Automated analysis of dependencies in advanced transaction models

Daniel Owens
Department of Computer Science, Colorado State University
Fort Collins, CO 80523, USA

Abstract

Researchers have formalized the notion of transactional dependencies and shown how these dependencies can be ordered and combined in such a way as to express various advanced transaction models. The problem has been that both the complexity of these models and the way in which the dependencies interact can make manual analysis cumbersome and error-prone. In this vein, this work enumerates fifteen different dependencies that are often found in such models. Using the given dependencies, transaction models can be built; however, the models must be checked for three different issues: redundant, conflicting, and "ignored" dependencies. These issues are described in the context of multiple dependencies per a subtransaction as well as over the course of multiple subtransactions. An automated technique for checking any given model that will attempt to simplify the model and detect both conflicts and superfluous dependencies is also described.

1 Introduction

Thanks to the introduction of more complex and longer running database transactions researchers have been forced to create specialized models that adequately describe and process such transaction models. The specialized models have since been turned into a more abstract model that has more applicability than any one of the individual models. While the abstract transaction-processing model has been well defined [9] for theoretical usage, it lacked both practical definitions and an automated process for detecting conflicts and redundancy between dependencies. Furthermore, researchers were lacking key theories required to completely minimize a given transaction model.

By building on the theoretical definitions and concepts presented by researchers, one can see that the abstract transaction-processing model is capable of usage by industry. The new practical dependencies provide a platform for further research into the actual usage of this model and, hopefully, eventually lead to the model being feasible for widespread, real-world use. Given this, the work provided herein is meant as an attractor to both researchers and industry, as well as to prove the abstract transaction-processing model's feasibility.

The goal of this work is to give practical definitions, based on the theoretical ones, for fifteen of the sixteen known dependencies. After the practical definitions are given it is possible to discuss ways in which dependencies can conflict or be redundant, regardless of whether this occurs over the same edge, multiple edges, or an opposing edge. A new concept, that of "ignored" dependencies, is then discussed. The work then takes those definitions and concepts and seeks to prove that models can be minimized in an automated method, relying on a proof-of-concept. Lastly, the results of the proof-of-concept are given to allow researchers and industry to use the results without having to run the proof-of-concept.

2 Practical definitions for the dependencies

- *Abort* (a): If T_i aborts, T_j must also abort; T_j cannot terminate prior to T_i.
- *Begin* (b): T_j cannot begin until T_i has begin; order is enforced.
- *Begin-on-abort* (ba): T_j cannot begin until T_i aborts; order is enforced. Note that if T_i never aborts, T_j is never run (e.g., if T_i commits, T_j is not run).
- *Begin-on-commit* (bc): T_j cannot begin until T_i commits; order is enforced. Note that if T_i never commits, T_j is neverrun (e.g., if T_i aborts, T_j is notrun).
- *Commit* (c): If both T_i and T_j commit, T_i must commit first; order is enforced.
- *Exclusion* (ex): If T_i commits and T_j is executing, T_j is aborted.
- *Force-begin-on-abort* (fba): If T_i aborts and T_j has not begun, T_j will begin.
- *Force-begin-on-begin* (fbb): If T_i begins and T_j has not begun, T_j will begin.
- *Force-begin-on-commit* (fbc): If T_i commits and T_j has not begun, T_j will begin.

- *Force-begin-on-termination* (fbt): If T_i terminates (either aborts or commits) and T_j has not begun, T_j will begin.

- *Force-commit-on-abort* (fca): If T_i aborts, T_j must commit; T_j must commit prior to T_i aborting.

- *Serial* (s): T_j cannot begin until T_i has terminated (either aborts or commits); order is enforced.

- *Strong commit* (sc): If T_i commits, T_j must also commit; T_i cannot terminate prior to T_j and if T_j aborts, so must T_i.

- *Termination* (t): T_j cannot terminate (commit or abort) before T_i terminates (commits or aborts).

- *Weak abort* (wa): If T_i aborts and T_j has not committed, T_j will abort.

A list of the theoretical definitions can be found in [11] and [9]. The theoretical definitions for the dependencies are generally similar to the practical definitions. The first difference exists in the abort dependency. The abort dependency, in theory, does not require an order; however, the only way to enforce the dependency in a practical model is to do just that. We say instead that T_j cannot terminate first because if it were to commit, it would still be possible that T_i aborts, thus violating the abort dependency.

The second difference is with the force-commit-on-abort dependency. This dependency requires that T_i not be allowed to begin until T_j commits to ensure that T_j does, in fact, commit. The new definition restricts this model to forcing T_j to commit just to allow T_i to run, but is logical.

The last difference is with strong commit. Strong commit now includes that T_i terminates after T_j so as to prevent T_j from aborting, as we cannot force something to commit. The differences are shown using the theoretical definitions in Section 10 so as to make the new definition differences relational and obvious.

3 SWI-Prolog

In order to automate the process of the detection of issues in a given transaction model, one must be able to build an application that can decide if two or more dependencies conflict or are able to be minimized. To do this, one needs a way of having the application be both intelligent and logical. These constraints led the author to choose SWI-Prolog, as Prolog is a logic language, and SWI-Prolog conforms to the ISO standard while still being highly flexible.

Prolog was used for the back-end, which meant that anyone can give the definitions of each of the dependencies as well as what constitutes a conflict situation or an include situation into the Prolog code. In order to discover problematic dependencies, two separate Prolog applications were created: one that detects conflicts and one that detects implied dependencies. A third was made to aid others in minimizing a given model without having to know why a dependency could not be added to the model. The third application (Appendix A) is merely a combination of the first two and first checks to

see that there are no conflicts and then checks to ensure that the dependency did not already exist either directly or as an implied dependency (as described in Section 12).

