*Project:* DRA FRONT END FILTER PROJECT

*Title:* An Execution Model for SWORD

*Ref:* DS/FMU/FEF/026 *Issue: Revision : 2.1* *Date:* 5 June 2016

*Status:* Approved *Type:* Specification

*Keywords:*

*Author:*

| *Name* | *Location* | *Signature* | *Date* |
|---|---|---|---|
| R. D. Arthan | WIN01 | | |

*Authorisation for Issue:*

| *Name* | *Function* | *Signature* | *Date* |
|---|---|---|---|
| R.B. Jones | HAT Manager | | |

*Abstract:* A formalisation of an execution model for the Front End implementation of SWORD for the DRA front end filter project RSRE 1C/6130.

*Distribution:* HAT FEF File
Simon Wiseman

# 0 DOCUMENT CONTROL

## 0.1 Contents List



## 0.2 Document Cross References

[1] DS/FMU/017. *Secure Database Technical Proposal.* High Assurance Team, ICL Secure Systems, WIN01, 21st January 1992.

[2] DS/FMU/FEF/004. *Specification of SSQL Semantics I.* G.M. Prout, ICL Secure Systems, WIN01.

[3] DS/FMU/FEF/005. *Specifications of hide and updateState*. G.M. Prout, ICL Secure Systems, WIN01.

[4] DS/FMU/FEF/014. *Specification of SSQL Semantics II.* G.M. Prout, ICL Secure Systems, WIN01.

[5] DS/FMU/FEF/021. *Specification of TSQL.* G.M. Prout, ICL Secure Systems, WIN01.

[6] DS/FMU/FEF/022. *SWORD Front End Architectural Model.* R.D. Arthan, ICL Secure Systems, WIN01.

[7] DS/FMU/FEF/024. *A HOL Specification of the SWORD Output Filter.* G.M. Prout, ICL Secure Systems, WIN01.

[8] DS/FMU/FEF/025. *Representation of an SSQL State as a TSQL State.* G.M. Prout, ICL Secure Systems, WIN01.

[9] DS/FMU/FEF/029. *Specification of Query Transformations in HOL (II).* R.D. Arthan, ICL Secure Systems, WIN01.

[10] DS/FMU/FEF/032. *Table Computations for SWORD.* R.D. Arthan, ICL Secure Systems, WIN01.

[11] DS/FMU/FEF/034. *Phase II Proof Strategy.* R.D. Arthan, ICL Secure Systems, WIN01.

[12] *The SWORD Front End Filter Specification.* Simon Wiseman, DRA, 21st January 1993.

[13] *SSQL Transformations.* Simon Wiseman, DRA, 14th January 1993.

### 0.3 Changes History

**Issue 1.1 (18 February 1993)** First draft.

**Issue 1.18 (5 November 1993)** *Condition_E_Lemma* removed.

**Issue** *Revision : 2.1* **(5 June 2016)** Final approved version.

**Issue 2.2** Removed dependency on ICL logo font

### 0.4 Changes Forecast

None.

# 1 GENERAL

## 1.1 Scope

This document gives a formal specification of part of the SWORD Front End giving the high-level security properties required of both the Query Transformations of [13] and of the Front End Filter of [12]. It constitutes part of deliverable D3 of work package 1a, as given in the Phase 2 Technical Proposal, [1].

## 1.2 Introduction

[6] gives a formal description of the top-level architecture of the Front End Implementation of SWORD and identifies three subsystems from which the system is constructed, namely, the SSQL Transformation Processor, the TSQL database, and the Output Filter. It is the purpose of this document to give a more concrete formulation of the critical requirements on these subsystems.

The current issue gives preliminary discussion of issues in formalising the critical requirements. Some formal material is included.

# 2 PRELIMINARIES

The following ProofPower instructions set the context for the proof tools and set up the new theory *fef026*, with parent the theory *fef029* in which the syntactic transformation of SSQL queries is formalised.

SML

```
open_theory "fef029";
(force_delete_theory "fef026" handle _ => ());
new_theory"fef026";
push_pc "hol";
```

# 3 REPRESENTATION OF DERIVED TABLES

The abstract formulation of the critical requirements on the subsystems of *FE_SWORD* given in [6] give little insight into *how* the subsystems are to be constructed. Further detail on the actual data types involved is required to discuss this.

Apart from some minor and easily remedied differences of organisation, the syntax of TSQL is a subset of the syntax of SSQL. For simplicity, it has been chosen to model the TSQL abstract syntax as being identical with the SSQL syntax. The TSQL semantics may then be specified reusing most of the SSQL semantics of [2, 3, 4] by imposing a state invariant which asserts that all classification information in the state, apart from where classifications are stored as data, is fixed at the lowest classification. A formal treatment of the TSQL semantics along these lines is given in [5].

The representation of a TSQL state as an SSQL state is defined informally in [12] and formally in [8]. In the general case, an SSQL table with $k$ columns is represented as a TSQL table with $1 + 3k$

columns: the first column in the TSQL table contain the row existence classes, successive blocks of 3 columns give the classification, the dinary data and the sterling data for the corresponding SSQL column. The SSQL state also associates some 'static' classification information about each table, namely: its class ($TS\_class$[1]) and its maximum row class ($TS\_maxRow$), and also about each column in a table, namely: its existence class ($CC\_exist$) and the maximum and minimum classifications for the data in the column ($CS\_min$, $CS\_max$). This information is not represented in the TSQL state (as modelled here, although in practice, it will be held in a TSQL table); however, it is used in the transformations of [12] in order to optimise the representation, so that, for example, the classification column is omitted for SSQL columns for which $CS\_min = CS\_max$.

The management of the optimisations discussed above and of the variable scoping rules of the query language make a significant contribution to the complexity of the transformations as specified in [12]. The scoping rules also complicate the semantics of both SSQL as given in [4] and hence of TSQL. The main security-relevant mechanisms used in the transformations are therefore best understood if the optimisations, the handling of the scoping rules for the two languages, and the security checks themselves are handled separately in a formal treatment.

Most of the complexity of the TSQL and SSQL semantics is in the *SELECT* query. Indeed, at least in *TSQL*, the *DELETE*, *INSERT* and *UPDATE* queries can be modelled as a *SELECT* query to compute a new table followed by an assignment or a merge of the new table into an existing table.

