(Un)decidability results for trigger design theories

A.P.J.M. Siebes, J.F.P. van den Akker and M.H. van der Voort

Computer Science/Department of Algorithmics and Architecture

CS-R9556 1995

Report CS-R9556
ISSN 0169-118X

(Un)decidability Results for Trigger Design Theories

Arno Siebes
Johan van den Akker
Leonie van der Voort

CWI
P.O.Box 94079, 1090 GB Amsterdam, The Netherlands
e-mail: {*arno,vdakker,leonie*}*@cwi.nl*

Abstract

Active databases are databases extended with production rules or triggers. Triggers have different uses in databases. Originally they were devised as a more flexible method of constraint enforcement. Triggers have found a much wider use however, because it is possible to code all dynamics of an information system in database triggers. Such use results in the presence of large sets of triggers in an active database. The mutual interaction of triggers in a set can lead to undesired results such as non-termination of trigger execution. To enable us to analyse a trigger set in advance for such behaviour we need a design theory for database triggers. In this report we consider a number of predicates on trigger sets. Most important are termination and confluence. We examine the decidability of these predicates in a number of simple trigger languages. We show that these predicates are decidable for very simple triggers that consists of local conditions and replacement of attribute values. We also show that a very small extension of the trigger language, viz. with non-local conditions, makes these predicates undecidable. The extension of the initial language with arithmetic in the form of the successor function preserves the decidability of termination and confluence.

CR Subject Classification (1991): H.2.1 [Information Systems]: Logical Design - *data models*, H.2.3 [Information Systems]: Languages - *data description languages (DDL); data manipulation languages (DML);*

Keywords: active databases, production rules, triggers, decidability, design theory.

Note: Johan van den Akker is supported by SION, the Foundation for Computer Science Research in the Netherlands under Project No. 612-323-424.

1 Introduction

The original role of rules, or triggers, in databases was the enforcement of integrity constraints. These constraints may either be static or dynamic. Static constraints describe the acceptable database states. Dynamic constraints specify the acceptable transitions between database states. Constraint specification is declarative, which means that the corrective action to be taken in case of a violation is decided by the underlying constraint enforcement system. In most cases this action will be the abort or roll-back of the transaction involved.

In a lot of cases we can take a better corrective action than aborting the transaction. In VLSI design, for example, we encounter a constraint specifying a minimum distance between two wires in order to avoid interference or manufacturing difficulties. If this constraint is only specified declaratively, enforcement of the constraint will involve aborting transactions violating the constraint. A better approach to correct this action would be to rearrange the wires in order to satisfy the constraint. In addition we may want to avoid the computational burden of continuously checking the design.

Rules allow us to specify *how* and *when* an integrity constraint must be enforced. A means to offer rules are *active objects*. An active object is an object with its own autonomous internal activity. The activity is specified by a condition-action pairs. The condition may be a predicate on the database state, a sequence of events or a combination of these. The action is usually defined in some kind of programming language. The semantics is roughly that whenever the condition is satisfied, the action is executed. In order to enforce an integrity constraint we specify the violating state or state transition by the condition and define a corrective action to be taken.

Of course much wider uses have been found for active objects than just constraint enforcement. Functionality of an information system that is reactive in nature can be implemented by active objects. An example is that we check for backorders each time a new load of an item is added to the inventory. Computerised traders in financial markets are also a form of active objects.

The additional functionality offered by active objects comes at a price. A designer who defines active objects needs also be concerned with the interactions of a set of active objects. For example, members of the set may mutually activate or deactivate each other. Because the autonomous nature of active objects, their behaviour is controlled through their definition only. A *design theory* defines properties on sets of active objects and provides algorithms to detect such properties.

Properties studied in the literature are *termination* and *confluence*. A set of triggers terminates if, starting with any initial database state, the conditions of all the triggers become false in a finite number of steps. That is, the database converges to a final state. A terminating set of triggers is called confluent if the final state is determined completely by the initial state and the trigger set.

The properties of a set of triggers depend, of course, on the run-time semantics of the triggers. There are two pre-dominant run-time models in the literature, viz., *set-oriented* and *instance-oriented* semantics. Under the former, a trigger executes simultaneously on *all* objects that satisfy its condition. Under the latter, the trigger executes, non-deterministically, on *one* object that satisfies this condition. In this paper we shall refer to these two semantics as set semantics and instance semantics.

These two types of semantics suggest a third property, which we call *indifference*, that relates the two semantics. A trigger set which is confluent under both semantics is indifferent if the unique final state under the two semantics is identical.

The development of a design theory for triggers has been advanced by targeting on either *sufficient conditions* or on *decidability*. Examples of the former approach are [5, 2]. In the context of the RDL rule system, Simon and deMaindreville [5] formulate a condition under which set and instance based rule execution coincide. Widom et al [9, 8], defined a production rule language for the Starburst database system. In [2] sufficient conditions for both termination and confluence of these production rules are formulated.

A property is called decidable if there exists an algorithm which given a set of triggers as input, decides in finite time whether this set satisfies the property or not. Examples of this approach are [1] and this paper. In this paper we start with a basic language that features conditions local to the active object and actions that replace values of attributes. After that we will consider extensions to both the condition and the action part separately. This paper and [1] are complementary, since Abiteboul and Simon concentrate on a language with deletions, while we only consider modifications.

Roadmap

The structure of the paper is as follows: First we give our data and rule model. Second we will define the predicates on rule sets that we consider. Then we will examine the decidability of these predicates for the simplest model. After that we will consider extension to non-local conditions and actions with simple arithmetic, both separately and combined. Concluding remarks and the direction future research should take round off the paper.

2 Model

In this section we define the models used in this report. First we define the full model, then we give the restricted models. The latter are interesting, because they have several decidable properties.

2.1 The Full Model

The data model is a simple object-oriented model. An object has attributes and methods. An attribute has a name and a type. The only types are integers and classes. Methods assign new values to attributes. A method consists of a heading and a body. The heading specifies the name and the parameters of the method. The body specifies the assignments to be made by the method. Some arithmetic can be done by a method, viz. addition and multiplication. An example of a class is the following class definition:

Class cell
Attributes
    no: Integer

neighbour : cell
Methods
multiply_no(factor:Integer) =
self except no:=no*factor
EndClass

This class has two attributes and one method. The keywords **self except** indicate that the object the method yields is the same one as before the method call, except for the changes specified.

The syntactical definitions of classes, queries and rules are given in BNF form. In these definition all terminals ending in $\langle \ldots \text{ Id}\rangle$ are identifiers taken from a universe of identifiers, defined in the usual way. The definition of a class is:

$$\begin{aligned}
\langle\text{Class}\rangle &\longrightarrow \textbf{Class}\ \langle\text{ClassId}\rangle && (1)\\
&\qquad \textbf{Attributes}\ \langle\text{AttributeList}\rangle\\
&\qquad \textbf{Methods}\ \langle\text{MethodList}\rangle\\
&\qquad \textbf{EndClass}\\
\langle\text{AttributeList}\rangle &\longrightarrow \langle\text{AttributeDecl}\rangle\ \langle\text{AttributeList}\rangle && (2)\\
\langle\text{AttributeList}\rangle &\longrightarrow \langle\text{AttributeDecl}\rangle && (3)\\
\langle\text{AttributeDecl}\rangle &\longrightarrow \langle\text{AttributeId}\rangle''\!:\!''\ \langle\text{TypeId}\rangle && (4)\\
\langle\text{TypeId}\rangle &\longrightarrow \text{Integer} && (5)\\
\langle\text{TypeId}\rangle &\longrightarrow \langle\text{ClassId}\rangle && (6)\\
\langle\text{MethodList}\rangle &\longrightarrow \langle\text{MethodDecl}\rangle\ \langle\text{MethodList}\rangle && (7)\\
\langle\text{MethodList}\rangle &\longrightarrow \langle\text{MethodDecl}\rangle && (8)\\
\langle\text{MethodDecl}\rangle &\longrightarrow \langle\text{MethodId}\rangle\ \texttt{(}\ \langle\text{ParameterList}\rangle\ \texttt{)} && (9)\\
&\qquad = \textbf{self except}\ \langle\text{AssignmentList}\rangle\\
\langle\text{ParameterList}\rangle &\longrightarrow \langle\text{ParameterDecl}\rangle\ \texttt{,}\ \langle\text{ParameterList}\rangle && (10)\\
\langle\text{ParameterList}\rangle &\longrightarrow \langle\text{ParameterDecl}\rangle && (11)\\
\langle\text{ParameterDecl}\rangle &\longrightarrow \langle\text{ParameterId}\rangle\ \texttt{:}\ \langle\text{TypeId}\rangle && (12)\\
\langle\text{AssignmentList}\rangle &\longrightarrow \langle\text{Assignment}\rangle\ \texttt{,}\ \langle\text{AssignmentList}\rangle && (13)\\
\langle\text{AssignmentList}\rangle &\longrightarrow \langle\text{Assignment}\rangle && (14)\\
\langle\text{Assignment}\rangle &\longrightarrow \langle\text{AttributeId}\rangle\ \texttt{:=}\ \langle\text{Expr}\rangle && (15)\\
\langle\text{Expr}\rangle &\longrightarrow \langle\text{BasicExpr}\rangle && (16)\\
\langle\text{BasicExpr}\rangle &\longrightarrow \texttt{Succ(}\ \langle\text{BasicExpr}\rangle\ \texttt{)} && (17)\\
\langle\text{BasicExpr}\rangle &\longrightarrow 0\ |\ 1\ |\ \ldots && (18)
\end{aligned}$$

When a method is called the parameters need to be assigned values. An example of this is the method call multiply_no(factor=3). The formal definition in BNF is:

$$\begin{aligned}
\langle\text{MethodCall}\rangle &\longrightarrow \langle\text{MethodId}\rangle\ \texttt{(}\ \langle\text{ActParamList}\rangle\ \texttt{)} && (19)\\
\langle\text{ActParamList}\rangle &\longrightarrow \langle\text{ActParam}\rangle\ \texttt{,}\ \langle\text{ActParamList}\rangle && (20)\\
\langle\text{ActParamList}\rangle &\longrightarrow \langle\text{ActParam}\rangle && (21)\\
\langle\text{ActParam}\rangle &\longrightarrow \langle\text{ParameterId}\rangle\ \texttt{=}\ \langle\text{Expr}\rangle && (22)
\end{aligned}$$

Although this defines syntactically correct objects, more is needed to define a meaningful hierarchy. The two sets of constraints that take care of this are uniqueness constraints and referential constraints.