One of the main benefits of using Prolog to write the actual detection engine is that Prolog can be used as a database. Users are able to assert information into the engine and check the database for conflicting assertions in a logical manner. Furthermore, Prolog naturally recurses through the database in attempts to discover whether the goal is known–if the goal is not known the query fails. One issue with the use of Prolog is that it always constructs a tree of assertions for any query and performs a depth-first search, so it can be inefficient. This can be overcome, however using threads, which are naturally supported by SWI-Prolog, thereby allowing multiple traversals of the tree concurrently. Prolog offers tremendous flexibility to the programming of the back-end and the database capabilities are invaluable. [3, 4, 5]

4 Java and InterProlog

Java is used to create the front-end for the automation process as well as to create a Graphical User Interface. First, a simple Java program was created to iterate through all combinations of the fifteen dependencies (excluding comparing the dependency against itself). The results of the first run whereby the edges ran in the same direction are contained in Table 2, which describes conflicts that arise over the same direction. Table 3 describes the results when the program was made to iterate through the combinations with the edges in opposing directions. A second application, the Graphical User Interface, from here-to-forth described as the front-end, is designed to allow those who do not know Prolog to use the proof-of-concept. It allows the user to add edges and displays a graph of the minimized, conflict-free, and ignored dependency-free subtransaction set with the proper dependencies.

The basis for using Java was two fold. First, Java naturally supports threads, which greatly increases the speed of the iteration process in multiprocessor environments. Second, SWI-Prolog's C++ headers seemed to be incompatible with the author's version of g++.

In order to have Java and Prolog exchange information, the use of InterProlog was required. While SWI-Prolog came with JPL, InterProlog offers greater power in the data exchange process between the Prolog engine and the Java engine. One problem with InterProlog, however, is that it used sockets to talk with the Prolog engine, rather than JNI (Java's native interpreter). The time taken to start and shutdown a Prolog engine builds on the runtime of any application using it. This is likely unnoticed by most users.

5 Analysis of the proof-of-concept

5.1 Tracing through the proof-of-concept

The backbone of the proof of concept relies on Prolog. The main file is transactionanalyzer.pl (Appendix A.1), which loads the driver, buildall.pl (Appendix A.2). The most

commonly used term[1] is likely to be add/3[2] as it allows us to see if we are able to add the desired information without creating a conflict and while still having a minimized model.

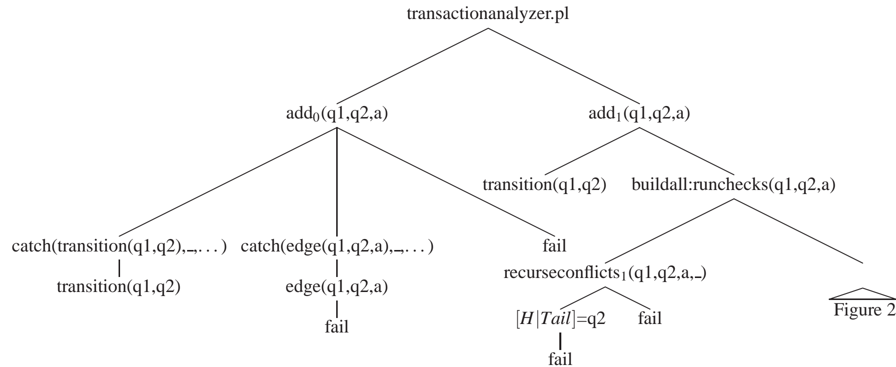

Figure 1: Partial call stack for add/3

Upon calling add/3, Prolog builds a tree containing all terms that might work[3]. Figure 1 shows part of the tree; it should be noted that Prolog builds the tree as need be and performs a depth first search. We begin in transactionanalyzer.pl and start with the first add/3 that might work; in this case, we actually use the first add/3 term in the file. This term has several conditions that must be met prior to the term being declared as true. First of all, we see that we make a call to the catch that checks if the transition even exists. We can safely assume that if the transition/2 term does not exist, we can add the edge without creating any issues. In order to trigger the catch we must try to find transition/2, so Prolog checks if the transition exists. If the term is unknown, the catch prevents any issues and instead adds in both the transition/2 and the edge/3. If the term is known and this specific transition exists, we do not add the edge, but continue through the tree. If the transition does not exist, this branch fails and we must backtrace and try another path through the tree.

Since the left branch failed altogether, we backtrace and begin to traverse the right branch. We use the second add/3 term in transactionanalyzer.pl, as designated by the subscript $_1$ in the tree. First, we check to see if the transition exists in the database, which it does in this case. We then use buildall.pl's runchecks/3 term, which makes calls of its own. runchecks/3 is true if all conditions exist in the database that define the term are true. The term is true if and only if there are no conflicts, no includes, the model is minimized, and we can successfully add the dependency's definition to our database. We begin evaluating the truth of the term with our usual tree structure and depth-first search.

Figure 2 shows us the continuation of the call stack from Figure 1's right branch. We begin this subtree from recurseconflicts/4 and have getallpaths/3 find all paths from

[1]A term consists of the functor and parameters listed in parentheses and separated by commas

[2]%functor%/n signifies that we are referring to those terms with n parameters and of the name %functor%

[3]e.g., if a term takes an empty list and you do not pass such an atom, this term will not be in the tree even though the name and arity may be correct

