Deseo Meeting Scheduler: Towards Automated Verification of Database Integrity Constraints

Ralph-Johan Back[1], Mikołaj Olszewski[1, 2] and Damián Soriano[1, 3]

[1] Åbo Akademi University, Joukahaisenkatu 3-5A, 20520 Turku, Finland
[2] TUCS, Turku Centre for Computer Science, Joukahaisenkatu 3-5B, 20520 Turku, Finland
[3] National University of Rosario, Faculty of Exact Sciences, Engineering and Surveying, Pellegrini 250, (2000) Rosario, Argentina
{backrj, molszews, dsoriano}@abo.fi

Abstract. Amongst the great diversity and quantity of software being developed nowadays, database applications form a large group of widely used software. They are considered to be of high importance, not only due to the general popularity they already gained, but also to the fact that they are applicable to various, also critical, domains. This in turn brings up the issue of correctness of the developed systems and need for the cautious modelling and verification. In this paper we present a method to formally verify a functionality of database application. We illustrate the technique by modelling in PVS a case study application that has a number of non-trivial behavioural and structural properties. The model we construct forms a theory, in which desired properties of the system can be stated and proven.

Keywords: formal methods, database, verification, modelling, PVS

1 Introduction

Database applications form a significant part of all software being produced today, especially when it comes to web-applications. Considering Web 2.0 and extensive use of web-based software that is supported by relational databases, it is essential to ensure the behavioural correctness of such software.

There are some methods of verifying correctness of a database application. For relational database schemas one could use object-relational mappings, namely Active Record pattern [1]. By using this approach database tables are translated into classes with objects (instances) corresponding to rows. The correctness of such object-oriented system can be proven correct using techniques developed for this type of software [2]. The constraints on the business logic, however, must then be placed in the client application code and not on the database side.

Constructing the software in a correct manner is another alternative. The *correct by construction* concept introduced by Dijkstra [3], is essential for safety critical systems. The programs are properly constructed by adding the functionality incrementally and at the same time proving the correctness. The efforts and costs of building systems this way are balanced by their importance to the tasks they perform. For other types of software the issues of correctness are often not considered, mainly because of the lack of any popular, lightweight method that can be included in the design process. The research is focused on providing such method to the community.

Programming with the use of invariants, described in different forms by van Emden [4], Reynolds [5] or Back [6] [7], has certainly the potential to become successful, since it is supported by the SOCOS programming platform [8]. The properties of the system to be constructed – integrity constraints in case of database applications – are used to form invariants that must be satisfied. The invariants are then proven based on the concrete, but formal, specification. However even the simplest programs can result in theorems too difficult or too time consuming to be proven manually, therefore automated theorem proving systems, like PVS Verification System [9], are used.

The specification is used to construct the model of the system. In case of databases entity relationship diagrams (ERD [10]) are often used to present such static model. Recently there has been some research on the verification of those diagrams in PVS [11]. The researched method allows verifying the correctness of mappings on different levels of abstraction. However, it is not possible to model behavioural properties of the system and prove that they preserve system invariants.

In our paper we address the above mentioned issues. By examining a case study [12] we show a technique that allows formal verification of both the database structure and behaviour. That is, provided that certain conditions are met, the database will preserve its invariants. Our method in this area enhances the work of Sheard and Stemple [13], where integrity constraints are treated as database transaction invariants. However, in our work we focused more on relational databases instead of object-oriented ones. We also did not cover transactions, rather concentrating our efforts on the invariants as the integrity constraints and possible operations that can be performed on the database schema. The method we developed could be fully automated and integrated into database design process, thus making it transparent and easy to use.

The rest of the paper is organised as follows. The case study is presented in the next section. Subsequently, we describe a formal model of the project's database, covering the structure, operations and triggers in separate subsections. We exemplify the verification technique with the table from the database schema of the case study. We also briefly describe the verification effort for the case study. We end the paper with some general conclusions and our future work directions.

Throughout the paper we use the terminology that is specific for relational databases developers community. By *database* (or *database schema*) we mean an organised, arranged collection of named *tables*, each of which contains named *fields* or *columns* and stores *rows*. Each row (*record*) contains a number of *values* that correspond to the columns of the table to which the row belongs. In addition, every row contains an *identifier* that distinguishes it from other rows of the same table. The *reference* is the link between two tables, where the *referencing field* of one table contains a value that is a valid identifier of the *referenced table*. The naming is influenced by SQL (Structured Query Language, descendant of SEQUEL [14]), language used to retrieval and management of data in relational databases. Despite being different from the one proposed by Codd [15] it is commonly used in the community. [16]

2 The case study

Deseo Meeting Scheduler [12] is an application that allows its users to schedule a meeting. Every registered user of Deseo Meeting Scheduler can set up a meeting by giving a number of proposed time slots for it (proposals). By creating the meeting the user becomes a host of the meeting. The host can issue invitations to other users for them to join the meeting. The invitation can be cancelled either by the invited user or by the host at any time. The invited user can give positive or negative answer to any of the proposals. The host can confirm the

proposal only if every invited user has given positive answer to it. The meeting can have at most one confirmed proposal that refers to it. The users are not allowed to agree to a proposal that conflicts (overlaps) with any proposal of confirmed meeting they are invited to. The users access the system by using a client application that connects to the database.