Uniqueness Constraints The uniqueness constraints state that there must be unique names for classes, attributes and methods.

Referential Constraints The referential constraints are concerned with the correctness of the references made to other entities in the method and attribute declarations.

1. All methods must be well-typed. Well-typedness means that all assignments are to the correct types.
2. All classes referred to in attribute or parameter declaration must exist in the class hierarchy.

A correct method call has the same number of actual parameters as the number of parameters specified in the method declaration. The criterion of well-typedness must also be satisfied by a method call.

The class definition above only describes a database schema, not an actual instance of a database. An instance of a database is determined by an extension and an interpretation. The extension assigns objects to each class in the database. The interpretation assigns values to the attributes.

The extension of a database assigns to each class a set of objects. An object is identified by a unique object identifier. Therefore we suppose the existence of a set of object identifiers *Oid*. The extension function then assigns a subset of *Oid* to each class, such that all object identifiers are assigned to one class only.

Definition 1 *Let H be a hierarchy. An extension $Ext : H \to \mathcal{P}Oid$ assigns to each class of H a set of objects such that if $C_1, C_2 \in H$ and $C_1 \neq C_2$, then $Ext(C_1) \neq Ext(C_2)$. $\mathcal{P}Oid$ denotes the power set of Oid.*

In addition to the classes we also have the type Integer that needs an extension.

Definition 2 *The extension of the type Integer is the set of natural numbers.*

With the extension we have sets of objects for each class. Relations between these sets are not yet defined however. The interpretation assigns values to all the attributes in the database. The simplest way to view the interpretation is that a table is maintained for every attribute in the database schema. This table has two columns, the first containing object identifiers. In the second column the value of the attribute in that object is given. An example is the following interpretation for the attribute “no” of the class “cell” defined above.

Oid	Value
345	3
874	25
902	16

In this example we have three objects. The value of the attribute no for object 874 is 25. The interpretation of an attribute is a function from the extension of the class it belongs to, to the extension of the type of the attribute. The extension of the type itself is either a class extension or the extension of the type Integer.

Definition 3 *Let Ext be an extension of a hierarchy H. An interpretation I for Ext and H assigns to each type declaration* $a : \tau$ *a function* $I(a : \tau) : Ext(C) \rightarrow Ext(\tau)$.

A database for a hierarchy is a pair of an extension and an interpretation.

Definition 4 *A database for a hierarchy H is a pair* (Ext, I) *where Ext is an extension for H and I an interpretation for Ext and H. The universe of all databases is denoted by* DB_H. *Individual database states are denoted by* $db, db_1, db_2, \ldots$.

If we look at the semantics of method execution, we need to relate two interpretation to each other. To be precise we need to describe the interpretation of the database after the method execution in terms of the interpretation before the method execution. For that we need the notion of a variant of an interpretation. A variant of an interpretation I is denoted by $I\{v/(a : \tau)(o)\}$. The variant is the same as I, except when $I(a : \tau)$ is applied to the object o, where it yields v. A property of variants that we state without proof is the independence of variants on different objects:

$$\begin{aligned}
&\forall o_1, o_2 \in Ext : \\
&o_1 \neq o_2 \\
&\rightarrow \\
&I\{v_1/(a : \tau)(o_1)\}\{v_2/(a : \tau)(o_2)\} = I\{v_2/(a : \tau)(o_2)\}\{v_1/(a : \tau)(o_1)\}
\end{aligned}$$

The function M that defines the semantics of method execution gives us a new interpretation of the database that is variant on the attributes modified.

Definition 5 *Let* $m(l_1 : \tau_1, \ldots, l_n : \tau_n) = \textbf{self except}\{a_i := l_i, succ(b_j)\}, i = 1 \ldots n, j = 1 \ldots m$ *be a method of class* $C \in H$ *with* $\forall i \in \{1, \ldots, n\} : "a_i : \tau_i" \in Attr(C), \forall j \in \{1, \ldots, m\} : "b_j : \sigma_j" \in Attr(C)$ *and* $\{"a_1 : \tau_1", \ldots, "a_n : \tau_n"\} \cap \{"b_1 : \sigma_1", \ldots, "b_m : \sigma_m"\} = \emptyset$. *o is an object in* $Ext(C)$ *and* $m(l_1 = v_1, \ldots, l_n = v_n)$ *is a correct method call. The function M is defined for the execution of m by o in the database* (Ext, I) *as:*

$$\begin{aligned}
&M(m(l_1 = v_1, \ldots, l_n = v_n)(o)(Ext, I)) = \\
&\quad (Ext, I\{v_1/(a_1 : \tau_1)(o)\}, \ldots, \{v_n/(a_n : \tau_n)(o)\}, \\
&\quad \{(I(b_1 : \sigma_1) + 1)/(b_1 : \sigma_1)(o)\}, \ldots, \{(I(b_m : \sigma_m) + 1)/(b_m : \sigma_m)(o)\})
\end{aligned}$$

In the object oriented setting a query is defined as a subclass of the class it queries. An example of a query is

```
Qclass Ten isa cell
Where
   no = 10
Endqclass
```

The formal definition in BNF of a query is:

$$\begin{array}{rcll}
\langle\text{QueryClass}\rangle & \longrightarrow & \textbf{Qclass}\,\langle\text{QclassId}\rangle\,\textbf{Isa}\,\langle\text{ClassID}\rangle & (23)\\
 & & \textbf{Where}\,\langle\text{SelectionCondition}\rangle & \\
 & & \textbf{Endqclass} & \\
\langle\text{SelectionCondition}\rangle & \longrightarrow & \langle\text{SelectionCondition}\rangle' \wedge' \langle\text{SelectionCondition}\rangle & (24)\\
\langle\text{SelectionCondition}\rangle & \longrightarrow & \langle\text{SelectionCondition}\rangle' \vee' \langle\text{SelectionCondition}\rangle & (25)\\
\langle\text{SelectionCondition}\rangle & \longrightarrow & \langle\text{SelectExpr}\rangle' =' \langle\text{SelectExpr}\rangle & (26)\\
\langle\text{SelectionCondition}\rangle & \longrightarrow & \langle\text{SelectExpr}\rangle' \neq' \langle\text{SelectExpr}\rangle & (27)\\
\langle\text{SelectExpr}\rangle & \longrightarrow & \langle\text{AttributeID}\rangle & (28)\\
\langle\text{SelectExpr}\rangle & \longrightarrow & 0 \mid 1 \mid \ldots & (29)\\
\langle\text{SelectExpr}\rangle & \longrightarrow & \langle\text{SelectExpr}\rangle' \circ' \langle\text{SelectExpr}\rangle & (30)
\end{array}$$

The obvious typing rules [7] should be applied in order to obtain well-typed selection expressions. The same constraints that apply to classes of course also apply to query classes. Since we do not consider inheritance, a query defines a simple subset of a class. The extension of a query class is formed by all objects of its superclass that satisfy the selection condition.

Rules are pairs of queries and actions. The query selects the objects on which the method call specified in the action part is executed. An example is the following rule, that selects cells with a value of no smaller than ten and multiplies it by two.

Rule Ten_by_Two = (Ten , multiply_no(factor=2))

The BNF definition of a rule is:

$$\langle\text{Rule}\rangle \longrightarrow \textbf{Rule}\,\langle\text{RuleId}\rangle' =\ \Big('\,\langle\text{QueryId}\rangle\,,\langle\text{MethodCall}\rangle'\Big)\ ' \qquad (31)$$

2.2 Model Mk I

This is the simplest model. We restrict the selection condition of a query to considering local attributes only. The methods are restricted to replacement of attributes by constant values only. This means that of the definition of classes and queries above, productions 17 and 30 are omitted.

An example of a class definition in this model is:

```
Class cell
Attributes
    no: Integer
    neighbour : cell
Methods
    new_value(number:Integer) =
        self except no:=number
EndClass
```

A query selecting all objects with a value smaller than 10 is in this model:

Qclass Ten **isa** cell
Where
    no = 10
Endqclass

This query is used in the following example of a rule in this model:

Rule Make_Ten_Five = (Ten, new_value(number=5))

2.3 Model Mk II

In this model we relax the restriction on the selection condition. It is allowed to look at attributes of other objects in the condition. The methods are still restricted to replacement by constant values. Thus the production 17 is omitted from the definition of the model.

