

Using assignment constraints to implement security rules of arbitrary complexity.

The present document investigates the possibilities to use a “novel” (insofar as this term is really warranted) type of (database-related) constraint, for the purpose of implementing (support for) security rules of arbitrary complexity.

Security – a definition.

To speak of “security rules” in a database context will first require a formal definition of what the term “security rule” does and does not cover. In its most general meaning, the term “security” covers something that can be somewhat informally defined as “identifying all the risks that an organization is exposed to, identifying the possible measures that can be taken for eliminating or reducing those risks, and implementing those measures”. Note that this includes risks such as “business discontinuity as a consequence of the entire board of directors getting killed in an airplane crash”. It is clear that we are not concerned with such kinds of risk, and that we will only be concerned with what could be termed “information security”, that is, security matters relating to the information (/data) that is kept by some organization. Even so, there are still two “major kinds of risk”, and those are quite distinct, that relate to business information: one is the kind of risks associated with “loss of data” (original documents getting lost in a fire, computer disks becoming unreadable, ...), and the other one is risks associated with “unauthorized access to data”. For the purpose of this document, we will only be dealing with “unauthorized access to data”.

As a general principle, security systems covering the risks of “unauthorized access”, build on three distinct concepts: objects, subjects and usage rules¹. Objects are the things that are managed by the secured system (database relvars, OS files, ...). Subjects are the “actors” (persons, external automated systems, ...) who initiate actions in the secured system that have an impact on the secured objects. Usage rules are the rules that determine whether or not such an action, initiated by some subject and impacting some object(s) within the secured system, is authorized.

In a database context, we can “translate” this to be a bit more specific, and state that database security rules are rules that determine whether or not some access to some relvar/data in the database, initiated by some subject, is authorized. “Some access” can clearly be either “write” access (any combination of inserts and deletes) or “read” access. These two cases will be dealt with separately. Note in particular the “initiated by some subject” part: a security rule always pertains to some identifiable user who is trying to achieve something wrt the data.

¹© Alfons Heirbaut, security officer at one of my former employers.

A new class of constraint.

Constraints can be categorized following several distinct schemes. Familiar “types” of constraint are type constraints, relvar constraints, database constraints, tuple constraints and transition constraints. Type constraints are typically defined as a constraint that involves only a single attribute in a single tuple in some relvar. Tuple constraints are typically defined as constraints that involve only a single tuple in some relvar. Relvar constraints are typically defined as constraints that involve only a single relvar in the database, but possibly more than one tuple of that relvar (keys are obviously the best example). Database constraints are typically defined as “all the rest, so long as the constraint involves only a single database state”. Transition constraints are typically not so formally defined, except for formulations such as “Transition constraints constrain which transitions between database states can be accomplished by any database transaction”.

For the purposes of the present document, we introduce the term “Assignment constraint” and define it as “Any constraint whose outcome cannot be determined by evaluating just a single database state (/value)”.

Transition constraints as typically defined, depend on two distinct database values for their outcome : the pre-update and the post-update value. This satisfies our criterion of “not being determined by a single database value”, so all transition constraints in this sense are assignment constraints by our definition. It is less clear whether all assignment constraints are also transition constraints. A constraint that takes into account the value of the current system date (“the birth date of any newly registered person must be earlier than the current system date”), or one that takes into account the identity of the party owning the transaction within which an update statement is attempted (enforcement of security rules of arbitrary complexity) might be regarded as not being a genuine transition constraint. We therefore prefer to adopt our newly introduced term for the concept, which at any rate we believe to be a better description of what it really is than “transition constraint”.

A model for database updates.

In order to propose a model for enforcing “security rules concerning database updates of arbitrary complexity”, we will need a formal model for “a database update”. One particular approach for doing this, is to define update statements as being assignments. With any update to some relvar, what happens conceptually is that that relvar is being given another value, one that consists of exactly the tuples that were already present, plus the tuples added and minus the tuples deleted. It should be obvious that no DBMS will ever actually apply this assignment as such to obtain the desired result. Rather, a DBMS will only add those tuples that actually need to be added, and delete only those that actually need to be deleted. So at the implementation level of a DBMS, an update operation on a relvar looks more like “insertions and deletions to be applied”, than it looks like “relvar names and values to be assigned to them”.