The security checks which apply to a *SELECT* query are best understood if we think of execution of the query as returning a *derived table*, which we think of as having two components: the table specification which contains static information about the table and its columns; and the list of rows comprising the table data proper. We will take a conceptual, unoptimised, view of derived tables and so use the following data types for them:

HOL Labelled Product

**DerColSpec**

| | |
|---|---|
| **DCS_name** | : *Ide LIST LIST*; |
| **DCS_min** | : *Class*; |
| **DCS_max** | : *Class* |

HOL Labelled Product

**DerTableSpec**

| | |
|---|---|
| **DTS_name** | : *Ide LIST LIST*; |
| **DTS_maxRow** | : *Class*; |
| **DTS_colSpecs** | : *DerColSpec LIST* |

HOL Labelled Product

**DerTableRow**

| | |
|---|---|
| **DTR_where** | : *Class*; |
| **DTR_row** | : *Class*; |
| **DTR_cols** | : $(Class \times Item)$ *LIST* |

[1] The names used here are as used in [2].

HOL Labelled Product

**DerTable**

| | | |
|---|---|---|
| **DT_spec** | : *DerTableSpec*; |
| **DT_rows** | : *DerTableRow LIST* |

The information in the above is intended to be extracted from information recorded for a table in the SSQL state or computed during derivation of the table, as described in the following table:

| Field | Description |
|---|---|
| *DCS_name* | The names by which the column may be known (computed from *DirectoryS* and the *ColSpec*s for the table initially) |
| *DCS_min*/*DCS_max* | The minimum and maximum classes for data in the column (taken from the *ColSpec*s for the column initially) |
| *DTS_name* | The names by which the table may be known (computed from *DirectoryS* initially) |
| *DTS_class* | The class of the table (computed from the *TableSpec* for the table initially) |
| *DTS_maxRow* | The maximum row existence class of the table (computed from the *TableSpec* for the table initially) |
| *DTS_colSpecs* | The column specifications for the table |
| *DTR_where* | The classification in this row of a *WHERE* clause used to construct this table. This information is only used during execution of transformed queries and by the output filter; it is not stored in the state and is taken as *lattice_bottom* for tables extracted directly from the state. |
| *DTR_row* | The existence classification of this row (taken from the SSQL state initially) |
| *DTR_cols* | The classication and data for each field in this row (taken from the SSQL state initially) |

We have chosen not to include the table or column existence classes here.

# 4 EXAMPLES

Before discussing the use of the above representation of a derived table within the formal framework, it may be helpful to consider some examples of the transformation of SSQL queries into TSQL. For the examples which follow, the main part of the transformation algorithm is defined in the description of $tuple\_list_{make\_outer}$ in [13].

**Example 1** The basic principle of operation of the Front End implementation of SWORD is apparent from the treatment of the simplest form of *SELECT* query:

SSQL Example

*SELECT* ∗ *FROM table*

In essence, this is transformed into a single TSQL query (with no check query) of the form[2]:

TSQL Example

$$\begin{array}{l} \mathit{SELECT}\ \mathit{cc},\ \mathit{rc},\ \mathit{col1}_s,\ \mathit{col1}_d,\ \mathit{col1}_c,\ ... \\ \mathit{FROM}\ \mathit{table}_t \\ \mathit{WHERE}\ \mathit{cc}\ \mathit{DOM}\ \mathit{rc} \end{array}$$

where $table_t$ is the TSQL table implementing *table*, *cc* is a constant classification equal to the clearance of the client, *rc* is the column containing the row classes, and the $col\mathcal{I}_{\mathcal{X}}$ are the columns containing the classes, sterling data and dinary data for the (SSQL) columns of *table*. Viewed conceptually as a derived table in the above sense, the result of this TSQL query represents the entirety of all rows in *table* whose existence the client is cleared to know, together with a column of where clause classes all equal to the clearance of the client. Filtering such a derived table amounts, in effect, to removing all rows whose class is not dominated by the client clearance and overwriting with a dummy value the data in all fields where the field classification is not dominated by the client clearance. Since, in this case, all of the classification information in the derived table is taken directly from the database (and so may be assumed to be correct), the result of this filtering removes all information content which the client is not cleared to see[3].

**Example 2** For a more complex SSQL query, classifications in the derived table which represents its output are computed during execution of the TSQL query. For example, consider:

SSQL Example

$$\mathit{SELECT}\ \mathit{col2} + \mathit{col3}\ \mathit{FROM}\ \mathit{table}\ \mathit{WHERE}\ \mathit{col1} < \mathit{col2}$$

Here, in each row, computation of the *WHERE* clause reveals information about values in the first two fields of the row. Again there is no check query and the transformed query might have the form:

TSQL Example

$$\begin{array}{l} \mathit{SELECT}\ \mathit{col1}_c\ \mathit{LUB}\ \mathit{col2}_c,\ \mathit{rc},\ \mathit{col2}_s + \mathit{col3}_s,\ \mathit{col2}_c\ \mathit{LUB}\ \mathit{col3}_c \\ \mathit{FROM}\ \mathit{table}_t \\ \mathit{WHERE}\ (\mathit{col1}_s < \mathit{col2}_s\ \mathit{OR}\ \mathit{NOT}\ \mathit{cc}\ \mathit{DOM}\ (\mathit{col1}_c\ \mathit{LUB}\ \mathit{col2}_c))\ \mathit{AND}\ \mathit{cc}\ \mathit{DOM}\ \mathit{rc} \end{array}$$

Here, the classification associated by the transformations with each computed data value $col2_s + col3_s$ in each row is the least upper bound of the classifications for *col2* and *col3* in that row, since the result of the addition reveals information about both of its operands.

Note that in the derived table passed to the filter, rows for which either the SSQL *WHERE* clause is true or the client is not cleared to compute the *WHERE* clause are included. I.e. the transformed query takes the *WHERE* clause as *true* for rows where the *WHERE* clause computation is a possible covert channel. From the security point of view the *WHERE* clause could equally well be taken as *false* for such rows, however, part of the desired filter functionality is to inform the client when such rows have been detected. What the filter then does is to issue a “may-not-be-complete” message and to delete the offending rows.

[2] In fact, the optimised query may omit some of the expressions in the select-list. In this case, *cc* would certainly be omitted, since there is no *WHERE* clause.

[3] In the current specifications [12, 13], the transformations add the *WHERE* clause as shown in the example and the filter does not need to eliminate rows whose existence the client is not cleared to know, since there will not be any. However, the row existence column is passed into the filter, since it comprises part of the output returned to the client (as is the class of the *WHERE* clause).

