can be automated, whereas the VDM expressions shown in this section are part of a larger model and cannot be extracted and translated into SQL statements automatically. But a series of query examples will show that the modelling approach using sets leads to VDM-SL expressions that are semantically quite similar to SQL statements, and so the insights gained by formulation of algorithms with complex queries in the formal world can be used to code these algorithms with confidence.

For the examples, some values are needed:

$$db = \mathsf{mk\text{-}}DB\,(\{\},\{\},\{\});$$

$$TaskNo : Number\text{-}38 = 42;$$

$$ProcessNo : Number\text{-}38 = 17;$$

First a simple query is considered: in order to start a new activity, the user has to specify which template (from the table STD_TASK) to use. So a set of all templates is needed. In the formal model, the following VDM-SL set comprehension expression provides such a set.

$$q1 = \{n \mid n \in db.STD\text{-}TASK\};$$

This set can be extracted from the database with the following SQL statement.

```
SELECT * FROM STD_TASK;
```

The calculation of process durations needs a list of the identifiers of all standard processes belonging to a specific task.

$$q2 = \{p.SP\text{-}NO \mid p \in db.STD\text{-}PROCESS \cdot p.ST\text{-}NO = TaskNo\};$$

Again, in SQL:

```
SELECT SP_NO FROM STD_PROCESS WHERE ST_NO = TaskNo;
```

Joins of two or more tables are also possible. Here is a set consisting of the names of all standard processes and their parent task:

$$\begin{aligned} q3 = \{&\mathsf{mk\text{-}}(t.ST\text{-}NAME, p.SP\text{-}NAME) \mid \\ &p \in db.STD\text{-}PROCESS, t \in db.STD\text{-}TASK \cdot \\ &\quad p.ST\text{-}NO = t.ST\text{-}NO\}; \end{aligned}$$

Again, in SQL:

```
SELECT ST_NAME, SP_NAME FROM STD_PROCESS, STD_TASK
WHERE STD_PROCESS.ST_NO = STD_TASK.ST_NO;
```

The following extracts the set of graph entries for a specific task from the table STD_GRAPH—a similar expression is used in the function *cycles-in-graph* in Section 4.

$$q4 = \{\mathsf{mk\text{-}}(g.FROM\text{-}SP, g.TO\text{-}SP) \mid g \in db.STD\text{-}GRAPH \cdot g.ST\text{-}NO = TaskNo\};$$

```
SELECT FROM_SP, TO_SP FROM STD_GRAPH
  WHERE ST_NO = TaskNo;
```

The last two examples deal with traversing the graph of a task, either in depth-first-search or in breadth-first-search. First it is necessary to find all starting points in the graph. A starting point is a node that has successors but no predecessors. This can be expressed as the set of all predecessors minus the set of all successors:

$$q5 = \{g.FROM\text{-}SP \mid g \in db.STD\text{-}GRAPH \cdot g.ST\text{-}NO = TaskNo\} \setminus \{g.TO\text{-}SP \mid g \in db.STD\text{-}GRAPH \cdot g.ST\text{-}NO = TaskNo\};$$

This non-trivial query was first modelled in VDM-SL and was then translated into SQL as two select statements.

```
SELECT DISTINCT FROM_SP FROM STD_GRAPH
  WHERE ST_NO = TaskNo
MINUS
SELECT  TO_SP FROM STD_GRAPH
  WHERE ST_NO = TaskNo;
```

Finally, the successors of a node are found this way:

$$q6 = \{g.TO\text{-}SP \mid g \in db.STD\text{-}GRAPH \cdot g.ST\text{-}NO = TaskNo \wedge g.FROM\text{-}SP = ProcessNo\}$$

```
SELECT TO_SP FROM STD_GRAPH
  WHERE ST_NO = TaskNo AND FROM_SP = ProcessNo;
```

This concludes the examination of database operations in the formal model.

