A Determinacy Testing Algorithm for Nondeterminate Flat Concurrent Logic Programming Languages

E. Tick — University of Oregon
M. Korsloot — Delft University of Technology

CIS-TR-90-18
November 1990

Abstract

This paper describes an algorithm for the code generation of determinacy testing for nondeterminate flat concurrent logic programming languages. Languages such as Andorra and Pandora require that procedure invocations suspend if there is more than one candidate clause potentially satisfying the goal. The algorithm described has been developed specifically for a variant of flat Pandora based on FGHC, although the concepts are general. We have extended Kliger and Shapiro's decision-graph construction algorithm to compile "don't know" procedures which must suspend for nondeterminate goal invocation. The determinacy test is compiled into a decision graph quite different from those of committed-choice procedures, but we argue that in most cases, the same low space complexity is retained.

Department of Computer and Information Science
University of Oregon

Contents

List of Figures

List of Tables

1 Introduction

"Lotta old computer hacks spent their *lives* programming machines."
Dogfight
M. Swanwick and W. Gibson [5]

This paper describes compilation techniques for the Andorra/Pandora family of parallel logic programming languages [1, 3, 4, 6, 11]. These languages share a common execution model wherein a goal invocation is suspended if it can potentially be satisfied by two or more clauses in a procedure definition. We call these "nondeterminate" goal invocations. If all goals suspend, causing deadlock, then one nondeterminate goal is selected and forced to execute, creating a choicepoint for potential backtracking.

In this paper, we limit ourselves specifically to our own variation of "flat" Pandora based on Flat Guarded Horn Clauses (FGHC) [10]. The language we consider has flat guards, no synchronization operator, and implicit synchronization rules, all as in FGHC. However, the compilation techniques we developed are applicable to the entire Andorra/Pandora family.

The techniques described are based on the decision-tree and decision-graph algorithms developed by S. Kliger and E. Shapiro [8]. The algorithm we introduce is for generating the code for a *determinism tester*: the code that checks whether two or more clauses can potentially satisfy a goal invocation. Our tester is a decision graph that is built in quite a different manner than graphs for committed-choice languages, as for example done by S. Taylor [9].

2 Pandora: Definitions and Example

In this section we introduce flat Pandora to put our later compilation techniques into clear perspective. Our flat Pandora programs consist of FGHC procedures categorized as either *don't care* or *don't know* procedures [3]. In a nutshell, a flat Pandora program executes like an FGHC program, i.e., concurrent fine-grain processes communicate to solve a problem. In addition, a don't-know goal can *backtrack* through its clauses in an attempt to satisfy its invocation.

At the time of its execution, a don't-care goal is treated as in any committed-choice language. If any clause head and guard can satisfy a goal invocation, then the goal *commits* to that clause and reduces. If the head and guard of two or more clauses can satisfy a goal invocation, then *any* one of the clauses is chosen for commitment. If no clause head and guard can satisfy the goal invocation, but one or more can *suspend*, then the goal suspends. Otherwise if no clause head and guard can succeed or suspend, then the goal fails. Note that unlike a pure committed-choice language where goal

failure implies program failure, in Pandora goal failure causes backtracking.

A don't-know goal is executed differently from a don't-care goal. If *only one* clause head and guard can satisfy the goal invocation, then the goal commits to that clause and reduces. However, if the head and guard of two or more clauses can succeed, then the goal suspends. If no clause head and guard can satisfy the goal invocation, but one or more can suspend, then the goal suspends as well. Otherwise if no clause head and guard can succeed or suspend, then the goal fails.

In summary, don't-know goals act similar to goals in a committed-choice language with the exception that if more than one clause can commit then the goal suspends. We call this a "nondeterminate goal invocation," and the test to determine if one or more clauses can commit is called the *determinism test.* Flat Pandora execution proceeds much like that of FGHC, with nondeterminate don't-know goal invocations suspending. At any point in time, the group of suspended goals consists of don't-care and don't-know goals. Unbound logical variables are "hooked" to these suspended goals, thus enabling resumption as in any committed-choice language implementation (e.g., [7]).

For don't-care goals, binding a hooked variable is the only method of resumption. Resumption causes the goal to be reconsidered for execution, although it does not guarantee reduction: the goal may suspend on another variable. For don't-know goals, it is also the case that binding a hooked variable will cause resumption, but it is not the only method. Certain suspended don't-know goals may not even have associated hooked variables because their procedure definitions are truly nondeterminate. In that case, the goals remain suspended until no executable goals remain, i.e., deadlock ensues.

