Electronic Journal of SADIO
http://www.dc.uba.ar/sadio/ejs/
vol. 1, no. 1, pp. 1–20 (1998)

Structural Testing of Active DataBases

Martín Balzamo *Martina Marré* *Daniel Yankelevich*

Departamento de Computación, FCEyN, Universidad de Buenos Aires, Argentina. {tincho/martina/dany}@dc.uba.ar

Abstract

Active databases (ADBs) are databases that include active components or agents that can execute actions. The rise of active databases in the picture of software development has a great impact on software systems and in the discipline of software engineering. However, we still lack the foundations that are needed to adequately support this new tool. These foundations are needed in order to properly apply known software engineering techniques to ADBs and systems that use them. Among the methods and techniques used to improve quality, we count systematic testing. In this work, we generalize structural testing techniques to ADB systems. We introduce a model of active databases, called *dbgraph*, suitable for testing. We show that dbgraphs can be used to generalize structural testing techniques for ADBs. Moreover, we introduce several new structural criteria aimed at find errors in a set of rules for an ADB. We also compare the strength of the coverage criteria presented in this work.

Supported in part by UBACyT grant EX186.

1 Introduction

Active databases (ADBs) are databases that include active components or agents that can execute actions. These databases do not just store data: they transform data, and may implement complex business rules and verifications.

The interest in active databases increased significantly in recent years. From the point of view of research, active databases present new interesting problems [6], and they inherit some others from the Artificial Intelligence (AI) field. In fact, the analogy between active databases and knowledge-based systems with inference rules is immediate.

From the point of view of industry, the interest in active databases has a more practical motivation. Many database vendors are including rules in their engines that may be triggered by the occurrence of certain events. These products do not include all the features offered in active database models, as used in academy. But in one way or the other, current products allow developers to define rules that are executed when a particular event occurs. Most of the main database engines include event/action rules of some kind, and many vendors promise to include more related features in future releases.

The current interest and potential future of active databases can also be illustrated by the inclusion of concepts such as rules and events in some standards and proposals, like ODMG and SQL3 [5].

The rise of active databases in the picture of software development has a great impact on software systems and in the discipline of software engineering. The *the facto standard* of client/server architectures in commercial software systems, which in most cases means passive database servers interacting with active clients, may be influenced if database servers might include rules.

However, we still lack the foundations that are needed to adequately support this new tool. These foundations are needed in order to properly apply known software engineering techniques to ADBs and systems that use them. The discipline of software engineering is now mature enough to promote best practices to apply in order to produce better software. These practices are acknowledged by most software professionals, and aim at positioning software development as an engineering discipline.

Among the methods and techniques used to improve quality, we count validation and verification techniques. In particular, systematic testing is one such *best practice* that is supported by many techniques and tools. Testing involves exercising an implementation, and is the predominant verification technique used in actual production.

Even if commercial systems are now in the market with ADB functionalities, no tool exists supporting systematic testing of active databases. In our view, tools can be developed only once the underlying model has been properly studied and well-known techniques are generalized to this new model. In the case of testing, tools are of great importance, because it is almost impossible to perform systematic testing on a real system without the aid of tools.

Validation and verification of ADBs have been studied from the point of view of static analysis. Static analysis, as opposed to dynamic analysis or testing, performs verification of the system without executing the code. Static verification techniques for ADBs have been presented, for example, in [8, 2].

Some work has been done in dynamic analysis of rule based systems, in the context of AI. The most relevant to ADBs is the work presented in [13]. In that work, the author apply notions from structural testing to rule based systems. While the motivations are very similar, and the goal is to apply structural testing techniques to a particular kind of systems, the models developed and the technical results are quite different. The reason is that the *use* of rule based knowledge bases and active databases is different. Structural techniques take into account the structure of systems, and structure is influenced by function [9]. For instance, the model of [13] for a set of rules are graphs in which each node represents a rule. Clearly, we need a finer granularity (more detail) if we want to analyze rules that may trigger complex actions (that might cause other rules to fire).

Our work is related to testing, and in general to verification, of event/action systems. However, active databases are a particular case of such systems, in which we do not deal with programs: the rules of an ADB have a particular, well-defined form with clear semantics. The semantics change from model to model. In this paper we follow the model of the Starburst system [18], but clearly the results can be easily applied to any model.

In this work, we generalize structural testing techniques to ADB systems. Our final goal is to use these techniques in real systems, and thus it includes the construction of an automatic tool. In this first step, we establish the basis of structural testing in this context. In particular, we show that structural techniques can be applied to active databases by introducing a model of active databases, called *dbgraph*, suitable for testing. Criteria are based on this model. Specifically, we introduce several new structural criteria aimed at find errors in a set of rules for an ADB.