From the initial specification we obtain the constraints and dependencies described in Table 1. The table lists only the entities and their properties that are related to the scheduling of the meetings.

Entity name	Dependencies	Constraints
`Users`	*None*	None
`Meetings`	*Hosted by the user*	There is only one host for each meeting.
`Proposals`	*Belongs to the meeting*	There cannot be two different proposals that start at the same time and refer to the same meeting. Every proposal upon creation must be unconfirmed. Proposals with a past date cannot be created. The proposal can be set to confirmed only if all invited users have agreed on it and it does not collide with host's other meetings.
`Invitations`	*Issued to the user for the meeting*	The host cannot be invited to the meeting. The user cannot have two invitations to the same meeting at any given time. The invitation can be cancelled (deleted) at any time. Once issued, the invitation cannot be modified (only cancelled).
`Answers`	*Given by invited user to the proposal*	Only invited users can give answers. The user cannot answer twice, but can change the answer once given. The user cannot agree to a proposal that collides with already confirmed meeting.

Table 1. Summary of initial specification.

The UML database model is presented in Figure 1. It follows naturally from the contents of Table 1. The constraints mentioned in the table define the core behaviour of the software. They can be implemented either on the database side or on the client application side. This decision is of high importance to the development process and should be made as soon as possible, but in a way that reduces the probability of a change to a minimum. [17].

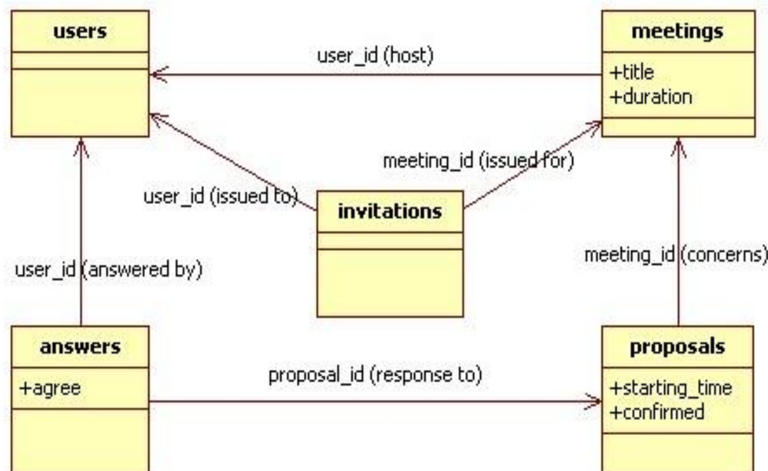

Figure 1. Database model of the Deseo Meeting Scheduler.

When implementing the integrity constraints on the client application side the resulting database is very fast and effective, as it is only used to store the data which correctness is ensured by the connecting clients. On the other hand, the implementation of the behaviour on the database side reduces the role of the clients to reading and writing data. The correctness is provided by the database itself – the data that is incorrect is rejected. This solution provides unified view of the data for client applications and facilitates the introduction of changes to functionality of the software, since there is only one place the changes must be made. The typical solution is often a balance of the above mentioned extremes – some part of the business logic is present on the database side, while the clients perform additional data checks on their own. In our case study we decided to include the core functionality (meeting scheduling) in the database itself by constructing advanced constraints and triggered functions. User authorisation mechanisms and basic type checking in our system is done by the client applications. This separation allows us to remove unnecessary code from the database, minimise the network traffic and to focus the development on the database so that it provides the functionality in the optimal way.

Because of the extensive use of triggered functions in our case study, we decided to use PostgreSQL Database Management System [18]. We also did not decide on the platform of the client applications, focusing our work on the database schema and proving the functionality it gives. This allows us to ensure that all client applications, regardless of their implementation, will provide correct functionality within the boundaries of the business logic that is coded in the database.

3 Formal model of the database

In this section we illustrate our formalisation technique with an example from the case study. In the rest of the section we operate on the invitations table due to a number of properties it has. Firstly, it references two other tables: users and meetings. Secondly, records of this table are read-only, i.e. once added they can be removed, but cannot be modified. Moreover, the table plays an important role in preserving an invariant of the system, as there cannot be any invitation that is issued to the host of the meeting. And finally, it has a non-trivial behavioural dependency, since removing the invitation results in removing the answers related to the meeting as well.

