Synthesizing Database Transactions

Xiaolei Qian
Department of Computer Science, Stanford University
and Kestrel Institute
3260 Hillview Ave., Palo Alto, CA 94304

Abstract

Database programming requires having the knowledge of database semantics both to maintain database integrity and to explore more optimization opportunities. Automated programming of database transactions is desirable and feasible. In general, transactions use simple constructs and algorithms; specifications of database semantics are available; and transactions perform small incremental updates to database contents. Automated programming in such a restricted but well-understood and important domain is promising.

We approach the synthesis of database transactions that preserve the validity of integrity constraints using deductive tachniques. A transaction logic is developed as the formalism with which the synthesis is conducted. Transactions are generated as the by-product of proving specifications in the logic. The Manna-Waldinger deductive-tableau system is extended with inference rules for the extraction of transactions from proofs, which require the cooperation of multiple tableaux.

1 Introduction

Databases are partial models of the real world that provide a means to record the knowledge and facts about certain aspects of the real world. We use an extension of the relational model[3] as the underlying modeling mechanism, where facts become tabular data and knowledge consists of logical statements about the data called integrity constraints. A database schema describes the semantics of the data stored in the database by specifying the structure of relations and the relationships between relations. We might view a database schema as a theory and a database as a model of the theory[14]. Integrity constraints represent the time-independent semantics of data and serve as the validity criteria of data.

In order to be a model of the evolving world, a database system must handle changes in the data and assure that data validity is preserved with respect to the integrity constraints. Changes are necessary when old facts become obsolete and new facts are introduced. Changes to databases are caused by the executions of transactions, which are programs that access and manipulate data in the database. Database programming is the activity about the specification, design, implementation, and maintenance of data-retrieval programs (queries) and data-manipulation programs (transactions) to achieve certain goals. Such goals might be to retrieve data that satisfy specific conditions, or to manipulate the contents of the database in specific ways.

Consider the Job Agency database in Figure 1 as a scenario. In the examples that follow, we will abbreviate database relations in the Job Agency database by their first capital letters. Suppose the database semantics is specified by the integrity constraints below:

1. disjoint(Applicant,Employee): Applicant and Employee relations are disjoint.
2. subset(Interview.NAME,Applicant.NAME): every Interview tuple refers to a valid Applicant tuple by NAME.
3. key(Applicant,NAME): no two Applicant tuples have the same NAME, i.e., the key of Applicant relation is the NAME attribute.

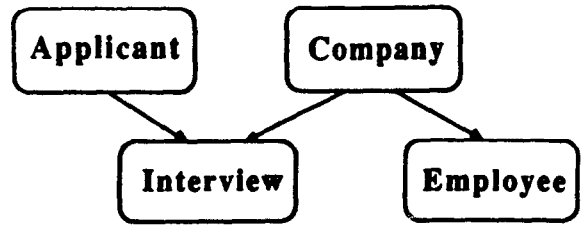

Figure 1: Job Agency Database

Suppose we need to enter into the Job Agency database the

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

fact that IBM has just hired a person named Jones. A se with the database may proceed as follows:

```
user:     Add Jones as an employee.
database: Done.
```

The user completes the session without realizing that database has run into serious inconsistency: Jones is recorded as a job applicant, and he has an interview sc uled with HP! The situation is not that bad, you may a Suppose the Job Agency database system enforces strict straint checking. In this case, the user is more likely t engaged in the following dialogue:

```
user:     Add Jones as an employee.
database: Sorry, I cannot do that.
user (looking puzzled):
          Why?
database: Because Jones is a job applicant.
user:     Oh, I forgot about that.
          Remove Jones as a job applicant.
database: Sorry, I cannot do that either.
user (getting mad):
          How come???
database: Because Jones has an interview with HP.
user:     I didn't know you keep track of interviews.
          Remove all interviews scheduled for Jones.
database: You are not authorized to do that.
user (jumping out of the chair):
          Forget it!
```

As a result, either database consistency is paralyzed or users are scared away from using the database. How can such a dilemma be resolved? In other words, how can database programming be more effective while semantic validity is properly maintained?

Database programming is an instance of programming. For forty years since John von Neumann first proposed stored-program computers, programming remains the primary means of instructing computers to perform work. Conventional programming is costly and error-prone. It does not have well-understood and precise principles. The automation of programming has been a dream since the birth of the first computer. Although the meaning has changed greatly, what was considered automated programming forty years ago becomes compiling now, the idea is still the same: we tell computers *what* to do and have them figure out *how* to do it. This is illustrated in Figure 2.

Automated programming techniques are commonly used in database programming. Query processing, for example, is a form of automated programming. Users specify what they want as queries in a high-level declarative language, such as relational calculus, and the query processor transforms them into executable query plans. Semantic query optimization is another form of automated programming. Users do not have to possess knowledge about the database semantics, and the query optimizer takes this knowledge into account to generate more efficient queries from users' query specifications. The fruitful research in these areas shows that automated programming of data-retrieval programs is not only possible but also feasible. Similar approaches can be taken to programming data-manipulation programs, namely transactions. In this paper, we address the issue of automated programming of database transactions.

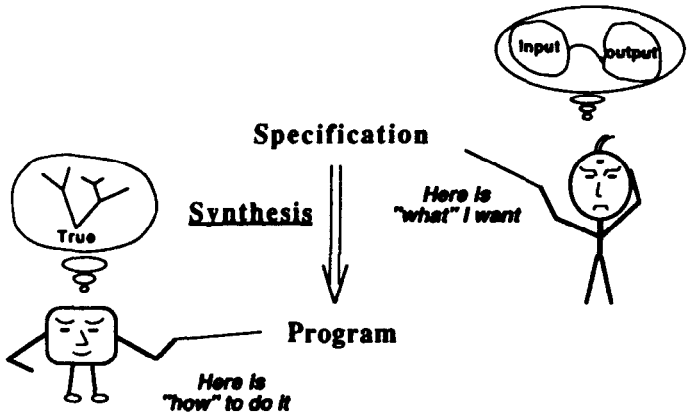

Figure 2: Automated Programming

Motivation: Desirability

Database programming requires knowledge of database semantics – the rules and constraints applied to the database so that it remains consistent and sharable by many users. Such knowledge is necessary for maintaining database integrity, and useful for exploring more optimization opportunities.

Let us reconsider the scenario where IBM has hired a person named Jones and needs to update its database. To program such a task, the programmer needs to know that, among other things, Jones must be removed from the Applicant relation if he is listed there, in order to preserve the validity of database semantics. In conducting such kind of reasoning, the programmer should also be aware of the fact that he does not have to check if other employees are looking for jobs — they could not be because the database in which the transaction will be executed is assumed to be valid to begin with.

The paradigm of database management makes automated programming more desirable. Most database systems today support the specification of database semantics to various degrees. However, specifications have mainly been used as documentation for programmers rather than as guidance to the system. We believe that it is now time to provide automated tools to exploit the use of semantics. Automation is necessary be-

cause programmers rarely have full access to database semantics. For reasons such as ease of use and security, databases often provide programmers with access to views, which are selected portions of data. If IBM is given access to a view which does not contain the relation Interview, we cannot expect a programmer at IBM to properly maintain the validity of the third constraint while coding the transaction to hire Jones.

Even if programmers do have full access to the database semantics, there is no guarantee that they will always completely understand the semantics. A large portion of transaction specifications comprises the specification of database semantics, which is interpreted to mean that the current database state is valid and the next database state should remain so. The specification of the transaction to hire Jones would look like:

for every input state
  if disjoint(Applicant,Employee) ∧
    subset(Interview.NAME,Applicant.NAME) ∧
    key(Applicant,NAME)
  then at output state
    Employee(Jones,...) ∧
    disjoint(Applicant,Employee) ∧
    subset(Interview.NAME,Applicant.NAME) ∧
    key(Applicant,NAME)

Because the specification of database semantics is oriented towards human understanding rather than machine execution, transaction programming involves routine translation from declarative specifications to procedural code. Without complete understanding of the semantics, correct translation would not be possible.