In the next section, we present background that is needed for the work, both on testing and active databases. In Section 3, we introduce the *dbgraph* model and we present some examples. Section 4 is devoted to the definition of testing criteria for active databases based on our model. Also, we use the inclusion relation [16] to compare the strength of the coverage criteria proposed. Finally, in Section 5, we discuss the conclusions and some future work.

2 Background

2.1 Testing

Software testing [15, 1, 4] consists of validating computer programs through the observation of a meaningful set of executions. A complete verification of a program at any stage of the development process can be obtained by performing a test for every element of the domain, i.e., for every possible input. If for each possible input of the program the actual behavior and the expected behavior agree, the program is verified; otherwise a fault has been found. This exhaustive testing method is the only dynamic analysis technique that would guarantee the validity of the program. Unfortunately, this technique is not practical. Frequently, domains are infinite or at least sufficiently large to make the number of required tests infeasible.

Therefore, program testing consists of the validation of the program through the selection of a meaningful subset of all possible executions of the program and verifying that the corresponding outputs are consistent with what the specification says. The subset of inputs selected to test the program is called *test (data) set* or *test suite*. The selection of test data can be guided by different *strategies* or *criteria* for choosing representative elements from the domain. A testing criteria groups together input elements determined by *test cases*. Thus, selection criteria require to test the program using at least one representative element of each subdomain.

A selection criteria can be based on the function or on the structure of a program. There is no controversy between the use of functional testing versus structural testing techniques: both are useful, both have limitations, both target different kinds of error [4].

In *structural testing* the structure of the program is examined in order to analyze the consistency of a component's implementation with its requirements. Typically, in structural based testing criteria the program structure is analyzed on a graphical representation called a *flowgraph* [12], and all the information used to select test cases is implicit on it. The test suite tries to cover some predefined features of the program's control flow (e.g., statements, decisions) or data flow (i.e., relations between a definition and a use of a variable). Control flow and data flow are both essential and complementary testing techniques. Structural testing is probably the most widely used class of program testing. The popularity of these techniques is mainly due to their simplicity and the resulting availability of software tools to assist the testing process.

One of the most difficult problems in testing is knowing when to stop [15]. On the one hand, it is not possible in general to give an answer to whether a test suite guarantees the absence of faults. Therefore, it is useful to have criteria to determine when a program has been tested "enough." On the other hand, we need a way to limit the cost of testing. In fact, if we had unlimited resources we could do all the testing we wanted. In real software projects, we are always short of time or money. Consequently, we need a way to know when to stop the testing process.

Either if the testing process has been planned in advance or the tests have been incorporated step by step, the testing process stops when some *completion criterion* or *adequacy criterion* is satisfied. In theory, the adequacy criterion should be related to some *fault-rate* [14] or some *coverage* [10]. In the second case, different coverages are used to determine when the program has been tested enough. The idea is to guarantee that each statement, decision, or other feature of the program has been executed at least once under some test. Structural testing criteria are frequently used to measure testing coverage, and thus stop the testing process.

2.2 Active Databases

Database systems, in general, are *passive*: they only give answer to queries or execute transactions required by a user or a program. In many applications it is important to monitor different situations and to react to them with certain actions. For instance, the global amount of expenses in an organization cannot exceed the budget. If it does, some information should be transformed or some processing is needed. Hence, particular conditions may trigger a (possibly complex) action. This can be solved in several ways:

In each application: Each application can check for a certain condition and execute the related action. This approach produces software that is difficult to maintain. Moreover, optimizations depend on particular details of each application, and the code is not reusable.

A daemon application: A program can be written to check for relevant conditions and execute actions accordingly. This is a better approach from the point of view of maintenance, but frequent checking causes low database performance. On the other hand, infrequent checking could signal events at wrong times.

An active database system: A database manager monitors situations of interest and executes related actions when they occur. This behavior is represented by event-condition-action rules. We will go through active database systems in the following.

An active database system [6] consists of a (passive) database and a set of *production rules* or *active rules*. The most popular form of active rule is the so-called *event-condition-action rules*, which specifies an action to be executed upon the occurrence of one or more events, provided that some specific condition holds. These rules take the form:
on *event*
if *condition*
then *action*

When its relevant events occur, a rule is said to be *triggered*; after triggering, a rule is *considered*, to see if its condition holds; finally, a considered rule with a true condition is *executed* by performing its action.

This kind of rules allow to implement referential integrity, triggers, alerts, update data derived from other information (e.g., statistics). Moreover, the inference power of the production rules makes active databases a suitable platform for expert systems and knowledge-based systems.

A database system not only provides tools for defining and storing rules, but it also allows their analysis and maintenance. It is common to activate and deactivate them.

Event Specification

In active databases, rules are in general associated with events related to data modification. In relational databases, data modification is carried out using **insert**, **delete** or **update** operations. For example:
define rule MonitorNewEmps
on insert to employee
if ...
then ...