3.1. Structure of the database

By structure of the database we mean a collection of tables. In other words, it is the model of the data. As mentioned previously, we use the invitations table as the example to our formalisation technique. The SQL code that constructs the invitations table is presented in Listing 1.

```
1.  CREATE TABLE invitations(
2.   id integer primary key,
3.   user_id integer REFERENCES users(id) on delete cascade not null,
4.   meeting_id integer REFERENCES meetings(id) on delete cascade not null
5.  );
```

Listing 1. SQL code that constructs invitations table.

The primary key (defined in line 2) identifies uniquely each row of the table. In most of the database applications the key field is named id, can hold natural or integer numbers and

the value of the field does not change once set for a record. However, in theory, there are no restrictions on the type of this field – it can store any type of data, as long as there are no two different rows with the same keys; it is also possible to change the value of the primary key. In addition in relational databases the records are stored unordered, as sets.

The invitations table also references users (line 3) and meetings (line 4), as the invitation must contain the information about the user it is issued to and the meeting it concerns. The `on cascade delete` directive ensures that the invitation will be deleted automatically when the referenced row (either in users or meetings table) gets removed. The `not null` constraint requires that the newly created or updated invitation does refer to valid rows in the other tables.

As said, the primary key is used to identify the record uniquely. In other words, the key can be used to obtain the value of the record's fields. Having noticed this fact, the field can be defined as a function from the primary key to the value of the field. The formal specification of the invitations table is presented in Listing 2.

```
1.  invitationId: TYPE = nat
2.  invitation_id_type: TYPE = set[invitationId]
3.  invitation_user_id_type: TYPE  = [invitationId -> userId]
4.  invitation_meeting_id_type: TYPE = [invitationId -> meetingId]
5.
6.  invitation_table_type: TYPE = [#
7.   idnt: invitation_id_type,
8.   user_id: invitation_user_id_type,
9.   meeting_id: invitation_meeting_id_type
10. #]
```

Listing 2. The PVS model for the invitations table.

In line 1 we introduce a type for the values of primary key that is synonymous with natural numbers. Line 2 defines a primary key of the table as the set. A row belongs to the table if and only if value of its primary key belongs to this set. Lines 3 and 4 introduce fields referring to primary keys of the dependent tables, users and meetings respectively. Lines 6-10 put together all the fields and creates a table type that is used in the rest of the code.

As previously mentioned, the idea of primary key is captured by the use of a set. Also note that all the declarations (except for the primary key) are similar; there is no distinction between a field referring to some other table and the not referring one. This is due to the typing: the return value of a function corresponding to a referring field is of type proper for the primary key of that table. This is very similar to the syntax of most commonly used SQL variants, as the reference is declared as a special type that points to the primary key of some table.

Having all the tables declared similarly, we declare the database schema as PVS record, where each field corresponds to one table, as presented in Listing 3.

```
1.  database_type: TYPE [#
2.   users: user_table_type,
3.   meetings: meeting_table_type,
4.   invitations: invitation_table_type,
5.   proposals: proposal_table_type,
6.   answers: answer_table_type
7.  #]
```

Listing 3. The declaration of the database in PVS.

3.2. Modelling simple operations

The structure of the database in itself provides only a model for the data. The dependencies between entities are stated and the type of data is set. However, the structure does not provide behavioural properties and constraints we demand from our system, as these are satisfied by operations performed on the data. All of them, regardless of complexity, are always translated to a sequence of four basic operations: select, update, delete and insert (also known as CRUD for create, read, update, destroy [19] [20]). In order to reason about the correctness of the specification that we want to implement, we have to model the basic operations as well. Then it is possible to state properties of the system and prove that they hold when the basic operations are executed. This can guarantee that the system behaves properly at all times.

3.2.1. Select

The select operation is used to read data from the table. Using predicative part of the query (also called *the where clause*) allows returning only those rows, for which the predicate is true. It is important to notice that the predicate depends on the table in question. This has a straightforward formalisation for the invitations table, as shown in Listing 4.

```
1.  selector_invitation_type: TYPE =
2.          [[invitationId, userId, meetingId] -> bool]
```

Listing 4. The selector for the invitations table.

With the selector we can model the operation that selects proper rows from the invitations table, as in Listing 5. The return value of the select operation is of the type of the table in question, as the returned rows have exactly the same structure as the table they have been selected from. Note that contrary to the SQL behaviour, the modelled operation returns all columns of the table.

```
1.  select_invitation(db: database_type,
2.                    selector: selector_invitation_type):
3.                                                invitation_table_type =
4.  (# idnt = {x: invitationId | member(x, idnt(invitations(db)))
5.                 AND selector(x, user_id(invitations(db))(x),
6.                 meeting_id(invitations(db))(x))},
7.     user_id = user_id(invitations(db)),
8.     meeting_id = meeting_id(invitations(db))
9.  #)
```

Listing 5. The PVS model for the select operation on the invitations table.

The behaviour of the function is very straightforward, as it exactly models the one of the corresponding SQL query. Returned are all the rows, which primary key defines an existing row and the selector predicate is satisfied (lines 4-6). The value of the key is then used to determine the values of the columns (lines 7-8) in the returned rows.