TTM defines the concept of “multiple assignment”. In the model for updates that we follow in this paper, this concept translates to an update statement having to be modeled as a set of “insertions and deletions”, where each of the “insertions and deletions” is marked/tagged with the name of the relvar to which they must be applied. A relational way to represent any database update operation then, is the following:

An update statement on a database is a **tuple**, whose attribute names are the **names of the relvars** to which updates should be applied, and whose attribute values are themselves **tuples of degree two**, where both attributes in such tuples are **relation-typed**, with one relation holding the tuples that the statement will be attempting to **insert** into the concerned relvar, and the other relation the obvious counterpart, and where those two relations are **disjoint**.

By way of example, the following tuple (of degree one) models an update statement that attempts to modify the name "Hugh Darwin" to "Hugh Darwen" in some tuple of a relvar named AUTHOR, with heading {AUTH#:... NAME:...}:

```

+-----+
! AUTHOR                                     !
+-----+
! +-----+ +-----+ !
! ! INSERTS                ! DELETES                ! !
! +-----+ +-----+ !
! ! +-----+ +-----+ ! +-----+ +-----+ ! !
! ! ! AUTH#    ! NAME                ! ! ! AUTH#    ! NAME                ! ! !
! ! +-----+ +-----+ ! +-----+ +-----+ ! !
! ! ! 1        ! Hugh Darwen          ! ! ! 1        ! Hugh Darwin          ! ! !
! ! +-----+ +-----+ ! +-----+ +-----+ ! !
! +-----+ +-----+ !
+-----+

```

This approach can obviously also model insert-only or delete-only types of update, using empty relations, and in just the same way, empty relations can be used to extend this "statement tuple" with attribute values for all relvars in the database to which no update was attempted. This helps establishing a potentially useful property of this "statement tuple": we can safely assume that its degree is always equal to the number of distinct base relvars in the database (and thus that its degree is equal to the degree of the database tuplevar).

Now, if this "statement tuple" is given some name, then we could usefully reference it in expressions. Where needed, this paper will further assume the name "STATEMENT" for this tuple. In so doing, we have created the possibility to define expressions such as (in **Tutorial D** syntax):

```

AUTHOR FROM STATEMENT
INSERTS FROM AUTHOR FROM STATEMENT

```

The former is a tuple-valued expression, which yields the tuple holding both the inserted relation and the deleted relation for database relvar AUTHOR. The latter expression yields the relation holding all the tuples that the issuing statement attempts to insert into relvar AUTHOR.

Now that we have established this formal model for a database update statement, another small side note regarding the transition/assignment constraint thing: a transition constraint is most naturally seen as some (boolean) function $g(D',D)$, in which D' and D denote the new and old database values, respectively. Assignment constraints can be most naturally seen as a function $h(D,S)$, in which D denotes the old database value (that is, the current database value at the time of constraint checking), and S denotes our STATEMENT tuple. To show that the case for introducing the term 'assignment constraint' is merely a matter of whether or not certain types of (contextual) information can be referenced by it (or not), observe the following equivalences between D , D' and S :

- $D' = D \text{ dbadd } S$
- $D = D' \text{ dbundo } S$
- $S = D' \text{ dbdelta } D$

In these, `dbadd` is the 'database addition' operator, which adds all `INSERTS FROM ... FROM STATEMENT` and removes all `DELETES FROM ... FROM STATEMENT` to/from the corresponding targets, `dbundo` is the operator that performs the opposite (adds all `DELETES FROM ... FROM STATEMENT` and removes all `INSERTS FROM ... FROM STATEMENT` to/from the corresponding targets), and `dbdelta` is obviously an operator to compute a `STATEMENT` tuple from two distinct database values. Iow, `dbadd` is an operator that performs 'database update' as typically implemented, and `dbdelta` is an operator that might typically be needed by systems that support direct relational assignment. Because of the first one, any $g(D', D)$ is identical to $g(\text{dbadd}(D, S), D)$, which is itself clearly "just some function $h(D, S)$ ". However, the view of ' $h(D, S)$ ' may be much more practical and/or intuitive for implementation purposes.

Attempted updates and actual updates

One issue that always surfaces in the context of update statements, is whether or not it should be considered an error if an attempt is made to add a tuple that is already present². Since it is hard for any DBMS to decide this issue, in general, on behalf of any and all users, the most elegant solution is for the DBMS to give the user the option. This implies that insert "comes in two distinct flavours": one that allows tuples-to-be-inserted to exist already, and one that will reject such inserts. In the "flavour" that allows said situations, the tuples that actually **will be** inserted, might be a proper subset of the tuples that the statement is **attempting** to insert.