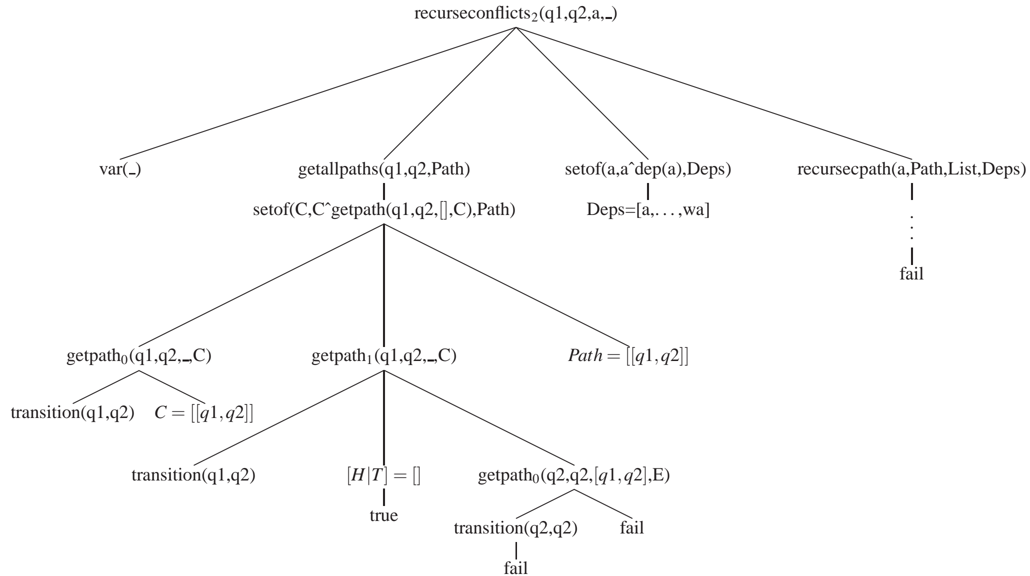

Figure 2: Partial call stack for recurseconflicts$_2$/4

the starting node to the destination node using setof/3, which is a native Prolog term. This term finds the set of all answers, orders them in the given order[4], and sets the third object passed to equal the created set. In this example, we branch to the first getpath/4 term in our database. The purpose of the getpath/4 terms is to find a path from a given node, to a given node such that non-trivial paths are not taken. This means that we will not take infinite loops, or even two, when present. Our first getpath/4 term merely attempts to see if there is a path from one node to the other, with no intermediates taken.

In this case, such a path exists, so C is set to [[q1,q2]]. Because setof/3 attempts to find all unique solutions, it will now call on the other matching term, which is the second getpath/4 term in the database. This term provides for finding a path over multiple nodes; if it fails, we know that no path exists over multiple nodes. In this example, we see that there exists a path from q1 to q2, so getpath/4 calls trying to find a path from q2 to the destination. The only way that this path would exist would be to have a loop on q2, which we do not, and so this branch fails. The setof/3 term collects the answers, organizes them, and then sets Path to be equal to the set of answers, which is now [[q1,q2]]. By mere chance, this is the result of only one call being successful and would likely not occur in a more complex example.

Now that we have a set of unique paths, we must traverse them, looking for anything that would conflict with our given dependency. We see that the traversal is as simple as checking at each node to see if conditions exist such that we would violate the requirements of our dependency or such that our dependency would violate the requirements already in place. After conflicts are checked for, if none exist, we look to

[4]For our purposes, we order alphabetically

make certain that the dependency is not already implied. If the dependency is not, we then make certain that anything it implies is removed from the database and now add the dependency's definition to the database. Lastly, the edge is added to the database and add/3 returns true.

5.2 Big O

A simple examination of the call stack reveals that the program speed is slowest when traversing $recurseconflicts_2/4$. This term performs two tasks that can take time: looping through all paths from Start to End, enumerating the paths while looping, and then actually traversing the paths and looking for any conflicts. The time taken to enumerate the paths from a subtree with m branches and n nodes per branch (all paths lead to the same leaf from the same root) is O(m*n). This is a rather trivial Big O, but is only narrowly beaten by $recursecpath_1/4$, which gives us the time taken looking for any conflicts.

Assume that we have a subtree with m branches and n nodes per branch. If we attempt to add either an edge with fca or sc going from the subtree root to the subtree leaf (all paths lead to the same leaf from the same root), we will be forced to check all edges for conflicts. Having already enumerated the paths, which took O(m*n) time, we must now examine for conflicts. We know that we will once again have at least O(m*n), since that is the number of nodes that we must examine en masse, but we also can see that we have to find one more piece of information: what the conflicts are for each of the dependencies on the edge to be added as well as those dependencies in the edges on the path.

This means that we introduce a third variable, x, which is the total number of dependencies being examined per edge (the number of dependencies on the edge being traversed, plus the number of dependencies on the edge to be added)). We are guaranteed that x be greater than or equal to two, as a result of both edges having at least one dependency. Our tree can have an infinite number of paths and depth, but we can only have a finite amount of dependencies per edge, which is further constrained by the issues that occur with the different dependencies, so we cannot take the limit as x approaches infinity. This means that we have x*m*n, which results in our order being (m*n) since x can be assumed to be a constant. This is the most complex order, so we can see that the proof-of-concept (Appendix A) has a worst case of O(m*n), since x is trivial.

6 Detecting issues with composite dependencies

6.1 Detecting conflicting dependencies

Because we can logically state that a given dependency conflicts with another, we can create a Prolog back-end that will detect such conflicts. Appendix A shows a proof-of-concept that can demonstrate the ease with which Prolog can deduce that two dependencies conflict either as composites or over a series of nodes.

If we return to Figure 2 we can see that the right branch continues through the depth-first search and eventually returns *fail*. A close look at the proof-of-concept shows that we will recursively traverse the paths discovered, making certain to traverse all of the paths, and collect information about all of the conflicts that could arise via each path. To detect conflicts with composites, we perform the same set of tasks that we would were we looking at conflicts over multiple nodes. We will collapse the paths from the source to the destination, querying for conflicts as we go. What we have after collapsing the nodes is a set containing all possible conflicts that would arise were we to add an edge from the source to the destination. Simply put, we now check our desired dependency to see if it is in the set. Either it is in the set or it is not. If it is not, we must move on and check for other violations that would cause our model to be either inefficient or incorrect.