Class definitions are not different from the Mk I model. Given the class definition of the previous section, a possible query is:

Qclass Neighbour_Ten **isa** cell
Where
    no$\circ$neighbour = 10
Endqclass

An example of a rule using this query is:

Rule Neighbour_Make_Ten_Five = (Neighbour_Ten, new_value(number=5))

2.4 Model Mk III

This model maintains the restriction on the selection condition to local attributes, but the actions allowed in the methods are modified. Limited arithmetic is allowed in the methods, viz. taking the successor of an attribute. Production 30 and 18 are omitted from the definition of the model.

In this model the class definitions are extended with some extra functionality in the methods, viz. arithmetic. An example of a class in this model is:

Class cell
Attributes
    no: Integer
    neighbour : cell
Methods
    add_two =
        self except no:=Succ(Succ(no))
EndClass

Queries in this model are still restricted to local attributes only, so the query Ten can also serve as an example for this model.

```
Qclass Ten isa cell
Where
    no = 10
Endqclass
```

An example rule in this model is the following rule that adds two to the attribute no in all objects where this attribute's value is ten:

```
Rule Add_two_to_ten = (Ten, add_two)
```

2.5 Model Mk IV

This is the least restricted model considered in this report, although it is still much too restricted to be of any practical use. The selection condition is allowed to look at attributes in other objects. Applying the successor function to an attribute is allowed in the method. This is the full model defined above.

Since the class definition is the same as in the Mk III model, the same class definition serves as an example:

```
Class cell
Attributes
    no: Integer
    neighbour : cell
Methods
    add_two =
        self except no:=Succ(Succ(no))
EndClass
```

The selection condition is defined in the same way as in the Mk II model. Therefore the same query is an example of a Mk IV query.

```
Qclass Neighbour_Ten isa cell
Where
    no∘neighbour = 10
Endqclass
```

The rules in this model are the least restricted ones in this paper. An example is the following rule that adds two to the value of the attribute no for those objects with a neighbour that has a value less than ten for that attribute.

```
Rule Neighbour_add_two =( Neighbour_Ten, add_two)
```

2.6 Trigger Semantics

Triggers can be applied to a database in two different ways [5]. This results in two different semantics for trigger execution. With set semantics the action is executed on all objects in the selected set simultaneously. Instance semantics means that the action is executed one at the time on the objects in the selected set. In the following definition the result of a trigger

T applied to a database db under set semantics is denoted by $M_s(T, db)$. Under instance semantics this is denoted by $M_i(T, db)$. The execution of a trigger, like a method execution, changes the interpretation of the database. Therefore we can define the resulting interpretation in terms of the variants induced by the trigger.

Definition 6 *Let $T = (Q, M(l_1 = b_1, \ldots, l_n = b_n))$ be a trigger in a hierarchy with M H defined as $M(l_1 : \tau_1, \ldots, l_n : \tau_n) = \textbf{self except}\{a_1 := l_1; \ldots; a_n := l_n\}$. $db = (Ext, I)$ is a database for H. $Pick : \mathcal{P}Oid \to Oid$ is a function that arbitrarily selects an object from a set of objects. $\vec{var}$ denotes the set of variants induced by trigger T.*

The execution of T under set semantics is defined by:

$$M_s(T, (Ext, I)) = \begin{cases} (Ext, I\{\vec{var}(o_1) \ldots \vec{var}(o_m)\} \\ \qquad \textit{if } Ext(Q) = \{o_1, \ldots, o_m\} \\ (Ext, I) \\ \qquad \textit{if } Ext(Q) = \emptyset \end{cases}$$

The execution of T under instance semantics is defined by:

$$M_i(T, (Ext, I)) = \begin{cases} (Ext, I\{\vec{var}(Pick(Ext(Q)))\}) & \textit{if } Ext(Q) \neq \emptyset \\ (Ext, I) & \textit{if } Ext(Q) = \emptyset \end{cases}$$

The above definition defines the result of a single trigger application. If a set of triggers is present on the database, we have an execution cycle. This cycle executes as long as there are triggers applicable to the database. During the cycle one of the set of applicable triggers is randomly chosen for execution and executed. After that the next iteration starts.

Definition 7 *Let TR be a set of triggers of the form $T = (Q_T, M_T)$ with an initial database db. Then the behaviour of TR under semantics sem is defined by:*

```
Execute(TR,db,sem) {
    While $\exists T \in TR : Ext_{db}(Q_T) \neq \emptyset$ do
        $T :=$ choose($\{T | T \in TR \wedge Ext_{db}(Q_T) \neq \emptyset$)
        $db := M_{sem}(T, db)$
    od
    return db
}
```

The process Execute(TR,db,sem) induces a set of execution sequences. An execution sequence gives a trace of the trigger execution. In the case of set semantics it is a sequence of triggers. In the case of instance semantics we also need to include the information which object the trigger was executed on. A sequence is defined as a function from the set of natural numbers to the set of triggers. The function assigns a trigger to each position in the sequence. If the function is total, i.e. it assigns a trigger to each position, then it is a sequence.

Definition 8 *TR is a set of triggers for a hierarchy H. If $Sq : \mathcal{N}^+ \to TR$ is a partial function whose support is a contiguous set starting at 1, then $Sq \in Seq^s$, $length(Sq) = |Support(Sq)|$.*

The ith element of a sequence Sq is denoted by Sq_i. It is a trigger $T_i = (Q_i(S_i), M_i)$.

Under instance semantics a sequence should also state to which object a trigger is applied at each place in the sequence.

Definition 9 *TR is a set of triggers for a hierarchy H. If $Sq : \mathcal{N}^+ \rightarrow TR \times Oid$ is a partial function whose support is a contiguous set starting at 1, then $Sq \in Seq^s$, $length(Sq) = |Support(Sq)|$.*

Here Sq_i is a pair of a trigger and an object $(T_i = (Q_i(S_i), M_i), o_i)$.

A sequence Sq from either Seq^s or Seq^i is also written as a list of elements: $[Sq(1); \ldots ; Sq(n)]$. Sequences may be concatenated in the usual way.

The execution of a trigger sequence is defined inductively in the following way:

Definition 10 *TR is a set of triggers for a hierarchy H. The execution of a sequence $Sq \in Seq^s \cup Seq^i$, where Seq^s and Seq^i are the set of sequences over TR, on a database db is defined as:*

1. *if* $Sq \equiv [Sq(1); \ldots ; Sq(n)]$
   then $M_{sem}(Sq, db) = M_{sem}([Sq(2); \ldots ; Sq(n)], M_{sem}(Sq(1), db))$

2. *if* $Sq \equiv [Sq(1); Sq(2); \ldots]$
   then $M_{sem}(Sq, db) = M_{sem}([Sq(2); \ldots], M_{sem}(Sq(1), db))$

Not every sequence is a valid execution sequence. A sequence $[Sq(1); \ldots ; Sq(n)]$ is an execution sequence, if each trigger $Sq(i+1)$ is activated after the execution of $[Sq(1); \ldots ; Sq(i)]$. After the last trigger of a sequence there are no more activated triggers, if the sequence is finite.

Definition 11 *Let TR be a set of triggers for a hierarchy H, $db \in DB_H$ a database, and Seq^s and Seq^i the sets of sequences over TR. The set $Seq^s(TR, db)$ of execution sequences over TR is db under set semantics is defined as follows:*

$$\begin{array}{ll}
\textit{if} & Sq \in Seq^s \wedge \\
& length(Sq) = n \wedge \\
& Ext_{db}(Q_1) \neq \emptyset \wedge \\
& \forall i \in \{2, \ldots, n\} : Ext_{M_s([Sq(1);\ldots;Sq(i-1)])}(Q_i) \neq \emptyset \wedge \\
& \forall T \in TR : Ext_{M_s(Sq,db)}(Q) = \emptyset \\
\textit{then} & Sq \in Seq^s(TR, db)
\end{array}$$

The set of execution sequences under instance semantics $Seq^i(TR, db)$ is defined analogously as follows:

$$\begin{array}{ll}
\textit{if} & Sq \in Seq^i \wedge \\
& lenght(Sq) = n \wedge \\
& o_i \in Ext_{db}(Q_1) \wedge \\
& \forall i \in \{2, \ldots, n\} : o_i \in Ext_{M_i([Sq(1);\ldots;Sq(i-1)])}(Q_i) \wedge \\
& \forall T \in TR : Ext_{M_i(Sq,db)}(Q) = \emptyset \\
\textit{then} & Sq \in Seq^i(TR, db)
\end{array}$$

3 Predicates

In the previous section we have defined the data and rule model and the semantics of trigger execution. Since we study the behaviour of trigger sets, we first need to define the predicates of trigger sets we are interested in.

3.1 Termination

Termination means that all executions of a trigger set on all possible database states terminate. In terms of execution sequences this means that all execution sequences on all databases must be finite.

Definition 12 (Termination) *Let TR be a set of triggers for a hierarchy H. Let sem denote either set or instance semantics.*

$$\begin{aligned} &Terminate(TR, sem) \overset{def}{=} \\ &\quad \forall db \in DB_H, \forall Sq \in Seq^{sem}(TR, db), \exists n \in \mathcal{N}^+ : length(Sq) = n \end{aligned}$$

3.2 Confluence

Confluence means that all possible executions of a trigger set yield the same final database state. Because a non-terminating execution does not yield a final database state, a preliminary requirement for confluence is that the trigger set is terminating. In terms of sequences confluence means that the results of all possible execution sequences of a trigger set must be equal to each other.