6 Further Work

6.1 Program Generation

A field that has so far not been explored is the translation back from VDM-SL to SQL. As shown in Section 5, the structure of some VDM-SL expressions is quite similar to SQL statements. It would be interesting to examine the possibilities of translating these expressions back to SQL and incorporating these changes automatically into code.

6.2 Derived Consistency Checks

Datatype and foreign key checking do not guarantee a consistent database. Often, further consistency checks have to be added, especially in client-server applications where the client programs cannot be trusted to supply well-formed data. Modern database engines provide *triggers*, small programs that can perform various checks before and after insert, update and delete operations. In a formal model, invariants, pre- and postconditions have similar duties. During formal design work, various assertions and invariants will be discovered that can be translated back into code to increase the robustness of the program. A way is needed to reliably identify these constructs in the formal model and aid the programmer in implementing them in the production code.

6.3 Automated Integration with Conventional Methods

The translation from SQL to VDM-SL enables the designer to work with the same conceptual model during dabase design and formal modelling. The automated construction of part of the formal model from the informal model encourages the designer to spend the effort to keep both models synchronized. Conceivably, other informal design methods could be integrated in a similar way. For example, Larsen et al. [18] have shown how to integrate VDM-SL and data flow diagrams. Their work shows that data flow diagrams structure the functional part of a VDM model similar to how entity-relationship diagrams help structure the data types of the formal model. Furthermore, entity-relationship diagrams and data flow diagrams integrate very well, too. A method and tool to integrate these diagramming and modelling functionalities would be very expressive (and impressive) indeed.

7 Related Work

Research on the formal aspects of databases was popular in the early 80's. Dines Bjørner [2, 5] describes formal models of relational, hierarchical and network-model databases. Neuhold et al. [8] formalize the inner workings of a database system, describing external, conceptual and inner level and their interplay. Both of these works describe database management systems on a very high level of detail and generality that is useful for the development of a DBMS, but not necessarily for modelling its behavior in the context of an application program.

Consequently, in later works many aspects of a database management system, such as the ability to store persistent data and the existence of a query language operating on stored data, are taken for granted and not explicitly modelled. As database technologies matured, the focus shifted to other areas of research, such as program generation and integration of formal methods in an existing development process. Günther et al. [22] describe the derivation of executable database programs from formal specifications, using stepwise refinement of a B [1] specification of database structure and operations until the constructs can be mapped

onto statements in the DBPL language. It remains to be explored if and how we can adapt this approach into our development process. De Barros' work [6] utilizes the Z method [21] for database modelling and program generation. The database model resulting from the transformation process outlined in our paper is similar to de Barros' model, although our focus is 180 degrees reversed: specification generation instead of program generation.

8 Concluding Remarks

The approach presented in this paper unites formal methods and conventional methods of database design. The following benefits were encountered during development work:

- Duplication of work is avoided; having defined entities and identified constraints for the database, the designer can use these items in the VDM model.
- Using the same structure in the model as in the finished database allows reasoning over the database design, exposing errors early in the design process.
- Tools exist for graphical database modelling that translate the models into SQL scripts that generate the database structure. It is quite easy to write a parser that translates SQL table definition statements into VDM-SL code.

Of course, every method has to be approached with caution. Some points that have to be observed when using the combination of VDM-SL and database modelling are:

- A certain level of expertise using formal methods is needed by the designer to resist over-specifying the formal model, especially during early design phases. The concrete data structures provided by the database model tempt the designer to "program in VDM-SL".
- The database design will impose its structure upon the formal model. This is not a disadvantage in itself, but it can lead to neglecting alternative design patterns if the data structures are perceived as "cast in stone".

The best path to success is probably to first design a very abstract specification without regarding database structure at all, then to identify entities and relationships and design the database. This work can then be used to write a detailed formal specification.

During development, the expressive power of formal methods and VDM in particular was very much appreciated. The expressive power of the formal notation was a great help when thinking about database operations in general and formulating complicated queries. The automated generation of part of the model in conjunction with the excellent type-checking capabilities of IFADs VDM-SL toolbox made it possible to keep the formal model up-to-date with the database model without any trouble. We are looking forward to working with VDM in our next projects as well.