To break deadlock, any suspended don't-know goal is forced to reduce. The fact that the goal was suspended implies that two or more clauses can commit to this goal invocation. Forced reduction must choose a clause to commit, after creating a choicepoint for potential backtracking. The choicepoint will direct execution to alternative clauses. Any clause with a non-failing head and guard can be reduced (as in Prolog). There is no distinction between input and output variables at this stage: the goal is *unified* with the clause heads until one is satisfied. Unlike don't-care goal reduction, output bindings may be performed during head and guard unifications.[1] However, certain guards cannot be forced to reduce with unbound inputs, e.g., `X>3` cannot be evaluated if `X` is unbound. These guards must suspend and another clause would be chosen. It could be the case that the "don't know" goal chosen to break the deadlock *cannot* reduce because of such tests. If this occurs, another goal must be chosen.

Given an "incorrect" program, all suspended goals may be don't-care in which case

[1] As for all bindings, trailing must be performed if the variable to be bound is older than the current choicepoint.

deadlock is fatal. It can also be the case that although don't-know goals exist, none of them can be forced to reduce because of guards that require bound variables. This is another form of fatal deadlock. Given a "correct" program, forcing reduction of a "don't know" goal may issue new don't-care goals and/or create bindings that resume old don't-care goals. However, some don't-know goals may generate only nondeterminate don't-know goal invocations and/or create bindings that do not resume old "don't care" goals. In this case, deadlock remains and another goal must be chosen for forced execution. In general, it is hoped and expected that the execution periods of determinate reduction of don't-care goals are much longer than the periods required to break deadlocks.

Choicepoints leading to multiple, independent OR-branches of the execution tree can potentially be searched in parallel, but this issue is orthogonal to the paper and will not be discussed.

To illustrate the flat Pandora procedural semantics, a small example is presented. Consider the following code:

```
:- dontknow a/3.

a(1,1,1).
a(2,1,1).
a(2,2,1).
a(2,2,2).

b(1,A) :- A=yes.
b(2,A) :- A=no.
```

Procedure `b/2` is assumed to be don't-care since it has no declaration. Suppose we make the following query:

```
?- a(X,Y,Z), b(Y,A), Z=2.
```

Assuming that the goals are evaluated in their sequential order, the sequence of actions executed is: `a/3` is found to be nondeterminate and suspends, `b/2` suspends, `Z` is bound to 2, `a/3` resumes and is found to be determinate (clause 4), `X` and `Y` are bound, `b/2` is resumed, and `A` is bound to `no`. In contrast, consider the following query:

```
?- a(X,Y,Z), b(Y,no), Z=1, X=2.
```

One possible sequence of actions executed is: `a/3` is found to be nondeterminate and suspends, `b/2` suspends, `Z` is bound, `a/3` resumes, found to be nondeterminate and suspends again, `X` is bound, `a/3` resumes, found to be nondeterminate and suspends a third time. Now deadlock ensues so `a/3` is *forced to reduce*, `Y` is bound to 1, `b/2` is resumed and *fails*, backtracking retries the execution of `a/3`, `Y` is bound to 2, `b/2` is resumed and succeeds. These two examples sufficiently illustrate the execution mechanisms of flat Pandora to understand the rest of this paper.

3 Decision Graphs: Background

The don't-care procedures in flat Pandora are compiled in a manner similar to the decision-graph algorithm given by Kliger and Shapiro for FCP [8]. Because we are restricting ourselves to FGHC, our notation can be made simpler than that of the FCP algorithm as we have no tell guards. We review the terminology and algorithm here as a foundation for our method of compiling don't-know procedures.

A guarded Horn clause is of the form:

$$H :- G_1, G_2, ..., G_m \mid B_1, B_2, ..., B_n.$$

where m and n are zero or positive integers. H is the clause head, G_i is a guard goal,[2] and B_i is a body goal. The commit operator '|' divides the clause into a passive part (the *guard*) and active part (the *body*). The first step in the compilation process is to translate source clauses into *canonical form.*

Definition: A *simple term* is either a constant, a variable, or a compound term in which the arguments are pairwise different variables. A *complex term* is a compound term in which there exists either a non-variable argument or two arguments with identical names. For example, $f(X,Y)$ is a simple term and $g(h(X))$ is a complex term. □