Definition 13 (Confluence) *Let TR be a set of triggers for a hierarchy H. Let sem denote either set or instance semantics.*

$$\begin{aligned} &Confluent(TR, sem) \overset{def}{=} \\ &\quad \forall db \in DB_H, \forall Sq \in Seq^{sem}(TR, db) : M_{sem}(Sq_1, db) = M_{sem}(Sq_2, db) \\ &\quad \wedge \\ &\quad Terminate(TR, sem) \end{aligned}$$

3.3 Termination in n steps

In most rule and data models termination is an undecidable property. Therefore we are sometimes interested in a stronger predicate, termination in a certain number of steps. This predicate is stronger than termination, because a trigger set not terminating in n steps might terminate in $n + 33$ steps.

It is important to define a step first. Under set semantics, we simply take one execution of one trigger as one step. We cannot do this under instance semantics, since a trigger application only executes on one object at the time. Because we do not wish to make our choice of n dependent of the size of the database, we count the number of trigger applications to one object. Thus n denotes the maximum number of times a trigger may execute under set semantics and the maximum number of times a trigger may execute on one object under instance semantics.

Definition 14 *If TR is a set of triggers for a hierarchy H and n a natural number, then*

$$\begin{aligned}
&Terminate(n, TR, set) \overset{def}{=} \\
&\quad \forall db \in DB_H, \forall Sq \in Seq^s(TR, db), \forall T \in TR : |\{i|Sq(i) = T\}| \leq n \\
&Terminate(n, TR, instance) \overset{def}{=} \\
&\quad \forall db \in DB_H, \forall o \in Ext_{db}, \forall Sq \in Seq^s(TR, db), \forall T \in TR : \\
&\quad |\{i|Sq(i) = (T, o)\}| \leq n
\end{aligned}$$

The execution sequences are finite, because all trigger sets and all databases are finite. Thus the definition implies termination of the trigger sets.

Proposition 1 *If TR is a set of triggers for a hierarchy H and sem denotes either set or instance semantics, then*

$$Terminate(n, TR, sem) \rightarrow Terminate(TR.sem)$$

Proof Obvious. □

3.4 Independence

Like we defined termination in n steps as a stronger alternative for termination, we can define a stronger alternative for confluence. This stronger predicate is independence. Instead of looking at a complete trigger set we examine a pair of triggers at the time. A pair is said to be independent, if the two triggers commute. This means that the result of their execution is the same for both possible orders of execution. A set of triggers is independent, if all possible pairs of triggers commute.

Definition 15 *If TR is a set of triggers for a hierarchy H, then:*

$$\begin{aligned}
&Independent(TR, set) \overset{def}{=} \\
&\quad \forall T_i, T_j \in TR, \forall db \in DB_H : \\
&\quad M_s(T_i, M_s(T_j, db)) = M_s(T_j, M_s(T_i, db))
\end{aligned}$$

$$\begin{aligned}
&Independent(TR, instance) \overset{def}{=} \\
&\quad \forall T_i, T_j \in TR, \forall db \in DB_H, \forall o_k, o_l \in Ext_{db} : \\
&\quad M_i(T_i(o_k), M_i(T_j(o_l), db)) = M_i(T_j(o_l), M_i(T_i(o_k), db))
\end{aligned}$$

A very useful property of an independent trigger set is that execution sequences can be rearranged. This comes in handy in a number of proofs. An example is the proof that independence implies confluence for terminating trigger sets [2].

Proposition 2 *If TR is a set of triggers and sem denotes either instance or set semantics, then:*

$$Terminate(n, TR, sem) \ \wedge \ Independent(TR, sem) \rightarrow Confluent(TR, sem)$$

Proof Let us first remark that $Terminate(n, TR, sem)$ implies $Terminate(TR, sem)$. Thus we have to prove that:

$$\forall db \in DB_H, \forall Sq_1, Sq_2 \in Seq^{sem}(TR, db)$$
$$M_{sem}(Sq_1, db) = M_{sem}(Sq_2, db)$$

Since both Sq_1 and Sq_2 are execution sequences, no trigger is activated after their execution. Thus for all database state $M_{sem}(Sq_1; Sq_2, db) = M_{sem}(Sq_1, db)$ and likewise $M_{sem}(Sq_2; Sq_1, db) = M_{sem}(Sq_2, db)$. Since all triggers are pairwise independent, $Sq_2; Sq_1$ can be rearranged into $Sq_1; Sq_2$. Thus $M_{sem}(Sq_2; Sq_1, db) = M_{sem}(Sq_1; Sq_2, db)$ and therefore $M_{sem}(Sq_1, db) = M_{sem}(Sq_2, db)$. □

3.5 Decidability

The last definition needed in this report is that of decidability. For this we use the standard notion of the existence of a decision procedure.

Definition 16 *A predicate is decidable, iff there exists an algorithm that on all possible input:*

1. *terminates*
2. *on termination gives the correct answer with regard to the truth of the predicate relative to the input.*

4 Decidability Results for the Model Mk I

In this section we look into the decidability issues of the most restricted model as described in Section 2.2. As can be expected in such a simple model, both termination and confluence are decidable properties of trigger sets in this language.

We will first look at termination of a single trigger. In this simple language it is not much of a problem, because the effect of the trigger on an object it executes on is idempotent.

Theorem 1 *In the model Mk I, given a singleton trigger set $TR = \{T\}$, $T = (Q_T, M_T)$ and semantics sem, $Terminate(TR, sem)$ and $Confluent(TR, sem)$ are decidable predicates.*

Proof To prove the decidability of termination and confluence, we show that we can construct a finite database state to represent all possible database states. This database state is called the *typical database state*. The typical database state is constructed relative to a trigger definition. From the attributes of the class hierarchy and the constants in the queries and the methods we construct a finite set of partition conditions. Every object in all of the possible database states satisfies one of these conditions. In addition method execution gives a uniform transition between these conditions. Based on this knowledge we construct a graph that enables us to decide termination and confluence.

We start by defining a set EC of elementary conditions on the attributes. Let A be the set of all attributes in the class on which T is defined. C_T is the set of all constants appearing in Q_T

and M_T. The set of elementary conditions EC with regard to T is defined by the following grammar:

$$\begin{array}{rcl} \langle\text{Econd}\rangle & \longrightarrow & \langle\text{Attr}\rangle\,\langle\text{EqualOrNot}\rangle\,\langle\text{BasicExpr}\rangle \\ \langle\text{EqualOrNot}\rangle & \longrightarrow & = \mid \neq \\ \langle\text{BasicExpr}\rangle & \longrightarrow & \langle\text{Attr}\rangle \mid \langle\text{Const}\rangle \end{array}$$

where the non-terminal $\langle\text{Attr}\rangle$ yields all elements of A and $\langle\text{Const}\rangle$ yields all elements of C_T.

Obviously EC does not take the types of attributes and constants into account. Therefore we restrict EC to the set of well-typed elementary conditions WEC as follows:

$$\forall x : \tau, y : \tau, \theta \in \{=, \neq\} : x\theta y \in EC \rightarrow x\theta y \in WEC$$

The elementary conditions only express equalities of one or two attributes at a time. To be able to express arbitrary conditions on object we obtain all composite conditions in the set $Cond$.

If $c \in WEC$ then $c \in Cond$
If $c_1, c_2 \in Cond$ then $c_1 \wedge c_2 \in Cond$

This definition yields a set that also contains inconsistent conditions. However we are able to decide what conditions are consistent.

Claim 1.1 *Determining the consistency of a condition $\phi \in Cond$ is decidable. If ϕ is consistent we can construct a database state that contains an object satisfying ϕ.*

Knowing that the consistency of a condition is decidable we can restrict our conditions to the set $CCond$ of consistent conditions.

$$CCond = \{c \in Cond | c \text{ is consistent}\}$$

Multiple conditions are satisfied by an object, because most conditions do not take all attributes of an object into account. To characterise an object we want those conditions that specify equalities of all attributes. To that end we define a relation on $CCond$:

$$\begin{array}{l} \forall \phi, \psi \in CCond, \phi = \phi_1 \wedge \ldots \wedge \phi_k, \psi = \psi_1 \wedge \ldots \wedge \psi_l : \\ \quad \forall i \in \{1 \ldots k\} \exists j \in \{1 \ldots l\} : \phi_i = \psi_j \\ \quad \rightarrow \\ \quad \psi > \phi \end{array}$$

This relation is a partial order. In this order the maxima are those conditions that incorporate all attribute-attribute and attribute-constant relations. Therefore we use these conditions as partitioning conditions.

$$PCond = \{c \in CCond | c \text{ is maximal}\}$$

The partition conditions characterise all possible objects in all possible databases relative to this trigger. Uniform transitions exist between partition conditions to represent the effect of method execution. These two properties are expressed in the following claim:

Claim 1.2 *Partition conditions satisfy the following properties:*

1. $\forall db \in DB, \forall o \in db : \exists! pc \in PCond : pc(o.db)$

2. $\forall db_1, db_2 \in DB, \forall o_1 \in db_1, \forall o_2 \in db_2$
   $pc(o_1, db_1) \wedge pc(o_2, db_2)$
   $\rightarrow$
   $\exists pc' \in PCond : pc'(o_1, Execute(T, db_1, set)) \wedge pc(o_2, Execute(T, db_2, set))$

   under set semantics and

   $\forall db_1, db_2 \in DB, \forall o_1 \in db_1, \forall o_2 \in db_2$
   $pc(o_1, db_1) \wedge pc(o_2, db_2)$
   $\rightarrow$
   $\exists pc' \in PCond :$
   $pc'(o_1, Execute(T(o_1), db_1, instance)) \wedge$
   $pc(o_2, Execute(T(o_2), db_2, instance))$

   under instance semantics.