**Example 3** In the above examples, there is no need for a check query since the data query can include all the information required for the filter to eliminate possible covert channels. The check query becomes necessary when the query includes a *GROUP BY* clause, e.g. consider the following query:

SSQL Example

$SELECT\ col1,\ COUNT\ (*)\ FROM\ table\ WHERE\ col2 > 0\ GROUP\ BY\ col1$

This is intended to return a table showing, for each value of *col1* appearing in *table* in a row having a positive *col2* value, the number of rows having that *col1* value and a positive *col2* value.

The query must not be allowed to reveal information about rows whose *col1* value the client is not cleared to see. This will result in a check query as follows:

TSQL Example

$SELECT\ TRUE$
$FROM\ table_t$
$WHERE\ cc\ DOM\ rc\ AND\ col2_s > 0\ AND\ NOT\ cc\ DOM\ col1_c$

Thus the check query will return a row for each row in *table* whose existence the client is cleared to know, whose *col2* value is such that the row would be included in the counts to be made by the data query and whose *col1* value the client is not cleared to know. If there are any such rows the data query should not be allowed to proceed.

**Example 4** The above examples involve selection from a single table. Selection from (the cartesian product of) several tables follows similar lines except that the row existence classes must be combined to give the existence class for the derived table. For, example, consider:

SSQL Example

$SELECT * FROM\ tableA,\ tableB$

This is transformed into the following query, which is similar to the selection from a single table but with the existence class in each row of the cartesian product taken to be the least upper bound of the existence classes of the two rows it is formed from.

TSQL Example

$SELECT\ cc,\ rcA\ LUB\ rcB,\ colA1_s,\ colA1_d,\ colA1_c,\ ...,\ colB1_s,\ colB1_d,\ colB1_c,\ ...$
$FROM\ tableA_t,\ tableB_t,$
$WHERE\ cc\ DOM\ (rcA\ LUB\ rcB)$

**Discussion** In fact, the above examples display most of the security relevant features of the transformations, with the following exceptions:

1. The more complex variants of a single select query, e.g. the *HAVING* clause.
2. More complex cases of the computation of the classification of the result of an expression. Example 2 above shows one of the simpler cases, where for each row, the expression $col2 + col3$ is assigned a classification which is the least upper bound of the classifications of *col2* and *col3* in that row). In other examples, e.g., the *and* and *or* expression forms, the result classification used depends on the actual values of (some of) the operands as well as on the classifications associated with them.

3. The details of the mapping of the SSQL name space onto the TSQL name space.

4. The handling of nested *SELECT* queries and the management of the scope of the name spaces within them.

The formal treatment in this document and in [10] is intended to address all of these issues, however the treatment of naming issues is somewhat simplified, bringing out the semantic issues but not all the syntactic ones.

# 5 EXECUTION MODEL (PART 1)

In order to simplify the handling both of nested *SELECT*s and of name space management, it is proposed to use a model of the TSQL execution mechanism in which the essence of the *SELECT* functionality is separated out. To do this, we restrict attention to transitions of the underlying DBMS whose effect can be viewed as having been obtained by the following steps:

1. Represent the visible state of the database as a list of derived tables, *dtl*;

2. Perform some computation on *dtl* resulting in a new derived table, *dt*;

3. Either filter *dt* to create the output for the client (*SELECT* query), or use *dt* and other information from the query to update the database (other queries).

Here, step 1 is intended to involve a fixed mapping on database states dependent on the SSQL database structure. The main security property enforced by step 1 is the removal of tables which the client is not cleared to see (cf. [13], in which such tables are not entered into the global symbol table).

In step 3, the filtering for the output from the *SELECT* queries is to be carried out by using the filter specified in [7]. The other sorts of query are to be processed on the assumption that *dt* encodes information about the modifications to be made to the table named in the query.

Step 2 is where the main security features of the transformations[13] are modelled. In particular, by bounding the class of computations permitted in this step, we can formalise the intuitions behind the clearances which are assigned by the transformations to the fields of derived tables.

The three-stage model of TSQL execution may be viewed as an alternative description of the function *TSQLtf* of [5], which reveals the semantic issues but ignores syntactic ones. It may be formalised as follows:

HOL Constant

$$\begin{array}{l} \mathbf{EM} : (State_t \to DerTable\ LIST) \\ \quad \to (Query \to (DerTable\ LIST \to DerTable \times Errors)) \\ \quad \to (Query \times (DerTable \times Errors) \times State_t \to State_t \times ANSWER) \\ \quad \to (Query,\ State_t)\ DBMS\_TYPE \end{array}$$

$$\begin{array}{l} \forall\ view;\ compile;\ act;\ query;\ st \bullet \\ \quad EM\ view\ compile\ act\ (query,\ st) \\ = \quad let \quad compute = compile\ query \\ \quad in\ let \quad viewed = view\ st \\ \quad in\ let \quad computed = compute\ viewed \\ \quad in \quad act\ (query,\ computed,\ st) \end{array}$$

Here the components, *view*, *compile* and *act* of the model represent the three stages as follows:

1. *view* represents the state of the database as a list of derived tables;
2. *compile* compiles a query to give the derived-table-forming function it computes;
3. *act* either updates the state or delivers an answer according to the type of *query* and the derived table computed by the compiled query.

# 6 REPRESENTING SSQL AND TSQL STATES AS DERIVED TABLES

This section describes how SSQL and TSQL states are viewed as lists of derived tables. A fairly explicit algorithm is specified for the SSQL states. The view for the TSQL states is then loosely specified in terms of the SSQL formulation.

The classification of a derived table in an SSQL state will be the least upper bound of the corresponding table classification and 'containing directory' classifications in the SSQL state.

## 6.1 Interpreting a State as a List of Tables

Obtain a list of pairs of directory name and directory from a state together with the upper bound of the clearances for the directory.

HOL Constant

$$\mathbf{StateDirs} : State \to\ (Class \times Ide\ LIST \times Directory)\ LIST$$

$$\begin{array}{l} \forall\ s \bullet \quad Elems\ (StateDirs\ s) \\ = \quad \{ \quad (c,\ (i,\ d)) \\ \quad\quad | \quad (i,\ d) \in repState\ s \\ \quad\quad \wedge \quad c = (Dir\_exist\ d\ lub\ Dir\_class\ d)\} \end{array}$$

Obtain a list of pairs of table name and table from a directory.

HOL Constant

$$\textbf{DirTables} : Directory \to (Ide \times TableSpec)\ LIST$$

$$\forall\ d \bullet Elems\ (DirTables\ d) = Dir\_tables\ d$$

Obtain a list of quadruples of class, directory name, table name and table from a state. This involves propagating the classes and directory names obtained for each directory *StateDirs* into the information returned for each table.