Transaction code is often reused over a long period of time through changing situations, while programmers do not stay in their positions very long. Moreover, tremendous optimization opportunities exist because transactions usually perform small, incremental changes to valid databases. A transaction in SQL that hires Jones with minimal incremental constraint maintenance can be:

delete x in Interview where x.NAME = Jones
delete x in Applicant where x.NAME = Jones
insert ⟨Jones...⟩ into Employee

Since manual optimization carries an extremely high risk of violating database semantics, it should be the responsibility of database management systems to synthesize executable code from transaction specifications that preserves the validity of database semantics.

Motivation: Feasibility

Database technology makes automated programming more feasible. Experience in program synthesis, especially deductive program synthesis, has revealed several major difficulties. First of all, large program specifications are very hard to write and to understand, requiring formal specification languages such as various forms of logic. Luckily, less effort is needed to write database transaction specifications because a large portion of them, namely the specification of database semantics, is already present. The user's specification of the task to hire Jones can be simply:

for every input state
  at the output state
    Employee(Jones,...)

Secondly, while program specifications express what needs to be done, the algorithmic information about how to do it is not specified and is almost impossible to derive fully automatically because of the tremendous search effort involved on the part of the synthesis system[17]. However, transactions are dominated by data manipulation tasks instead of complex computations. On the average, eighty percent of the common transaction code is about integrity constraint checking and enforcement. Simple algorithms and control structures — like case analysis — suffice.

Thirdly, programming languages are complex objects with a large number of powerful constructs such as pointers, loops, procedures, and recursive data structures. In order to perform program synthesis, the semantics of these constructs must be completely specified first, which is again a very large and difficult task[9]. For the same reasons, however, a small number of simple language constructs — insert, delete, modify, test, and iterate for example — constitutes a reasonable transaction language. Our task to hire Jones can be programed using these constructs:

if Applicant(Jones,...) then
  foreach x in Interview where x.NAME = Jones do
    delete x from Interview;
  delete (Jones,...) from Applicant;
insert (Jones,...) into Employee.

Finally, situations become even worse when programs are to be synthesized that manipulate computational states destructively. A large number of axioms and rules need to be formulated in order to completely describe the effects of various language constructs on states. Fortunately, database states are relatively simple to characterize in terms of a finite set of relations. We do not have to consider exceptional conditions not stated by the constraints. We are often able to specify precisely the effect of every language construct on database states. After executing a transaction that hires Jones and does nothing else, we should be able to infer for example that no one else, neither job applicants nor employees, are affected in terms of their job status.

The paper is organized as follows. In Section 2, we formally define the syntax and semantics of a transaction language as our database programming language. A transaction logic is developed in Section 3 as the formalism for specifying and reasoning about transactions. In Section 4 we extend the Manna-Waldinger deductive-tableau system as the system for deductive synthesis. New rules are designed to eliminate nonconstructive proofs and to extract transactions from constructive proofs. Finally, a brief literature survey and concluding

remarks are provided in Section 5. The material of this paper is extracted from [16]. Because of the limitation on space, we will emphasize the key ideas and illustrate them with examples and intuitive arguments. Interested readers might consult [16] for detailed formalizations and proofs.

2 A Transaction Language

The transaction language takes a first-order many-sorted language as a sublanguage. There is an atom sort *atom* and for every $n \geq 1$ there is an n-ary tuple sort *ntup*. The function symbols of the sublanguage include a countable number of constants; n unary *tuple selector* functions s_i^n of sort $ntup \rightarrow atom$ for $i = 1, \ldots, n$; and one n-ary *tuple constructor* function c^n of sort $(atom, \ldots, atom) \rightarrow ntup$ for every *ntup* sort. When no ambiguity is possible, we will neglect the superscripts and write $s_1, \ldots, s_n$ and c instead. The predicate symbols of the sublanguage include a countable number of uninterpreted predicate symbols over *atom* which we call relation symbols. As a notational convention, for n-ary predicate symbol R and n-ary tuple e, the atomic formula $R(e)$ is treated as an abbreviation of $R(s_1(e), \ldots, s_n(e))$.

If R is an n-ary relation, x is an n-ary tuple variable, e is an n-ary tuple, and p is a formula, then the construction of transactions is characterized recursively by the following rules:

1. The tuple insertion $insert_R(e)$, tuple deletion $delete_R(e)$, and tuple modification $modify_R(x, e)$ are transactions.
2. If t_1, t_2 are transactions, then the sequential-composition $t_1;;t_2$ is a transaction.
3. If t_1, t_2 are transactions, then the conditional-branch **if** p **then** t_1 **else** t_2 and **if** p **then** t_1 are transactions.
4. If t is a transaction, then the bounded-iteration **foreach** x **in** $R|p$ **do** t and **foreach** x **in** p **do** t are transactions.

For conditional-branch statement **if** p **then** t_1 **else** t_2, formula p is the *test predicate*. For bounded-iteration statement **foreach** x **in** $R|p$ **do** t, relation R is the *loop relation*, tuple variable x is the *cursor ranging over* R, formula p is the *selection predicate*, $R|p$ is the *header*, and transaction t is the *body*. Cursor x is *bound* in the body and *free* outside. For transaction t, a relation R is an *update relation* in t if t includes a basic statement $insert_R(e)$, $delete_R(e)$, or $modify_R(x, e)$. R is an *access relation* in t if it appears in either the header of a bounded-iteration statement or the test predicate of a conditional-branch statement.

From the construction of transactions, it is clear that any relation in a transaction is either an update or an access relation, and a relation is accessed if and only if it appears in places other than basic statements. A transaction t is *well-formed* if:

1. the cursors of nested bounded-iteration statements have different names;
2. modify statements are always nested in bounded-iteration statements;
3. for basic statements of the form $delete_R(x)$ or $modify_R(x, e)$ nested in bounded-iteration statements, the immediately containing bounded-iteration statement is of the form **foreach** x **in** $R|p$ **do** u; and
4. for bounded-iteration statements of the form **foreach** x **in** $R|p$ **do** u, no relation is both an update relation and an access relation in u.

The first two conditions are for convenience. The last two conditions imply that deletions and modifications can only be performed on cursors ranging over loop relations, and they can be nested in at most one bounded-iteration statement. The reason for these restrictions will become clear when we discuss transaction semantics. Unless specified otherwise, we assume hereafter that transactions are always well-formed.

Example 1 Assuming R does not occur in q, the transaction below is well-formed by our definition, where relation R is updated but not accessed in the body of the outer bounded-iteration statement.

foreach x **in** $R|p(x)$ **do**
  if $s_i(x) = a$ **then** $delete_R(x)$
  else $modify_R(x, f(x))$;
  foreach y **in** $S|q(x, y)$ **do** $insert_R(g(x, y))$. ◇

Given a *universe* $\mathcal{U}$ of elements, a *state* on $\mathcal{U}$ is an interpretation that assigns: an atom or n-ary tuple over $\mathcal{U}$ to every atom or n-ary tuple variable, an n-ary function over $\mathcal{U}$ to every n-ary function symbol, and an n-ary relation over $\mathcal{U}$ to every n-ary relation symbol. An *update* is a mapping on the collection of states over certain fixed universe. An update is *relational* if output states differ from input states only in the values of relation symbols. The semantics of transactions assigns an update $\kappa(t)$ to every subtransaction t of a transaction. In particular it assigns a relational update $\kappa(t)$ to every transaction t. The assertion $(I, J) \in \kappa(t)$ represents the fact that the computation of t starting at state I, if terminates, will terminate in state J. Transaction t *expresses* update σ if $\kappa(t) = \sigma$.

Informally, the semantics can be stated as following. After the execution of $insert_R(e)$ in state I where e is not a member of R, e will become a member of R. After the execution of $delete_R(e)$ in state I, where e is a member of R, e will no longer be a member of R. After the execution of $modify_R(x, e)$ in state I, where the value of tuple variable x is a member of R, e will replace x as a member of R and tuple variable x will have value e.