It can be observed that the expression denoting the **actual insertions** and the expression denoting the **actual deletions** are, respectively³:

```
(INSERTS FROM <relvarName> FROM STATEMENT) MINUS <relvarName>.
(DELETES FROM <relvarName> FROM STATEMENT) INTERSECT <relvarName>.
```

²or to delete one that isn't. Anything said about insert here also applies, mutatis mutandis, to delete.

³Note that this is obviously only true "at the beginning" of the statement's execution, i.e. when the inserts and/or deletes concerned have not yet actually been carried out.

For purposes of brevity, we define the following shorthands for these constructs:

Shorthand	Equivalent to
I(<relvarName>)	INSERTS FROM <relvarName> FROM STATEMENT
D(<relvarName>)	DELETES FROM <relvarName> FROM STATEMENT
AI(<relvarName>)	I(<relvarName>) MINUS <relvarName>
AD(<relvarName>)	D(<relvarName>) INTERSECT <relvarName>

One thing to be pointed out explicitly: this definition implies that we would be able to speak of the AI()/AD() of base relvars only. However, it makes equally perfect sense to be able to speak of the AI()/AD() of any virtual relvar, or of the AI()/AD() of just any relational expression, regardless of whether or not such expression actually defines a virtual relvar. Any further mention of AI(...) and/or AD(...) is to be interpreted in that sense : the thing between brackets can always be just any relational expression, and only if that expression is a simple base relvar reference, does the AI(...) and/or AD(...) expression constitute a reference to an attribute in the STATEMENT tuple. In all other cases, these expressions merely denote some relation that can be computed⁴. How so, is, however, not the purpose of this document.

In the context of security rules, the distinction between “attempted updates” and “actual updates” matters, because any user who is aware of the distinction could exploit it to obtain knowledge he might not be entitled to have. If an insert does not fail on a security violation, then the issuing user could conclude that the data he attempted to insert were already present, and this opens up to abusing the database update mechanisms for purposes of unauthorized querying.

This observation seems to suggest that it is probably desirable to have the STATEMENT tuple include not only the actual insertions and deletions, but also the attempted ones.

A note on Update

So far, we have only been speaking of insert and delete as the fundamental types of update. The operation usually called 'update', which takes some tuples out of a relvar and replaces them with some other tuple values, is to be seen as exactly that: the simultaneous appearance of certain tuple inserts alongside certain other tuple deletes in the same relvar.

For the purposes of this paper, there is no benefit to be had in regarding the 'update' update operation as something distinct from insert and delete, neither at the 'model level' of the approach here described, nor at its implementation level.

⁴From the perspective of good language design, such conventions are extremely sloppy. But we are not proposing any formal computing language constructs.

A model for the process of database updating

At this point, it might be useful to pin down precisely the sequence of events, as they are envisaged by this paper, that occur as a consequence of some attempted database update. That sequence of events is as follows:

- Parse, or otherwise interpret, the update request, thereby determining, first the attempted insertions and deletions, and then the $AI() / AD()$ for each base relvar, and preparing the STATEMENT tuple to contain this information.
- Based upon which $I() / D() / AI() / AD()$ are nonempty, determine the set of constraints to be checked. These can include both database constraints and assignment constraints.
- Check each one of that set of constraints, doing all necessary computations using the STATEMENT tuple and the current value of the database.
- If no violations are found, then carry out the actual updates. The fact that the INSERTS / DELETES relations have already been computed and are available through the STATEMENT tuple is of course very useful here.

The first step is the one that will be executing, a.o., the algorithm laid out in prescription 21 of TTM, regarding multiple assignment. Meaning in particular that any concerns about multiple assignment are orthogonal to what is being talked about here. In particular, a multiple assignment that mentions the same assignment target more than once can perfectly be dealt with according to the rules laid out in said prescription, the only thing needed is that at the end of that first step, an accurate account should be available of all the tuples that are actually being deleted/inserted. It might be thought that there could be some ambiguity here concerning the attempted insertions/deletions, but this isn't really the case: if some tuple is "inserted and deleted again" (perhaps even multiple times) due to the "chaining" of updates to the same target, then it is still the case that "at the end of the chain", it will be unambiguously defined whether or not such a tuple really is part of the update attempt or not.