HOL Constant

$$\textbf{StateTables} : State \to (Class \times Ide\ LIST \times Ide \times TableSpec) LIST$$

$$\begin{array}{l}
\forall s \bullet StateTables\ s = \\
\quad let\ (cl,illdl) = Split(StateDirs\ s) \\
\quad in\ let\ (ill,dl) = Split\ illdl \\
\quad in\ let\ (itll) = (Map\ DirTables\ dl) \\
\quad in\ let\ f\ c\ i\ it = (c,\ i,it) \\
\quad in\ let\ g\ (c,\ i,\ its) = Map\ (f\ c\ i)\ its \\
\quad in\ let\ h\ cl\ is\ itss = Map\ g\ (Combine\ cl\ (Combine\ is\ itss)) \\
\quad in\ Flat\ (h\ cl\ ill\ itll)
\end{array}$$

## 6.2 Mapping SSQL Tables to Derived Tables

First, obtain the static column information for a derived table from a *ColSpec*.

HOL Constant

$$\textbf{ColSpec}_{\mathbf{d}} : (Ide\ LIST \times Ide \times TableSpec) \to ColSpec \to DerColSpec$$

$$\begin{array}{l}
\forall\ d_name\ t_name\ t\ cs \bullet ColSpec_d\ (d_name,t_name,t)\ cs = \\
\qquad let\ cc = (TS_cons\ t)\ @\ (CS_consGroup\ cs) \\
\qquad in\ let\ tc_name = Cons\ t_name[CS_ide\ cs] \\
\qquad in\ \ MkDerColSpec \\
\qquad\qquad [[CS_ide\ cs];tc_name;d_name \frown tc_name] \\
\qquad\qquad (CS_min\ cs) \\
\qquad\qquad (CS_max\ cs)
\end{array}$$

Obtain the static table information for a derived table from a *TableSpec*.

HOL Constant

$$\mathbf{TableSpec_d} : (Class \times Ide\ LIST \times Ide\ \times TableSpec) \to DerTableSpec$$

$$\begin{array}{l}
\forall\ c\ d_name\ t_name\ t \bullet TableSpec_d\ (c,\ d_name,t_name,t) = \\
\quad MkDerTableSpec \\
\quad\quad [[t_name];d_name \frown [t_name]] \\
\quad\quad (TS_maxRow\ t) \\
\quad\quad (RelList(Squash\{(n,cs) | \exists\ cs' \bullet\ cs' \in TS_colspecs\ t \\
\quad\quad\quad \wedge\ cs = ColSpec_d\ (d_name,t_name,t)\ cs' \wedge n = CS_posn\ cs'\}))
\end{array}$$

Obtain the information in a derived table row from a *TableSpec*, a *Row* and a classification. The where clause class is set to bottom since no where clause has been evaluated for the row.

HOL Constant

$$\mathbf{TableRow_d} : TableSpec \to Row \to DerTableRow$$

$$\begin{array}{l}
\forall t\ r \bullet TableRow_d\ t\ r = \\
\quad let\ f = \lambda\ d \bullet (Dat_class\ d, Dat_item\ d) \\
\quad in\quad MkDerTableRow \\
\quad\quad lattice_bottom \\
\quad\quad (R_exist\ r) \\
\quad\quad (RelList(Squash \\
\quad\quad \{(n,ic) | n \in Dom(R_data\ r) \wedge ic = f((R_data\ r)\ @\ n)\}))
\end{array}$$

Now obtain the derived table from a *TableSpec* and a classification.

HOL Constant

$$\mathbf{Table_d} : (Class \times Ide\ LIST \times Ide\ \times TableSpec) \to DerTable$$

$$\begin{array}{l}
\forall\ c\ d_name\ t_name\ t \bullet Table_d\ (c,\ d_name,t_name,t) = \\
\quad MkDerTable \\
\quad\quad (TableSpec_d\ (c,\ d_name,t_name,t)) \\
\quad\quad (Map\ (TableRow_d\ t)\ (TS_rows\ t))
\end{array}$$

## 6.3 Viewing an SSQL State as a List of Derived Tables

Finally the mapping from an SSQL *State* : *Exp* to a list of derived tables.

HOL Constant

$$\mathbf{View_s} : State \to DerTable\ LIST$$

$$\forall\ s \bullet View_s\ s = Map\ Table_d\ (StateTables\ s)$$

### 6.4 Viewing a TSQL State as a List of Derived Tables

The view of a TSQL state is obtained using the view function for SSQL and the representation function of [8].

HOL Constant

$$\mathbf{View_t} : State_t \to DerTable\ LIST$$

$$\forall\ s_t \bullet \forall s_s \bullet\ s_t = reprState\ s_s \Rightarrow View_t\ s_t = View_s\ s_s$$

Note that the consistency of the above implicit definition requires that any two SSQL states which are identified by *reprState* must be the same when viewed as lists of derived tables.

# 7 ACTION FUNCTION

A derived table is flattened into a list of data tuples using the following function:

HOL Constant

$$\mathbf{GiveData} : DerTable \to Data\ LIST\ LIST$$

$$\begin{array}{lll}
\forall dt \bullet & GiveData\ dt = & \\
let & class_item\ c = ValuedItemItem(MkValuedItem\ sterling\ (ClassVal\ c)) & \\
in\ let & item_data\ i = MkData\ lattice_bottom\ i & \\
in\ let & class_data\ c = item_data\ (class_item\ c) & \\
in\ let & cell_cols\ (c,\ i) = [class_data\ c;\ item_data\ i] & \\
in\ let & row_data\ r & \\
 & = & Flat \\
 & & (Cons \\
 & & [class_data\ (DTR_where\ r);\ class_data\ (DTR_row\ r)] \\
 & & (Map\ cell_cols\ (DTR_cols\ r))) \\
in & Map\ row_data(DT_rows\ dt) &
\end{array}$$

The following, which is intended to determine whether or not a query is a *SELECT* query might be moved to an earlier document, I think:

HOL Constant

$$\mathbf{is\_select} : Query \to BOOL$$

$$\forall q \bullet\ is\_select\ q \Leftrightarrow \exists\ t \bullet q = select\ t$$

We do not yet have a specification of any of the database update operations and so the update function must be supplied as a parameter to the function defining the action of the system:

HOL Constant

$$
\begin{array}{lll}
\mathbf{Act_t} & : & (Query \times (DerTable \times Errors) \times State_t \to State_t) \\
 & \to & Query \times (DerTable \times Errors) \times State_t \to State_t \times ANSWER
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\forall upd;\ query;\ dt;\ errs;\ st \bullet} \\
 & \multicolumn{2}{l}{Act_t\ upd\ (query,\ (dt,\ errs),\ st)} \\
= & if & \neg errs = [] \\
 & then & (st,\ ([],\ errs)) \\
 & else\ if & is\_select\ query \\
 & then & (st,\ (GiveData\ dt,\ errs)) \\
 & else & (upd\ (query,\ (dt,\ errs),\ st),\ ([],\ []))
\end{array}
$$

# 8 CRITICAL PROPERTIES

We can now attempt to state critical requirements on the main subsystems of the architectural model of [6] by attempting to bound the allowable untrusted queries which are executed. Since we are currently only concerned with the *SELECT* query, only condition *subsys_secureE* from [6] is relevant.