In some languages, events are associated with data retrieval. That is to say, an action can be executed when specific data are read. For instance:

define rule MonitorSalAccess
on retrieve salary
if ...
then ...

In other languages, it is possible to work with transaction events. A rule can be associated with **commit**, **abort** or **prepare-to-commit** operations.

Another kind of events allowed in some managers are time dependent events. They can be absolute (*1/1/95, 8:00:00 AM*), relative (*five seconds after download*), or periodic (*Every Friday at 7:00:00 PM*).

Finally, some languages have operators to compose events. For instance, disjunction or other logical operators.

Condition Specification

A rule condition is a predicate or a query over the database's data. The condition is satisfied according to the predicate's truth or to whether the result of the query is empty. Sometimes, the condition does not exist, meaning that the action will always be executed. Several languages allow referring to data value before and after data modification. For example:

define rule MonitorRaise
on update to employee.salary
if employee.salary $> 1.1 *$ **old** employee.salary
then ...

Action Specification

Rule actions specify the operations to be executed over the database. The operations can be implicitly or explicitly given. For instance, a *rollback* (not written) in a referential integrity rule is an implicit operation. *Delete*, *update*, or *insert* queries are explicit operations. In these queries, deleted data, inserted data, or modified data can be manipulated before and after the occurrence of the event. Finally, an action can be a *commit* or *rollback*. For example:

define rule NewEmps
on insert to employee
then delete employee **where new**.name="WHA*"

In this example there is no condition, the inserted data are used, and the action is executed on each update. This rule says that no register with *name* field beginning with "WHA" can be added to the table *employee*.

Rules Execution

When an event occurs the manager could have to make a choice between different rules related to the same event. This problem can be handled in different ways:

- by forbidding the existence of more than one rule associated with the same event;
- by setting an order over the rules, and executing them according to this order;
- by executing all of them in parallel.

Another problem to be solved is the treatment of infinite loops. The manager can count how many times a rule calls itself and, for instance, stop it after a prefixed number of times.

Finally, it is important for the user to know the rules processing granularity: a tuple or a set of tuples can be modified. The process manager can evaluate *net effect*, or analyze each single operation.

Starburst

Every active database system or language has an underlying model, that fixes the particular meaning and form of execution of the rules. In this work, we apply the techniques to the Starbust model.

Starburst rules are based on the notion of *transition*. A transition is a database change resulting from execution of a sequence of data manipulation operations. Rules consider only *net effect* of transitions [17], meaning that (1) if a tuple is updated several times, only the composite update is considered; (2) if a tuple is updated and then deleted, only the deletion is considered; (3) if a tuple is inserted and then updated, this is considered as inserting the updated tuple; (4) if a tuple is inserted and then deleted, this is not considered at all.

The syntax for defining a rule is:
create rule *name* **on** *table*
when *event*
[**if** *condition*]
then *action*
[**precedes** *rule list*]
[**follows** *rule list*]

An *event* can be an **insert**, **delete**, or **update** operation. A rule can be activated by a transition only if at least one operation occurs in the net effect of the transition. The *condition* is optional, and has the form of a SQL query. The *action* is an insert, update, delete, rollback, or select. The *action* is executed if

- the *condition* does not exist, or
- the *condition* exists, and the result is not empty.

Finally, there is a way of specify a partial order between rules, by means of two statements: *precedes* and *follows*. This order is used if two rules can be activated simultaneously, in order to *guide* the scheduling algorithm.

Testing techniques must be used to complement static analysis techniques. This fact is immediate if the properties of interest are not decidable for the language studied. For instance, if the properties of interest are $p_1, p_2, ...p_n$ and, given any program P, there are algorithms to check the validity of p_i for P, the importance of testing could be discussed. Even in that case, the actual execution of the program may give useful information - in general, it is not true that one knows everything that must be checked in a program.

However, it is important to know the expressive power of the language, in order to have complexity measures of the problem to solve. In our case, it is easy to see that the Starburst model is Turing complete, i.e. any computable function can be expressed by means of a set of rules and a database schema. This fact is given without proof in this paper (see [3]), but one way to prove it, is to translate any Turing machine in a set of rules and codify the tape in one table. A consequence of this fact is that it is not possible to give algorithms to perform static and automatic analysis of programs.

On this work, we use the Starburst model to apply our terminology. This model is rather simple, while preserving the interesting properties of active databases and rules. The techniques presented can be generalized to more complex models if desired. Industrial products, in general, use quite simple models.

3 Dbgraphs

In this section we introduce the notion of *active database flowgraphs* or *dbgraphs*. We assume that the basic concepts of graph theory [11] and of program flowgraphs [12] are known to the reader.

Dbgraphs model rules and their relationship in an active database. Each rule is represented as a dbgraph. A node represents a *decision* (a rule point at which the control flow diverges) or a *junction* (a rule point at which the control flow merges). An arc represents a possible course of the control flow inside the rule.