This also explains how 'update-as-a-distinct-type-of-assignment' makes no difference: every update can always equivalently be represented as a combined delete/insert, and the only thing needed, is a detailed account of which those deletes/inserts are.

However, a possible issue with "direct relational assignments" needs to be drawn attention to. The issue being, if a database update is issued under the form of a 'direct' relational assignment, then what exactly are the 'attempted deletes' and 'attempted inserts'? The 'actual' deletes and inserts can be unambiguously computed (if R denotes an 'old' relvar value and R' the 'new' relvar value, then they are $R \text{ MINUS } R'$ and $R' \text{ MINUS } R$, respectively), but this is not the case for the 'attempted' updates. This is problematic because enforcement of security rules had better be based on 'attempted' updates. A 'direct' relational assignment might also be regarded as an attempt to "first make completely empty, then add all the tuples of the new value". That is, the attempted delete is equal to R , and the attempted insert is equal to R' . And the assignment might even be regarded as an attempt to delete U , the universal relation of the applicable relation type. The outcome of evaluating a security rule might be different, depending on the choice made here. The most sensible option, however, seems to be to assume that the attempted updates are equal to the actual updates here, since the other options do not have the property that the attempted inserts and attempted deletes are disjoint.

Note that in “extreme” cases, some “inconsistencies” might arise. Take, e.g., the no-op insert `INSERT R INTO R`. This “clearly” suggests that the “attempted insert” for the target R is the value of R itself. The actual insert for R will be empty, of course. But now observe what happens if we “translate” this insert into its equivalent assignment `: R := R UNION R;`. If we compute the “attempted insertions” from this assignment, in accordance with the conclusion from the preceding paragraph, then we get an “attempted insert”

`(R UNION R) MINUS R = {}!`

The conclusion to be drawn from this inconsistency might be that it is not so good an idea after all to base enforcement of security rules on the attempted updates, and that we should really be using the actual updates, but then it is unclear what can be done about the user abusing the update mechanism to issue queries he wouldn't otherwise be allowed to do.

Also in this step, all transformations needed for dealing with updates to virtual relvars, are expected to be done. TTM defines a detailed proposal for what these transformations might be, but once again, this concern of view updating does not affect our proposed mechanisms in any way, just so long as the mechanisms that handle the “view updating part” see to it that in the end, a detailed account is available of which deletions/insertions must be applied to the actual base relvars. As for the updates that are “attempted” to a base relvar, as a consequence of an assignment to a virtual one, avoiding possible ambiguity (or rather, avoiding non-determinacy or avoiding having to choose between multiple equivalent solutions), will be a matter of carefully (and scrutinously) defining the “translation” from assignments-to-virtual-relvars to assignments-to-base-relvars.

And as a final point, it is also expected that all database updates deriving from declared update triggers, compensatory actions and suchlike, will be computed. Observe that this is in disagreement with the “view updating” chapter in DBE, which states that “all cascades should be done after the update itself has been done”.

When to check a security rule (and when not to)

Step two of this model for the process of database updating is to determine the “set of constraints” that need to be checked, given some particular database update, and with the “set of constraints” possibly including database constraints as well as assignment constraints. For database constraints, this process of determination involves some nontrivial computations, but fortunately this text is not about database constraints.

As for the “set of assignment constraints” to be checked, we assume that “when to check a constraint” is part of the declaration of the constraint. It might be possible for the system to determine this info automatically using just the declaration of the assignment constraint itself, this issue will be revisited. Regardless of whether the “when-to-check” information is provided “manually” by the user or inferred from the declaration by the system, we also assume that the “when-to-check” info is explicitly recorded in the catalog.

A Tutorial D-like syntax for declaring assignment constraints could thus be something like :

```
ASSIGNMENT CONSTRAINT <name or identifier> <boolean expression> CHECK UPON
<assignment constraint check trigger list>

<assignment constraint check trigger> := <update type spec> <base relvar name>

<update type spec> := INSERT TO | DELETE FROM
```

This “when-to-check” info could even be compiled into a module containing database update statements. Of course, when such information is “compiled into” a runnable module, great care must be taken that such a module gets recompiled whenever that is needed, e.g. if a new constraint is defined, then all compiled update statements containing an update operation type that appears in the CHECK UPON list of the new constraint, might need to be recompiled so as to include the new constraint as one to be checked.