The semantics of the sequential-composition $t_1;;t_2$ is the composition of the semantics of t_1 and t_2. The semantics of the conditional-branch **if** p **then** t_1 **else** t_2 is the mapping that maps states where p is true to those specified by the semantics of t_1 and states where p is false to those specified by the semantics of t_2. The semantics of the bounded-iteration **foreach** x **in** $R|p$ **do** t is based on parallel execution. The set $\{x|R(x) \wedge p\}$ is bound before the execution begins. Each round of the iteration starts as if R contains a single tuple x

in the set and executes the body t. The result of the iteration is the union of the corresponding relations in the result of every round of the iteration. The semantics of loop relations is illustrated in Figure 3.

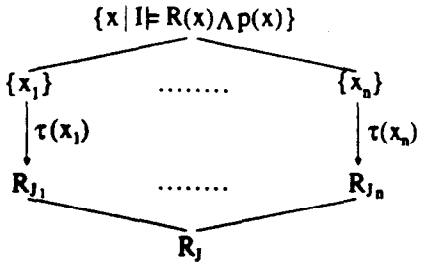

Figure 3: Loop Relation Semantics

More formally, the semantics of transactions is specified by the following rules, where e^I denotes the value assigned to e in state I, $I[o/v]$ denotes the state everywhere identical to I except that it replaces the value of o by v, "$\circ$" denotes the map composition operator, and the union of two states I and J, which assign the same values to variables, is taken to be the state in which variables remain the same and relations are unions of corresponding relations in I, J.

1. For relation R, tuple variable x, and tuple e,

$$\kappa(insert_R(e)) = \{(I,J) | J = I[R/(R^I \cup \{e^I\})]\}$$
$$\kappa(delete_R(e)) = \{(I,J) | J = I[R/(R^I \setminus \{e^I\})]\}$$
$$\kappa(modify_R(x,e)) = \{(I,J) | J = \textit{if } x^I \in R^I \textit{ then } I[R/(R^I \setminus \{x^I\} \cup \{e^I\}), x/e^I] \textit{ else } I\}$$

2. For two transactions t_1 and t_2,

$$\kappa(t_1;;t_2) = \kappa(t_1) \circ \kappa(t_2)$$

3. For formula p and transactions t_1, t_2,

$$\kappa(\textbf{if } p \textbf{ then } t_1 \textbf{ else } t_2) = \{(I,J) | \textit{if } p^I \textit{ then } (I,J) \in \kappa(t_1) \textit{ else } (I,J) \in \kappa(t_2)\}$$

4. For tuple variable x, relation R, formula p, and transaction t,

$$\kappa(\textbf{foreach } x \textbf{ in } R|p \textbf{ do } t) = \{(I,J) | S = \{v | v \in R^I \wedge p^I[x/v]\} \wedge \textstyle\bigwedge_{v \in S}(I[R/\{v\}, x/v], J_v) \in \kappa(t) \wedge J' = \bigcup_{v \in S} J_v[x/x^I] \wedge J = J'[R/(R^{J'} \cup \{v | v \in R^I \wedge \neg p^I[x/v]\})]\}$$

The reason for the last two conditions of well-formedness is mainly due to the parallel-execution semantics. For loop relations, deletions and modifications to tuples other than the cursor make no sense since each round of the iteration only has access to the cursor. Therefore, we require that tuples different from the cursor are not deleted or modified. Because the execution is parallel, any reference to intermediate states would be counter-intuitive. Hence we demand that relations are not both accessed and updated in the body of an iteration.

3 A Theory of Transactions

We motivate the definition of our transaction logic by considering its expected properties. First of all, states should be an integral part of the vocabulary, which means that they have to be explicit objects. The set of states should characterize the space of database evolution. Secondly, transaction language constructs should be functions in the logic such that transactions can be represented as expressions mapping states to states that can be composed to form new transactions. State transitions should correspond to executions of transactions, therefore, the concept of transaction execution should be explicit in the logic. Thirdly, logical formulas may refer to various states and state transitions. The semantics of transactions should be represented as logical axioms, while integrity constraints and transaction specifications should be expressible as logical sentences. Finally, transactions should be executable programs, in the sense that they can access and manipulate only an implicit current state rather than several hypothetical states. To avoid the formation of non-executable transactions, only a subset of the expressions in the language should be considered as transactions.

The transaction logic is an instance of many-sorted, first-order, situational logic that takes the language of transactions defined in Section 2 as a sublanguage. We distinguish two classes of objects in the logic: ordinary objects, such as states, attributes, or tuples; and program objects that return states, attribute values, or tuples as the result of execution. These two classes of objects are represented in the underlying language by two classes of sorts respectively: the *situational sorts* and the *fluent sorts*. Each situational sort has an associated fluent sort, and vice versa. There are three types of fluent sorts and associated situational sorts:

1. the fluent state sort *f-state* and the situational state sort *s-state* as the domain of database evolution,
2. the fluent atom sort *f-atom* and the situational atom sort *s-atom* as the domain for attribute values, and
3. the fluent n-ary tuple sort *f-ntup* and the situational n-ary tuple sort *s-ntup* for every $n \geq 1$ as the domains for tuples of atoms.

We call the atom and n-ary tuple sorts the *object sorts* and denote them by *f-object* and *s-object* respectively. For each fluent function symbol f of sort $(i_1, \ldots, i_{n-1}) \rightarrow i_n$, where $i_1, \ldots, i_n$ are fluent sorts, there is an associated situational function symbol f' of sort $(s\text{-}state, i'_1, \ldots, i'_{n-1}) \rightarrow i'_n$, where $i'_1, \ldots, i'_n$ are the situational sorts associated with $i_1, \ldots, i_n$ respectively. Similarly, for each fluent predicate symbol P of sort $(i_1, \ldots, i_n)$ there is an associated situational predicate symbol P' of sort $(s\text{-}state, i'_1, \ldots, i'_n)$.

Expressions are also classified into two classes: (1) expressions of situational sorts are *situational expressions* or *s-expressions*, which denote values in specific states; and (2) expressions of fluent sorts are *fluent expressions* or *f-expressions*, which denote mappings from states to values. Fluent expressions evaluated at specific states denote values in those states. As notational conventions, we will denote an f-expression by e and the s-expression associated with e by e'. Logical symbols will be overloaded for situational and fluent formulas. For example, the s-expressions

$$\textit{salary}'(w, e'),\ \textit{work-in-project}'(w, e', p'),\ \textit{hire}'(w, e'),$$

where w, e', p' are s-expressions, denote respectively the salary of employee e' at state w, the truth value of the assertion "employee e' works in project p' at state w", and the state obtained after hiring person e' at state w. On the other hand, the f-expressions

$$\textit{salary}(e),\ \textit{work-in-project}(e, p),\ \textit{hire}(e),$$

where e, p are f-expressions, denote respectively a query about the salary of employee e, a boolean query about whether e works in project p, and a transaction to hire e. They yield an object (salary of e), a truth value (e works in p), and a state (after hiring e) only when they are evaluated in a particular state.