Let R be a rule. In Figure 3 we present the two possible dbgraphs for a single rule in an active database. There are two possible decisions in a rule: one corresponding to the *if* condition, and the other corresponding to the *where* condition. The dbgraph in Figure 3A corresponds to a "complete rule," i.e., the if condition and the where condition are not empty. There is an *initial node* of the rule representing the activation of the rule and the if condition. Arc a represents the satisfaction of the if condition. Arc b represents that the if condition does not hold. Arc c represents that the action in the rule is executed. And arc d represents the case in which the where in the action is empty, and the action is not executed. The head of arcs c and d are called *final nodes* of the rule, representing the two possible states after the execution of the rule.

When the if condition of a rule is empty, then the rule is represented by the dbgraph in Figure 3B.

Now let $\Re$ be a set of rules. For each $R \in \Re$, let the corresponding dbgraph be $G_R = (N_R, E_R)$. The dbgraph corresponding to the set of rules $\Re$ represents every rule and the interaction between rules, i.e., the possible activation of more rules when a rule is being executed. Hence, in a dbgraph we have two different kinds of arcs: those inside a rule (as we have seen for a single rule), and those between rules or from the final node of a rule to the initial node of a rule, indicating the possible activation of a rule. We consider that a rule R may trigger another rule Q if the condition of Q includes a field of a table that can be modified by R. Note that this invocation relation can be statically detected.

Thus, the dbgraph $G_\Re = (N_\Re, E_\Re \uplus F_\Re)$ corresponding to the set of rules $\Re$ can be constructed in the following way:

1. For each rule $R \in \Re$, for each node $n \in N_R$, then the node $n_R \in N_\Re$.

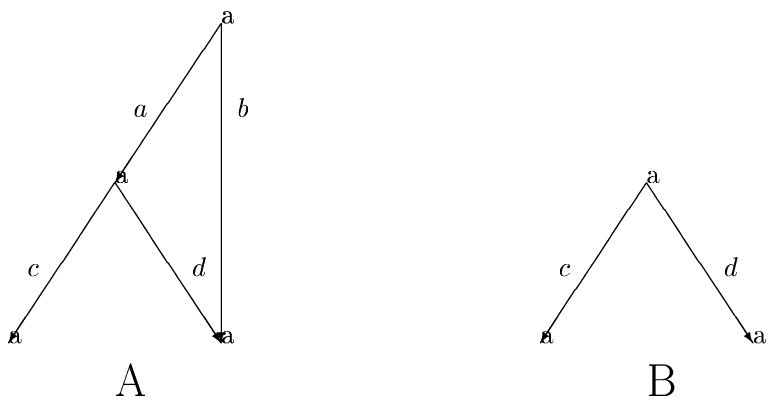

Figure 1: Dbgraphs for a single rule

2. For each rule $R \in \Re$, for each arc $e \in E_R$, $e = (o, d)$, then the arc $e_R = (o_R, d_R)$ is in $E_\Re$.

3. If an execution of rule R can activate rule Q, then there exist an arc in $F_\Re$ from the final node c_R of R to the initial node of Q. Such an arc is called a *firing arc*.

4. Let a_i and a_f two new nodes in $N_\Re$. For each $R \in \Re$, there exists an arc in $F_\Re$ from a_i to the initial node n of R. There exists an arc in $E_\Re$ from each final node n in $N_\Re$ to a_f if there is no arc exiting n.

5. There are no other arcs or nodes in $G_\Re$ then those implied by item 1 through item 4.

Notice that, by construction, every node can be reached from a_i and can reach a_f.

The construction of the dbgraph for an active database can be automated, since the information needed to construct it can be obtained statically.

Let us consider an extension of an example presented in [2]. The following simple database schema contains 3 tables.
emp(id, rank, salary)
bonus(emp_id, amount)
sales(emp_id, month, number)

Table *emp* records each employee rank and salary. Table *bonus* records a bonus amount to be awarded to each employee. Table *sales* records each employee's number of sales on a monthly basis.

The following rule increases an employee's salary by 10 whenever that employee posts sales greater than 50 units for a month.
create rule D *on sales*
when inserted
then update emp
set $salary = salary + 10$
where id in (select emp-id from inserted where number > 50)

The next rule increases an employee's rank by 1 whenever that employee posts sales greater than 100 for a month.
create rule K *on sales*
when inserted
then update emp
set $rank = rank + 1$
where id in (select emp-id from inserted where number > 100)

The next rule increases an employee's salary by 10% whenever that employee's rank reaches 15 (we assume that ranks do not decrease), provided that the sum of all employee's salaries is less that 20000 for a month.
create rule T *on emp*
when updated(rank)
if Select sum(salary) as total where total < 20000 *then update emp*
set $salary = 1.1 * salary$
where id in (select id from new-updated where $rank = 15$*)*

Finally, the following rule increases an employee's rank whenever that employee receives a raise provided that some conditions are satisfied.
create rule V *on emp*
when update(salary)
then update emp
set $rank = rank + 1$
where id in (select id from new-updated where $rank < 5$ *and* $salary < 1000$*)*

In Figure 2 we present the dbgraph for the four rules above. There are firing arcs from D to V, from V to T, from K to T and from T to V, and from node a_i to the inital nodes of all rules.