An example of a composite conflict is when you have an edge $T_i \rightarrow T_j$ contain the dependencies strong commit and terminate. Strong commit requires that if T_i commits, T_j must commit. We cannot force something to commit, so we have to have T_j terminate first. If T_j aborts, T_i must also abort. If, however, T_j commits, T_i is allowed to terminate in any state. By having terminate on the same edge as strong commit, we are attempting to have T_i terminate before T_j is allowed to. The restrictions on the edge are now such that nothing can occur and the database cannot proceed; thus, we have a conflict and since the conflict is between two dependencies on the same edge we say that we have a conflict between composite dependencies.

If we were to separate how we detect composite dependency conflicts we would simply examine all direct edges for conflicts. This would mean that we would not build a path, but rather would call directly to cdefs(%dependency%,%from%,%to%). For efficiency, however, the detection of composite conflicts has been merged with the detection of conflicts over multiple nodes, since we have to check both prior to accepting the transition as being conflict free. This optimization serves two purposes: complexity is reduced and redundant calls are eliminated. If we were to perform the checks independent of one another, we would actually check for composite conflicts twice: once on the direct call and again when traversing the path and recursively checking for conflicts.

6.2 Detecting implied dependencies

One form of superfluous dependencies that can arise in a database are those that are implied, but exist as an edge. If the abort dependency exists on the edge $T_i \rightarrow T_j$ and we were to add a composite dependency such as the terminate dependency, we would be attempting to add a dependency to the edge that already exists. This is because terminate states that T_i will terminate prior to T_j terminating. Abort already states that T_i will terminate prior to T_j, even though it defines the way with which the termination will occur (T_j aborts if T_i aborts). Therefore we can say that the terminate dependency already exists on the edge, however indirect the existence is, and regardless of the stricter definition imposed by abort.

Detecting implied dependencies is as simple as checking to see if all of the restrictions imposed by a given dependency already exist [9]. If they do, the dependency already exists in the model, even if the existence is an indirect one. If even one condi-

tion is not already present in the database, the dependency is not implied and may be added without redundancy.

6.3 Detecting other superfluous dependencies

There exists a second form of superfluous dependencies, which has gone unnoticed by researchers: those that are ignored. Certain dependencies prevent a case in which other dependencies can be run. This is not a conflict, however, because a conflict requires an environment such that neither of the two dependencies may run without violating the other. In this case, one dependency creates an environment such that the other cannot be run, however it is not prevented from running by the other dependency.

A simple example of this would be if the commit dependency were a composite with the begin-on-abort dependency over the subtransaction pair $T_i \rightarrow T_j$. The commit dependency states that if both nodes commit, T_i must commit before T_j. The begin-on-abort dependency also imposes order restrictions on the edge; it requires that T_j not begin until T_i aborts. The way in which begin-on-abort works prevents a situation such that T_i and T_j both commit as T_j will not be run if T_i commits. This means that the commit dependency will not be run. Given this, we must look at the requirements of the commit dependency; if it imparts restrictions that prevent begin-on-abort from being allowed to run a conflict exists. Commit, however, does not require that both subtransactions commit, only that if they do, they commit in a certain order. Given this, we know that commit and begin-on-abort do not conflict, but that commit is overridden and ignored. Therefore, we say that we have begin-on-abort as an overriding dependency and commit as an ignored dependency when the two are paired on the same edge.

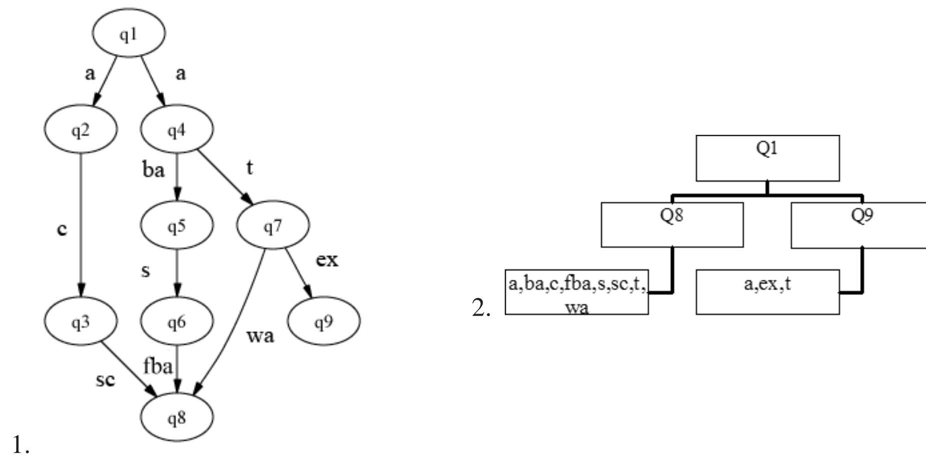

Figure 3: Simple subtransaction model

7 Detecting conflicting dependencies over multiple nodes

Conflicts can be detected over multiple dependencies simply enough using the proof-of-concept (Appendix A). If a model were created whereby we have the subtransactions in Figure 3.1, we can see that we have a conflict by collapsing the nodes going from q_1 to q_8, as Figure 3.2 shows. We now can clearly see that we cannot add the edge from q_1 to q_8. The logic behind this is largely that requirements are introduced such that we cannot traverse the desired edge and the intermediate edges without having conflicting requirements. It should be noted that the approach taken traverses all paths from nodeA to nodeX, so the order does not matter. For instance, if we were to take a simple model that has two edges, $r_1 \rightarrow_{sc} r_2$ and $r_2 \rightarrow_{sc} r_3$. If we were to try adding a third edge to connect r_1 directly to r_3 and give it any dependency that conflicts with the strong commit dependency, the proof-of-concept would return that the call failed. If we were to add the edge containing the conflicting dependency first and then add in the strong commit edges, the call would fail again.