The f-expressions do not refer to states explicitly. In order to determine the object, the state, or the truth value that an f-expression e designates with respect to a specific state w, we provide two situational functions and one situational predicate that relate w and e:

1. the *access* function ":" of sort (*s-state*, *f-object*) $\rightarrow$ *s-object*,
2. the *execution* function ";" of sort (*s-state*, *f-state*) $\rightarrow$ *s-state*, and
3. the *evaluation* predicate "$::_p$" of sort (*s-state*) for every f-formula p,

where the access function is overloaded for all the object sorts. They have the following informal semantics. The access function $:(w, e)$ takes an f-expression e of sort *f-object* and returns the value of e at state w. The execution function $;(w, e)$ takes an f-expression e of sort *f-state* and returns the state after executing e at state w. The evaluation predicate $::_e(w)$ evaluates to true if and only if f-formula e is true at state w. By convention, the s-expressions $:(w, e)$, $;(w, e)$, and $::_e(w)$ are written as $w{:}e$, $w;e$, and $w::e$ respectively. For example, the s-expressions

$$w{:}\textit{salary}(e),\ w::\textit{work-in-project}(e, p),\ w;\textit{hire}(e),$$

denote the salary of employee e at state w, the truth value of the assertion "employee e works in project p" at state w, and the state after hiring e at state w. In general, the behavior of situational functions and predicates are governed by a set of linkage axioms. For state w, n-ary f-function f of sort (*f-object*, ..., *f-object*) $\rightarrow$ *f-object*, n-ary f-function g of sort (*f-object*, ..., *f-object*) $\rightarrow$ *f-state*, n-ary f-predicate P of sort (*f-object*, ..., *f-object*), f-expressions $x_1, \ldots, x_n$ of sort *f-object*, and f-expressions τ_1, τ_2 of sort *f-state*,

$w{:}f(x_1, \ldots, x_n) = f'(w, w{:}x_1, \ldots, w{:}x_n)$	*object-linkage*
$w;g(x_1, \ldots, x_n) = g'(w, w{:}x_1, \ldots, w{:}x_n)$	*state-linkage*
$w::P(x_1, \ldots, x_n) = P'(w, w{:}x_1, \ldots, w{:}x_n)$	*predicate-linkage*
$w::(\tau_1 = \tau_2) \equiv (w;\tau_1 = w;\tau_2)$	*equality-linkage*
$w::(\neg p) \equiv \neg(w::p)$	$\neg$*-linkage*
$w::(p \wedge q) \equiv w::p \wedge w::q$	$\wedge$*-linkage*
$w::(\forall x)p \equiv (\forall x)w::p$	$\forall$*-linkage*

Transactions are essentially mappings from states to states that can be represented as state-valued f-expressions. There is one *fluent constant* and three *basic fluent functions* for every $n \geq 1$ and n-ary relation R:

1. the identity constant Λ of sort *f-state*,
2. the insert function $\textit{insert}_R$ of sort *f-ntup* $\rightarrow$ *f-state*,
3. the delete function $\textit{delete}_R$ of sort *f-ntup* $\rightarrow$ *f-state*, and
4. the modify function $\textit{modify}_R$ of sort (*f-ntup*, *f-ntup*) $\rightarrow$ *f-state*.

The semantics of the basic fluent functions is the same as the semantics of the corresponding basic statements in the transaction language specified in Section 2. The identity constant Λ corresponds to the *null* transaction, which performs no actions at all:

$w;\Lambda = w$	*identity-frame*

In order to completely specify the effect of evaluating basic fluent functions, we need two groups of axioms. The *action axioms* below specify the parts of the states that are changed as the result of evaluating the fluent functions and how they are changed.

$R'(\textit{insert}'_R(w, e'), e')$	*insert-action*
$R'(w, e') \rightarrow \neg R'(\textit{delete}'_R(w, e'), e')$	*delete-action*
$R'(w, x) \rightarrow R'(\textit{modify}'_R(w, x, e'), e')$	*modify-action-1*
$w::R(x) \rightarrow (w;\textit{modify}_R(x, e)){:}x = w{:}e$	*modify-action-2*

The insert-action axiom says that a tuple e will be in R after inserting e into R. Similarly, the delete-action axiom tells us that after deleting a tuple e from relation R, e will not be in R. The modify-action axioms state that after performing an action $\textit{modify}_R(x, e)$, two parts of the state may change. Relation R will have e as a member, and cursor x will have value e. On the other hand, the *frame axioms* below specify the exact scope of the changes, from which we can deduce what other parts of the states remain the same.

$$(R'(w,e') \vee e' = d') \equiv R'(insert'_R(w,d'),e') \qquad \textit{insert-frame-1}$$
$$(w;insert_R(d)):e = w:e \qquad \textit{insert-frame-2}$$
$$(R'(w,e') \wedge e' \neq d') \equiv R'(delete'_R(w,d'),e') \qquad \textit{delete-frame-1}$$
$$(w;delete_R(d)):e = w:e \qquad \textit{delete-frame-2}$$
$$\left(\begin{array}{l} \textit{if } R'(w,x) \\ \textit{then } (R'(w,e') \wedge e' \neq x) \vee e' = d' \\ \textit{else } R'(w,e') \end{array}\right) \equiv R'(modify'_R(w,x,d'),e') \qquad \textit{modify-frame-1}$$
$$(w;modify_R(x,d)):e \stackrel{c}{=} \textit{if } w::R(x) \textit{ then } w:e[x/d] \textit{ else } w:e \qquad \textit{modify-frame-2}$$

where the conditional equality $\stackrel{c}{=}$ in the last frame axiom holds if e does not contain cursors ranging over R other than x. The insert-frame axioms express the fact that a tuple e is in relation R after inserting tuple d into R if and only if either e is in R before the insertion or $e = d$. Likewise, the delete-frame axioms state that a tuple e is in relation R after deleting tuple d from R if and only if e is in R before the deletion and $e \neq d$. The values of cursors will not change as the result of insert or delete actions. The modify-frame axioms are more complicated. They tell us that a tuple e is in relation R after performing the action $modify_R(x,d)$ if and only if, either e is in R before the action and is different from x, or e agrees with d. To explain the meaning of the last frame axiom, we should bear in mind that cursors in transactions behave like pointers to tuples in relations: two cursors may have different names but point to the same tuple. The axiom gives a sufficient condition: the value of e is $e[x/d]$ after the action if e does not contain other cursors ranging over relation R.

Because state-valued f-expressions are mappings from states into states, we can compose them to form new f-expressions through the use of *composition fluent functions*, corresponding to the control structures in the transaction language. There are three such functions:

1. the *sequential-composition* function ";;" of sort $(f\text{-}state, f\text{-}state) \rightarrow f\text{-}state$,
2. the *conditional-branch* functions $if\text{-}then\text{-}else_p$ of sort $(f\text{-}state, f\text{-}state) \rightarrow f\text{-}state$ and $if\text{-}then_p$ of sort $f\text{-}state \rightarrow f\text{-}state$ for every f-formula p, and
3. the *bounded-iteration* functions $foreach\text{-}do_{R,p}$ and $foreach\text{-}do_p$ of sort $f\text{-}state \rightarrow f\text{-}state$ for every $n \geq 1$, n-ary relation R and f-formula p.

For f-expressions τ, τ_1, τ_2 of sort *f-state*, n-ary relation R, and f-formula p, the semantics of composition fluent functions $;;(\tau_1,\tau_2)$, $if\text{-}then\text{-}else_p(\tau_1,\tau_2)$, and $foreach\text{-}do_{R,p}(\tau)$ is the same as the semantics of the corresponding transactions specified in Section 2. For notational convenience, we will use the following syntactic forms of composition fluent functions:

$$\tau_1;;\tau_2, \textbf{ if } p \textbf{ then } \tau_1 \textbf{ else } \tau_2, \textbf{ foreach } x \textbf{ in } R|p \textbf{ do } \tau,$$

where x is a variable of sort *f-ntup*, called the *cursor* of the bounded-iteration function. There are two axioms concerning the identity constant and the sequential-composition function, which tell us that the sequential-composition function is associative and has an identity Λ:

$$(\tau;;\tau_1);;\tau_2 = \tau;;(\tau_1;;\tau_2) \qquad \textit{composition-associativity}$$
$$\Lambda;;\tau = \tau;;\Lambda = \tau \qquad \textit{identity-fluent}$$

The interaction of situational functions and predicates with composition fluent functions are also characterized by a set of axioms that relate the evaluation of composite fluents with the evaluation of component fluents. For state w, f-expressions τ_1, τ_2 of sort *f-state*, and f-formula p, we have:

$$w;(\tau_1;;\tau_2) = (w;\tau_1);\tau_2 \qquad \textit{composition-linkage}$$
$$w;(\textbf{if } p \textbf{ then } \tau_1 \textbf{ else } \tau_2) = (\textit{if } w::p \textit{ then } w;\tau_1 \textit{ else } w;\tau_2) \qquad \textit{condition-linkage}$$