A *path*, i.e., a sequence of adjacent arcs, from a_i in a dbgraph represents a possible sequence of firings of rules in the ADB. If there is no input that exercises a specific path, then the path is said to be *infeasible*. A *complete path* in a dbgraph is a path such that the first arc in the sequence is a_i and the last arc is a_f. For instance, ${e^i}_K, K_C, {e^i}_{KT}, T_a, T_b, {e^i}_{KV}, V_d, {e^f}_{Vd}$ is a complete path in the dbgraph of Figure 2.

In this work, *test cases* for the ADB are associated with *paths* in a dbgraph. Our goal is to select a set of paths in a dbgraph, in order to test the rules. In the next section, we propose several criteria useful to select test paths in a dbgraph, or to evaluate testing completeness based on the coverage of a dbgraph.

We note that the generation of test data is out of the scope of this work.

4 Coverage Criteria

There are many, possibly infinite, (complete) paths in a dbgraph. Various criteria, or test strategies, can be followed to select a suitable and finite subset of test paths. In this section we introduce several control-flow based coverage criteria for dbgraphs. These criteria use the information in a dbgraph to select a subset of all the complete paths. They can be used to generate test cases or to measure testing thoroughness (i.e., once the rules in a ADB have been tested with test data generated by using some other test generation method, these criteria can be used to check how thorough those test cases have been).

4.1 Coverage Criteria

Let $\Re$ be a set of rules. Let $G_\Re = (N_\Re, E_\Re \uplus F_\Re)$ be the dbgraph for $\Re$. First we present several test criteria that are extensions of sequential testing strategies.

All-Paths This is the strongest criterion presented in this work. It requires to exercise every path in $G_\Re$, i.e., the unique set of paths that satisfies this criterion is the set of all paths in $G_\Re$. Notice that, if $G_\Re$ contains a cycle, then there is an infinite number of paths in $G_\Re$.

All-k-Cycles This criterion provides a restriction of All-Paths, by limiting the iterations of loops in a path. The paths selected are those that do not iterate loops more than k times, for a given integer k. This is to say, a set of paths satisfies this criterion if it includes all paths in $G_\Re$ that do not iterate loops more than k times.

All-Arcs This criterion requires exercising all the arcs in the dbgraph at least once under some test. Thus, a set of paths $\wp$ satisfies this criterion if for each arc e in the dbgraph there is at least a path in $\wp$ including e.

All-Nodes This criterion requires that all the nodes in the dbgraph be exercised at least once under some test. Hence, a set of paths $\wp$ satisfies this criterion if for each node n in the dbgraph there is at least a path in $\wp$ including n.

Now we present several testing criteria that use specific information provided by ADB.

All-Actions This strategy guarantees that every action in every rule of the ADB is exercised at least once under some test. Then, it requires to exercise all the arcs representing an action in the dbgraph. This is to say, a set of paths $\wp$ satisfies this criterion if for each arc e that represents an action in a rule in the dbgraph, there is at least a path in $\wp$ passing through e.

For instance, if

$p_1 = {e^i}_D, D_c, {e^i}_{DV}, V_d, {e^f}_{Vd}$, and

$p_2 = {e^i}_K, K_c, {e^i}_{KT}, T_a, T_c, {e^d}_{TV}, V_c, {e^d}_{VT}, T_b, {e^f}_{Kd}$,

thus, the set of complete paths $\{p_1, p_2\}$ satisfies the All-Actions criterion for the dbgraph in Figure 2.

This criterion adopts a pure state-transition point of view. On this view, only actions are of interest because only them might modify the state of the database. This coverage exercises all actions at least once, and hence all possible "modifications" of the state of the database are executed at least once. Relations among actions are not taken into account.

All-Rules This criterion guarantees that every rule is exercised at least once under some test. Thus, it requires to exercise all the initial nodes in the dbgraph. This is to say, a set of paths $\wp$ satisfies this criterion if for each initial node n in the dbgraph, there is at least a path in $\wp$ including n.

For instance, the set of complete paths $\{p_1, p_2\}$ satisfies the All-Rules criterion for the dbgraph in Figure 2.

This criterion formalizes the intuitive idea of "trying at least one every rule." In the actual testing of an actual database, it guarantees that no rules remain unexplored.

