ABSTRACT
Federated databases are useful in data integration, because they can provide data from other databases, without physically merging that data. Prolog-based federated databases are based on predicate logic and use the Prolog system to generate results for queries. In this paper we constructed two formal criterions that express that such a Prolog-based federated database is correct. We distinguish between strong correctness and weak correctness, for which we both provided a criterion. Furthermore, the criterions are based on a formal representation of the Prolog-based federated database in conjunction with a non-Prolog based federated database.

Keywords
Federated database, Prolog, Formal, Correctness, Proof

1. INTRODUCTION
Currently there is a lot of information available through the use of computer systems. For some subjects there are even multiple systems that contain information about that same subject. Typically, those different systems organize their data in different manners. Also, one data source may have information on a specific subject that is not available in another data source.

To retrieve information from one single system is generally easy, because a query can be written based on the data schema, and the result of that query can be computed immediately. To automatically retrieve information from multiple systems containing information about the subject, is a bit more complicated. To compute a result for a query, the relevant information in all the systems should be combined to deliver a 'complete' result.

There are several approaches to solve the problem that arises with different data sources. We will discuss two approaches in the following subsections.

1.1 Super-system
One approach is to merge all the data from the data sources into one 'super' data source, where all the information from the original data sources is available [1]. This is a time consuming operation, because the data in every source has to be processed and then moved to its new location. This would be suitable when two or more systems actually are to be merged into one big super-system, for instance with a corporate fusion. However, when this is not the case, and the original data sources keep changing, the drawback of this approach is that we have to keep merging the data continuously to ensure that the super-system is up to date. This obviously is not desirable, and therefore a more flexible approach is needed.

1.2 Federated database
Another approach is the use of a federated database [2, 3]. A federated database is a virtual database which uses underlying so-called component databases to store the data. It is comparable to a non-materialized view in a database with the underlying table(s) containing the actual data. A federated database has its own schema, so a virtual counterpart of the super-system mentioned in the previous subsection can be created. On the outside it will be identical to that super-system, but under the hood there are significant differences. Because when a query is executed on a federated database, the data has to be pulled from the underlying component databases. This process involves query decomposition, query translation to component schema(s), and merging the results using the federated schema to return this to the querier.

This approach offers the same functionality as the super-system discussed earlier, but it is clear that this is more flexible and easier to update, because no data has to be moved in the process. Also, when new information is inserted in one of the component databases, this is immediately available in the federated database.

Chakravarthy, Whang and Navathe [4, 5, 6] use a logical approach in the design of a federated database. This logical approach means that the data and relations in a data source are represented by predicate logic. The Prolog system is then used to query this set of predicates. This approach is flexible, because it does not rely on a single RDBMS (Relational Database Management System) implementation. It is also extensible, because new facts or more logic can be added to the system on the fly.

But before going further, we will now first give an overview of Prolog.

1.3 Prolog
Prolog is a declarative programming language based on predicate logic. A Prolog program is constructed from facts and rules [7, 8]. A fact is a predicate that holds in all cases. A rule is a predicate that holds when certain given other predicates hold. (Horn clauses) When such a set of facts and rules is created, Prolog queries can be executed on this program. Through the use of unification and backtracking, Prolog then tries to find a solution to the query using the provided facts and rules. Unification binds a free variable to a certain available value. Backtracking is the process where all possible unifications are evaluated until a correct proof is found. This mechanism for finding solutions to a query can be useful in federated databases. This is because Prolog ac-
tually decomposes the query automatically when a rule is based on another rule or fact. When the underlying facts or rules are evaluated the results are combined to give the result for the first rule. So, not only query decomposition is done automatically, but Prolog also takes care of the combining of the results after wards.

1.4 Problem statement
While Chakravarthy et al. presented a method for a federated database with the use of Prolog, no formal proof is given for its correctness. This was also not found in other literature on this topic. However, we find it necessary that system designers do prove that such Prolog-based federated databases behave correctly. But before anything can be proven formally, we have to know what exactly needs to be proven. In other words, how can we express the requirements of such a system in a formal correctness criterion? In this paper we will present such a criterion which can then be used by others to prove – or disprove – the correctness of a Prolog-based federated database.

The paper will further be organized as follows. First, we will formulate the main- and sub-research questions. Then the sub-questions will be answered, which will result in the construction of the formal correctness criterion. After that we will end up with the conclusions and discuss possible future work.

2. RESEARCH QUESTIONS
We will now discuss the research questions on which this research is based. The main research question is:

• What formal criterion expresses that a Prolog-based federated database is correct?