In general, the evaluation of f-expressions at different states may result in different values. Those f-expressions that always evaluate to the same values at every state are called *rigid expressions*. In reasoning about the effect of transaction executions on the database states, we do not have to be concerned with rigid expressions because their values will not be affected by state transitions. In other words, if e is a rigid expression then for any two states s and w, we know that $s:e = w:e$, $s;e = w;e$, and $s::e \equiv w::e$. For rigid f-function symbol f and rigid f-predicate symbol P, we will denote the corresponding s-function and s-predicate by the same symbols f and P respectively, rather than by the symbols f' and P' with one extra state argument. Rigidity is specified by the axioms below:

$$='(w,x,y) \equiv x = y \qquad \textit{equality-rigidity}$$
$$c'(w,x_1,\ldots,x_n) = c(x_1,\ldots,x_n) \qquad \textit{construction-rigidity}$$
$$s'_i(w,x) = s_i(x) \qquad \textit{selection-rigidity}$$
$$w:a = a \quad \text{for constant } a \qquad \textit{atom-rigidity}$$
$$w::true \equiv true \qquad \textit{truth-rigidity}$$

An s-expression e' is *primitive at* w, where w is an s-expression of sort *s-state*, if we can find an f-expression e such that e' is equal to $w:e$, $w::e$, or $w;e$. Intuitively, a primitive expression can always be evaluated at single states.

4 Deductive Synthesis

Transaction specifications are logical assertions that express *what* properties state transitions have in terms of the relationships between input and output states of transactions, without explicitly providing the transactions involved. *Transactions*, on the other hand, are procedural expressions that describe *how* state transitions happen. A specification is intuitively an s-formula over two states: $\mathcal{Q}(s_0, s_f, \bar{x})$, where s_0, s_f are state variables and $\bar{x}$ is a sequence of object variables, that specifies a relationship between the input state s_0 and the output state s_f. A transaction is taken to be a state-valued f-expression. A specification $\mathcal{Q}(s_0, s_f, \bar{x})$ is satisfied by a transaction t if, for every input state s_0 and all possible values of input parameters $\bar{x}$, the resulting state of executing t at s_0 makes $\mathcal{Q}(s_0, s_0;t, \bar{x})$ true.

Example 2 The Hire-Employee problem outlined in Section 1 can be specified using our transaction logic. The transaction specification to hire person a is $E'(s_f, a)$, where a is an input parameter. The integrity constraints that represent the semantics of the Job-Agency database can be specified as follows:

$$(\forall s)(\forall x)(E'(s,x) \rightarrow \neg A'(s,x)) \wedge$$
$$(\forall s)(\forall x)(I'(s,x) \rightarrow (\exists y)(A'(s,y) \wedge applicant(x) = name(y))) \wedge$$
$$(\forall s)(\forall x)(\forall y)(A'(s,x) \wedge A'(s,y) \wedge name(x) = name(y) \rightarrow x = y)$$

One version of the Hire-Employee transaction that satisfies the specification and preserves the validity of integrity constraints can be expressed as:

if $A(a)$ **then**
  foreach x **in** $I|applicant(x) = name(a)$ **do**
    $delete_I(x)$;;
  $delete_A(a)$
else Λ;;
$insert_E(a)$. $\diamond$

The process of *transaction synthesis* is the gradual transformation from declarative specifications to procedural implementations. Our approach to transaction synthesis is by deductive reasoning. For transaction specification $Q(s_0, s_f, \bar{x})$, transaction synthesis is formulated as the process of finding a transaction t that satisfies the specification. In other words, we show constructively the validity of the *specification theorem*

$$(\forall s_0)(\forall \bar{x})(\exists \tau) Q(s_0, s_0;\tau, \bar{x})$$

Suppose we also need to maintain the validity of a set IC of constraints of the form $(\forall s)\alpha(s)$. Let us denote by $C(s)$ the conjunction of s-formulas $\alpha(s)$ where $(\forall s)\alpha(s)$ is in IC. To synthesize transactions from the specification $Q(s_0, s_f, \bar{x})$ that preserve the validity of constraints in IC is to find a transaction t such that for every valid input state s_0 and all possible values of input parameters $\bar{x}$, the resulting state of executing t on s_0 is another valid state s_f that satisfies the specification. Or more formally, we show constructively the validity of the *expanded specification theorem*

$$(\forall s_0)(\forall \bar{x})(\exists \tau)(C(s_0) \rightarrow Q(s_0, s_0;\tau, \bar{x}) \wedge C(s_0;\tau))$$

The process of transaction synthesis is thus formalized as the process of finding constructive proofs of specification theorems. By restricting the deduction to be constructive, a qualified transaction can be extracted from a proof of the specification theorem. The process of transaction synthesis is illustrated by Figure 4.

4.1 Deductive-Tableau Synthesis System

An appropriate proof system is necessary in order to carry out the deductive transaction synthesis outlined above. We extend the deductive-tableau proof system for first-order logic developed by Manna and Waldinger[12] to fulfill our need. Proofs in the system are represented as tables or *deductive tableaux*.

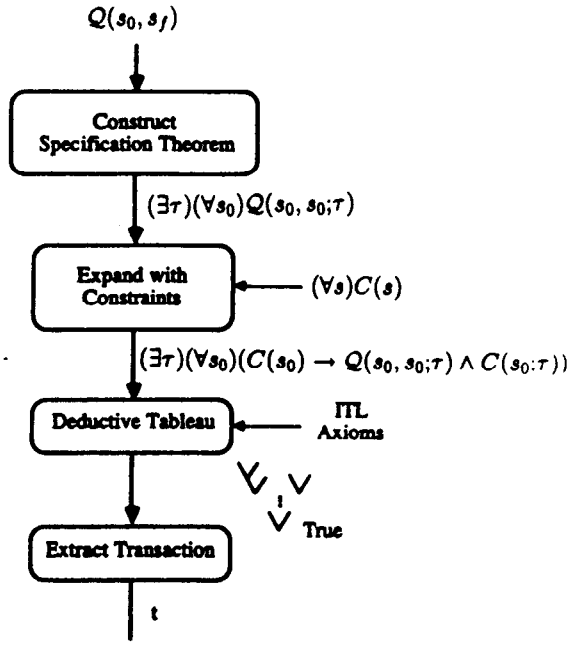

Figure 4: Transaction Synthesis Process

A deductive tableau consists of two lists of s-formulas: the assertion list $\alpha_1, \ldots, \alpha_m$ and the goal list $\gamma_1, \ldots, \gamma_n$ for some non-negative integers m and n. Every assertion or goal β may have an associated transaction entry ω which is an s-expression of sort *s-state*, denoted as $\beta|\omega$. A deductive tableau is represented as a table with three columns, each row has either a goal or an assertion but not both:

assertion	goal	transaction
α_1		
$\vdots$		
α_m		μ
	γ_1	ν
	$\vdots$	
	γ_n	

A deductive tableau is valid when:

> *for any interpretation, if every assertion is true for all possible variable assignments under the interpretation, then at least one goal is true for some variable assignments under the same interpretation.*

More precisely, the deductive tableau above has the same

meaning as the following sentence:

$$(\forall \bar{x})\alpha_1 \wedge \ldots \wedge (\forall \bar{x})\alpha_m \rightarrow (\exists \bar{x})\gamma_1 \vee \ldots \vee (\exists \bar{x})\gamma_n$$

A subformula has *positive polarity* if it is nested within an even number of negations in a goal, or an odd number of negations in an assertion. It has *negative polarity* if it is nested within an odd number of negations in a goal, or an even number of negations in an assertion. A quantification has *universal force* if it is universal and positive, or existential and negative. It has *existential force* if it is existential and positive, or universal and negative.