Claim 1.1 said that we can for each consistent condition construct a database state satisfying that condition. Because we may assume without loss of generality that object identifiers are unique over all db_ϕ, we can construct a database state $db = \bigcup_{\phi \in PCond} db_\phi$ that is typical i.e. $\forall c \in PCond, \exists o \in db : c(o, db)$.

We have shown that partition conditions represent all possible object states and that the effects of method application is uniformly represented by a transition from one partition condition to another. We now proceed by constructing a graph that represents these transitions for all partitions. The graphs SG and IG are defined for set and instance semantics respectively as follows:

$$Nodes(SG) = Nodes(IG) = PCond$$

$$\begin{aligned}
&\forall c_1, c_2 \in PCond : \\
&\quad \exists o \in Q_T(db) \wedge c_1(o, db) \wedge c_2(o, Execute(T, db, set)) \\
&\quad \rightarrow \\
&\quad (c_1, c_2) \in Edges(SG) \\
&\\
&\quad \exists o \in Q_T(db) \wedge c_1(o, db) \wedge c_2(o, Execute(T(o), db, instance)) \\
&\quad \rightarrow \\
&\quad (c_1, c_2) \in Edges(IG)
\end{aligned}$$

Using this graph we can reduce the problem of termination to the problem of cycle detection, which is a decidable problem. Confluence reduces to finding a unique sink from each node in the graph, which is also a decidable problem.

Left to be proven are the claims we made with regard to the decidability of consistency of a condition and the properties of partition conditions.

Proof of Claim 1.1 We give an algorithm that checks a condition $\phi = \bigwedge_{i=1}^{n} \phi_i$ for consistency. Without loss of generality we may assume that the ϕ_i are ordered according to the following criteria:

1. Equalities before inequalities.
2. Attribute-constant equalities before attribute-attribute equalities.
3. The attribute-constant equalities are sorted by attribute.
4. The attribute equalities are put in the form $a_i = a_j$ such that $i < j$ and then sorted lexicographically by the pairs (a_i, a_j).

The algorithm proceeds by constructing an object o with attributes $a_1, \ldots, a_n$, that satisfies the condition ϕ. For the construction we need a set of dummy variables *Dummy* with the following properties:

$$\begin{aligned} &\mathit{Dummy} = \{D_1, \ldots, D_n\} \\ &\quad \text{such that} \quad (1)\ i \neq j \rightarrow D_i \neq D_j \\ &\qquad\qquad\qquad (2)\ a_i : \tau \rightarrow D_i : \tau \end{aligned}$$

The algorithm is as follows:

```
**Algorithm** consistency-check
**Begin**
  **Check** $\forall a_i \in A$:
    **Case** $\exists c_1, c_2 \in C_T : a_i = c_1 \wedge a_i = c_2$:
      Exit(Unsuccessfully)
    **Case** $\exists! c_1 \in C_T : a_i = c_1$:
      $o.a_1 := c_1$
    **Otherwise:**
      $o.a_i := D_i$
  **Endcheck**

  **Check** $\forall \phi_k$ of the form $a_i = a_j$:
    **Case** $a_i = a_j \wedge o.a_i = c_1 \wedge o.a_j = c_2 \wedge c_1 \neq c_2$:
      Exit(Unsuccessfully)
    **Case** $a_i = a_j \wedge o.a_i = c_1 \wedge o.a_j = c_2 \wedge c_1 = c_2$:
      Next
    **Case** $a_i = a_j \wedge o.a_i = c_1 \wedge o.a_j = D_j$:
      $a_j := c_1$
    **Case** $a_i = a_j \wedge o.a_i = D_i \wedge o.a_j = c_1$:
      $a_i := c_1$
      ReplaceAll($D_i, c_1$)
    **Case** $a_i = a_j \wedge o.a_i = D_i \wedge o.a_j = D_j$:
      $a_j := D_i$
  **Endcheck**

  **Check** $\forall \phi_k$ of the form $a_i \neq v$
```

```
        Case $a_i \neq a_j \wedge o.a_i = o.a_j$:
            Exit(Unsuccessfully)
        Case $a_i \neq c \wedge o.a_i = c$:
            Exit(Unsuccessfully)
        Otherwise:
            Next
    Endcheck

    Exit(Successfully)

End.
```

Successful termination of this algorithm means that it was able to construct an object satisfying the given condition ϕ, implying that it is consistent. If the algorithm terminates unsuccessfully, then the condition is inconsistent. If the condition is consistent, we can construct a database state from the object and the set of dummy values.

Proof of Claim 1.2 The first part of this claim was

$$\forall db \in DB, \forall o \in db : \exists! pc \in PCond : pc(o.db)$$

The existence of a $pc \in PCond$ is obvious from the fact that all possible equalities between attributes and constants are included in WEC from which the partition conditions are constructed. The existence of a unique $pc \in PCond$ follows from the maximality of the partition conditions.

The second part of the claim was that

$$\begin{aligned}&\forall db_1, db_2 \in DB, \forall o_1 \in db_1, \forall o_2 \in db_2\\ &\quad pc(o_1, db_1) \wedge pc(o_2, db_2)\\ &\quad \rightarrow\\ &\quad \exists pc' \in PCond : pc'(o_1, Execute(T, db_1, set)) \wedge pc(o_2, Execute(T, db_2, set))\end{aligned}$$

under set semantics and

$$\begin{aligned}&\forall db_1, db_2 \in DB, \forall o_1 \in db_1, \forall o_2 \in db_2\\ &\quad pc(o_1, db_1) \wedge pc(o_2, db_2)\\ &\quad \rightarrow\\ &\quad \exists pc' \in PCond :\\ &\qquad pc'(o_1, Execute(T(o_1), db_1, instance))\\ &\qquad \wedge pc(o_2, Execute(T(o_2), db_2, instance))\end{aligned}$$

under instance semantics.

This lemma follows from the maximality of the partition conditions and the fact that the changes made by M_T are identical to both objects. □

The method used for deciding termination of a singleton trigger set can be extended to a trigger set with more than one trigger. The method used is the same except that we must construct a graph using more than one trigger.

Theorem 2 *Given a trigger set $\mathcal{T}$ and semantics sem, the predicates $Terminate(\mathcal{T}, sem)$ and $Confluence(\mathcal{T}, sem)$ are decidable in the model Mk I.*

Proof Since the conditions and actions of the triggers are local to an object, we can treat this problem for each class separately. For each class C we construct a set of partition conditions $PCond_C$ using exactly the same method as used in the proof of Theorem 1. We then merge these sets into one set of partition conditions $PCond = \bigcup_C PCond_C$. Clearly, since each $PCond_C$ induces a typical database state tdb_C, their union $PCond$ also induces a typical database state $tdb = \bigcup_C tdb_C$.

We then proceed with the construction of a graph. The presence of more than one trigger has some effect on the drawing of the graph. The graphs SG for set semantics and IG for instance semantics are defined as:

$$Nodes(SG) = Nodes(IG) = PCond$$

$$\begin{aligned}
&\forall c_1, c_2 \in PCond : \\
&\quad \exists T \in \mathcal{T}, \exists o \in Q_T(db) \wedge c_1(o, db) \wedge c_2(o, Execute(T, db)) \\
&\quad \rightarrow \\
&\quad (c_1, c_2) \in Edges(SG)
\end{aligned}$$

$$\begin{aligned}
&\forall c_1, c_2 \in PCond : \\
&\quad \exists T \in \mathcal{T}, \exists o \in Q_T(db) \wedge c_1(o, db) \wedge c_2(o, Execute(T(o), db)) \\
&\quad \rightarrow \\
&\quad (c_1, c_2) \in Edges(IG)
\end{aligned}$$

Again deciding termination reduces to cycle detection and deciding confluence to finding a unique sink for each node. □

5 Decidability Results for the Model Mk II

In this section we look into decidability of the predicates in the model Mk II. The only difference between the model in this section and the Mk I model is that we now allow attributes of other object in the selection condition of a query. This is enough to make termination an undecidable property. Some stronger properties are decidable however, such as termination in n steps and independence.

First we look at termination of a singleton trigger set. This is a decidable predicate in this model.

Theorem 3 *In the Mk II model, given a singleton trigger set $\mathcal{T}$ and semantics sem the predicate $Terminate(\mathcal{T}, sem)$ is decidable.*

Proof We show that we can construct a typical database state in this model. A complication is that the number of possible conditions is infinite.

Since path expressions are allowed in the condition of a rule, the set of elementary conditions EC is generated by the following grammar:

$$\langle \text{Econd} \rangle \longrightarrow \langle \text{AttrExpr} \rangle \, \langle \text{EqualOrNot} \rangle \, \langle \text{BasicExpr} \rangle$$

$$\begin{array}{rcl} \langle\text{EqualOrNot}\rangle & \longrightarrow & = \mid \neq \\ \langle\text{BasicExpr}\rangle & \longrightarrow & \langle\text{AttrExpr}\rangle \mid \langle\text{Const}\rangle \\ \langle\text{AttrExpr}\rangle & \longrightarrow & \langle\text{Attr}\rangle \mid \langle\text{Attr}\rangle\,.\,\langle\text{AttrExpr}\rangle \end{array}$$

where the non-terminal $\langle\text{Attr}\rangle$ yields all attributes in the hierarchy and $\langle\text{Const}\rangle$ yields all constants used in the trigger set. We use the obvious typing rules to restrict EC to the set of well-typed elementary conditions WEC. We collect all possible conditions with conjunction and disjunction into the set $Cond$.