Definition: A complex term is *flattened* into a pair (F, S), where F is a simple term and S is a *constraint set.* To flatten a complex term the following rules are applied with S initially empty. Each of k instances of a shared variable X is replaced by a unique variable $\{Z_{i+j} \mid 1 \leq i \leq k\}$ and $S := S \cup \{Z_{j+1} = Z_{j+2}, Z_{j+1} = Z_{j+3}, \ldots\}$, for C($k$,2) pairs.[3] A constant argument α is replaced by a unique variable Z_j, and $S := \{Z_j = \alpha\} \cup S$. A complex argument is flattened into (F', S') and replaced by a unique variable Z_j, and $S := \{Z_j = F'\} \cup S' \cup S$. □

Definition: Given a clause $C =$ "$H :- G \mid B$." then its normalized form is $C' =$ "$H' :- G' \mid B$." A complex term in $H \cup G$ is flattened into a pair (F, S) and replaced by a unique variable Z_i in C'. Furthermore $G' := \{Z_i = F\} \cup S \cup G'$. □

Normalization is needed to simplify a clause into a trivial head and an extended guard containing constituent constraints. This form facilitates code generation of triples. Normalization however is not enough: we need to rename variables among the clauses belonging to the same procedure such that variables corresponding to the same depth within the same procedure argument have the same name. This characteristic is necessary for indexing purposes, as is shown later.

Definition: A simple term $H = p(X_1, X_2, ..., X_n)$ is *expanded*, with respect to a set of constraints G, into a tree, $tree(H)$, with n branches labeled $1, \ldots, n$. For a variable

[2] more precisely, an "ask" guard.

[3] Index j is chosen to produce unique variables.

X_i and term T that appear in a constraint $g = \{X_i = T\} \in G$ or $g = \{T = X_i\}$, the subtree at root i is T expanded with respect to $G \setminus \{g\}$. Otherwise the subtree at root i is the leaf X_i. □

Definition: A normalized clause $C =$ "$H :- G \mid B.$" is *renamed into canonical form* by expanding H with respect to G into $tree(H)$. Each variable X in C is replaced by Z_p where p is the *label sequence* from the root of $tree(H)$ to the vertex corresponding to X. □

Using this sequence gives a complete ordering of all variables and retains identical names for corresponding variables among clauses. For instance, $f(g(X,Y))$ has normalized form $f(A) :- A = g(B,C)$ and canonical form $f(Z_1) :- Z_1 = g(Z_{1,1}, Z_{1,2})$.

Definition: A canonical-form procedure consists of canonical-form clauses, each of the form $< i, G >$ for clause number i with guard G. The head is not needed because it has been flattened away. The head arguments are always named $Z_1, Z_2, ..., Z_n$ for an arity-n procedure. The body is not listed because the algorithms we discuss in this paper do not deal with body compilation. □

From this point on in the paper, all references to "procedures" and "clauses" implicitly assume canonical forms. Both don't-care and don't-know procedures are syntactically identical and are converted into canonical form. The goals in guard G consist of builtin predicates such as $=/2$, $\neq/2$, $>/2$, var/1, otherwise, etc. We call these *constraints* or *tests*. For efficiency, we can represent G as a bit-vector corresponding to inclusion in the union of all guards of a procedure. This makes set operations on guards very fast.

Definition: Variables within a clause are partially ordered on their label sequences: $Z_{i,j,...,k} \geq_\psi Z_{i,j,...,k,l,m,...}$. In words, a variable is ψ-greater than or equal to another variable if their label sequences share the same prefix and the latter sequence is longer or equal. For example, $Z_1 \geq_\psi Z_{1,2}$, whereas Z_2 and $Z_{1,2}$ have no ordering. □

Definition: The constraints within a canonical-form clause form a relation with respect to $\geq_\psi$, defined as follows:

$$g(Z_a, ..., Z_b) \geq_\psi g(Z_c, ..., Z_d) \ \underline{iff} \ \{\exists\, i \in a, ..., b \wedge j \in c, ..., d \mid Z_i \geq_\psi Z_j\}$$

□

For example, $\{Z_1 = Z_2\} \geq_\psi var(Z_{1,2})$, whereas both $\{Z_{1,2} = Z_2\} \geq_\psi \{Z_1 = Z_{2,1}\}$ and $\{Z_1 = Z_{2,1}\} \geq_\psi \{Z_{1,2} = Z_2\}$.