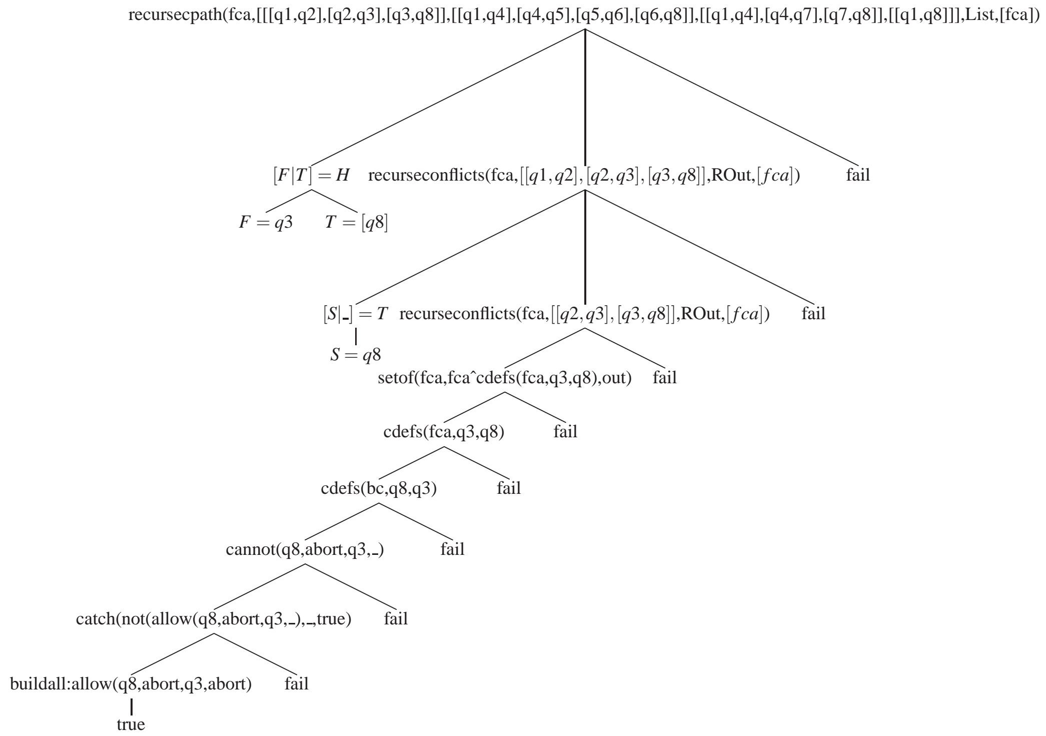

Figure 4: Call stack for recursecpath$_2$/4

The complexity involved in checking all paths is greatly reduced thanks to recursion, however the call stack can become large and cumbersome to diagram. In short, if we are attempting to build this model and wish to add the edge $q_1 \rightarrow_{fca} q_8$ we would create a call stack similar to Figure 4. It should be noted that we do *not* conflict for

any other reason than that there are multiple paths that start from a single location, traverse to another location, and the *paths* conflict with one another. When detecting for conflicts over multiple nodes we are actually checking to see if there is another path that places constraints such that neither path can be traversed [9]. The complexity in checking conflicts over multiple nodes and paths is largely due to a model's complexity.

8 Minimizing a model

The minimizing of a model occurs on-the-fly in the proof-of-concept. This is because implied dependencies and conflicting dependencies are never added to the database, as well as the removal of implied dependencies [9]. If terminate were added to an edge prior to abort, the method for detecting implied dependencies would not remove terminate. Instead, a static term is used: minimize/3 (Appendix A.2). The term makes certain that if a case occurs such as the above, any dependencies that are implied by the new dependency are removed from the edge. For instance, in the above case, before the abort dependency is actually added to the database and edge terminate would be removed from the edge. This helps to ensure that the database has been minimized, other than just looking for conflicts or making certain that the edge being added is not already implied.

9 Detecting equivalent models

Two models are said to be equivalent if they imply the same things [9]. Given this, equivalence can be found if a list of all implied dependencies is created for each model and compared. The back-end provides a simple way of listing the implied dependencies, but a front end must be used to make the comparison. The limitation is simply that multiple engines must be run with a model in each engine. Simply call recurseincludes/4 for each model and see if all transitions in either model are implied in the other model. If they are, there exists an equivalence relation between the two models.

10 Practical dependencies

Dependency	Practical Dependency
$T_i \rightarrow_a T_j$	$T_i \rightarrow_t T_j$
$T_i \rightarrow_{fca} T_j$	$T_j \rightarrow_{bc} T_i$
$T_i \rightarrow_{sc} T_j$	$T_j \rightarrow_a T_i$,$T_j \rightarrow_t T_i$

Table 1: Dependency inclusion relationships

There exist a handful of dependencies that are required by a given dependency in order for that dependency to function. An example of this would be with the abort dependency. This dependency states that if T_i aborts, T_j must abort. We can note that no order is explicitly stated, but that we have to have an implicit order. If T_i were

to abort after T_j has committed, a problem would exist. We assume that there is an implicit order such that T_i terminates first, preventing such a scenario. By making such an assumption, we are stating that there exists a practical dependency, $_t$, on the same edge.

By using this logic, we can see that only three dependencies seem to require any practical dependencies to enforce their definitions: abort, force-commit-on-abort, and strong commit. Force-commit-on-abort requires that T_j commit if T_i aborts; while more restrictive than the definition, the only practical solution is to place a begin-on-commit dependency on the reverse edge. Strong commit requires that if T_i commits, T_j must also commit. Since we cannot *force* something to commit, we must state that if T_j aborts, T_i aborts and rely on the practical abort dependency[5].