Because of the complexity of conditions with path expressions, we introduce the length of a condition. Intuitively this is a measure of the distance of the attributes of interest to the condition from the local object. The length of a condition is recursively defined as:

1. $length(e) = 1$, where e is an attribute identifier or a constant.
2. $length(a.e) = length(e) + 1$, where a is an attribute and e a path expression.
3. $length(\bigwedge_i \bigvee_j (e\omega f)_{ij}) = max(\{max(length(e), length(f))\}_{ij})$, where e is a path expression and f a path expression or a constant.

The set of conditions restricted to a maximum length n is denoted by $Cond_n$.

Consistency can be checked by a slight modification of the algorithm in the proof of Theorem 1. The difference in the conditions is in the possibility of referring to other objects. This means that in order to show that a database satisfying the condition exists, we need to construct more than one object. In fact we construct all objects referred to in the condition. Using this consistency check and a partial order defined as previously, we arrive at the set of partition condition of length n:

$$PCond_n = \{c \in Cond_n | c \text{ is consistent} \wedge c \text{ is maximal wrt } >\}$$

As before this set induces a typical database state tdb_n, that can be constructed using the consistency checking algorithm.

Now we can again construct a graph to encode the execution of a trigger. A complication is that an object can move from one partition condition to another, because of a change in another object. Therefore we label the transition to indicate whether it is a direct or an indirect transition. A direct transition, labelled da, is caused by the direct application of a trigger on the object. An indirect transition, labelled in, is caused by the execution of a trigger on another object.

We construct a graph for instance semantics $IG_{n.\mathcal{T}}$ and set semantics $SG_{n,\mathcal{T}}$ separately.

1. $\forall n \in \mathcal{N} : Nodes(SG_{n,\mathcal{T}}) = Nodes(IG_{n,\mathcal{T}}) = PCond_n$
2. $\forall n \in \mathcal{N}, \forall c_1, c_2 \in PCond_n :$
   $\exists o \in Q_T(tdb_n) : c_1(o, tdb_n) \wedge c_2(o, Execute(T, tdb_n, set))$
   $\rightarrow (c_1, c_2, da) \in Edges(SG_{n,\mathcal{T}})$
   $\exists o \in Q_T(tdb_n) : c_1(o, tdb_n) \wedge c_2(o, Execute(T(o), tdb_n, instance))$
   $\rightarrow (c_1, c_2, da) \in Edges(IG_{n,\mathcal{T}})$

3. $\forall n \in \mathcal{N}, \forall c_1, c_2 \in PCond_n$:
   $\exists o \notin Q_T(tdb_n) : c_1(o, tdb_n) \wedge c_2(o, Execute(T, tdb_n, set))$
   $\rightarrow (c_1, c_2, in) \in Edges(SG_{n,\mathcal{T}})$
   $\exists o_1 \in Q_T(tdb_n) : c_1(o, tdb_n) \wedge (\exists o_2 \in tdb_n : o_2 \in Q_T(tdb_n) \wedge$
   $o_1 \neq o_2 \wedge c_2(o_1, Execute(T(o_2), tdb_n, instance))$
   $\rightarrow (c_1, c_2, in) \in Edges(IG_{n,\mathcal{T}})$

This graph encodes the life cycle of an object under the execution of the singleton trigger set $\mathcal{T} = \{T\}$, if we take $n = length(C_T)$. This graph has the useful property, that:

Claim 3.1 *In the graphs $SG_{n,\mathcal{T}}$ and $IG_{n,\mathcal{T}}$ there are no cycles of a length greater than* 1.

Thus, deciding termination amounts to the detection of *da*-cycles of length 1. This is a decidable problem. Confluence of a trigger set can be decided by checking whether each object has a unique sink.

Proof of Claim 3.1 The first thing to be noted, is that we have taken the value of n, such that any change that is seen by an object through the trigger condition is included in the partition condition. The use of disjunction in the partition conditions in addition means that other objects' states that are indifferent to the local object are included in the partition condition. We should also keep in mind that application of the trigger to an object is idempotent.

There are three possible configurations for cycles of length greater than 1:

1. Cycle consisting of *da*-edges only
2. Cycle consisting of *in*-edges only
3. Cycle consisting of at least one *da*-edge with the other edges *in*-edges.

Because of the idempotence of trigger application cycles consisting only of *da*-edges must be of length 1. A cycle consisting of more than one *da*-edge would mean that the local state of the object changes after the first application of the trigger to that object. That would be in contradiction to the idempotence of trigger application in this model.

The value of n is chosen in such a way that all variables of importance to the object are incorporated in the partition conditions. Since there is no change in the local state during an *in*-transition, there must be another object that changes. However, the local state of any object changes at most once during any execution of a trigger. Therefore it is not possible that an object's state returns to its initial state after the first trigger application to it. This also means that it is not possible that all objects that refer to a partition condition return to their initial states. This is exactly what would happen if an *in*-cycle were present in the graph. Therefore a cycle of more than one *in*-edge cannot exist.

For the next case, we first show that we need only consider mixed *da*-*in*-cycles with one *da*-edge. Suppose a mixed cycle contains more than one *da*-edge, one from c_1 to c_2 and the second from c_i to c_{i+1}. The local state of the object does not change with the second application of the trigger. Therefore there must be a da-edge from c_1 to c_{i+1}. This means that there always is a shorter cycle, if there are more than one da-edges.

Now we consider a cycle consisting of one da-edge from c_1 to c_2 and for the rest of in-edges. The argument for the non-existence of such a cycle is the same as for the non-existence of a cycle of only in-edges. It is even more obvious, because the existence of such a mixed cycle requires that the object itself reverts to its initial state.

Thus, every cycle in the graph must be of length 1. As a result of this we can construct a database for each path through the graph. It is obvious that a database can be constructed for a single transition in the graph. The method used is the consistency checking algorithm mentioned earlier. This can be extended in a straightforward way for an acyclic graph. It is also obvious that we can construct a database that follows a cycle of length 1. □

Although termination is decidable for a singleton trigger set, this is not the case for a set of more than one trigger. The reason is that the language in this model is powerful enough to simulate a Turing Machine. In a Turing Machine all replacements of values on the tape is by constant symbols. Only in moving between cells we need communication between cells, which can be done by reading a status attribute at the neighbouring cell. We will first show how a Turing Machine is emulated using triggers.

We will give the implementation of a Turing Machine in this trigger language. The tape is represented by the database state, while the transition function is represented by the trigger set. The following class definition implements a cell on the tape.

```
Class Cell
Attributes
    left-neighbour : Cell
    right-neighbour : Cell
    value : Symbol
    state : State
    next : { left, right, neutral }
    current : { yes, no }
    from : Cell
Methods
    execute($x_1$:Symbol, $x_2$:{left,right}, $x_3$:State) = self except
        value:=$x_1$, next:=$x_2$, state:=$x_3$
    become-current-left() = self except
        from := right-neighbour, current:=yes
    become-current-right() = self except
        from := left-neighbour, current:=yes
    not-current-anymore() = self except
        current:=no, next:=neutral
Endclass
```

In this definition the types Symbol, State and all finite sets can be considered subsets of Integer, with the constants denoting a certain number. No extra functionality is added by using these types.

The transition table is recorded in the triggers that each record one entry of the transition table. These triggers are activated when a cell becomes the current cell. To make sure the right cell is designated as the current cell we need a number of bookkeeping triggers, two each for moving the head to the left and the right.

The execution of a transition is triggered on a cell is triggered when the cell is current and the previous cell is finished. The query is:

```
**Qclass** do-i-execute($s$,$v$) **Isa** Cell
**Where**
    current=yes $\wedge$
    from.next=neutral $\wedge$
    from.state=$s$ $\wedge$
    value=$v$
**Endqclass**
```

The execution rule for the transition determined by the state s and value v thus becomes:

Rule Execute(s,v)=(do-i-execute(s,v),execute($x_1 = c_1, x_2 = c_2, x_3 = c_3$))

where c_1 denotes the new symbol for the cell, c_2 the direction the head moves to and c_3 the new state of the Turing Machine.

After the transition is executed, the head must be moved to the next cell. If the head moves to the left, the cell that is next, must change its status to current. It knows it may do this if its right neighbour has recorded that the head moves left. The query thus becomes:

```
**Qclass** am-i-next-left **Isa** Cell
**Where**
    right-neighbour.next=left $\wedge$
    right-neighbour.current=yes
**Endqclass**
```

The action to be taken is given in the method become-current-left. The rule thus becomes:

Rule to-be-current-left=(am-i-next-left,become-current-left())

When the cell the head moves to has registered that it is the current cell, the previous cell must set its current attribute to no and erase the movement data in next.

```
**Qclass** am-i-done-left **Isa** Cell
**Where**
    next=left $\wedge$
    current=yes $\wedge$
    left-neighbour.current=yes
**Endqclass**
```

The action of this rule is given by the method not-current-anymore. The rule definition therefore is:

Rule was-current-left=(am-i-done-left, not-current-anymore())

The latter two rules are also defined for a head movement to the right. Their definition is analogous to the rules for the left movement with the appropriate substitutions of left and right.