First of all, we need to specify an analogue of *hide* for derived tables in order to state the information flow constraints which apply to computations on them.

HOL Constant

$$
\mathbf{HideDerTableRow} : Class \to DerTableRow \to DerTableRow
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\forall cc\ r \bullet\ HideDerTableRow\ cc\ r} \\
= & let & d = ValuedItemItem(MkValuedItem\ sterling\ dummyVal) \\
 & in\ let & hc\ (c,\ i) = if\ cc\ dominates\ c\ then\ (c,\ i)\ else\ (c,\ d) \\
 & in & MkDerTableRow \\
 & & (DTR\_where\ r) \\
 & & (DTR\_row\ r) \\
 & & (Map\ hc\ (DTR\_cols\ r))
\end{array}
$$

Note that in the following, rows which the client is not cleared to evaluate the *WHERE* clause are left in. Although such rows are eliminated by the Output Filter they may occur as intermediate results in the table computations of [10], e.g., in inner *SELECT*s.

HOL Constant

$$
\mathbf{HideDerTableData} : Class \to DerTableRow\ LIST \to DerTableRow\ LIST
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\forall cc\ rs \bullet HideDerTableData\ cc\ rs} \\
= & let & okr = \{r | cc\ dominates\ DTR\_row\ r\} \\
 & in & Map\ (HideDerTableRow\ cc)\ (rs \upharpoonright okr)
\end{array}
$$

HOL Constant

$$\mathbf{HideDerTable} : Class \to DerTable \to DerTable$$

$$\forall cc\ t\bullet\ HideDerTable\ cc\ t$$
$$=\quad MkDerTable\ (DT\_spec\ t)\ (HideDerTableData\ cc\ (DT\_rows\ t))$$

Given a function, $f$, on lists of derived tables, let us say that the *risk inputs* of $f$ at a given class, $c$ are those inputs for which $f$ reveals information which should not be visible at class $c$:

HOL Constant

$$\mathbf{RiskInputs} : Class \to (DerTable\ LIST \to DerTable \times Errors) \to (DerTable\ LIST\ \mathbb{P})$$

$$\begin{array}{lll}
\forall c\ f\bullet & & \\
RiskInputs\ c\ f & & \\
= & & \\
\{ & ts & \\
\mid & \exists ts_0\ \bullet & \\
 & Map\ (HideDerTable\ c)\ ts_0 = Map\ (HideDerTable\ c)\ ts & \\
\wedge & ( & \neg HideDerTable\ c\ (Fst(f\ ts_0)) = HideDerTable\ c\ (Fst(f\ ts)) \\
 & \vee & \neg Snd\ (f\ ts_0) = Snd\ (f\ ts))\}
\end{array}$$

We can now assert the critical requirements on the *SELECT* query processing (corresponding to *subsys_secureE* of [6]). using the following auxiliary:

HOL Constant

$$\begin{array}{lll}
\mathbf{ConditionE} & : & (Query \to (DerTable\ LIST \to DerTable \times Errors)) \\
 & \to & Class \\
 & \to & ((Query \times (Query + ONE) \times {'}PARS) + Errors)\ \mathbb{P}
\end{array}$$

$$\begin{array}{lllll}
\forall compile\ cc\bullet & & & & \\
ConditionE\ compile\ cc & & & & \\
= & & & & \\
\{ & stp_res & & & \\
\mid & isError\ stp_res\ \vee & & & \\
 & let & (dq,\ ocq,\ pars) = destVal\ stp_res & & \\
 & in\ let & dcomp = compile\ dq & & \\
 & in & \forall ri\bullet & ri \in RiskInputs\ cc\ dcomp & \\
 & & \Rightarrow & IsL\ ocq & \\
 & & \wedge & is_select(OutL\ ocq) & \\
 & & \wedge & let & ccomp = compile\ (OutL\ ocq) \\
 & & & in & \neg DT_rows(Fst\ (ccomp\ ri)) = [] \\
 & & & \vee & \neg Snd\ (ccomp\ ri) = []\}
\end{array}$$

HOL Constant

$$\begin{array}{lll} \mathbf{STP\_secure\_E} & : & (Query \to (DerTable\ LIST \to DerTable \times Errors)) \\ & \to & (Query,\ 'PARS)\ STP\_TYPE\ \mathbb{P} \end{array}$$

$$\begin{array}{l} \forall compile \bullet \\ STP\_secure\_E\ compile \\ = \\ \{ \quad stp \\ \mid \quad \forall\ q\ c \bullet\ stp(q,\ c) \in ConditionE\ compile\ c\} \end{array}$$

(and similarly, presumably for the other 4 conditions from [6], although something would need to be said about the details of *upd*).

We can now instantiate the generic formulation of the execution model to the particular view and action functions of the previous sections.

HOL Constant

$$\begin{array}{lll} \mathbf{EM_1} & : & (Query \to (DerTable\ LIST \to DerTable \times Errors)) \\ & \to & (Query \times (DerTable \times Errors) \times State_t \to State_t) \\ & \to & (Query,\ State_t)\ DBMS\_TYPE \end{array}$$

$$\begin{array}{ll} & \forall\ compile\ upd \bullet \\ & EM_1\ compile\ upd \\ = & EM\ \ View_t\ compile\ (Act_t\ upd) \end{array}$$

A compiler and associated database update operation are correct with respect to the TSQL semantics if the above construction implements the TSQL semantics as specified by *TSQLtf* in [5].