Practical dependencies are not always logically derivable, but often must be directly input into the back-end code. By doing this, the user does not have to add the practical dependency transition, such as they would if it were not coded into the back-end. This also means that we can be certain that all checks include *everything* required by any given dependency.

11 Conflicting dependencies

11.1 Conflicting composite dependencies

Dependency $T_i \rightarrow T_j$	Conflicting Dependency $T_i \rightarrow T_j$
a	fca,sc
b	fca
ba	bc,fbb,fbc,fbt,fca,sc
bc	ba,fba,fbb,fbt,fca,sc
c	fca,sc
ex	sc
fba	bc
fbb	ba,bc
fbc	ba
fbt	ba,bc
fca	a,b,ba,bc,c,s,sc,t,wa
s	fca,sc
sc	a,ba,bc,c,ex,fca,s,t
t	fca,sc
wa	fca

Table 2: Dependencies that conflict when presented in the same direction

Dependencies can conflict in two separate ways, the first being when presented as

[5]Terminate is a practical dependency of strong commit because abort is a practical dependency of strong commit

composites. Table 2 lists all know instances where composites conflict. Such conflicts occur in cases such as: $T_i \rightarrow_a T_j$ and $T_i \rightarrow_{fca} T_j$. The abort dependency requires that if T_i aborts, so must T_j, while force-commit-on-abort states that if T_i aborts, T_j will commit. The two dependencies cannot be fulfilled, so are said to conflict as one requires T_j to abort and the other to commit. Conflicts such as this are the easiest to detect because you lack the complexity of working with multiple subtransactions.

11.2 Dependencies that conflict over multiple subtransactions

The second way that dependencies can conflict is when they are somehow linked over multiple transactions. Take, for instance, Figure 3.1. In this case if we add another edge that goes from q_1 to q_8 and has sc or fca as a dependency, we have a conflict. This is because we have an edge going in the opposite direction, so we create a path that goes from q_8 to q_1. These sorts of conflicts can be very difficult to detect by hand in a complex model as the relations might not be clear or are easily missed. A major motivation for automating the detection process, such as we have, came about because of the complexities involved in checking a model by hand. While the above example is

Dependency $T_i \rightarrow T_j$	Conflicting Dependency $T_j \rightarrow T_i$
a	a,ba,bc,c,fca,s,t
b	b,ba,bc,s
ba	a,b,ba,bc,fca,s,t
bc	a,b,ba,bc,c,s,sc,t
c	bc,c
ex	s
fba	fca,s
fbb	fca,s
fbc	s
fbt	fca,s
fca	a,ba,fba,fbb,fbt,fca,s,sc,wa
s	a,b,ba,bc,c,ex,fba,fbb,fbc,fbt,fca,s,sc,t,wa
sc	bc,fca,s,sc
t	a,ba,bc,c,s,t
wa	fca,s

Table 3: Dependencies that conflict when present in opposing directions

simple enough to follow and such conflicts are listed in Table 2, this is not always the case. Conflicts can occur when two dependencies are presented in opposing directions. The relations created therein can become complex and cumbersome but are mapped out in Table 3.

Dependency $T_i \rightarrow T_j$	Implied Dependency $T_i \rightarrow T_j$
a	c,wa
ba	b,s,t
bc	b,c,s,t
fbb	fba,fbc,fbt
fbt	fba,fbc
s	b,c,t
t	c

Table 4: Dependency inclusion relationships

12 Implied dependencies

Dependencies can imply other dependencies. This fact is unimportant to preventing conflicts, but it is very important to the speed, efficiency, and readability of a set of subtransactions. A model cannot be said to be minimized or efficient if there are repeated dependencies; similarly, if there exist dependencies that are already implied in a edge, the model cannot be said to be efficient and minimized.

Assume that we have a model where there is at least one subtransaction pair, T_i and T_j. The pair contains multiple dependencies, two of which are the abort dependency and the weak abort dependency. Abort states that if T_i aborts, T_j must also abort. Weak abort states that if T_i aborts and T_j has not committed, T_j must abort. Given these definitions, one can say that abort is a stricter form of weak abort, including all of the requirements that weak abort implies, however it always forces T_j to abort if T_i aborts, while weak abort does not make such a requirement.

Prolog, given the definitions, can easily detect all include-relationships. The approach is simply to see if a dependency has all of its requirements in the Prolog database prior to allowing the dependency to be added to the edge. If all of the dependency's requirements are already present, it is safe to assume that the dependency is already in the database, be it by inclusion or by already existing. The trouble comes when adding abort after weak abort has already been added. In such a case, a search will be performed to remove any edge known to be implied by the given dependency. Table 4 lists all known implied dependencies, where the left column shows the dependency of interest and the right column shows all dependencies that are implied by the dependency of interest.

13 Ignored dependencies

One of the least discussed issues that can occur but that is very important in minimizing a model is that of instances where dependencies are essentially ignored. This occurs when one dependency specifies that nothing can occur until a specific action occurs. If another dependency is introduced that is only used when a certain action that is not governed by the above dependency, this new dependency can never achieve its goal.

Overriding Dependency $T_i \rightarrow T_j$	Ignored Dependency $T_i \rightarrow T_j$
ba	c,ex
bc	a,wa
fca	c

Table 5: Overriding and ignored dependencies

This is not a conflict; however, as the two dependencies are indeed satisfiable, it is just that the second dependency will never have an instance where it is actually used.

For instance, the dependencies $T_i \rightarrow_{ba} T_j$ and $T_i \rightarrow_c T_j$ may exist in a given model prior to minimization. Begin-on-abort will block T_j until T_i aborts, so T_i and T_j cannot both commit (T_j will do nothing if T_i commits). Therefore, it can be said that $_{ba}$ causes $_c$ to be ignored. Table 5 lists all known ignored dependencies, where the left column shows the overriding dependency and the right column shows the ignored dependency.