Whenever there is a need of selecting some records, proper selector must be constructed prior to calling the function and then passed to the select operation. For example, selecting all invitations issued to the user *U* can be done with the lines of code shown in Listing 6. The selector is defined in lines 1-2, where the Boolean condition in line 2 exactly matches the condition from the example. Line 3 executes the select function.

```
1.  selector(inv: invitationId, usr: userId, meet: mettingId): bool =
2.                (usr = U)
3.  select_invitation(db, selector)
```

Listing 6. Example use of the selector and the select function.

3.2.2. Delete

Deletion removes the rows from the specified table. Again, predicate is used to determine conditions that must be fulfilled in order for the row to be deleted. Once the delete operation is executed, the database is in the new state – the rows affected by the query have been removed. It is shown in Listing 7.

```
1.  delete_invitation(db: database_type,
2.                    selector: selector_invitation_type): database_type =
3.  LET new_invitations =
4.       (# idnt := {x: invitationId | member(x, idnt(invitations(db)))
5.                   AND NOT selector(x, user_id(invitations(db))(x),
6.                                       meeting_id(invitations(db))(x))},
7.          user_id := user_id(invitations(db)),
8.          meeting_id := meeting_id(invitations(db)) #),
9.      db1 = db WITH [invitations := new_invitations]
10. IN db1
```

Listing 7. The PVS model for the delete operation on the invitations table.

Again, the model captures the behaviour of its SQL correspondence. The invitations table after the removal (line 3) contains all the existing rows for which the selector is not true (lines 4-6). That is, all the rows that are indicated by the selector will be removed from the table. In the new database state only the invitations table is affected (line 9) and other tables are left intact.

As with the select operation, the deletion consists of constructing the selector and executing the function. For example, to delete all invitations that were issued for the meeting *M*, one has to write the code given in Listing 8.

```
1.  select(inv: invitationId, usr: userId, meet: meetingId): bool =
2.              (meet = M)
3.  delete_invitation(db, select)
```

Listing 8. Example use of the delete function.

3.2.3. Update

The update operation differs from the two previously described, as it not only selects proper rows, but also updates them respectively. Therefore, not only the selector is needed. We define the updater function to represent the corresponding part of the SQL query, as given in Listing 9.

```
1.  updater_invitation_type: TYPE =
2.            [[invitationId, userId, meetingId] -> [userId, meetingId]]
```

Listing 9. The updater function for the invitations table.

For every row passed to the updater, the function is expected to return the new value for the record. Notice that the function is defined in a way that does not alter the primary key. In practice it is undesirable to change the value of the primary key, as it can possibly lead to inconsistencies and difficulties hard to track, and is virtually never done in database applications.

As with the delete function, after executing the update the database is in the new state. The old rows are replaced with the new ones, in accordance with the updater function. The updating function is given in Listing 10. We use the projection functions built in PVS. The functions, named `proj_x`, return x^{th} parameter from the ones that are passed to them.

```
1.  update_invitation(db: database_type,
2.                    selector: selector_invitation_type,
3.                    updater: updater_invitation_type): database_type =
4.  LET cond1(inv: invitationId): bool =
5.      LET old_usr = user_id(invitations(db))(inv),
6.          old_meet = meeting_id(invitations(db))(inv),
7.          new_usr = proj_1(updater(inv, old_usr, old_meet)),
8.          new_meet = proj_2(updater(inv, old_usr, old_meet))
9.      IN member(new_usr, idnt(users(db))) AND
10.        member(new_meet, idnt(meetings(db))),
11.     new_user_id(inv: invitationId): userId =
12.     LET usr = user_id(invitation(db))(inv),
13.         meet = meeting_id(invitation(db))(inv)
14.     IN (COND NOT selector(x, usr, meet) -> usr,
15.              NOT cond1(x) -> usr,
16.              ELSE -> proj_1(updater(inv, usr, meet))
17.         ENDCOND),
18.     new_meeting_id(inv: invitationId): meetingId =
19.       LET usr = user_id(invitation(db))(inv),
20.         meet = meeting_id(invitation(db))(inv)
21.       IN (COND NOT selector(x., usr, meet) -> meet,
22.              NOT cond1(x) -> meet,
23.              ELSE -> proj_2(updater(inv, usr, meet))
24.          ENDCOND),
25.     new_invitations = (# idnt := idnt(invitations(db)),
26.                          user_id := new_user_id,
27.                          meeting_id := new_meeting_id #),
28.     db1 = db WITH [invitations := new_invitations]
29. IN db1
```

Listing 10. The PVS model for the update query on the table invitations.