The main research question is decomposed into the following sub-questions:

1. How can Prolog-based federated databases be represented formally?
2. What non-formal requirements should be incorporated in the criterion?
3. How can these results be combined into a formal correctness criterion?

These questions will be answered in the upcoming sections, but first we will discuss some simplifications that we made.

3. SIMPLIFICATIONS
In order to give the criterion in our available amount of time, some simplifications had to be made. That is, the component databases that we use, are just single tables. A federated database with these component databases is then defined by a database view over the source tables. Although these simplifications seem quite radical, the essence of a federated database is still preserved. Our goal is to provide the first step. The definition of a more general criterion is left for future research.

4. FORMAL REPRESENTATION
In this section we will create a formal representation of a Prolog-based federated database. The representation we give here may not be applicable to all cases, but we intend to be as general as possible. First, we will discuss the non-Prolog-based (or conventional) federated database.

4.1 Non-Prolog based federated database
With our simplifications of the previous section in mind, the federated database is a view over multiple tables. To express this more formally, consider two databases \( DB_1, DB_2 \in DBS \). Where \( DBS \) is the collection of all databases. In this case, with our applied simplifications, this means the collection of all database tables and views. Furthermore, let \( VWS \) be the collection of all SQL View definitions. We use a view definition \( \otimes \in VWS \) as an infix operator, such that \( x \otimes y \in DBS \), when \( x, y \in DBS \).

The federated database is then the view over \( DB_1 \) and \( DB_2 \), so:

\[ DB_1 \otimes DB_2 \in DBS \]

The main use of a federated database, however, is querying its content. We will therefore also include this within our formal representation. To do this, let \( QRS \) be the collection of all SQL queries. The query \( Q \in QRS \) we will then use as a function on a database such that \( Q(x) \in DBS \) when \( x \in DBS \). We will denote the result of a query on the federated database with \( DBS \in DBS \).

Querying the federated database is then:

\[ Q(DB_1 \otimes DB_2) = DBS \]

4.2 Prolog-based federated database
Now that we have given a formal representation of a non-Prolog-based federated database, we can extend this representation to the Prolog-based situation. To do this, we have to translate every part in the non-Prolog based definition to Prolog counterparts. The databases \( DB_1, DB_2 \in DBS \) will be translated into Prolog programs, denoted by \( P_1, P_2 \in PPS \) respectively, where \( PPS \) is the collection of all Prolog programs. A Prolog program consists of a set of Prolog facts and/or Prolog rules. Furthermore, the SQL view definition \( \otimes \in VWS \) will also be translated into a Prolog program, denoted by \( \otimes \in PPS \). Like \( \otimes \) we also use \( \oplus \) as an infix operator, but now such that \( x \oplus y \in PPS \), when \( x, y \in PPS \).

The SQL query \( Q \in QRS \) will be translated into a Prolog query denoted by \( PQ \in PPS \), with \( PQS \) as the collection of all Prolog queries. And like \( Q \), we will use the Prolog query \( PQ \) as a function, but now such that \( PQ(x) \in PPS \) when \( x \in PPS \). Finally, the result of that Prolog query \( PQ \) is denoted with \( PFS \in PPS \).

To accomplish these translations, we introduce the following family translation functions:

\[ T_D : DBS \rightarrow PPS \]
\[ T_D(DB_1) = P_1 \in PPS, \text{ where } DB_1 \in DBS. \]

\[ T_V : VWS \rightarrow PPS \]
\[ T_V(\otimes) = \otimes \in PPS, \text{ where } \otimes \in VWS. \]

\[ T_Q : QRS \rightarrow PQS \]
\[ T_Q(Q) = PQ \in PQS, \text{ where } Q \in QRS. \]

The translation functions along with the database and Prolog layer are shown in figure 1. In this figure the non-Prolog-based situation is presented with the top layer of symbols. The layer beneath represents the Prolog counterpart of that system, which is
connected with the upper layer through the use of the translation functions. When the translation functions $T_D$ and $T_V$ are used to translate the databases $DB_1$, $DB_2$ and the view definition $\oplus$, a Prolog program indicated by the larger rectangle is created. This program represents the Prolog-based federated database, and can be queried by Prolog queries. So when the SQL query ($Q$) is translated through the use of $T_Q$, that Prolog query ($P_Q$) will be executed on the Prolog program. This causes Prolog to generate the Prolog result ($P_R$) represented with the bottom right rectangle in the diagram. A Prolog result is in this case – with our simplifications – just a list of Prolog facts.