To give a better insight in how these triggers implement a Turing machine, we will show how these triggers achieve the movement of the head to the left. The two objects of interest are shown with their contents. We start right after the application of an Execute rule on the right cell. The objects' attribute are then as follows:

	Oid=*Newcell*	Oid=*Oldcell*	
	current=no	current=yes	
	next=neutral	next=left	
	from=?	from=?	
	state=?	state=s	
	value=v_1	value=v_2	
	right-neighbour=*Oldcell*	right-neighbour=?	
	left-neighbour=?	left-neighbour=*Newcell*	

Object *Newcell* satisfies the query am-i-next-left, so the rule to-be-current-left is executed on this object resulting in the following attribute valuations:

	Oid=*Newcell*	Oid=*Oldcell*	
	current=yes	current=yes	
	next=neutral	next=left	
	from=*Oldcell*	from=?	
	state=?	state=s	
	value=v_1	value=v_2	
	right-neighbour=*Oldcell*	right-neighbour=?	
	left-neighbour=?	left-neighbour=*Newcell*	

Now that the current status has been set for *Newcell*, *Oldcell* must lose its current status. This is done by the rule was-current-left, that is now executed on *Oldcell*. The result is:

	Oid=*Newcell*	Oid=*Oldcell*	
	current=yes	current=no	
	next=neutral	next=neutral	
	from=*Oldcell*	from=?	
	state=?	state=s	
	value=v_1	value=v_2	
	right-neighbour=*Oldcell*	right-neighbour=?	
	left-neighbour=?	left-neighbour=*Newcell*	

Now the condition for the rule Execute(s,v_1) is satisfied and its action is executed. After that the head is again moved by the bookkeeping triggers.

A final note on the emulation of a Turing Machine by a trigger set and a database is on starting the Turing Machine. The Execute(s,v) rules are triggered by the neutral state of the previous cell in the execution. However, there is no previous cell at the start. Therefore we need an extra cell that is not part of the tape and has its next attribute set to neutral. The from attribute of the cell where the head is positioned at the start of the execution is set to this extra cell.

It is obvious that the head movement is done correctly by the bookkeeping triggers given. For each entry in the transition table of the Turing Machine we can define an execute rule. A transition consists of a tape symbol and a state of the machine resulting in writing a new value on the tape, moving the head left or right and a new state. These can be translated to triggers by filling in these values for s, v, c_1, c_2 and c_3.

In order to show that this emulation of a Turing Machine is the same under instance and set semantics, we have to show that there is no trigger that is invoked on more than one object at the same time. When we look at the conditions of the rules, we see that the attributes of governing the trigger application are current and next. If we start with a correct input state only one cell object will have current set to yes and all next attributes will be set to neutral. It is obvious that as long as there is only one current object any rule is only executed on one object. The only time two objects have their current attribute set to yes is in the movement of the head. This however immediately triggers the was-current-{left,right} rule that sets the current attribute of the previous cell to no. No other rule is triggered in this situation.

Lemma 1 *For each Turing Machine TM there exists a pair $(\mathcal{T}, db)$ with $\mathcal{T}$ a trigger set and db a database state, such that $(\mathcal{T}, db)$ implements TM under both instance and set semantics.*

The fact that we can emulate a Turing Machine in this trigger model, gives a clear indication of the decidability of termination of a trigger set. The halting problem for Turing Machines is known to be undecidable. Therefore termination of a set of triggers is also an undecidable problem.

Theorem 4 *Let $\mathcal{T}$ be a trigger set in model Mk II and let sem denote either instance or set semantics. Then $Terminate(\mathcal{T}, sem)$ and $Confluent(\mathcal{T}, sem)$ are undecidable predicates.*

Proof According to Lemma 1 everything that can be computed on a Turing machine, can be computed by using triggers of this model. Suppose we could decide termination of a set of triggers in the Mk II model. Then we could solve the problem whether a Turing Machines terminates on every input by translating the Turing Machine to a trigger set. However this problem is undecidable for Turing Machines [3]. Therefore termination of a set of triggers in this model must be undecidable.

As we have shown with the Turing Machine simulation, this trigger language codes the class of unary partially computable functions. Clearly the set of confluent trigger sets is a non-empty, proper subset of the set of all possible trigger sets. Rice's theorem [4] thus implies that confluence is undecidable. $\square$

The possibility of simulating a Turing Machine points us to a stronger predicate that is decidable for Turing Machines, viz. termination of a set of triggers in n steps. The restriction to a limited number of steps imposes an upper bound on the time a decision procedure can take. We can simply run the trigger set on a typical database state of sufficient length until we reach the maximum number of steps. We then check whether trigger execution has terminated or not.

A similar strategy is applied to solve a stronger predicate than confluence, pairwise independence of triggers. This means that the application of two triggers is commutative. In both possible execution sequences for two triggers the resulting database state is the same. If all triggers are pairwise independent and the trigger set is terminating, then the trigger set is confluent. We can test this on the typical database state by running both possible executions of each pair and then comparing the result. Thus we have the following theorem:

Theorem 5 *For a trigger set $\mathcal{T}$, semantics sem and an integer number $n > 0$ the predicates $Terminate(n, \mathcal{T}, sem)$ and $Independent(\mathcal{T}, sem)$ are decidable in the model Mk II.*

A consequence of this is that for independent trigger sets termination and confluence are decidable.

Corollary 1 *For an independent trigger set $\mathcal{T}$ and semantics sem, $Terminate(\mathcal{T}, sem)$ and $Confluent(\mathcal{T}, sem)$ are decidable predicates.*

Proof If all triggers in $\mathcal{T}$ are independent, we can rearrange an execution sequence of $\mathcal{T}$ at will. Let $\mathcal{T} = \{T_1, \ldots, T_n\}$. Then any execution sequence of $\mathcal{T}$ can be rearranged into the form $T_1; \ldots; t_1; T_2, ; \ldots; T_2; \ldots; T_n; \ldots; T_n\}$. For each $T_i \in \mathcal{T}$ individually we can decide termination. If each $T_i \in \mathcal{T}$ terminates individually, then it is obvious that $\mathcal{T}$ terminates.

Because we can rearrange an execution sequence of an independent trigger set $\mathcal{T}$ at will, it is obvious $\mathcal{T}$ is confluent. □

6 Decidability Results for the Model Mk III

In this section we examine an extension of the actions relative to model Mk I. The ex5tension is that we allow arithmetic in the action of the rules. We show that for very simple arithmetic termination and confluence remain decidable predicates.

We start with the Mk III model that incorporates the simplest arithmetic function we can think of, viz. the successor function. If we allow the successor function in the action, we can decide termination of a singleton trigger set.

Theorem 6 *Given a singleton trigger set T and semantics sem, $Terminate(T, sem)$ is a decidable predicate in the Mk III model.*

Proof We show that T is a simple vector operation, that is within the decidable part of arithmetic. If we have only a successor function, the action can be written as follows:

$$\vec{a} := \vec{a} + \vec{c}$$

Thus the result of m applications of the trigger is:

$$\vec{a}_m = \vec{a}_0 + m\vec{c}$$

Suppose that the condition is a set of conditions of the form $a_i = a_j$, $a_i \neq a_j$, $a_i = k$ and $a_i \neq k$, where a_i and a_j are attributes and k is a constant. Then deciding termination amounts to deciding the corresponding expressions:

$$\forall \vec{a} \exists m : [\vec{a} + m\vec{c}]_i \neq [\vec{a} + m\vec{c}]_j$$

$$\forall \vec{a} \exists m : [\vec{a} + m\vec{c}]_i = [\vec{a} + m\vec{c}]_j$$

$$\forall \vec{a} \exists m : [\vec{a} + m\vec{c}]_i \neq k$$

$$\forall \vec{a} \exists m : [\vec{a} + m\vec{c}]_i = k$$

These are clearly in the decidable part of arithmetic [6]. □

Because of the strict locality of condition and action and the commutation of the action, confluence is guaranteed in this situation.

Before we look at the decidability of predicates on multiple trigger sets, we first explain the method used in the following proofs to check the validity of an execution sequence. The essence of the method is that we derive constraints on the initial values of the attributes from the conditions of the applied triggers. If contradicting constraints are derived, the sequence is not possible.

We check the existence of a starting point, i.e. an initial valuation of the attributes, for a trigger sequence *seq*. If such a starting point exists, the trigger sequence *seq* is a valid execution sequence. The initial valuation of the attributes a_1 to a_m is i_1 to i_m. T_1 is the first trigger in *seq*. The condition of T_1 is $a_j = a_k$. Therefore we have the constraint $i_j = i_k$ on the initial valuation of the attributes. We then execute the action of the trigger on the valuation vector and check the condition of the next trigger T_2 in the sequence. Suppose that its condition is $a_k = a_h$ and the values of these attributes are $i_k + 2$ and $i_h + 4$, respectively. Thus we derive the constraint on the initial valuation $i_k = i_h + 2$.

It is obvious that the derivation of a constraint that contradicts previous constraints during the execution of *seq*, means that there is no initial valuation of the attributes for *seq*. Therefore *seq* is not a valid execution sequence.

We now apply this method in order to prove decidability of termination for multiple trigger sets in the Mk III model.

Theorem 7 *Let $\mathcal{T}$ be a trigger set in the Mk III model. Let the semantics sem be either Instance or Set. Then, $Terminate(\mathcal{T}, sem)$ is decidable.*

Proof In order to decide the predicate we view the actions of the triggers as vector additions. The goal of the decision procedure is to find a combination of triggers, that makes the conditions of the triggers true.