During the update the integrity of the data must be checked – the fields that point to other tables must contain values that are existing primary keys of those tables. Therefore we declare conditions that check, given an invitation, if the user and meeting point to the existing objects in the database (lines 9-10). Once these are met, the functions corresponding to the fields of invitations table are updated to store new values (lines 11-24). After all the functions are updated, the new state of the database is created (line 28), where the contents of the invitations table are updated. It can be seen that the primary keys have not changed. As we have mentioned it previously, such behaviour is not recommended; therefore our model disallows doing so.

Listing 11 gives an example of how the function can be used. We want to change the invitations from the meeting *M1* to *M2* for all the users that have been already invited to *M1* (at this point we forget about initial requirement that disallows us to do so).

```
1.  M1: meetingId
2.  M2: meetingId
3.  users_set = set[userId]
4.
5.  selector(inv: invitationId, usr: userId, meet: meetingId): bool =
6.                member(usr, users_set) AND
7.                meet = M1
8.  updater(inv: invitationId, usr: userId, meet: meetingId):
9.               [userId, meetingId] = (usr, M2)
10.
11. update_invitation(db, selector, updater)
```

Listing 11. Example of the update function.

3.2.4. Insert

Adding the record to the database does not require selectors or updaters, as it takes the data of the new row and simply inserts it to the database. In principle inserting must take care of initial integrity constraints (user and meeting referred to in the new invitation must be present in the corresponding tables of the database) and ensure that the record to be inserted has new, unused primary key value. This can be modelled as given in Listing 12.

```
1.  insert_invitation(db: database_type,
2.                    inv: invitationId,
3.                    usr: userId,
4.                    meet: meetingId): database_type =
5.  LET new_invitations = (#
6.          idnt := add(inv, idnt(invitations(db))),
7.          user_id := user_id(invitations(db)) WITH [inv := usr],
8.          meeting_id := meeting_id(invitations(db)) WITH [inv := meet]
9.      #),
10.     db1 = (COND member(inv, idnt(invitations(db))) -> db,
11.                 NOT member(usr, idnt(users(db))) -> db,
12.                 NOT member(meet, idnt(meetings(db))) -> db,
13.                 ELSE -> LET db2 = db WITH
14.                                [invitations := new_invitations]
15.                         IN db2
16.            ENDCOND)
17. IN db1
```

Listing 12. The PVS model for the insert operation on invitations table.

The insertion function checks whether the new identifier is unused (line 10) and that the user and meeting of new row exist in the database (lines 11-12). In case of not fulfilling the initial constraints no change to the database is made. Otherwise, the new state of the database is returned, with invitations table containing the new row (lines 5-9, 13-15).

3.3. Modelling complex operations

In order to maintain complex constraints the need arises to perform a fixed sequence of simple commands. This typically happens when an operation modifying the data is executed. Such fixed sequence is then triggered before or after the operation itself, hence the name *trigger*. There can be a number of triggers for any operation. By using triggers virtually any data constraint can be preserved – the data not meeting the condition is not processed.

Often triggers are used implicitly, without stating them in code. For example, the SQL directive `on delete cascade` that can be used in row declaration, is executed in response to the delete operation. Although being part of the *de facto* standard syntax, its behaviour is converted to a trigger function.

Triggers are integral part of the database, so the implementation of client applications is not altered whenever the triggers change. In our case study triggers are used to maintain data integrity and to provide desired functionality.

3.3.1. Delete trigger

In the case study one of the functional requirements states that whenever an invitation to a meeting is deleted, all answers made by the invited user to the proposals concerning the meeting should be deleted. The dependency is not direct, as the answers are not pointing to the invitations, but to the users (following the intuitive understanding – it is the user who answers,

not the invitation). Such condition cannot be expressed with the standard SQL syntax, so triggers have to be used. Otherwise, the client application would have to implement this behaviour. This functionality is provided by the trigger presented in Listing 13.

```
1.  create function delete_answers_uninvited() returns trigger as $dai$
2.  begin
3.   DELETE FROM answers
4.   WHERE user_id = OLD.user_id AND proposal_id IN
5.         (SELECT id FROM proposals
6.          WHERE meeting_id = OLD. meeting_id);
7.   RETURN NULL;
8.  end;
9.  $dai$ language 'plpgsql';
10. create trigger tr_delete_answers_uninvited after delete on invitations
11. for each row execute procedure delete_answers_uninvited();
```

Listing 13. The trigger for deleting answers related to the removed invitation.

Nested selection (lines 5-6) is used to limit the deletion only to those answers, that are related to the proposal concerning the meeting of the invitation and are given by the user that the removed invitation was issued to (lines 3-4). Since the trigger is executed after the deletion takes place, the value it returns is ignored. Returning `NULL` value (line 7) in such cases is considered to be good practice of programming.