Figure 1: Overview

4.3 Example
We will now first demonstrate a simple example of the system shown in the overview in figure 1. Consider the following database tables, which will represent $DB_1$ and $DB_2$:

- $db\_person(person\_id : int, name : string)$
- $db\_phone(person\_id : int, number : int)$

With the following content:

<table>
<thead>
<tr>
<th>person_id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>'Bob'</td>
</tr>
<tr>
<td>2</td>
<td>'John'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>person_id</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1234</td>
</tr>
<tr>
<td>2</td>
<td>5678</td>
</tr>
</tbody>
</table>

Furthermore, let the view definition $\oplus$ be defined by:

- create view $db\_fed$ as
  select name, number
  from $db\_person$ ps, $db\_phone$ ph
  where ps.person\_id = ph.person\_id

This creates the view $db\_fed(name : string, number : int)$, which represents the federated database. Let the query $Q$ be defined as:

- select number
  from $db\_fed$
  where name = 'Bob'

This query $Q$, executed on the federated database, will deliver the following database result $DB_R$:

<table>
<thead>
<tr>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
</tr>
</tbody>
</table>

Now, to illustrate the Prolog-based federated database, we translate these parts to Prolog programs. The implementations of the translation functions will be omitted in this paper, because this would be too detailed, and therefore bypass the goal of the example, clarification.

The translation function $T_D$ will translate db\_person and db\_phone into the following Prolog facts ($P_1$ & $P_2$):

- $p\_person(1, 'Bob').$
- $p\_person(2, 'John').$
- $p\_phone(1, 1234).$
- $p\_phone(2, 5678).$

The translation function $T_V$ will translate the create view statement $\oplus$ to the following Prolog rule ($\otimes$):

- $p\_fed(NAME, NUMBER) :-$
  $p\_person(PERSON\_ID, NAME),$
  $p\_phone(PERSON\_ID, NUMBER).$

The uppercased fields denote variables. The common variable PERSON\_ID in $p\_person$ and in $p\_phone$ enforces that the value of that field is the same in both facts. Thereby it represents the where clause in the SQL query $Q$.

The translation function $T_Q$ translates the SQL query $Q$ to the Prolog query ($P_Q$):

- $?- p\_fed('Bob', NUMBER).$

And finally, when this Prolog query $P_Q$ is executed on the Prolog-based federated database, the following result ($P_R$) is generated:

- $p\_fed('Bob', 1234).$

In the resulting fact, the variable NUMBER in the Prolog query is replaced with a possible solution that Prolog found for this variable. When more solutions are available, a fact is given for every solution.

5. REQUIREMENTS OF CORRECTNESS
To declare an implementation of a Prolog-based federated database as correct, certain requirements have to be met. The requirements we use are selected by the author. A possible user of the criterion should thereby wonder if he or she agrees with the chosen requirements, before the criterion is used in a proof.

The main requirement of the system is – of course – that it has to be correct. But what we mean with ‘correct’ should be expressed more precisely. We will interpret ‘correct’ in this paper as follows. Correct means that the system will do exactly what
we expect it to do. In terms of a database system, this means that if we execute a query on well-known data, the result of that query is exactly what we expected it to be. This expected result is the result that we can acquire through the use of a trusted other method. In our case this would be the non-Prolog-based federated database discussed earlier (see top layer in figure 1).

This is the main requirement, but it is not complete yet. Because when a system is correct once, it does not necessary mean it is correct in every case. Thereby we need to extend the requirement to that the system should always do exactly what we expected it to do.

The requirements will then be:

1. The Prolog-based federated database should return a result that is equivalent to the result acquired through the use of the non-Prolog based system, on a given query.
2. Requirement 1 holds for all possible queries on every possible data.

To express this more formally, consider the functions:

- \( \text{prolog} :: \text{QRS, DBS} \rightarrow \text{PPS} \)
- \( \text{nonprolog} :: \text{QRS, DBS} \rightarrow \text{DBS} \)

These functions represent the use of the Prolog-based federated database and the use of the non-Prolog based federated database respectively. So therefore, these functions represent the acquired result and the expected result.

The requirements are then:

- \( \forall \text{Q, DBS} \ (\text{prolog}(Q, DB) \equiv \text{nonprolog}(Q, DB)) \)

When this requirement is met, the system can be considered correct.