14 Conclusion

There now exists a method for logically minimizing a model and detecting conflicts. The proof-of-concept demonstrates the ease with which such tasks can be undertaken in a logic language such as Prolog. Furthermore, clear definitions are provided for what makes two dependencies conflict, what makes a dependency redundant, and what makes a dependency overridden by another. Comprehensives tables for the conflicts and superfluous dependencies are also provided, as well as practical dependencies that can arise on an edge containing composite dependencies. Most importantly, a method for detecting conflicts and ignored dependencies over multiple nodes has been defined, thereby allowing for complicated models to be created and tested.

The proof-of-concept provided allows any Prolog programmer to easily add definitions to the database and is written in the ISO standard without using complex statements. As a result, a clean, easy to adapt proof-of-concept that should be relatively simple to follow has been created. In doing this, the methods used can be proven, allowing the community to repeat and build on this research, and allowing those willing to work directly with a back-end to examine even the most complex model for conflicts when the model contains the fifteen dependencies defined.

While this work has added the ability to automate the process of examining and minimizing a model, it leaves more to be done. The methods for detecting conflicts and implied dependencies are completely automated, but the detection of "ignored" dependencies has not been completely automated. While some ignored relations can be detected by Prolog, such as $T_i \rightarrow_{bc} T_j$ and $T_i \rightarrow_a T_j$, others (such as $T_i \rightarrow_{ba} T_j$ and $T_i \rightarrow_c T_j$) do not seem to be able to be derived given the definitions of the dependencies. Instead, these are practically ignored, not logically ignored. The conflict detection module of the program developed as a part of this research can only detect some of the ignored dependencies.

In addition, there is room for improving the speed of the proof-of-concept, as well

as decreasing the memory footprint, which could not be done using the implemented programming languages and interludes (such improvements were sacrificed for decreased complexity and increased ease of programming). Furthermore, it might be useful to consider the actual state by emulating the transaction scheduler. By doing this, it would be possible to have more specific definitions for the dependencies as well as include such dependencies as weak begin-on-commit. Weak begin-on-commit needs to be better defined, however, before it can be automated.

Finally, the addition of security constraints to both the back-end and front-end will increase the capabilities of the program. By adding the security constraints, developers and researchers will be better able to create perfectly programmed databases and less likely to create an unstable database model. Prior to security constraints being added, however, they must be better defined.

15 Acknowledgment

The author has benefited greatly from discussions with Dr. Indrakshi Ray of Colorado State University. This work was supported in part by the United States National Science Foundation under Award IIS-0242258. The opinions reflected in this paper are those of the author and do not necessarily represent those of the NSF or any other party.

References

[1] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[2] Yuetang Deng, Phyllis Frankl, and David Chays. *Testing Database Transaction Consistency*, 2003.

[3] Nigel Ford. *PROLOG Programming*. John Wiley, New York, 1989.

[4] Ulf Nilsson and Jan Maluszynski. *Logic, Programming And Prolog*. John Wiley, New York, second edition, 1995.

[5] Richard O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, MA, 1990.

[6] Luigi Palopoli, Domenico Sacc, and Domenico Ursino. An automatic technique for detecting type conflicts in database schemes. In *Proceedings of the Seventh International Conference on Information and Knowledge Management*, pages 306–313, New York, 1998. ACM Press.

[7] Xiaolei Qian. Synthesizing database transactions. In *Proceedings 1990 VLDB Conference, Volume 1: 16th International Conference on Very Large Data Bases*, pages 552–565, Brisbane, Australia, 1990. Morgan Kaufmann.

[8] Xiaolei Qian. The deductive synthesis of database transactions. *ACM Transactions on Database Systems*, 18(4):626–677, December 1993.

[9] Indrakshi Ray and Tai Xin. Analysis of dependencies in advanced transaction models. *Distributed and Parallel Databases*, 20(1):5–27, July 2006.

[10] Tim Sheard and David Stemple. Automatic verification of database transaction saftey. *ACM Transactions on Database Systems*, 14(3):322–368, September 1989.

[11] Tai Xin, Yajie Zhu, and Indrakshi Ray. Reliable scheduling of advanced transactions. *Data and Applications Security XIX*, 3654:124–138, 2005.

A Proof-of-concept

Complete source code can be found at http://danielsecuritiesinc.com/papers/ata.html

A.1 transactionanalyzer.pl

```
#!/usr/bin/pl -L0 -T0 -G0 -s

/* Proof-of-concept created by Daniel Owens
 * Version 2.0
 * Copyright (C) 2006 Daniel Owens
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 */                                                                 10

/*
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 */

/*
 * You should have received a copy of the GNU General Public License   20
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
 * USA.
 */

:- ensure_loaded([buildall]).

/*
 * This is called if the database is empty.
 */                                                                 30
add(A, B, Dep) :-
        catch(transition(A, B), _, (assert(transition(A, B)),
                assert(edge(A, B, Dep)))),
        catch(edge(A, B, Dep), _, fail),
        !, %If the edge and transition exist, we don't need to examine the other
           %add/3's
        runchecks(A, B, Dep).
/*
 * This is called if the transition exists, but the edge does not.
 */                                                                 40
```

```
add(A, B, Dep) :-
        transition(A, B),
        runchecks(A, B, Dep),
        !, %Speed up time by not letting it backtrace if we fail from here on
        not(edge(A, B, Dep)),
        assert(edge(A, B, Dep)).
/*
 * This is called if the database is not empty, but the transition and edge
 * do not exist.
 */                                                                          50
add(A, B, Dep) :-
        not(transition(A, B)),
        assert(transition(A, B)),
        (runchecks(A, B, Dep)
                ->true
                ;(retract(transition(A, B)),
                        fail)),
        !, %Speed up time by not letting it backtrace if we fail from here on
        not(edge(A, B, Dep)),
        assert(edge(A, B, Dep)).                                             60

getpath(A, B, List) :-
        transition(A, B),
        assert(black(B)), %Signals that the goal was found
        List=[[A,B]].
getpath(A, B, List) :-
        transition(A, C),
        assert(grey(A)),
        not(grey(C)), %Checks to see if C has already been explored
        getpath(C, B, E),                                                    70
        black(B), %Checks to see if the Goal was found
        merge_set([[A, C]], E, List).

getallpaths(A, B, List) :-
        setof(C, C^getpath(A, B, C), List), %Get all unique paths
        retractall(grey(X)),
        retractall(black(X)).
```