The trigger enhances the functionality of the original delete operation on the invitations table. In other words, deleting the row automatically deletes all answers that are related to it through the user and the meeting fields. Once the trigger is created, it is not possible to delete only the invitation row without altering the other tables. Therefore, we decided to model the trigger in PVS as if it was part of the delete function, as shown in Listing 14.

```
1.  delete_invitation(db: database_type,
2.                        selector: selector_invitation_type):
3.                                                     database_type =
4.  LET new_invitations =
5.       (# idnt := {x: invitationId | member(x, idnt(invitations(db)))
6.                   AND NOT selector(x, user_id(invitations(db))(x),
7.                           meeting_id(invitations(db))(x))},
8.          user_id := user_id(invitations(db)),
9.          meeting_id := meeting_id(invitations(db)) #),
10.     db1 = db WITH [invitations := new_invitations],
11.     del_rows = select_invitation(db, selector),
12.     sel1(ans: answerId,usr: userId,prop: proposalId,ok: bool): bool =
13.          EXISTS (x: invitationId):
14.           LET sel2(prop_sel2: proposalId, meet_sel2: meetingId,
15.                    str_sel2: nat, conf_sel2: bool): bool =
16.                        meet_sel2 = meeting_id(del_rows)(x),
17.               tbl1 = select_proposal(db1, sel2)
18.           IN member(x, idnt(del_raws)) AND
19.              usr = user_id(del_rows)(x) AND
20.              member(prop, idnt(tbl1)),
21.     db2 = delete_answer(db1, sel1)
22. IN db2
```

Listing 14. The PVS model of modified delete operation on table invitations.

At first we select all the records that satisfy the selector of the operation and save them to a new database state (lines 4-10). Subsequently, we construct a selector for the answers table that is based on proposals referred to the meeting of the invitation we remove (lines 12-

20) and use it to delete the answers (line 21). Having proven this function with a number of theorems that check the desired behaviour, we have the confidence that the database state is consistent after deleting the invitation.

3.3.2. Update trigger

The functional requirements state that once the invitation is issued, it has to remain unchanged (frozen). Of course, the invitation can be safely deleted, but it cannot be updated. The standard SQL syntax is not sufficient to perform this task; therefore, we have to use triggers again. Since we want to prevent a change from happening, we use the trigger that is executed before the operation takes place – in case of an attempt to change the frozen fields the update is abandoned and no changes happen. The trigger is shown in Listing 15.

```
1.  create function disable_invitation_change() returns trigger as $dic$
2.  begin
3.  IF NEW.user_id <> old_user_id OR NEW.meeting_id <> OLD.meeting_id
4.    THEN RETURN NULL;
5.    ELSE RETURN NEW;
6.  END IF;
7.  end;
8.  $dic$ language 'plpgsql'
9.
10. create trigger tr_disable_invitation_change before update
11. on invitations for each row
12. execute procedure disable_invitation_change(),
```

Listing 15. The trigger that prevents updates to already issued invitation.

The new values of the record to be updated are checked whether they match the previous ones (line 3). If this condition is not met, the operation is abandoned by returning `NULL` value (line 4), otherwise the updating takes place (line 5). As in the previous situation, the trigger extends the original update operation, this time placing additional condition that must be fulfilled before the operation can be performed. We use the same approach as previously by incorporating the trigger into the update operation, as shown in Listing 16.

```
1.  update_invitation(db: database_type,
2.                    selector: selector_invitation_type,
3.                    updater: updater_invitation_type): database_type =
4.  LET cond1(inv: invitationId): bool =
5.      LET old_usr = user_id(invitations(db))(inv),
6.          old_meet = meeting_id(invitations(db))(inv),
7.          new_usr = proj_1(updater(inv, old_usr, old_meet)),
8.          new_meet = proj_2(updater(inv, old_usr, old_meet))
9.      IN member(new_usr, idnt(users(db))) AND
10.        member(new_meet, idnt(meetings(db))),
11.     cond2(inv: invitationId): bool =
12.     LET old_usr = user_id(invitations(db))(inv),
13.         old_meet = meeting_id(invitations(db))(inv),
14.         new_usr = proj_1(updater(inv, old_usr, old_meet)),
15.         new_meet = proj_2(updater(inv, old_usr, old_meet))
16.     IN new_usr = old_usr AND new_meet = old_meet,
17.     new_user_id(inv: invitationId): userId =
18.        LET usr = user_id(invitation(db))(inv),
19.            meet = meeting_id(invitation(db))(inv)
20.        IN (COND NOT selector(x, usr, meet) -> usr,
21.                 NOT cond1(x) -> usr,
```

```
22.                 NOT cond2(x) -> usr,
23.                 ELSE -> proj_1(updater(inv, usr, meet))
24.            ENDCOND),
25.     new_meeting_id(inv: invitationId): meetingId =
26.        LET usr = user_id(invitation(db))(inv),
27.            meet = meeting_id(invitation(db))(inv)
28.        IN (COND NOT selector(x, usr, meet) -> meet,
29.                 NOT cond1(x) -> meet,
30.                 NOT cond2(x) -> meet,
31.                 ELSE -> proj_2(updater(inv, usr, meet))
32.            ENDCOND),
33.     new_invitations = (# idnt := idnt(invitations(db)),
34.                          user_id := new_user_id,
35.                          meeting_id := new_meeting_id #),
36.     db1 = db WITH [invitations := new_invitations]
37. IN db1
```
Listing 16. The PVS model of modified update operation on table invitations.