6. CORRECTNESS CRITERION

Now that we have given a formal representation of the Prolog-based federated database, and stated the requirements of the system, we can merge those two results into a formal correctness criterion. To do this, we first take a second look at the requirements.

According to requirement 1, the system should return a result that is equivalent to the result acquired by the non-Prolog based federated database. We denoted this result with \( DB_r \) (subsection 4.1). In figure 2 this idea is shown graphically. The Prolog-based federated database is shown as a black box. How the translations or the Prolog program itself works, is irrelevant here. The only thing that matters, is that given the provided input, the generated output is the known database result \( DB_r \).

But because the Prolog-based federated database delivers its results in Prolog facts instead of a database table, in order to check for equality with \( DB_r \), we should also introduce the following inverse translation function:

- \( \text{TP} :: \text{PPS} \rightarrow \text{DBS} \)
  
  This function translates a Prolog result to a database, such that \( \text{TP}(P) \in \text{DBS} \), when \( P \in \text{PPS} \).

However, this inverse function of \( T_P \) only exists if \( T_P \) is injective. That is, if different elements in the database correspond to different elements in the Prolog result.

We therefore distinguish between the situation where \( T_P \) exists and where \( T_P \) does not exists. For the first case we formulated the strong correctness criterion. This is presented in the next subsection. For the second case we formulated an alternative criterion. This criterion we call the weak correctness criterion, and this is treated in the subsection thereafter.

6.1 Strong criterion

The strong criterion assumes that \( T_P \) exists. Therefore we can apply this function to the Prolog result \( P_B \) from our Prolog-based federated database. If we look again at figure 1, this would be an upward arrow from the Prolog result \( P_B \). When the system is correct, this arrow leads to the same database result which is already present in the diagram. That is, the database result we would acquire if only the non-Prolog-based federated database was used \( (DB_R) \).

The Prolog-based federated database extended with the \( T_P \) function on \( P_B \) then fits in the blackbox in figure 2. This verifies that this test is in line with requirement 1.

We define the Prolog-based federated database, together with the translation functions \( (T_D, T_V, T_P) \) strong correct, when the following is true:

- \( \forall \text{DBS, DBS', DB} \ (DBS' = T_D(P_B)) \)

Where:

- \( DB_R = Q(DB_1 \oplus DB_2) \)
- \( P_B = PQ(P_1 \oplus P_2) \)

and

- \( P_1 = T_D(DB_1) \)
- \( P_2 = T_D(DB_2) \)
- \( P_B = T_D(DB_R) \)
- \( @ = T_V(@) \)
- \( PQ = T_P(Q) \)
Or with everything combined:

\[
\forall_{DB, DB, Q, Q, R, S, T} Q(DB \oplus DB) = T_P(PQ(T_T(DB) \otimes T_T(DB)))
\]

Where:

- \( \otimes = T_T(\otimes) \)
- \( PQ = T_T(Q) \)

When we denote a ‘translation’ with [...] (without distinguishing between the different kind of translations), and we denote the inverse translation with [...]’, we get:

\[
\forall_{DB, DB, Q, Q, R, S, T} Q(DB \oplus DB)' = [Q( [DB_1] [DB_2] ) ]'
\]

A proof for a specific implementation can now be constructed using this criterion and the induction technique. Induction is used to prove that the criterion holds for all queries and all databases.

### 6.2 Weak criterion

As we have seen in the previous subsection, the strong criterion relies on the existence of the \( T_T \) translation function. Because when this function does not exist, the Prolog result cannot be translated to a database result, and therefore the system does not ‘fit’ into the blackbox of figure 2. However, when this function does not exist, we have found that it is still useful to check correctness in another way. Because when the translation function \( T_T \) is not injective, it does not necessary mean that the whole system is completely incorrect.

Consider for example the following situation. The translation function \( T_T \), translates all elements from the database to Prolog correctly, but it does not create doubles in Prolog. So, when double elements exist in the database, there is only one corresponding element in Prolog. Therefore the translation function \( T_T \) is not injective, and the inverse function \( T_T' \) does not exist. In this case, the system therefore does not comply to the strong criterion. However, if we would accept this behavior, we are still interested in checking if the rest of the Prolog-based federated database works correctly.

To construct the weak criterion we basically lowered our expectations of the system. We no longer demand that the translated Prolog result is equal to the database result. Instead, we require that the Prolog result is equal to the translated database result. This is a weaker criterion because the Prolog result is now compared to a modified database result. This can be done within the boundaries of the requirements, because the equivalence in requirement 1 still holds.