All-Firing-Arcs This criterion requires to exercise all the firing arcs in the dbgraph. To exercise these arcs is important since they represent the calling of a rule by the system and the calling between rules. This is to say, a set of paths $\wp$ satisfies this criterion if for each arc e in $F_{\Re}$, there is at least a path in $\wp$ including e.

For instance, if

$p_3 = {e^i}_V, V_d, {e^f}_{Vd}$,

$p_4 = {e^i}_T, T_b, {e^f}_{Td}$,

thus, the set of complete paths $\{p_1, p_2, p_3, p_4\}$ satisfies the All-Firing-Arcs criterion for the dbgraph in Figure 2.

This criterion reflects that it is important, besides checking every rule, to check every form in which a rule can be activated. Firing arcs between rules can appear in an unexpected way: for instance, a new rule includes an action that modifies a field that triggers another rule. Hence, these forms of activation must be verified; it is not enough to check rules. This criterion can be used when merging databases with rules, when adding new rules to a database, or when upgrading an application that uses rules. On all these cases, it is clever to analyze possible activation among old and new rules.

4.2 Comparison of Active DataBases Structural Testing Criteria

We have presented several coverage criteria for active databases. However, there is no information about how these methods compare. One common way of comparing criteria is to use the inclusion or subsumption relations [16]. In this work, we use the inclusion relation to compare the strength of the coverage criteria proposed in the last section. Let c_1 and c_2 be two coverage criteria from those presented in the last section. We say that c_1 *includes* c_2, written as $c_1 \to c_2$, if for every set of paths $\wp$ satisfying c_1, $\wp$ satisfies c_2 as well. If neither c_1 nor c_2 include the other, then c_1 and c_2 are said to be *incomparable*.

For the criteria presented in this paper, we have the following facts:

- All-Paths $\to$ All-k-Cycles.
- All-k-Cycles $\to$ All-Arcs.
- All-Arcs $\to$ All-Nodes.
- All-Nodes $\to$ All-Actions, since the only way to reach the node head of the arc representing an action, is to pass through that arc.

  All-Actions does not include All-Nodes. As an example, we consider the dbgraph for a single complete rule. The loop-free path $p = a, c$ covers the rule for the All-Actions criterion. However, this path does not cover the HEAD of d, and then the All-Nodes criterion is not satisfied.
- All-Arcs $\to$ All-Firing-Arcs, since the set of firing arcs is a subset of all the arcs in the dbgraph.

  All-Firing-Arcs does not include All-Arcs. For example, we consider again the dbgraph for a single complete rule, and the path p covering c. In this case, d is not covered, and then All-Arcs is not satisfied.
- All-Firing-Arcs $\to$ All-Rules, since to reach the initial node of a rule, the path must contain at least one firing arc (remember that arcs starting at a_i are also firing arcs).

  All-Rules does not include All-Firing-Arcs. For example, a path covering the arc d satisfies the All-Rules for the dbgraph of a single complete rule. However, this path does not satisfy All-Firing-Arcs since the arc from a_i to the initial node of the rule is not covered.

- All-Actions $\rightarrow$ All-Rules, since every rule contains an action and to cover the action of a rule you must pass through the initial node of that rule.

  All-Rules does not include All-Actions. For example, a path covering the arc d satisfies the All-Rules for the dbgraph of a single complete rule. However, this path does not satisfy All-Actions, since arc c is not covered.

- All-Actions and All-Firing-Arcs are incomparable.

  First, suppose that there is a firing arc that triggers the rule R, and the action in R does not trigger any rule. Suppose that a test case is chosen to cover that firing arc, such that it does not cover the action in R. Thus, the covering of All-Firing-Arcs does not guarantee the covering of All-Actions.

  Now, $p_1 = {e^i}_D, D_c, {e^i}_{DV}, V_c, {e^i}_{VK}, T_a, T_c, {e^i}_{KV}, V_d, {e^f}_{Vd}$, and $p_2 = {e^i}_K, K_c, {e^i}_{KT}, T_b, {e^f}_{Td}$ are two complete paths in the dbgraph of Figure 2. And $\{p_1, p_2\}$ satisfies All-Actions, but does not satisfy All-Firing-Arcs. Thus, the covering of All-Actions does not guarantee the covering of All-Firing-Arcs.

- All-Nodes and All-Firing-Arcs are incomparable. Let $p_3 = {e^i}_D, D_d, {e^f}_{Dd}$, and $p_4 = {e^i}_K, K_d, {e^f}_{Kd}$. Thus, the set of complete paths $\{p_1, p_2, p_3, p_4\}$ in the dbgraph of Figure 2 satisfies All-Nodes, but not All-Firing-Arcs, since arc ${e^i}_T$ is not covered.