It can be seen that the constraint needed for the invitation to be updated is stated as a condition (lines 4-16) that is evaluated before the database is changed (lines 20-24, 28-32). Whenever a new constraint is needed, it is simply added as a new branch in the conditional statement. Such definition of update allows proving that the functional requirement, which forces once issued invitation to remain read-only, is satisfied.

3.3.3. Insert trigger

Another property we expect from our system states that the invitation to the meeting cannot be issued to the host of the meeting. Again, this constraint is too complex to be expressed in SQL only. The corresponding trigger has to prevent an insertion to the invitations table if the user invited to the meeting is the host of it. Therefore it has to be executed before the operation takes place. The SQL code is presented in Listing 17.

```
1.  create function disable_inviting_host() returns trigger as $dih$
2.  declare
3.    host integer;
4.  begin
5.  SELECT INTO host user_id FROM meetings WHERE id = NEW.meeting_id;
6.  IF host = NEW.user_id
7.    THEN RETURN NULL;
8.    ELSE RETURN NEW;
9.  END IF
10. end;
11. $dih$ language 'plpgsql'
12.
13. create trigger tr_disable_inviting_host before insert on invitations
14. for each row execute procedure disable_inviting_host();
```
Listing 17. The trigger that prevents issuing an invitation to the host of the meeting.

We apply the standard technique used to get only one, specific information from a single row (line 5) in order to check the condition. It can be modelled even more straightforwardly by using quantifiers and not the selector, as shown in Listing 18.

```
1.  insert_invitation(db: database_type,
2.                    inv: invitationId,
3.                    usr: userId,
```

```
4.                    meet: meetingId): database_type =
5.  LET host = user_id(meetings(db))(meet),
6.      cond1 = (host = usr),
7.      new_invitations = (#
8.          idnt := add(inv, idnt(invitations(db))),
9.          user_id := user_id(invitations(db)) WITH [inv := usr],
10.         meeting_id := meeting_id(invitations(db)) WITH [inv := meet]
11.     #),
12.     db1 = (COND member(inv, idnt(invitations(db))) -> db,
13.                 NOT member(usr, idnt(users(db))) -> db,
14.                 NOT member(meet, idnt(meetings(db))) -> db,
15.                 cond1 -> db,
16.                 ELSE -> LET db2 = db WITH
17.                                [invitations := new_invitations]
18.                         IN db2
19.            ENDCOND)
20. IN db1
```
Listing 18. The PVS model of modified insert operation.

As with the previously modelled operation, the condition is added and evaluated before the insertion takes place (lines 5-6). Having this function we can state that the host not being invited to any meeting is an invariant over the insertion operation. First we define a predicate that corresponds to this property, as shown in Listing 19.

```
1.  not_inv_host(db: database_type): bool =
2.       FORALL (i:invitationId): member(i, idnt(invitations(db))) =>
3.         user_id(invitations(db))(i) /=
4.         user_id(meetings(db))(meeting_id(invitations(db))(id))
```
Listing 19. The predicate corresponding to the property of the system.

The predicate is used to form a theorem presented in Listing 20. It states that the property is an invariant over the operation of adding new invitations. In other words, adding new invitation preserves the property. In the theory our model forms this theorem was proven simply by expanding the definition of the predicate and then applying basic `(grind)` strategy in PVS.

```
1.  insert_invitation_th7: THEOREM
2.    FORALL (db:database_type, inv: invitationId,
3.            usr: userId, meet: meetingId):
4.      LET new_db = insert_invitation(db, inv, usr, meet)
5.      IN
6.      not_inv_host(db) => not_inv_host(new_db)
```
Listing 20. The theorem stating that the predicate is an invariant of the insert operation.

We have just shown that the host cannot be invited and earlier we have shown that the invitation cannot be changed once issued, therefore all operations will preserve such invariant. In other words, we have constructed a PVS theory in which this property can be proven true.

4 Verification effort

By using our approach we have verified the database schema of Deseo Meeting Scheduler with its structural and behavioural features. In our PVS model we have stated and

proven 167 theorems and lemmas corresponding to various properties of the system [21]. They can be divided in two groups.

The majority of theorems describe the desired behaviour of operations that can be performed on database tables and is relatively simple to prove. They can be considered as equivalent to unit tests in the SQL development. By proving these theorems we ensure that the operations behave as expected and yield desired results.

The theorems in the second group correspond to system properties and invariants. By proving them we gain the confidence that the system as whole works properly. We ensure that it is correct according to the specification regardless of the order or number of simple operations performed. Those theorems can be seen as equivalent to functional tests and guarantee the correct behaviour of the system on higher level of abstraction.