Given a specification theorem $(\forall s_0)(\forall \bar{x})(\exists \tau)\mathcal{Q}(s_0, s_0;\tau, \bar{x})$, we remove its quantifiers on state s_0, parameters $\bar{x}$, and transaction τ by skolemization, enter the resulting s-formula $\mathcal{Q}(s_0, s_0;\tau, \bar{x})$ into a deductive tableau as the initial goal, and enter the axioms of our transaction logic into the deductive tableau as assertions. The transaction entry for the initial goal is $s_0;\tau$. An s-expression ω satisfies the initial goal if $\mathcal{Q}(s_0, \omega, \bar{x})$ is valid. The semantics of transaction entries is:

> *for any goal (or assertion) in the tableau, if the goal (or the negation of the assertion) is valid for some variable assignment, then the corresponding transaction entry under the same variable assignment satisfies the initial goal.*

assertion	goal	transaction
	$\mathcal{Q}(s_0, s_0;\tau, \bar{x})$	$s_0;\tau$
	$\vdots$	$\vdots$
	true	ω

The synthesis system consists of deduction rules that manipulate the deductive tableaux in a validity-preserving and semantics-preserving manner. A proof proceeds by both forward reasoning (adding new assertions to the tableau) and backward reasoning (adding new goals to the tableau) without changing the validity of tableaux or the semantics of transaction entries. A proof *terminates* if either *false* is generated as an assertion (refutation) or *true* is generated as a goal (confirmation), and the transaction entry associated with the assertion or goal is primitive at s_0.

The extra requirement that proofs terminate with primitive transaction entries means essentially that proofs in the deductive-tableau synthesis system is *constructive*: we are able to extract executable transactions from these proofs at termination, and the transactions extracted are guaranteed to satisfy initial specifications.

Example 3 Throughout this section, we use the Hire-Employee transaction in Example 2 to demonstrate the deductive synthesis of transactions. The specification of the transaction is $E'(s_f, a)$. The expanded specification theorem constructed from this specification and the integrity constraints in Example 2 looks like the following, where $p(x, y)$ and $q(x, y)$ denote $applicant(x) = name(y)$ and $name(x) = name(y)$ respectively:

$$(\forall s_0)(\forall a)(\exists \tau)$$
$$\left(\begin{array}{l}(\forall x)(E'(s_0, x) \rightarrow \neg A'(s_0, x))\wedge \\ (\forall x)(I'(s_0, x) \rightarrow (\exists y)(A'(s_0, y) \wedge p(x, y)))\wedge \\ (\forall x)(\forall y)(A'(s_0, x) \wedge A'(s_0, y) \wedge q(x, y) \rightarrow x = y)\end{array}\right)$$
$$\rightarrow \left(\begin{array}{l}E'(s_0;\tau, a)\wedge \\ (\forall x)(E'(s_0;\tau, x) \rightarrow \neg A'(s_0;\tau, x))\wedge \\ (\forall x)(I'(s_0;\tau, x) \rightarrow (\exists y)(A'(s_0;\tau, y) \wedge p(x, y)))\wedge \\ (\forall x)(\forall y)(A'(s_0;\tau, x) \wedge A'(s_0;\tau, y) \wedge q(x, y) \rightarrow x = y)\end{array}\right)$$

goal	transaction
G1. $C(s_0) \rightarrow E'(s_0;\tau, a) \wedge C(s_0;\tau)$	$s_0;\tau$

As notational conventions, the three constraints are denoted as C_i for $i = 1, 2, 3$, and their conjunction is denoted by C. The initial tableau looks like the above. ◇

4.2 Basic Statements

The deduction rules responsible for synthesizing basic statements are the resolution rules. One version of the resolution

assertion	goal	transaction
	G1. $C(s_0) \rightarrow E'(s_0;\tau, a) \wedge C(s_0;\tau)$	$s_0;\tau$
A1. $C(s_0)$		$s_0;\tau$
	G2. $\boxed{E'(s_0;\tau, a)}^{+} \wedge C(s_0;\tau)$	$s_0;\tau$
A2. $\boxed{E'(insert'_E(w, x), x)}^{-}$		
	G3. $C(s_0;insert_E(a))$	$s_0;insert_E(a)$

Table 1: Extracting Basic Statements

assertion	goal	transaction
	G2. $\boxed{E'(s_0;\tau, a)}^{+} \wedge C(s_0;\tau)$	$s_0;\tau$
A2. $\boxed{E'(insert'_E(w,x),x)}^{-}$		
	G3'. $C(s_0;\tau_1;insert_E(a))$	$s_0;\tau_1;insert_E(a)$

assertion	goal	transaction
	G3'. $C(s_0;\tau_1;insert_E(a))$	$s_0;\tau_1;insert_E(a)$
	$\vdots$	$\vdots$
	G4. $\boxed{\neg A'(s_0;\tau_1, a)}^{+} \wedge C(s_0;\tau_1)$	$s_0;\tau_1;insert_E(a)$
A3. $\dfrac{A'(w,x) \rightarrow}{\boxed{\neg A'(delete'_A(w,x),x)}^{-}}$		
	G5. $A'(s_0;\tau_2, a) \wedge C(s_0;\tau_2;delete_A(a))$	$s_0;\tau_2;delete_A(a);insert_E(a)$

Table 2: Extracting Sequential-Composition Statements

rules is shown below. The rule says: if there is in the tableau an assertion $\alpha[p]$ with negative subformula p and a goal $\gamma[q]|\mu$ with positive subformula q, and θ is a most general unifier of p and q, then we can add $(\neg\alpha)\theta[false] \wedge \gamma\theta[true]$ as a new goal to the tableau with the instantiated transaction entry $\mu\theta$.

assertion	goal	transaction
$\alpha[p^-]$		
	$\gamma[q^+]$	μ
	$(\neg\alpha)\theta[false] \wedge \gamma\theta[true]$	$\mu\theta$

Example 4 The initial goal G1 of Example 3 is obviously equivalent to assertion A1 and goal G2 taken together in the tableau in Table 1. G2 and the insert-action axiom A2 have boxed subformulas that unify with respect to the rigidity axiom $a = w{:}a$ and the state-linkage axiom $w;insert_E(x) = insert'_E(w, w{:}x)$, with a most general unifier θ: $\{w \leftarrow s_0, \tau \leftarrow insert_E(a), x \leftarrow a\}$. Therefore we apply the resolution rule to obtain a new goal G3, together with one step in the synthesis: the generation of a basic statement $insert_E(a)$ in the transaction column. ◇

4.3 Sequential-Composition Statements

The synthesis of sequential-composition statements is achieved by successive applications of resolution rules. This requires that the composition-linkage axiom $w;(\tau_1;;\tau_2) = (w;\tau_1);\tau_2$ be built into the equational unification algorithm, as the following example illustrates.

Example 5 We cannot proceed with the synthesis at goal G3 in Example 4, because it is an assertion about a specific state (namely the state after hiring a at the input state), which is true or false but we have no control over its truth value. By building the composition-linkage axiom into the equational unification algorithm, we obtain goal G3′ instead by applying the resolution rule to G2 and A2 with a most general unifier $\{w \leftarrow s_0;\tau_1, \tau \leftarrow \tau_1;;insert_E(a), x \leftarrow a\}$, where τ_1 is a new transaction variable, as shown by the first tableau in Table 2. Now the goal G3′, which says that the state after hiring a is valid, is equivalent to G4 in the second tableau in Table 2 by the insert-frame axioms, which says that the state before hiring a should be valid and a should not be in relation Applicant. Now G4 and the delete-action axiom A3 have boxed subformulas that unify with a most general unifier $\{w \leftarrow s_0;\tau_2, \tau_1 \leftarrow \tau_2;;delete_A(a), x \leftarrow a\}$. So we apply the resolution rule once again to obtain a new goal G5, together with one more step in the synthesis: a sequential-composition statement $delete_A(a);;insert_E(a)$ is generated. ◇

4.4 Conditional-Branch Statements

The synthesis of conditional-branch statements relies on extended resolution rules, one version of which is shown below. The rule states: if there is a goal $\gamma[p]||\mu$ with positive subformula p and an assertion $\alpha[q]||\nu$ with negative subformula q, and θ is a most general unifier of p and q, then we can add the new goal $(\neg\alpha)\theta[false] \wedge \gamma\theta[true]$ to the tableau with a conditional-branch transaction entry *if* $p\theta$ *then* $\mu\theta$ *else* $\nu\theta$.

assertion	goal	transaction
	$\gamma[p^+]$	μ
$\alpha[q^-]$		ν
	$\neg\alpha\theta[false] \wedge \gamma\theta[true]$	*if* $p\theta$ *then* $\mu\theta$ *else* $\nu\theta$

However, the applications of extended resolution rules may cause transaction entries associated with new rows to be non-primitive, which means that the proof may not be constructive and we may not be able to extract a transaction from the proof even if the tableau is proven valid. To avoid the non-primitivity situation, the following *primitivity condition* is imposed in our synthesis system:

> *The application of a deduction rule is ignored if the resulting transaction entry is not primitive at* s_0.