  Let $p_5 = {e^i}_T, T_b, {e^f}_{Td}$, and $p_6 = {e^i}_V, V_d, {e^f}_{Vd}$. Then, the HEAD of arc K_d is not covered by the set of complete paths $\{p_1, p_2, p_5, p_6\}$, that satisfies All-Firing-Arcs.

Then, the family of criteria presented in this paper is partially ordered by inclusion, as shown in Figure 3. In fact, a criterion c_1 includes a criterion c_2 if and only if the inclusion is explicitly shown in Figure 3 or follows from the transitivity of the relations.

5 Conclusions and Future Work

In this work, we have presented dbgraphs, a model for active databases. We have shown that it can be used to generalize structural testing techniques for active databases. We have also defined new testing criteria based on the information provided by the rules in an ADB. In this way, the criteria introduced are not just a generalization of existing criteria: they take into account the nature of active databases.

In order to establish whether these criteria are useful or not, adequate experimentation must be performed. Only after using the criteria in a significant set of cases we might take definitive conclusions. Therefore, the next step is to build a testing tool based on the results presented here. This tool will construct a model from a set of rules (getting the information from the Data Dictionary or metamodel) and will help in checking satisfaction of criteria and in the selection of inputs. It is also very simple to generalize structural complexity metrics [7] to the case of active databases, letting the tool calculate such metrics.

Moreover, when the tool will be ready, it will be easier to perform experiments with different kind of systems to both refine the model and define new criteria. Besides, it will help to analyze whether some concepts (for instance, structural complexity) are meaningful in this context.

The model presented in this work is based on the control flow in a rule and between rules. We plan to further extend the results obtained to data flow structural testing.

Our work is based on the Starburst model. This model is very simple. However, the results can be generalized, because the complexity of other models might be handled using techniques already applied in procedural languages [1]. The *integration* of techniques is also an interesting problem. For instance, suppose we are analyzing a complete system, part implemented via the active database and part in a procedural language. It is too simplistic to consider that if we cover both parts separately we are covering it as a whole. Somehow, we need to integrate what is done in the procedural part with the model for the active database rules. Database languages used in industry allow actions to be specified in a procedural language. In order to perform experimentation on real systems, we must solve this integration between ruled based and procedural models for structural testing.

References

[1] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky. Validation, verification and testing of computer software. *ACM Comp. Surveys*, 14(2):159–192, June 1982.

[2] A. Aiken, J. Hellerstein, and J. Widom. Static analysis techniques for predicting the behavior of active database rules. *ACM Trans. on Database Systems*, 20(1):3–41, March 1995.

[3] M. Balzamo. *Testing sobre Bases de Datos Activas.* Tesis de Licenciatura, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 1997.

[4] B. Beizer. *Software Testing Techniques, Second Edition.* Van Nostrand Reinhold, New York, 1990.

[5] R. G. Cattell. *Object Data Management.* Addison Wesley, 1994.

[6] U. Dayal, E. Hanson, and J. Widom. Active database systems. *Modern Database Systems: The Object Model, Interoperability, and Beyond. W. Kim, Ed. ACM Press, NY*, 434–456, 1994.

[7] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A rigorous and practical approach.* PWS Pu. Co., 1996.

[8] P. Fraternali and L. Tanca. A structured approach for the definition of the semantics of active databases. *ACM Trans. on Database Systems*, 20(4):414–471, March 1995.

[9] R. P. Gabriel. *Patters of Software.* Oxford University Press, 1996.

[10] J. B. Goodenough and S. L. Gerhart. Towards a theory of testing: data selection criteria. In *Current trends in programming methodology, vol. 2, R. T. Yeh Ed. Prentice-Hall, Englewood Cliffs, N. J.*, pages 44–79, 1977.

[11] F. Harary. *Graph Theory.* Addison Wesley, 1969.

[12] M. S. Hecht. *Flow Analysis of Computer Programs.* North Holland, 1977.

[13] J. D. Kiper. Structural Testing of Rule-Based Expert Systems. *ACM Trans. on Soft. Eng. and Methodology*, 1(2):168–187, April 1992.

[14] J. D. Musa and A. F. Ackerman. Quantifying software validation: when to stop testing? *IEEE Software*, 19–27, May 1989.

[15] G. J. Myers. *The Art of Software Testing.* Wiley, New York, 1979.

[16] E. J. Weyuker and S. N. Weiss and D. Hamlet. Comparison of program testing strategies. *Proc. ACM SIGSOFT Fourth Symposium on Software Testing, Analysis, and Verification (TAV4)*, 1–10, 1991.

[17] J. Widom. The starburst rule system: language design, implementation and applications. *Special Issue on Active Databases, IEEE Eng. Bull.*, 15(4):15–18, 1992.

[18] J. Widom, R. Cochrane, and B. Linsday. Implementing set-oriented production rules as an extension to starburst. In *Proceedings of 17th International Conference on Very Large Data Bases, VLDB Endowment*, pages 275–285, 1991.