Since conditions and actions of the triggers are strictly local, we need only consider the termination of a set of triggers defined on the same object o.

To start with we note that if a trigger in $\mathcal{T}$ is individually non-terminating, $\mathcal{T}$ is non-terminating. Termination is decidable for an individual trigger. Therefore in the following decision procedure we may assume that each trigger is terminating individually.

For the procedure to decide termination of a trigger set, we only need to consider those triggers that have conditions of the form $a_i = a_j$. It is obvious that triggers with conditions of the form $a_i = k$ or $a_j \neq k$ either never terminate individually or terminate in a finite number of steps. Thus, we need not consider them if we are looking for a finite pattern of trigger applications that can be repeated infinitely. Likewise, a trigger with a condition of the form $a_i \neq a_j$ will never be individually terminating in this model. For the same reasons, only conditions of the form $a_i = a_j$ are of importance in conjunctions, since we can always find valuations where the part of the form $a_i \neq a_j$ will always be true.

The values of all attributes $a_1, \ldots, a_n$ of an object o can be represented by a vector $\vec{a} = (a_1\ a_2 \cdots a_n)$ The action of a trigger $T \in \mathcal{T}$ may apply the successor function zero or more times to the attributes of the object o. We define the action of T as being the vector $\vec{s} = (s_1\ s_2 \cdots s_n)$ where s_j indicates the number of times T applies the successor function to a_j.

Applying the successor function n times to an attribute a_i has the same result as adding n to it. Therefore the action of a trigger T can be represented by:

$$\vec{a} := \vec{a} + \vec{s}$$

The extension of this to the effect of multiple trigger applications is obvious.

The effect of the application of a trigger T is always the same. Therefore non-termination of a finite trigger set means that the execution sequence of the set has some finite repeating pattern in it. The repetition of a certain pattern means that the truth of the condition of a trigger T_i is preserved by the complete execution of the repeating pattern. Because conditions are of the form $a_i = a_j$, this means that the numbers added to a_i and a_j should be equal. This leads to the following necessary condition for non-termination of a trigger set $\{T_1, \ldots, T_k\}$:

$$m_1\vec{s}_1 + \ldots + m_k\vec{s}_k = \vec{c} \tag{32}$$

where m_i is the number of applications of T_i in the repeating pattern and $\vec{s}_i$ is the action of T_i. $\vec{c}$ is a vector that codes the conditions of the triggers. If a trigger T has the condition $a_i = a_j$, this is reflected in $\vec{c}$ by putting the same constant η at c_i and c_j.

The condition above is necessary for non-termination, but not sufficient. For example it might be satisfied by taking 5 times T_1 and T_2 and T_3 once each. Clearly this cannot be translated to a valid execution sequence if each trigger terminates individually, because it requires successive applications of T_1.

We can use Condition 32 to generate trigger combinations that are candidates for non-termination. We test a candidate to see if it is indeed a non-terminating trigger set. This is done by considering all possible execution sequences of the candidate.

Definition 11 defines a valid execution sequence. It is obvious that we can test whether a sequence is a valid execution sequence in this model.

Let $\mathcal{C}$ be a bag of triggers containing m_1 times $T_1, \ldots$ and m_k times T_k with $\{T_1, \ldots, T_k\} \subseteq \mathcal{T}$ such that Condition 32 is satisfied by $T_1, \ldots, T_k$ with the associated $m_1, \ldots, m_k$. In order to decide whether $\mathcal{C}$ gives rise to a non-terminating execution sequence, we test each possible sequence of the elements of $\mathcal{C}$ for being a valid execution sequence.

Those sequences of $\mathcal{C}$ that are valid execution sequences can be tested for non-termination by checking whether they can be repeated. Suppose a sequence *seq* is a valid execution sequence that starts with $T_i; T_j; \ldots; T_l$. In order to check for non-termination we need to check whether *seq* can be executed for a second time. In other words, we check whether *seq*; *seq* is a valid execution sequence.

If *seq*; *seq* is also a valid execution sequence of $\mathcal{T}$, each trigger-place combination in *seq* can be repeated. Since the effect of all trigger applications in *seq* is the same each time *seq* is executed, this means that *seq* can be repeated infinitely.

In worst case we need to check all possible subsets of $\mathcal{T}$ for Condition 32. The number of subsets of $\mathcal{T}$ is finite. Checking Condition 32 amounts to solving a system of linear equations, which is decidable. The number of possible sequences of the elements of $\mathcal{C}$ is at most $(\sum_{i=1}^{k} m_i)!$. Since all subproblems to be solved in sequence are decidable, the problem of termination of $\mathcal{T}$ is decidable. $\Box$

7 Decidability in Extended Models

In the previous two sections we have examined two extensions of the original Mk I model. Introducing limited arithmetic preserves decidability of termination and confluence. The ability to inspect other objects' attributes in the condition already makes it impossible to decide termination and confluence. The stronger predicates termination in n steps and independence are decidable in the Mk II model however. Combining both extensions of the original model gives us the Mk IV model.

The decidability of various predicates is at best the same as in the separate extensions, the Mk II and Mk III models. Therefore it is obvious that termination and confluence of a trigger set are undecidable. The same proof as used in the Mk II model is applicable to the Mk IV model, since its trigger language is a subset of the language of the Mk IV model.

The decidability of the predicates independence and termination in n steps in the Mk IV model is less obvious. The key to the decidability of these predicates is the construction of a typical database state. For this we need to be able to determine in advance what attribute valuations are possible as a result of the execution of a set of triggers. The problem in determining this in the Mk IV model is in the interaction between applications of the successor functions and the assignment of an attribute value to another assignment.

Other extensions to consider are extensions of the arithmetic allowed in the Mk III model. The expressions we must decide in the Mk III model are in the decidable part of arithmetic. As soon as we have a combination of addition and multiplication in these expressions, they are no longer decidable [6]. It is an open question whether we can avoid this combination in deciding termination and confluence of a trigger set.

Another open issue is the construction of a typical database state for models that combine arithmetic and constant assignment. A necessary condition to this construction is that the attributes vary over a finite set. This condition cannot be satisfied if we wish to decide termination. Termination in n steps limits the number of times each trigger may execute, however. Thus, it should be possible to determine what finite set the attributes can vary over.

8 Conclusions

In this report we have explored some borders of the decidability of database triggers. In the simplest model for condition-action rules termination and confluence are decidable. Next we considered two separate extensions of the model. The first of these extensions concerned the reference to attributes of other objects in the condition. The second considered extension of the action language with arithmetic. We saw that only very limited arithmetic in the form of the successor function is allowed in order to preserve decidability of termination and confluence. The combination of these two extensions does not change the decidability results. A possible further extension is the addition of an event specification to the condition-action rules to form event-condition-action rules as used in many practical systems. Because the event stream is separate from the rest of the data model, this will most probably not affect the decidability results in this report.

The results of this report make clear that properties of sets of database triggers are decidable only for a very simple rule language. Most practically useful active facilities in database systems will have to incorporate a more complex rule language. Therefore we will have to find other ways to obtain the desired properties of trigger sets.

One of the approaches to guarantee termination and confluence are sufficient conditions. It is possible to formulate conditions on trigger sets such that sets satisfying these conditions are guaranteed to be terminating and confluent. Not all terminating trigger sets will satisfy a sufficient condition for termination. A number of terminating trigger sets will thus be rejected for use. We can also take a purely empirical path and monitor the system in use. If the number of successive trigger applications exceeds a preset limit, we cancel the trigger execution. This prevents a system getting stuck in a trigger execution, but does nothing to prevent the re-occurrence of the endless trigger application. Another problem of monitoring is that we cannot monitor for a property like confluence. We would have to repeat a trigger execution twice in exactly the same database state in order to determine whether a set of triggers is confluent.

If we want to overcome the drawbacks of both mentioned approaches, we need a more sophisticated method. We propose to build a learning capability into the active objects such that it is able to learn to avoid the situation where it gets into, for example, a non-terminating trigger application. In order to avoid such situations an active object will be able to change its triggers. An open question is the complexity of the learning method involved. Pragmatic strategies might work very well in most cases. The development of more complicated techniques will draw from game theory. However, at present there have been little results, viz. on non-cooperating players with conflicting interests, that can be applied in this setting.

References

[1] Serge Abiteboul and Eric Simon. Fundamental properties of deterministic and non-deterministic extensions of datalog. *Journal of Theoretical Computer Science*, 78:137-158, 1991.

[2] Alexander Aiken, Jennifer Widom, and Joseph M. Hellerstein. Behavior of database production rules:termination, confluence, and observable determinism. In *Proceedings of the 1992 ACM SIGMOD International Conference on the Management of Data*, pages 59-68, 1992.

[3] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, Englewood Cliffs, NJ, USA, 1981.

[4] I.C.C. Phillips. Recursion theory. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, volume 1. Background:Mathematical Structures*, pages 79-188. Clarendon, Oxford, UK, 1992.

[5] Eric Simon and Christophe de Maindreville. Deciding whether a production rule is relational computable. In *Proceedings of the ICDT 88*, LNCS 326, pages 205-222. Springer, 1988.

[6] Craig Smoryński. *Logical Number Theory I - An Introduction.* Springer Verlag, Berlin, Germany, 1991.

[7] M.H. van der Voort. *A Design Theory for Database Triggers.* PhD thesis, Universiteit van Amsterdam, The Netherlands, September 1994.

[8] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 275-285, 1991.

[9] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the 1990 ACM SIGMOD Conference on the Management of Data*, pages 259-270, 1990.