Example 6 The transaction synthesized in Example 5 is not quite what we want: it tries to delete a from relation Applicant even if a is not a member of Applicant. What is needed is a conditional-branch statement that only performs such a deletion when necessary. Suppose the identity-frame axiom is built into the equational unification algorithm. Goals G4 and G5 in the tableau above have boxed subformulas that unify, with a most general unifier $\{\tau_1 \leftarrow \tau_2;;\Lambda\}$. So we apply the extended resolution rule to obtain goal G6, together with the synthesis of a conditional-branch statement if $A(a)$ then $delete_A(a)$ else Λ.

	goal	transaction
G4.	$\neg\boxed{A'(s_0;\tau_1, a)}\wedge$ $C(s_0;\tau_1)$	$s_0;\tau_1;$ $insert_E(a)$
G5.	$\boxed{A'(s_0;\tau_2, a)}^+\wedge$ $C(s_0;\tau_2;delete_A(a))$	$s_0;\tau_2;$ $delete_A(a);$ $insert_E(a)$
G6.	$C(s_0;\tau_2)\wedge$ $C(s_0;\tau_2;delete_A(a))$	$s_0;\tau_2;$ if $A(a)$ then $delete_A(a)$ else $\Lambda;$ $insert_E(a)$

4.5 Bounded-Iteration Statements

By intuition, a bounded-iteration **foreach** x **in** $R|p$ **do** τ achieves certain goals for every tuple x in relation R that satisfies the selection predicate p. Therefore, when we face with a universally quantified goal, a bounded-iteration statement should be generated. Hence the skolemization rules for quantifiers of universal force should serve as the iteration-introduction rules.

The skolemization rules for quantifiers of universal force need the cooperation of two deductive tableaux. Whenever a quantifier of universal force needs to be skolemized, a new tableau is called for the synthesis of the body of some bounded-iteration statement. If the new tableau does this successfully, the bounded-iteration statement is entered in the original tableau as one step of its synthesis. Of course the new tableau may create other tableaux as well. A simplified version of the rule is stated below.

If $\alpha[(\forall x)p(x)]||\mu$ is an assertion or goal with a subformula $(\forall x)p(x)$ primitive at $s_0;\tau_1$ that has universal force and is not contained in the scopes of any quantifiers in tableau Γ, where τ_1 is a transaction variable, $\bar{x} = [x_1, \ldots, x_m]$ are the only free variables in α, variables x_i, z are distinct for $1 \leq i \leq m$, and f is an m-ary function symbol not occurring in Γ, then we create a new tableau Δ with initial goal $p[z/f(\bar{x}), \tau_1/\rho]||s_0;\rho$, where ρ is a new transaction variable.

For any goal $true|s_0;\bar{u}$ in Δ and new variable x such that $\bar{u}$ has the form **if** $R(f(\bar{x}))\wedge q$ **then** u and t of the form **foreach** x **in** $R|q[f(\bar{x})/x]$ **do** $u[f(\bar{x})/x]$ is well-formed, we add $\alpha\theta[true]||\mu\theta$ to the assertion or goal list of Γ, where τ_2 is a new transaction variable and $\theta = \{\tau_1 \leftarrow \tau_2;;t\}$.

goal	transaction
$\alpha[(\forall x)^\forall p(x)]$	μ
$\alpha\theta[true]$	$\mu[\tau_1/\tau_2;;$**foreach** x **in** $R|q$ **do** $u]$

goal	transaction
$p[z/f(\bar{x}), \tau_1/\rho]$	$s_0;\rho$
$\vdots$	$\vdots$
true	$s_0;$**if** $R(f(\bar{x}))\wedge q$ **then** u

We may think of the skolemization rules intuitively as achieving a universally quantified goal by requesting the cooperation of two tableaux. In particular, when one tableau tries to prove a universally quantified goal by generating a bounded-iteration statement, it calls another tableau for the proof of a "lemma", which at the same time constructs the body u of a bounded-iteration statement. If the second tableau obtains the proof successfully, the first tableau takes it as one step in its synthesis.

Example 7 In the previous examples, we synthesized code to achieve the user's goal to hire a as well as to enforce the disjointness constraint C_1. When basic statement $delete_A(a)$ is generated, the referential integrity constraint C_2 can be violated if a has scheduled interviews. A bounded-iteration statement should be generated to remove a's interviews.

goal	transaction
G5. $A'(s_0;r_2,a) \wedge C(s_0;r_2;delete_A(a))$	$s_0;r_2;delete_A(a);insert_E(a)$
⋮	⋮
G6'. $A'(s_0;r_2,a) \wedge C(s_0;r_2) \wedge \boxed{(\forall x)(I'(s_0;r_2,x) \rightarrow \neg p(x,a))}$	$s_0;r_2;$ $delete_A(a);$ $insert_E(a)$
G7. $A'(s_0;r_3;t,a) \wedge C(s_0;r_3;t)$	$s_0;r_3;$ **foreach** x **in** $I\|p(x,a)$ **do** $delete_I(x);$ $delete_A(a);$ $insert_E(a)$

goal	transaction
$I'(s_0;\rho,b) \rightarrow \neg p(b,a)$	$s_0;\rho$
⋮	⋮
true	s_0;**if** $I(b) \wedge p(b,a)$ **then** $delete_I(b)$

Suppose we choose to delay the synthesis of conditional-branch statements as we did in Example 6. G5 in the first tableau above can be reduced to G6' by applications of frame axioms and $C_3(s_0;r_2)$, where $p(x,a)$ denotes the atomic s-formula $applicant(x) = name(a)$. G6' has a quantified subformula of universal force to which the skolemization rules are applicable. A second tableau is called to prove the matrix of the quantification and to synthesize the body of a bounded-iteration statement: **if** $I(b) \wedge applicant(b) = name(a)$ **then** $delete_I(b)$. The result obtained in the second tableau is then entered into the first tableau to generate a new goal G7 and synthesize a bounded-iteration t: **foreach** x **in** $I|applicant(x) = name(a)$ **do** $delete_I(x)$. ◇

4.6 Hire-Employee Transaction

Example 8 The goal G7 in the previous example has a subformula $A'(s_0;r_3;t,a)$ that is not affected by t. Hence it can be reduced to G8, which has a boxed subformula unifying with the first conjunct of G4. The resulting transaction entry contains a conditional-branch statement, which is identical to the Hire-Employee transaction that we have in mind. We do not need to proceed with goal G9, because it follows from assertion A1 by instantiating r_3 to Λ.

goal	transaction
G4. $\boxed{A'(s_0;r_1,a)} \wedge C(s_0;r_1)$	$s_0;r_1;insert_E(a)$
G8. $\boxed{A'(s_0;r_3,a)} \wedge C(s_0;r_3;t)$	$s_0;r_3;$ **foreach** x **in** $I\|p(x,a)$ **do** $delete_I(x);$ $delete_A(a);$ $insert_E(a)$
G9. $C(s_0;r_3) \wedge C(s_0;r_3;t)$	$s_0;r_3;$ **if** $A(a)$ **then** **foreach** x **in** $I\|p(x,a)$ **do** $delete_I(x);$ $delete_A(a)$ **else** $\Lambda;$ $insert_E(a)$

To summarize, the deductive synthesis of the Hire-Employee transaction can be depicted by the flow graph of Figure 5, where nodes denote goals and edges denote synthesized statements. The deduction proceeds top-down, while the extraction of the transaction proceeds bottom-up. ◇

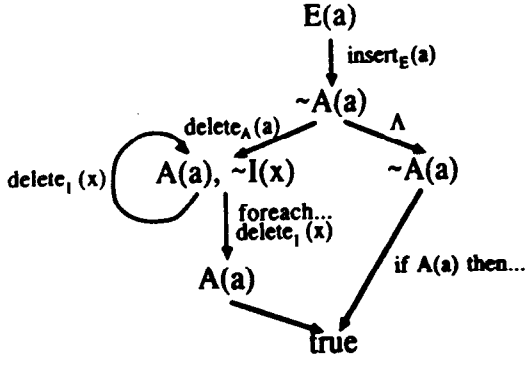

Figure 5: Synthesis of Hire-Employee Transaction

[5] Georgeff, M., Lansky, A. (editors), *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, Morgan Kaufmann, 1987.