So the expected result in the weak criterion is then the database result \( DB_R \), translated using the original \( T_T \) translation function to represent this result as Prolog facts.

Or more formally:

- \( T_T(DB_R) \)

We define the Prolog-based federated database, together with the translation functions \( T_T, T_V, T_P \) weak correct, when the following is true:

\[
\forall_{DB, DB, Q, Q, R, S, T} T_T(DB_R) = P_R
\]

Where:

- \( DB_R = Q(DB_1 \oplus DB_2) \)
- \( P_R = PQ(P_1 \oplus P_2) \)

and

- \( P_1 = T_P(DB_1) \)
- \( P_2 = T_P(DB_2) \)
- \( \otimes = T_V(\otimes) \)
- \( PQ = T_T(Q) \)

When we denote a ‘translation’ with [...] (without distinguishing between the different kind of translations), we get:

\[
\forall_{DB, DB, Q, Q, R, S, T} T_T(Q(DB_1 \oplus DB_2)) = PQ(T_T(DB_1) \otimes T_T(DB_2))
\]

Where:

- \( \otimes = T_V(\otimes) \)
- \( PQ = T_T(Q) \)

### 6.3 Examples

We will now demonstrate two examples of the use of the correctness criteria.

#### 6.3.1 Example 1

Consider the same example as in subsection 4.3. Furthermore, let \( T_T \) be injective, so that the inverse translation function \( T_T' \) exists. We want to check if the system in the example is strong correct. In order to claim strong correctness, the following should be true:

- \( \forall_{DB, DB, Q, Q, R, S, T} DB_R = T_T'(P_R) \)

From subsection 4.3 we already know that \( DB_R \) is:

<table>
<thead>
<tr>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
</tr>
</tbody>
</table>

We also know that \( P_R \) is:

- \( p\_fed('Bob', 1234) \).
Say now, that $T_D$ translates $P_R$ into:

```
number
1234
```

Now we have shown that in this specific case the system is correct. In order to comply to the strong correctness criterion, we should now prove that the system is correct with every query on every data. The details of that proof by induction are omitted, because therefore we need formal specifications of the translation functions. These specifications are very detailed and implementation specific, and are outside the scope of this paper.

### 6.3.2 Example 2

Consider the same example as in subsection 4.3, except for the translation function $T_V$. Furthermore let $T_D$ be the same as in the previous example. So $T_D$ is injective, and $T_D'$ exists. We, again, want to check if the system is strong correct.

Let $T_V$ be a translation function that translates the view definition $\oplus$ to:

- $p_{\text{fed}}(\text{NAME, NUMBER}) :-$
- $p_{\text{person}}(\text{PERSON_ID, NAME}),$
- $p_{\text{phone}}(\text{ID, NUMBER}).$

The result of the Prolog query $PQ$ will then be defined by:

- $p_{\text{fed}}(\text{’Bob’, 1234}).$
- $p_{\text{fed}}(\text{’Bob’, 5678}).$

When we use the inverse translation function $T_D'$ on this result, we get:

```
number
1234
5678
```

This clearly does not satisfy the condition of the strong correctness criterion:

- $\forall DB_1, DB_2 \in DBS, Q \in QRS DB_R = T_D'(P_R)$

The translation function $T_V$ created the Prolog rule without connecting the tables with the common person_id field. This resulted in a Cartesian product of the two tables. And when querying for Bob’s phone-number, the system incorrectly also returned John’s phone-number as well.

### 7. CONCLUSIONS

We have presented two formal criterions for the correctness of a Prolog-based federated database. We constructed these criterions by first creating a formal representation of a simplified version of a Prolog-based federated database. This formal representation gave a clear view of the situation, and became the basis for the rest of the theory. We continued, by translating non-formal definitions of correctness to formal requirements of the Prolog-based federated database. By combining those results we were able to construct the strong correctness criterion. However, we found that a weaker criterion would also be useful in case we could not use the strong criterion. And so, besides the strong correctness criterion, the weak correctness criterion was introduced as well. For both criterions we provided a description in natural language, as well as a formal definition.

The results we acquired in this research are based on a simplified version of the Prolog-based federated database. That is, only single tables were used as component databases, and a regular view over those tables represented the federated database. Hereby we covered only the essence of proving the correctness of a Prolog-based federated database. Future research could therefore extend our results to a more general correctness criterion which would be applicable to real Prolog-based federated databases.

### Acknowledgments

I would like to thank dr. M.M. Fokkinga for his assistance on formal methods as well as for the general coaching during the research.

### REFERENCES