HOL Constant

$$\begin{array}{lll} \mathbf{Correct\_Compile} & : & ((Query \to (DerTable\ LIST \to DerTable \times Errors)) \\ & \times & (Query \times (DerTable \times Errors) \times State_t \to State_t))\ \mathbb{P} \end{array}$$

$$Correct\_Compile\ = \{(compile,\ upd) \mid EM_1\ compile\ upd\ =\ TSQLtf\}$$

To relate the three-stage model of TSQL execution to the actual system constructed from the TSQL transition function, filter of SSQL transformation processor of [5, 7, 9], it is conjectured in [11], conjecture *EM_SecureE*, that if a compiler and associated database update operation satisfy the above correctness criterion, and if the SSQL Query Transformation Processor lies in the set *STP_secure_E* determined by the compiler, then the system components satisfy property E of [6] with respect to the representation.

Connections between the above notions and the critical properties for the SWORD system as a whole are formalised and discussed in [11].

# 9 CLOSING DOWN

The following ProofPower instruction restores the previous proof context.

SML

```
pop_pc();
```

# 10 THE THEORY fef026

## 10.1 Parents

*fef029*

## 10.2 Children

*fef031 fef032*

## 10.3 Constants

| | |
|---|---|
| **DCS_max** | $DerColSpec \to Class$ |
| **DCS_min** | $DerColSpec \to Class$ |
| **DCS_name** | $DerColSpec \to Col\ LIST$ |
| **MkDerColSpec** | $Col\ LIST \to Class \to Class \to DerColSpec$ |
| **DTS_colSpecs** | $DerTableSpec \to DerColSpec\ LIST$ |
| **DTS_maxRow** | $DerTableSpec \to Class$ |
| **DTS_name** | $DerTableSpec \to Col\ LIST$ |
| **MkDerTableSpec** | $Col\ LIST \to Class \to DerColSpec\ LIST \to DerTableSpec$ |
| **DTR_cols** | $DerTableRow \to (Class \times ValuedItem\ OPT)\ LIST$ |
| **DTR_row** | $DerTableRow \to Class$ |
| **DTR_where** | $DerTableRow \to Class$ |
| **MkDerTableRow** | $Class \to Class \to (Class \times ValuedItem\ OPT)\ LIST \to DerTableRow$ |
| **DT_rows** | $DerTable \to DerTableRow\ LIST$ |
| **DT_spec** | $DerTable \to DerTableSpec$ |
| **MkDerTable** | $DerTableSpec \to DerTableRow\ LIST \to DerTable$ |
| **EM** | $(State_t \to DerTable\ LIST) \to (Query \to DerTable\ LIST \to DerTable \times Errors) \to (Query \times (DerTable \times Errors) \times State_t \to State_t \times ANSWER) \to tf_t$ |
| **StateDirs** | $State \to (Class \times Col \times Directory)\ LIST$ |
| **DirTables** | $Directory \to (Op \times TableSpec)\ LIST$ |
| **StateTables** | $State \to (Class \times Col \times Op \times TableSpec)\ LIST$ |
| **$\mathbf{ColSpec_d}$** | $Col \times Op \times TableSpec \to ColSpec \to DerColSpec$ |
| **$\mathbf{TableSpec_d}$** | $Class \times Col \times Op \times TableSpec \to DerTableSpec$ |
| **$\mathbf{TableRow_d}$** | $TableSpec \to Row \to DerTableRow$ |
| **$\mathbf{Table_d}$** | $Class \times Col \times Op \times TableSpec \to DerTable$ |
| **$\mathbf{View_s}$** | $State \to DerTable\ LIST$ |
| **$\mathbf{View_t}$** | $State_t \to DerTable\ LIST$ |
| **GiveData** | $DerTable \to Select$ |

**is_select** $Query \to Bool$

**Act_t** $(Query \times (DerTable \times Errors) \times State_t \to State_t)$
$\to Query \times (DerTable \times Errors) \times State_t$
$\to State_t \times ANSWER$

**HideDerTableRow**
$Class \to DerTableRow \to DerTableRow$

**HideDerTableData**
$Class \to DerTableRow\ LIST \to DerTableRow\ LIST$

**HideDerTable** $Class \to DerTable \to DerTable$

**RiskInputs** $Class$
$\to (DerTable\ LIST \to DerTable \times Errors)$
$\to DerTable\ LIST\ \mathbb{P}$

**ConditionE** $(Query \to DerTable\ LIST \to DerTable \times Errors)$
$\to Class$
$\to (Query \times Query\ OPT \times 'PARS)\ RESULT\ \mathbb{P}$

**STP_secure_E** $(Query \to DerTable\ LIST \to DerTable \times Errors)$
$\to (Query, 'PARS)\ STP\_TYPE\ \mathbb{P}$

**EM_1** $(Query \to DerTable\ LIST \to DerTable \times Errors)$
$\to (Query \times (DerTable \times Errors) \times State_t$
$\to State_t)$
$\to tf_t$

**Correct_Compile**
$(Query \to DerTable\ LIST \to DerTable \times Errors)$
$\leftrightarrow (Query \times (DerTable \times Errors) \times State_t$
$\to State_t)$

## 10.4 Types

**DerColSpec**
**DerTableSpec**
**DerTableRow**
**DerTable**

## 10.5 Definitions

**DerColSpec** $\vdash \exists\ f \bullet\ TypeDefn\ (\lambda\ x \bullet\ true)\ f$
**MkDerColSpec**
**DCS_name**
**DCS_min**
**DCS_max** $\vdash \forall\ t\ x1\ x2\ x3$
$\bullet\ DCS\_name\ (MkDerColSpec\ x1\ x2\ x3) = x1$
$\land\ DCS\_min\ (MkDerColSpec\ x1\ x2\ x3) = x2$
$\land\ DCS\_max\ (MkDerColSpec\ x1\ x2\ x3) = x3$
$\land\ MkDerColSpec$
$(DCS\_name\ t)$
$(DCS\_min\ t)$
$(DCS\_max\ t)$
$= t$

**DerTableSpec** $\vdash \exists f \bullet \; TypeDefn\ (\lambda\ x \bullet\ true)\ f$
**MkDerTableSpec**
**DTS_name**
**DTS_maxRow**
**DTS_colSpecs** $\vdash \forall\ t\ x1\ x2\ x3$

$$\begin{aligned}&\bullet\ DTS\_name\ (MkDerTableSpec\ x1\ x2\ x3) = x1\\ &\quad \wedge DTS\_maxRow\ (MkDerTableSpec\ x1\ x2\ x3) = x2\\ &\quad \wedge DTS\_colSpecs\ (MkDerTableSpec\ x1\ x2\ x3) = x3\\ &\quad \wedge MkDerTableSpec\\ &\qquad (DTS\_name\ t)\\ &\qquad (DTS\_maxRow\ t)\\ &\qquad (DTS\_colSpecs\ t)\\ &\quad\ = t\end{aligned}$$