[6] Green, C., "Application of Theorem Proving to Problem Solving"; *Proceedings of the International Joint Conference on Artificial Intelligence*, Washington, D.C., 1969, 219-239.

[7] King, J., "Query Optimization by Semantic Reasoning"; PhD Dissertation, *Technical Report* STAN-CS-81-857, Department of Computer Science, Stanford University, 1981.

[8] Manna, Z., Waldinger, R., "A Deductive Approach to Program Synthesis"; *ACM Transactions on Programming Languages and Systems* **2**:1, January 1980, 90-121.

[9] Manna, Z., Waldinger, R., "Problematic Features of Programming Languages: A Situational-Calculus Approach"; *Acta Informatica* **16**, 1981, 371-426.

[10] Manna, Z, Waldinger, R., "The Deductive Synthesis of Imperative LISP Programs"; *Proceedings of AAAI*, 1987, 155-160.

[11] Manna, Z, Waldinger, R., "How to Clear a Block: A Theory of Plans"; *Journal of Automated Reasoning* **3**, 1987, 343-377.

[12] Manna, Z., Waldinger, R., *The Logical Basis for Computer Programming, Vol.2: Deductive Techniques*; Addison-Wesley, to be published.

[13] McCarthy, J., "Situations, Actions, and Causal Laws"; *Semantic Information Processing*, M. Minsky (editor), MIT Press, 1968, 410-417.

[14] Nicolas, J-M., Gallaire, H., "Data Base: Theory vs. Interpretation"; *Logic and Databases*, H. Gallaire and J. Minker (editors), Plenum Press, 1978, 33-54.

[15] Pednault, E., "Toward a Mathematical Theory of Plan Synthesis"; PhD Dissertation, Computer Science Department, Stanford University, 1987.

[16] Qian, X., "The Deductive Synthesis of Database Transactions"; PhD Dissertation, *Technical Report* STAN-CS-89-1291, Department of Computer Science, Stanford University, 1989.

[17] Rich, C., Waters, R. (editors), *Readings in Artificial Intelligence and Software Enngineering*, Morgan Kaufmann, 1986.

[18] Sheard, T., Stemple, D., "Automatic Verification of Database Transaction Safety"; *ACM Transactions on Database Systems* **14**:3, September 1989, 322-368.

[19] Ullman, J., *Principles of Database and Knowledge-Base Systems*, Computer Science Press, 1988.

[20] Waldinger, R., Lee, R., "PROW: A Step Toward Automatic Program Writing"; *Proceedings of the International Joint Conference on Artificial Intelligence*, Washington, D.C., 1969, 241-252.

5 Discussion

Program correctness has been approached by either program verification or program synthesis. The application of program verification to transaction programming is illustrated in [2, 18]. Program synthesis techniques are typically classified into two broad classes: deductive versus transformational approaches[17]. It is reasonable to view the transformational approach as extending the deductive approach with proven lemmas or correctness-preserving transformation rules. In deductive program synthesis, specifications are theorems and programs are (constructive) proofs. Research in this area has concentrated primarily on the generation of applicative programs[8].

The deductive synthesis of imperative programs has been approached with various forms of situational logic, which was first introduced by [13] as a very convenient formalism for describing situations, actions, and causality. [1] used the formalism to reason about programs that manipulate the states of a computation. It was used in PROW[20] to synthesize imperative programs. Manna and Waldinger took situational logic as a framework to formalize ALGOL-like language constructs[9], such as pointers and procedure invocation. They also proposed a restricted variant, which we used to build our transaction theory, to avoid synthesizing non-executable programs. This variant was applied to the deductive synthesis of imperative LISP programs[10]. Mathematical induction was used as a proof construct from which recursive programs can be extracted. Specifications often must be generalized first in order for the induction to be carried through.

A related area of study is robot planning in AI[5]. Situational logic was first used in QA3[6] to synthesize robot plans, which are straight line programs of basic actions. Hoare logic was used in plan synthesis methods based on goal reduction[15], which has the difficulty in generating plans with control structures more powerful than sequential composition. Manna and Waldinger recently adapted program synthesis techniques to the automated generation of recursive plans[11]. They noticed however that fully rigorous theorem proving might not be suited to planning, where imprecise inference is often necessary.

Transformational synthesis has been used intensively in database programming, such as query processing[19] and semantic query optimization[7], where declarative query specifications are transformed into executable and efficient query plans. In [4] program transformation techniques were applied to the synthesis of iterative programs from relational query specifications. There has been essentially no work in the deductive synthesis of database transactions.

We applied deductive program synthesis techniques to the deductive synthesis of database transactions that preserve the validity of integrity constraints. With our synthesis system a large class of database transactions can be generated from logical specifications. The synthesis process is formalized as the process of proving the validity of specification theorems constructively and extracting qualified transactions from the proofs. We extended the Manna-Waldinger deductive-tableau synthesis system, with special-purpose deduction rules to extract bounded-iteration statements from constructive proofs of universally quantified specifications. It requires the cooperation of multiple tableaux.

In program synthesis, the degree of automation is counterproportional to the degree of ingenuity needed. The less human guidance is involved in a synthesis system, the more automated the system is. Our choice of focus on the synthesis of transactions enables us on one hand to generate a much more expressive class of transactions than iteration-free programs, and on the other hand to reduce greatly the human guidance necessary in synthesizing recursive programs, such as generalizing specifications or inventing well-founded relations.

Automation holds the only hope in solving the software crisis. Yet, forty years of experience in automated programming research tells us that general-purpose, fully-automated program synthesis — just like general-purpose problem solving systems — is unlikely to succeed in the foreseeable future. Therefore we believe that program synthesis in restricted but better-understood domains is more promising. Transaction programming in databases is such a simple and yet important domain. This work stands as a solid proof of our claim and represents significant progress towards the goal of automated programming of database transactions.

Acknowledgement

The author has benefited greatly from discussions with Gio Wiederhold, Richard Waldinger, Cordell Green, and Peter Ladkin. This work was supported in part by the Defense Advanced Research Projects Agency under Contract N39-84-C-0211 for Knowledge Based Management Systems and by Rome Air Development Center under Contract 30602-86-C-0026 for Knowledge-Based Software Assistant.

References

[1] Burstall, R., "Formal Description of Program Structure and Semantics in First-Order Logic"; *Machine Intelligence* 5, B. Meltzer and D. Michie (editors), Edinburgh University Press, Edinburgh, Scotland, 1969, 79-98.

[2] Casanova, M., Bernstein, P., "A Formal System for Reasoning about Programs Accessing a Relational Database"; *ACM Transactions on Programming Languages and Systems* 2:3, July 1980, 386-414.

[3] Codd, E. F., "A Relational Model of Data for Large Shared Data Banks"; *Communications ACM* 13:6, 1970, 377-387.

[4] Freytag, J., Goodman, N., "On the Translation of Relational Queries into Iterative Programs"; *ACM Transactions on Database Systems* 14:1, 1989, 1-27.