Similar observations obviously hold if the <boolean expression> of an assignment constraint is compiled into a “runnable module” containing an update statement, then if that boolean expression references some virtual relvar V, and the relational expression defining V is modified, then this compiled update statements holding this reference to V must be recompiled.

Of course, in this case there is the obvious proviso that it is impossible at compile time to inspect the actual nonemptiness in the STATEMENT tuple, as that requires the specific update request to have been parsed, and matched with the current value of the database. However, what is possible at compile time nonetheless, is to determine the possible nonemptiness of some AD()/AI() in the STATEMENT tuple as it could result from some particular update request. For example, a compiled statement 'INSERT ... INTO V', where V is a virtual relvar defined as 'R1 JOIN R2', could contain the information that some assignment constraint must be checked because of a CHECK UPON INSERT TO R1 defined for it⁵, even if some particular invocation of that compiled insert statement would actually result in no update at all!

A consequence for updating virtual relvars

It must be pointed out that, with said model of database updating process, it is impossible for the portion of the system handling the “view updating” part, to take into account the existence of any constraints and/or their precise nature. That is, there is no possibility for the system to take the declared constraints into account when determining which one of multiple possible “solutions” that exist with “ambiguous” updates (delete through join and insert through union) is the one to be chosen.

For example, an insert-through-union is, by definition, ambiguous, but it becomes unambiguous if it were known that the issuing user does not have any update authority at all on one of the two arguments of the union.

One way to overcome this might be to have step 1 of the updating process compute, not a single STATEMENT tuple, but a whole collection of possible STATEMENT tuples instead, perhaps ordered in descending order of preferability, and have step 2 do the constraint checking on each individual STATEMENT tuple of the collection, until one is found that “passes all tests”.

It will be clear that it would be very desirable to have very advanced heuristics in the computation of the STATEMENT tuples, so as to minimize the workload of the subsequent “find the acceptable one” part of the process, and it will also be clear that such scenario's are nowhere near trivial to implement.

⁵Note the “could”. This approach makes for a kind of “assignment constraint inheritance” for virtual relvars, in that any virtual relvar always “inherits” all the assignment constraint checks defined for the base relvars in terms of which the virtual relvar is defined, but the alternative of NOT having such “automatic inheritance” may be equally viable. The issue also exists for “read” accesses, and will be revisited.

Using assignment constraints for enforcing database update security rules

With all this in place, two claims can now be made regarding the enforcing of “security rules of arbitrary complexity” :

- Assignment constraints as introduced in the former chapters are a necessary and sufficient means to support both the declaration and the enforcement of security rules that are “truly” of arbitrary complexity,
- and no “special” or “specific” difficulties will arise for the implementor when it comes to their enforcement in the most optimal way possible.

To justify the first claim, we first observe the property that a database security rule always involves some kind of user identity. Assignment constraints as defined in this document, are the only type of constraint that can be allowed to include such information, which is contextual (and that explains the 'necessary' part of the claim). The “arbitrary complexity” aspect of a security rule, manifests itself in the fact that any security rule is itself an arbitrary proposition in logic. The expression of these arbitrary propositions is allowed for in our model through the <boolean expression> in the definition of the assignment constraint, because obviously such an expression can also be of arbitrary complexity (and that explains the 'sufficient' part of the claim).

To comment a bit further, contrast this approach with the more 'traditional' approach, which usually consists of providing a few (and usually very small in number) “dedicated” tables in the catalog, and which simply serve to record the data contained in the various GRANT and REVOKE statements issued by the database and/or security administrator. Typically, the latter approach requires the admins to define security rules at the level of each individual users, and with such systems it is typically very difficult, very laborious or even outright impossible to define security rules at the level of, say, the user's department. At any rate, the set of security rules which are expressible/definable/enforceable in the system will always be limited to just those whose propositional structure matches exactly with the propositional structure of the predicates that correspond to the catalog relvars that are offered by the system. The consequence of this is that there will, almost by definition, always exist certain predicates/propositions that are not expressible in this 'more traditional' approach.