**DerTableRow** $\vdash \exists f \bullet \; TypeDefn\ (\lambda\ x \bullet\ true)\ f$
**MkDerTableRow**
**DTR_where**
**DTR_row**
**DTR_cols** $\vdash \forall\ t\ x1\ x2\ x3$

$$\begin{aligned}&\bullet\ DTR\_where\ (MkDerTableRow\ x1\ x2\ x3) = x1\\ &\quad \wedge DTR\_row\ (MkDerTableRow\ x1\ x2\ x3) = x2\\ &\quad \wedge DTR\_cols\ (MkDerTableRow\ x1\ x2\ x3) = x3\\ &\quad \wedge MkDerTableRow\\ &\qquad (DTR\_where\ t)\\ &\qquad (DTR\_row\ t)\\ &\qquad (DTR\_cols\ t)\\ &\quad\ = t\end{aligned}$$

**DerTable** $\vdash \exists f \bullet \; TypeDefn\ (\lambda\ x \bullet\ true)\ f$
**MkDerTable**
**DT_spec**
**DT_rows** $\vdash \forall\ t\ x1\ x2$

$$\begin{aligned}&\bullet\ DT\_spec\ (MkDerTable\ x1\ x2) = x1\\ &\quad \wedge DT\_rows\ (MkDerTable\ x1\ x2) = x2\\ &\quad \wedge MkDerTable\ (DT\_spec\ t)\ (DT\_rows\ t) = t\end{aligned}$$

**EM** $\vdash \forall\ view\ compile\ act\ query\ st$

$$\begin{aligned}&\bullet\ EM\ view\ compile\ act\ (query,\ st)\\ &\quad = (let\ compute = compile\ query\\ &\qquad in\ let\ viewed = view\ st\\ &\qquad\quad in\ let\ computed = compute\ viewed\\ &\qquad\qquad in\ act\ (query,\ computed,\ st))\end{aligned}$$

**StateDirs** $\vdash ConstSpec$

$$\begin{aligned}&(\lambda\ StateDirs'\\ &\quad \bullet \forall\ s\\ &\qquad \bullet\ Elems\ (StateDirs'\ s)\\ &\qquad\quad = \{(c,\ i,\ d)\\ &\qquad\quad |(i,\ d) \in repState\ s\\ &\qquad\qquad \wedge c = Dir\_exist\ d\ lub\ Dir\_class\ d\})\\ &StateDirs\end{aligned}$$

**DirTables** $\vdash ConstSpec$

$$
\begin{array}{ll}
 & (\lambda\ DirTables' \\
 & \quad \bullet\ \forall\ d \bullet\ Elems\ (DirTables'\ d) = Dir\_tables\ d) \\
 & DirTables \\
\textbf{StateTables} & \vdash \forall\ s \\
 & \quad \bullet\ StateTables\ s \\
 & \qquad = (let\ (cl,\ illdl) = Split\ (StateDirs\ s) \\
 & \qquad\quad in\ let\ (ill,\ dl) = Split\ illdl \\
 & \qquad\qquad in\ let\ itll = Map\ DirTables\ dl \\
 & \qquad\qquad\quad in\ let\ f\ c\ i\ it = (c,\ i,\ it) \\
 & \qquad\qquad\qquad in\ let\ g\ (c,\ i,\ its) = Map\ (f\ c\ i)\ its \\
 & \qquad\qquad\qquad\quad in\ let\ h\ cl\ is\ itss \\
 & \qquad\qquad\qquad\qquad\quad = Map \\
 & \qquad\qquad\qquad\qquad\qquad g \\
 & \qquad\qquad\qquad\qquad\qquad (Combine\ cl\ (Combine\ is\ itss)) \\
 & \qquad\qquad\qquad\qquad in\ Flat\ (h\ cl\ ill\ itll)) \\
\textbf{ColSpec}_{\mathbf{d}} & \vdash \forall\ d\_name\ t\_name\ t\ cs \\
 & \quad \bullet\ ColSpec_d\ (d\_name,\ t\_name,\ t)\ cs \\
 & \qquad = (let\ cc = TS\_cons\ t\ @\ CS\_consGroup\ cs \\
 & \qquad\quad in\ let\ tc\_name = [t\_name;\ CS\_ide\ cs] \\
 & \qquad\qquad in\ MkDerColSpec \\
 & \qquad\qquad\quad [[CS\_ide\ cs];\ tc\_name;\ d\_name\ @\ tc\_name] \\
 & \qquad\qquad\quad (CS\_min\ cs) \\
 & \qquad\qquad\quad (CS\_max\ cs)) \\
\textbf{TableSpec}_{\mathbf{d}} & \vdash \forall\ c\ d\_name\ t\_name\ t \\
 & \quad \bullet\ TableSpec_d\ (c,\ d\_name,\ t\_name,\ t) \\
 & \qquad = MkDerTableSpec \\
 & \qquad\quad [[t\_name];\ d\_name\ @\ [t\_name]] \\
 & \qquad\quad (TS\_maxRow\ t) \\
 & \qquad\quad (RelList \\
 & \qquad\qquad\quad (Squash \\
 & \qquad\qquad\qquad \{(n,\ cs) \\
 & \qquad\qquad\qquad\quad | \exists\ cs' \\
 & \qquad\qquad\qquad\qquad \bullet\ cs' \in TS\_colspecs\ t \\
 & \qquad\qquad\qquad\qquad\quad \wedge\ cs \\
 & \qquad\qquad\qquad\qquad\qquad = ColSpec_d \\
 & \qquad\qquad\qquad\qquad\qquad\quad (d\_name,\ t\_name,\ t) \\
 & \qquad\qquad\qquad\qquad\qquad\quad cs' \\
 & \qquad\qquad\qquad\qquad\quad \wedge\ n = CS\_posn\ cs'\})) \\
\textbf{TableRow}_{\mathbf{d}} & \vdash \forall\ t\ r \\
 & \quad \bullet\ TableRow_d\ t\ r \\
 & \qquad = (let\ f\ d = (Dat\_class\ d,\ Dat\_item\ d) \\
 & \qquad\quad in\ MkDerTableRow \\
 & \qquad\qquad lattice\_bottom \\
 & \qquad\qquad (R\_exist\ r) \\
 & \qquad\qquad (RelList \\
 & \qquad\qquad\qquad (Squash \\
 & \qquad\qquad\qquad\quad \{(n,\ ic) \\
 & \qquad\qquad\qquad\qquad | n \in Dom\ (R\_data\ r)
\end{array}
$$