References

[1] J.-R. Abrial. *The B-Book, Assigning programs to meanings.* Cambridge University Press, 1996. ISBN 0521 49619 5(hardback).

[2] D. Bjørner. Formalization of data base models. *Abstract Software Specifications*, pages 144–215, 1980.

[3] Thomas A. Bruce. *Designing Quality Databases with IDEF1X Information Models.* Dorset House, 1992. ISBN 0-932633-18-8.

[4] P.P. Chen. The entity-relationship model, toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[5] D.Bjørner. Formalization of database management systems. In *Formal Specification and Software Development*, chapter 12, pages 379–442. Prentice-Hall, 1982.

[6] Roberto S.M. de Barros. Deriving relational database programs from formal specifications. In M. Bertran M. Naftalin, T. Denvir, editor, *FME'94: Industrial Benefit of Formal Methods*, pages 703–723. Springer-Verlag, October 1994.

[7] Martin Dunstan, Tom Kelsey, Steve Linton, and Ursula Martin. Lightweight formal methods for computer algebra systems. In *ISSAC '98: International Symposium on Symbolic and Algebraic Computation*, University of Rostock, Germany, August 1998.

[8] Ths. Olnhoff E. Neuhold. Building data base management systems through formal specifications. In *Lecture Notes in Computer Science*, pages 169–209. Springer-Verlag, LNCS Vol. 107, 1981.

[9] S. M. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering, Special Issue on Formal Methods in Software Practice*, 24(1), 1998. (Technical Report #NASA-IVV-97-015).

[10] Steve Easterbrook, Jack Callahan, and Zhong Zhang. FLAVORS: Formal, lightweight approaches to validation of requirements specifications. FLAVOR is an ongoing project at NASAs Software Research Labratory (SRL), `http://research.ivv.nasa.gov/projects/FLAVORS/`.

[11] John Fitzgerald and Peter Gorm Larsen. *Modelling Sytems, Practical Tools and Techniques.* Cambridge University Press, 1998.

[12] Chris George et al. *The Raise Development Method.* The BCS Practitioner Series. Prentice Hall, 1995.

[13] IFAD. IFAD's products. URL: http://www.ifad.dk/Products/products.htm.

[14] Knowledge Based Systems Inc. IDEF1X overview. WWW at URL http://www.idef.com/overviews/idef1x.htm.

[15] Daniel Jackson and Jeannette Wing. Formal methods light: Lightweight formal methods. *IEEE Computer*, 29(4):21–22, April 1996.

[16] Cliff B. Jones. *Systematic Software Development Using VDM.* Series in Computer Science. Prentice-Hall, 2nd. edition, 1990.

[17] Cliff B. Jones. Formal methods light: A rigorous approach to formal methods. *IEEE Computer*, 29(4):20–21, April 1996.

[18] Peter Gorm Larsen, Jan van Katwijk, Nico Plat, Kees Pronk, and Hans Toetenel. Towards an integrated combination of SA and VDM. In *Structured Analysis and Formal Methods*, June 1991.

[19] P.G. Larsen and B. S. Hansen and H. Brunn N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin et. al. Information Technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, ISO/IEC 13817-1, December 1996.

[20] Sotiris C. Skevoulis. A Light-Weight Approach to Formal Methods. DePaul University Software Engineering Symposium, Spring 1997, `http://saturn.cs.depaul.edu/~se/SES/ses.htm`.

[21] J. M. Spivey. *The Z Notation*. Series in Computer Science. Prentice-Hall, 1989.

[22] & Ingrid Wetzel Thomas Günther, Klaus-Dieter Schewe. On the derivation of executable database programs from formal specifications. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 351–366. Formal Methods Europe, Springer-Verlag, April 1993. Lecture Notes in Computer Science 670.

[23] Edward Yourdon. *Modern Structured Analysis*. Yourdon Press Computing Series. Prentice Hall, 1989. ISBN 0-13-598624-9.