To justify the second claim, it suffices to observe that “enforcing the security rule” amounts to nothing more than just evaluating the <boolean expression>, which itself likely involves references to the transaction context (user identity), to the statement tuple (because that's what holds the update the user is attempting), and possibly the current value of the database itself too. Doing this in the most optimal way possible involves nothing more than letting the query optimizer do its optimization work on this boolean expression, just like it does on any other query.

However, there is a little bit more to be said about the nature of the boolean expression, and certain advantages that can be had if certain conventions are adhered to in this respect. To which we will proceed after having given some examples.

Examples

The following examples will be based on the usual suppliers-and-parts database, but the following relvars will also be available:

- The transaction context, the declaration of which would be something like
VAR RELATION {USERID CHAR, SYSDATE DATE} C KEY {};
- An employee relvar:
VAR RELATION {EMPID CHAR, USERID CHAR, DEPTID CHAR} EMP
KEY {EMPID} KEY {USERID};
- A departments relvar:
VAR RELATION {DEPTID CHAR, MGREMPID CHAR} DEPT
KEY {DEPTID} KEY {MGREMPID};
- A salary relvar:
VAR RELATION {EMPID CHAR, SALARY MONEY} SAL
KEY {EMPID};

User HD cannot update the suppliers relvar.

```
IEMPTY((I(SUPPLIER) UNION D(SUPPLIER)) JOIN (C WHERE USERID='HD'))
```

(“If the current user is HD, then his attempts to update SUPPLIER must be none”)

User HD can update suppliers data, but cannot delete any suppliers.

```
IEMPTY((D(SUPPLIER) NOT MATCHING I(SUPPLIER){S#}) JOIN (C WHERE USERID='HD'))
```

(“If the current user is HD, then his attempts to delete anything from SUPPLIER that doesn't have a corresponding replacement, must be none”)

Only staff members of department HR can update the employee relvar (but the manager of that department cannot).

```
IEMPTY((D(EMP) UNION I(EMP)) JOIN (C MATCHING ((EMP WHERE DEPTID <> 'HR') {EMPID DEPTID} UNION (RENAME DEPT ...)))
```

Only staff members of department HR can update the salary relvar (but excluding the tuple that pertains to him/herself).

```
IEMPTY((D(EMP) UNION I(EMP)) JOIN C JOIN ((EMP WHERE DEPTID <> 'HR') {EMPID DEPTID} UNION (RENAME DEPT ...)))
```

A special convention for the boolean expression

As already observed, a security rule is just a rule to the effect that some given database update S (the STATEMENT tuple), initiated by some user U (whose identity is part of the contextual information C), must satisfy some logical predicate. A more convoluted way of saying the same is that applying the negation of that same logical predicate to S and C, must yield an empty result. Sort of saying “since the only acceptable updates are the acceptable ones, there cannot be any unacceptable ones”. Or a bit less ridiculously so, “All attempted updates S must be member of the acceptable ones.”, or yet “The set of attempted updates {S}, minus the set of acceptable ones, must give an empty set.”.

What this boils down to, is that it is always possible to express the <boolean expression> in the form of an invocation of `ISEMPTY(<somerelementaryexpression>)`. This is not to say that the expression will always be simpler, but nonetheless we will assume in what follows that this convention is indeed adopted, that is, it is indeed the case that all assignment constraints are expressed in said form.

The benefit is that when this convention is adopted, it becomes possible for the system to infer from it all the kinds of updates that might cause a violation of the rule, and that it might consequently become superfluous (/unnecessary) for the user to have to explicitly specify the `CHECK UPON` list.

A formal method for computing the set of violating operations

Assume that indeed all assignment constraints are defined as an `ISEMPTY(...)` invocation. Violations of this constraint can thus only occur if some update statement *S* is issued that makes the resulting <relational expression> value of some constraint nonempty. But the complete set of possible causes of this happening can be determined at compile time !

Example

Consider a -rather silly, but don't mind about that- assignment constraint `ISEMPTY((R1 MINUS D(R1)) UNION I(R1))`.⁶ We can list all the possible causes of such an expression not evaluating to the empty relation.

The overall relational expression here, is a relational union. This union being nonempty is possible if either of its arguments is nonempty. The second argument of the union is the attempted inserts to *R1*, thus we know we should be checking this constraint whenever such inserts are attempted. The first argument of the relational union is itself a relational difference. This expression being nonempty is possible only if its minuend is nonempty. The subtrahend doesn't matter, so we know that attempts to delete anything from *R1* cannot possibly cause this constraint to be violated.