$\land\ ic = f\ (R\_data\ r\ @\ n)\})))$

**$\mathbf{Table_d}$** $\vdash \forall\ c\ d\_name\ t\_name\ t$
$\bullet\ Table_d\ (c,\ d\_name,\ t\_name,\ t)$
$= MkDerTable$
$(TableSpec_d\ (c,\ d\_name,\ t\_name,\ t))$
$(Map\ (TableRow_d\ t)\ (TS\_rows\ t))$

**$\mathbf{View_s}$** $\vdash \forall\ s\bullet\ View_s\ s = Map\ Table_d\ (StateTables\ s)$

**$\mathbf{View_t}$** $\vdash ConstSpec$
$(\lambda\ View'_t$
$\bullet \forall\ s_t\ s_s$
$\bullet\ s_t = reprState\ s_s$
$\Rightarrow View'_t\ s_t = View_s\ s_s)$
$View_t$

**GiveData** $\vdash \forall\ dt$
$\bullet\ GiveData\ dt$
$= (let\ class\_item\ c$
$= ValuedItemItem$
$(MkValuedItem\ sterling\ (ClassVal\ c))$
$in\ let\ item\_data\ i = MkData\ lattice\_bottom\ i$
$in\ let\ class\_data\ c$
$= item\_data\ (class\_item\ c)$
$in\ let\ cell\_cols\ (c,\ i)$
$= [class\_data\ c;\ item\_data\ i]$
$in\ let\ row\_data\ r$
$= Flat$
$(Cons$
$[class\_data\ (DTR\_where\ r);$
$class\_data\ (DTR\_row\ r)]$
$(Map\ cell\_cols\ (DTR\_cols\ r)))$
$in\ Map\ row\_data\ (DT\_rows\ dt))$

**is_select** $\vdash \forall\ q\bullet\ is\_select\ q \Leftrightarrow (\exists\ t\bullet\ q = select\ t)$

**$\mathbf{Act_t}$** $\vdash \forall\ upd\ query\ dt\ errs\ st$
$\bullet\ Act_t\ upd\ (query,\ (dt,\ errs),\ st)$
$= (if\ \neg\ errs = []$
$then\ (st,\ [],\ errs)$
$else\ if\ is\_select\ query$
$then\ (st,\ GiveData\ dt,\ errs)$
$else\ (upd\ (query,\ (dt,\ errs),\ st),\ [],\ []))$

**HideDerTableRow**
$\vdash \forall\ cc\ r$
$\bullet\ HideDerTableRow\ cc\ r$
$= (let\ d$
$= ValuedItemItem$
$(MkValuedItem\ sterling\ dummyVal)$
$in\ let\ hc\ (c,\ i)$
$= (if\ cc\ dominates\ c$
$then\ (c,\ i)$
$else\ (c,\ d))$

$$\begin{array}{l}
\quad in\ MkDerTableRow \\
\quad\ (DTR_where\ r) \\
\quad\ (DTR_row\ r) \\
\quad\ (Map\ hc\ (DTR_cols\ r)))
\end{array}$$

**HideDerTableData**

$$\begin{array}{l}
\vdash \forall\ cc\ rs \\
\quad \bullet\ HideDerTableData\ cc\ rs \\
\qquad = (let\ okr = \{r | cc\ dominates\ DTR_row\ r\} \\
\qquad\ in\ Map\ (HideDerTableRow\ cc)\ (rs \upharpoonright okr))
\end{array}$$

**HideDerTable**

$$\begin{array}{l}
\vdash \forall\ cc\ t \\
\quad \bullet\ HideDerTable\ cc\ t \\
\qquad = MkDerTable \\
\qquad\ (DT_spec\ t) \\
\qquad\ (HideDerTableData\ cc\ (DT_rows\ t))
\end{array}$$

**RiskInputs**

$$\begin{array}{l}
\vdash \forall\ c\ f \\
\quad \bullet\ RiskInputs\ c\ f \\
\qquad = \{ts \\
\qquad | \exists\ ts_0 \\
\qquad\quad \bullet\ Map\ (HideDerTable\ c)\ ts_0 \\
\qquad\qquad = Map\ (HideDerTable\ c)\ ts \\
\qquad\quad\ \wedge (\neg\ HideDerTable\ c\ (Fst\ (f\ ts_0)) \\
\qquad\qquad\quad = HideDerTable\ c\ (Fst\ (f\ ts)) \\
\qquad\qquad \vee \neg\ Snd\ (f\ ts_0) = Snd\ (f\ ts))\}
\end{array}$$

**ConditionE**

$$\begin{array}{l}
\vdash \forall\ compile\ cc \\
\quad \bullet\ ConditionE\ compile\ cc \\
\qquad = \{stp_res \\
\qquad | isError\ stp_res \\
\qquad\quad \vee (let\ (dq,\ ocq,\ pars) = destVal\ stp_res \\
\qquad\quad\ in\ let\ dcomp = compile\ dq \\
\qquad\quad\quad in\ \forall\ ri \\
\qquad\quad\quad \bullet\ ri \in RiskInputs\ cc\ dcomp \\
\qquad\qquad\quad \Rightarrow IsL\ ocq \\
\qquad\qquad\qquad \wedge\ is_select\ (OutL\ ocq) \\
\qquad\qquad\qquad \wedge (let\ ccomp = compile\ (OutL\ ocq) \\
\qquad\qquad\qquad\quad in\ \neg\ DT_rows\ (Fst\ (ccomp\ ri)) \\
\qquad\qquad\qquad\qquad = [] \\
\qquad\qquad\qquad\quad \vee \neg\ Snd\ (ccomp\ ri) = []))\}
\end{array}$$

**STP_secure_E**

$$\begin{array}{l}
\vdash \forall\ compile \\
\quad \bullet\ STP_secure_E\ compile \\
\qquad = \{stp | \forall\ q\ c \bullet\ stp\ (q,\ c) \in ConditionE\ compile\ c\}
\end{array}$$

**EM₁**

$$\begin{array}{l}
\vdash \forall\ compile\ upd \\
\quad \bullet\ EM_1\ compile\ upd = EM\ \ View_t\ compile\ (Act_t\ upd)
\end{array}$$

**Correct_Compile**

$$\begin{array}{l}
\vdash Correct_Compile \\
\quad = \{(compile,\ upd) | EM_1\ compile\ upd = TSQLtf\}
\end{array}$$

# 11 INDEX