A.2 buildall.pl

```
:- ensure_loaded([conflictdefs,impliesdefs,recurseconflicts,recurseincludes]).

/*                                                                           80
 * Check for conflicts, includes, then minimize the transaction and add the
 * definition.
```

```
 */
runchecks(Node1, Node2, Dependency) :-
        recurseconflicts(Node1, Node2, Dependency, _),
        not(idefs(Dependency, Node1, Node2)),
        minimize(Dependency, Node1, Node2),
        adddef(Dependency, Node1, Node2).

/*                                                              90
 * Minimize the transaction by removing edges that are included by the
 * definition of the dependency. The included dependencies must be fed to
 * Prolog explicitly in this case to avoid having to teach Prolog the
 * included dependencies every run.
 */
minimize(a,A,B) :-
        (edge(A,B,wa)
                ->retractall(edge(A,B,wa))
                ;true),
        minimize(t,A,B),                                        100
        minimize(wa,A,B).
minimize(b,_,_).
minimize(ba,A,B) :-
        (edge(A,B,s)
                ->retractall(edge(A,B,s))
                ;true),
        minimize(s,A,B).
minimize(bc,A,B) :-
        (edge(A,B,s)
                ->retractall(edge(A,B,s))                       110
                ;true),
        minimize(s,A,B).
minimize(c,_,_).
minimize(ex,_,_).
minimize(fba,_,_).
minimize(fbb,A,B) :-
        (edge(A,B,fbt)
                ->retractall(edge(A,B,fbt))
                ;true),
minimize(fbt,A,B).                                              120
minimize(fbc,_,_).
minimize(fbt,A,B) :-
        (edge(A,B,fba)
                ->retractall(edge(A,B,fba))
                ;true),
        (edge(A,B,fbc)
                ->retractall(edge(A,B,fbc))
                ;true),
```

```
        minimize(fba,A,B),
        minimize(fbc,A,B).                                              130
minimize(fca,A,B) :−
        minimize(bc,B,A).
minimize(s,A,B) :−
        (edge(A,B,b)
                −>retractall(edge(A,B,b))
                ;true),
        (edge(A,B,t)
                −>retractall(edge(A,B,t))
                ;true),
        minimize(b,A,B),                                                140
        minimize(t,A,B).
minimize(sc,A,B) :−
        minimize(a,B,A).
minimize(t,A,B) :−
        (edge(A,B,c)
                −>retractall(edge(A,B,c))
                ;true),
        minimize(c,A,B).
minimize(wa,_,_).
```

A.3 recurseconflicts.pl

```
antithesis(_, [], Empty) :−                                             150
        Empty=[]. %Empty is set to an empty list

/*
 * The antithesis/3 term finds and creates a list of all known dependencies
 * that are not in the given list. This is used to create the list of
 * dependencies that conflict.
 */
antithesis(OrigList, Deps, Conflicts) :−
        [Dep|NewDeps]=Deps,
        !, %Speed up time by not letting it backtrace if we fail from here on 160
        antithesis(OrigList, NewDeps, E),
        (member(Dep,OrigList)
                −>Conflict=[]
                ;Conflict=[Dep]),
        merge_set(Conflict, E, Conflicts).

/*
 * Merely provides a more abstracted term for discovering the conflicts from
 * Start to End.
 */                                                                     170
```

```
listconflicts(Start, End, List) :-
        recurseconflicts(Start, End, _, List).

recurseconflicts(_, [], Empty, _) :-
        Empty=[]. %Empty is set to an empty list

/*
 * Recurses through the transitions; the list expected is a single list.
 */
recurseconflicts(Dependency, Path, List, DependencyList) :-                180
        [H|Tail]=Path, %pull off a transaction pair from Path
        !, %Speed up time by not letting it backtrace if we fail from here on
        recurseconflicts(Dependency, Tail, ROut, DependencyList),
        [F|T]=H, %get Node1
        [S|_]=T, %get Node2
        setof(Dependency,Dependency^cdefs(Dependency, F, S),Out),
        antithesis(Out,DependencyList,ConflictsList),
        merge_set(ConflictsList, ROut, List).

/*                                                                         190
 * Calls to recursecpath/4, which then recurses through a double list.
 */
recurseconflicts(Start, End, Dependency, List) :-
        var(List),
        !, %Speed up time by not letting it backtrace if we fail from here on
        getallpaths(Start,End, Path), %get all paths to End from Start
        setof(Dependency,Dependency^dep(Dependency),DependencyList),
        recursecpath(Dependency,Path,List,DependencyList).

recursecpath( _, [], Empty, _) :-                                          200
        Empty=[].

/*
 * Recurses through the double list, which contains path information,
 * extracting the single lists and having recurseconflicts examine the
 * individual paths.
 */
recursecpath(Dependency, Path, List, DependencyList) :-
        [Head|Tail]=Path, %pull off a transaction pair from Path,
        !, %Speed up time by not letting it backtrace if we fail from here on 210
        recurseconflicts(Dependency, Head, ROut, DependencyList),
        recursecpath(Dependency, Tail, OtherRoutes, DependencyList),
        merge_set(OtherRoutes, ROut, List).
```