Finally, we are left with the case of *R1* itself being nonempty. With the example as specified, this is of course hardly possible, except in the case where the assignment constraint was newly introduced at a time when *R1* was already nonempty. The general problem case we're looking at here is the <relational expression> being nonempty, without this being attributable to any `I(Rx)` and/or `D(Rx)`. Another such example would be a rule to the effect that "department managers are not allowed to update the database". This can be expressed by a constraint `ISEMPTY(C JOIN (RENAME DEPT ...))`. However, this definition does not give us any clue as to when (and when not) to check this constraint, since there are no references at all to any `I(Rx)` and/or `D(Rx)`.

⁶This constraint effectively says that after any update, *R1* must be empty.

General algorithm and relational operator specifics

With the proviso of possible exceptional treatment⁷ needed for cases such as the two just mentioned, this example has unveiled a possible algorithm for computing the complete set of violating operations of any assignment constraint, based upon its relational argument when defined in the form of an `IS_EMPTY(...)` invocation :

For the relational expression that is the argument of the `IS_EMPTY` operator, traverse the parse tree, collecting all references to attributes in the `STATEMENT` tuple, taking note whether they are about insertions (`I (Rx)`) or deletions (`D (Rx)`).

It will be clear that for most invocations of the relational operators that make up the relational expression, it will be the case that all arguments must themselves also be traversed. Only exceptions to this rule are the `MINUS` operator used in the running example, and its nephew `SEMIMINUS`, for which only the minuend argument is to be traversed in search of `STATEMENT` tuple attribute references.

User feedback in the event of violations

There is yet another justification for the choice of expressing assignment constraints in the form of some `ISEMPTY(...)` expression: following that convention will yield, in the event of a violation, and simply as a natural consequence of evaluating the constraint, precisely all the details that the security officer would likely want to see logged.

In theory, the same information could also be used to provide the user with “meaningful feedback” and with very precise details of the reason why his update request was rejected. In the context of enforcing data access rules, however, it may be questionable whether we really want to provide the user with all those details ! Especially if we consider that some of those details might be data values coming from a relvar that that user isn't even allowed to read !

Authorizations for read access

Rules for authorization of read-mode database access are typically defined at the relvar level. This typically implies that a user either has access to the entire (value of the) relvar, or no access at all. More fine-grained security rules (e.g. preventing a user from accessing certain specific attributes, or certain tuples that satisfy some specific condition), must typically be implemented by defining virtual relvars.

E.g., implementing a rule to the effect that user XYZ can read all attributes of the `EMPLOYEE` relvar, except the `SALARY` attribute, must typically be done by (a) defining a virtual relvar `EMPLOYEE_V`, which is the projection on `ALL BUT SALARY` of `EMPLOYEE`, (b) granting user XYZ read access to `EMPLOYEE_V`, and (c) not granting user XYZ read access to `EMPLOYEE` itself.

Likewise, implementing a rule to the effect that user XYZ can read only credit applications for amounts < 5000, must typically be implemented by defining a virtual relvar, which is a restriction⁸ on the actual database relvar that holds all credit applications.

⁷And where the nature of that “exceptional treatment” is still unclear.

These two distinct cases of more fine-grained read access authorization, can be labeled as 'attribute-level security' and 'tuple-level security', respectively. For the purposes of this document, there is little left to say about the case of attribute-level security, except to note that the solution (virtual relvars) might give rise to a problem of view updating, in casu, update-through-projection. Now, if there is a requirement to also grant write access to a user who has(/needs) read access to a proper subset of the attribute set of some database relvar R, then the most likely feasible solution will not be to define R as a database relvar, and the projection of R onto the accessible attributes as a virtual relvar V, but will instead probably be to decompose R into two distinct database relvars, one holding the accessible attributes, the other the inaccessible ones, and (if useful) define a virtual relvar V (probably a join of the two database relvars) holding all the attributes.

⁸But note that the virtual relvar can in fact be just any arbitrary relational expression, that is, it can involve joins and aggregations and what have you. In fact, in a sense, the definition of the virtual relvar itself is also (part of) the definition of the read authorization for the user (and the same holds for the case of 'attribute-level access authorization', where it is the projection attribute list that defines the accessible attributes).