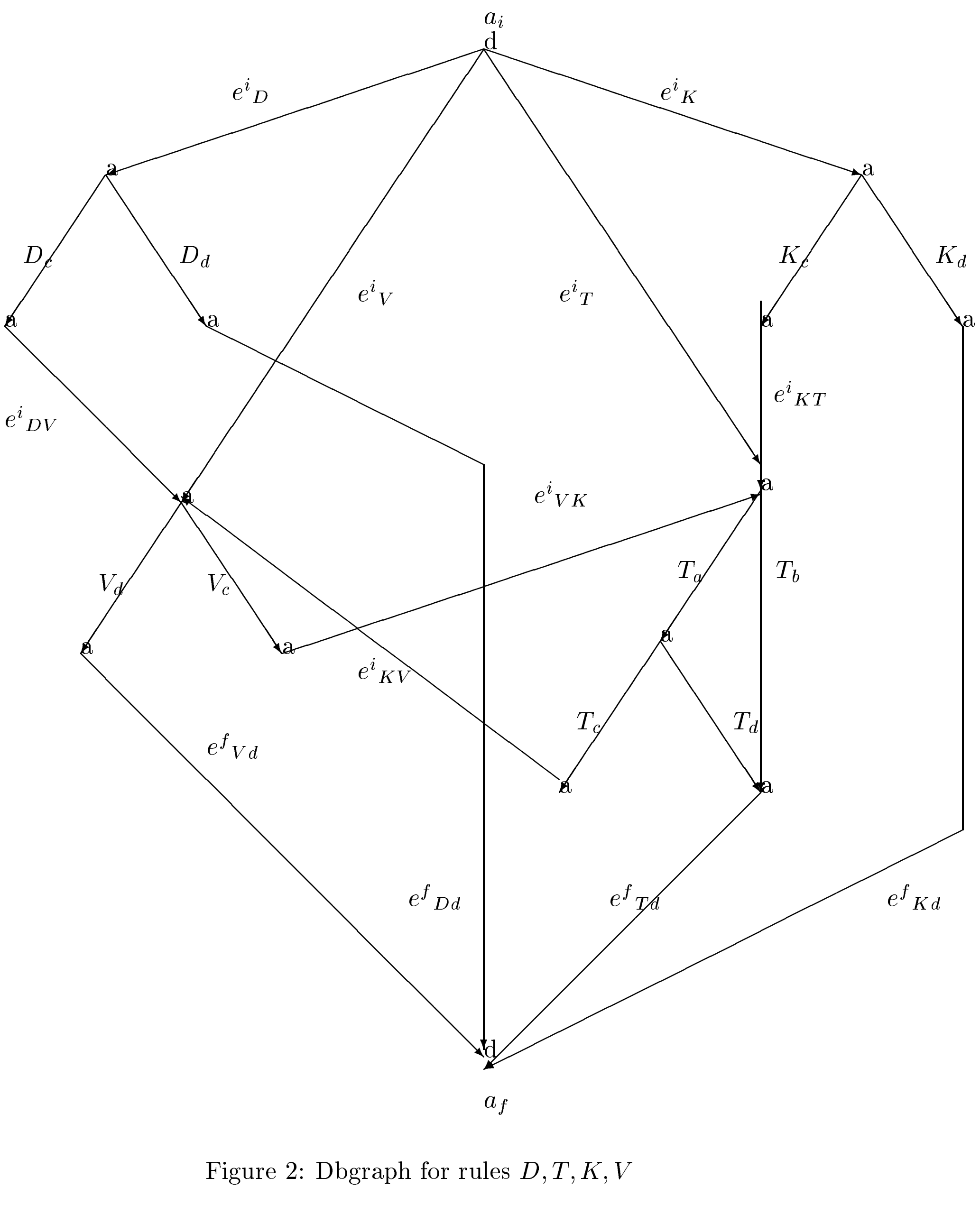

Figure 2: Dbgraph for rules D, T, K, V

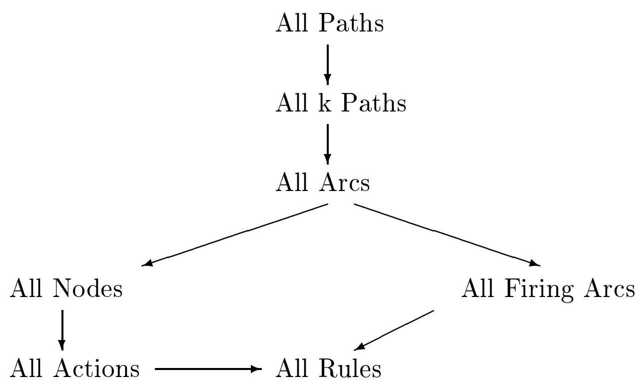

Figure 3: Relationship among the Structural Testing Criteria for Active DataBases