Main advantage of our formalisation method is that the theorems can be proven with straightforward PVS proving strategies. Most of the theorems in the case study were proven with `(grind)` command, and only a few required more dedicated approach.

5 Conclusions and future work

Since the relevance of the database applications as a type of developed software is significant, so is the need for their correctness. Therefore, it is essential to have a lightweight verification method for checking the correctness of such systems. With the case study project we illustrated a novel approach to the verification of the database applications. Based on precise requirements we created SQL code for the database and showed equivalent model in PVS. We used this PVS theory to state and prove theorems that correspond to properties of the system.

Our modelling and formalisation approach is based on sets and functions. We wanted to use simple and easy to understand framework relying on solid and well known mathematical background. There is a separate type for the primary key of each table and the possible values that identify existing rows are held in a set of elements of such type. The fields (or attributes) are then represented as functions from the type of the primary key to the type of the field. With this approach we can, in a natural way, state the properties of the system we want to construct. Moreover, the resulting theorems are simple to prove and can be proven mostly automatically.

We are encouraged with the results of our work and believe that the technique we developed can be generalised, so that it could be applied to other projects as well. Furthermore, we have experienced that main steps of our modelling technique are repeatable, thus suitable for automation and code generation. In the broader perspective we aim to construct a programming environment that would allow to model databases and prove their correctness at the same time. Our current efforts are focused towards the generalisation of the method we developed, so that it is more modular and more reusable.

Acknowledgements

We would like to thank Prof. Iván Porres and Dr Viorel Preoteasa for fruitful discussions on the topic of modelling and verification. We also thank M.Sc. Johannes Eriksson for collaboration on the meeting scheduler application.

References

1. Fowler Martin, Rice David, *Patterns of Enterprise Application Architecture*. Addison-Weasley (2003).
2. Barnett Mike, DeLine Robert, Fähndrich Manuel, Leino K., Schulte Wolfram, *Verification of object-oriented programs with invariants*, Journal of Object Technology, (2004).
3. Dijkstra Edsger, *A constructive approach to the problem of program correctness*. BIT (1968).
4. van Emden M., *Programming with verification conditions*. In: *IEEE Transactions on Software Engineering*, (1979).
5. Reynolds J., *Programming with transition diagrams*. In: Gries D., *Programming methodology*, Springer-Verlag, Berlin (1978).
6. Back Ralph-Johan, *Invariant based programs and their correctness*. In: Biermann W., Guiho G., Kodratoff Y., *Automatic Program Construction Techniques*, MacMillan Publishing Company (1983).
7. Back Ralph-Johan, *Invariant based programming*. In: Donatelli Susanna, Thiagarajan P., *Lecture Notes in Computer Science*, Springer (2006).
8. Back Ralph-Johan, Eriksson Johannes, Myréen Magnus, *Testing and Verifying Invariant Based Programs in the SOCOS Environment*. TUCS Technical Report (2006)
9. PVS Verification and Specification System, http://pvs.csl.sri.com/
10. Chen Peter, *The Entity-Relationship Model-Toward a Unified View of Data*, ACM Transactions on Database Systems, (1976).
11. Choppella Venkatesh, Sengupta Arijit, Robertson Edward, Johnson Steven, *Preliminary Explorations in Specifying and Validating Entity-Relationship Models in PVS*. AFM, Atlanta, USA (2007)
12. Back Ralph-Johan, Olszewski Mikołaj, Porres Iván, Preoteasa Viorel, Eriksson Johannes, Soriano Damián, *Deseo Meeting Scheduler - Specification*. Software Construction Laboratory, Åbo Akademi University (2007).
13. Sheard T., Stemple D., *Automatic verification of database transaction safety*, ACM Transactions on Database Systems, (1989), pp.322-368.
14. Chamberlin Donald, Boyce Raymond, *SEQUEL: A Structured Query Language*, Proceedings of the 1974 ACM SIGFIDET Workshop on Data Description, Access and Control, (1974).
15. Codd E., *A relational model of data for large shared data banks*, Communications of ACM, (1970).
16. Relational database, http://en.wikipedia.org/w/index.php?title=Relational_database&oldid=247489980
17. Pląska Marta, *Development of Database Applications (M.Sc. Thesis)*. University of Gdańsk, Gdańsk (2006).
18. PostgreSQL, http://www.postgresql.org
19. Kilov H., *From semantic to object-oriented data modelling*, Proceedings of the First International Conference on Volume, (1990).
20. Kilov Haim, *Business Specifications: The Key to Successful Software Engineering*. Prentice Hall (1998).
21. Back Ralph-Johan, Olszewski Mikołaj, Soriano Damián, *Deseo Meeting Scheduler: Database Schema with Verified Business Logic*. To appear in TUCS Technical Report series, Turku Centre for Computer Science, Turku (2009)