Definition: The *residual* of a clause $C =< i, A >$ with respect to a constraint g is

denoted as $\Re(C,g)$.[4]

$$\Re(C,g) = \begin{cases} <i, A \setminus G> & G = \{g' \in A \mid g \Rightarrow g'\} \neq \emptyset \\ \emptyset & otherwise \end{cases}$$

□

Definition: The *otherwise-residual* of a clause $C =< i, A >$ with respect to a constraint set Γ is denoted as $\Re_o(C,\Gamma)$.

$$\Re_o(C,\Gamma) = \begin{cases} \emptyset & G = \{g \in \Gamma, g' \in A \mid g' \Rightarrow g\} \neq \emptyset \\ <i, A> & otherwise \end{cases}$$

□

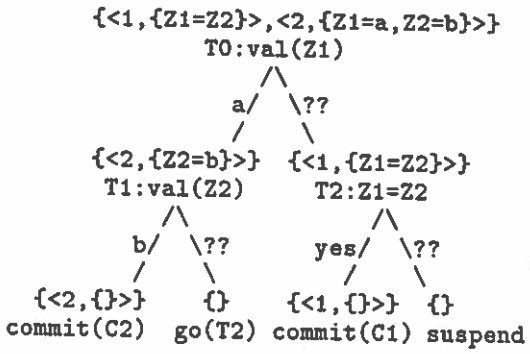
Definition: The residual and otherwise-residual of a procedure, $\Re(P,g)$ and $\Re_o(P,g)$, are the union of all residuals corresponding to the clauses in P. □

4 Committed-Choice Compilation: Kliger's Method

Kliger's algorithm [8] for compilation of don't-care (committed-choice) procedures is reviewed in this section (see Figure 1). The algorithm we discuss is slightly modified for FGHC execution, i.e., no tell guards. The code-generation function *decision-graph(P,C)* is passed procedure P and continuation C. The initial continuation is a suspend instruction that will be explained later. The resulting value of the function is an abstract code tree that can easily be flattened into a linear code sequence. To illustrate the algorithm, consider the following don't-care procedure:

```
f(X,X) :- ...
f(a,b) :- ...
```

The final code graph produced is:



[4]This (and the next) definition are due to Kliger. One may think of this residual as a don't-care residual $\Re_{dc}(C,g)$ to be consistent with later terminology.

decision-graph(P, C)

if ($P = \emptyset$) then return(`go(C)`);

if ($<i, \emptyset> \in P$) then

— any clause with empty guard can commit

return(`commit(i)`);

else

if ($P = \{<i, G>\}$) then

— we can generate code for a single clause

return(`attempt(i,G) else go(C)`);

else

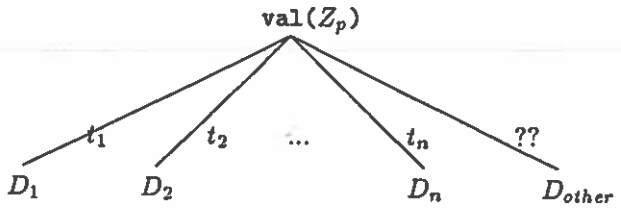
— multiple clauses, none with empty guards

— first choose indexing variable and collect constraints

$\Gamma :=$ index(P);

if ($\Gamma = \{Z_p = t_1, ..., Z_p = t_n\}$) then

— return the following code tree:



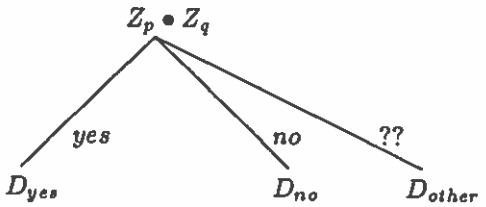
$\{ \forall i \in 1 \ldots n \mid D_i :=$ decision-graph($\Re(P, Z_p = t_i), D_{other}$) $\}$

$D_{other} :=$ decision-graph($\Re_o(P, \Gamma)$,C);

else

if ($\Gamma = \{Z_p \bullet Z_q, \neg(Z_p \bullet Z_q)\}$) then

— return the following code tree:



$D_{yes} :=$ decision-graph($\Re(P, Z_p \bullet Z_q), D_{other}$);

$D_{no} :=$ decision-graph($\Re(P, \neg(Z_p \bullet Z_q)), D_{other}$);

$D_{other} :=$ decision-graph($\Re_o(P, Z_p \bullet Z_q), C$);

Figure 1: Decision Graph Construction Algorithm (Based on Kliger).

Each node is labeled with its residual P, followed by a set of constraints Γ, as returned by the indexer. Each leaf is either **suspend**, `fail`, **go**, or a code segment. A branch labeled '??' is an otherwise-residual branch, i.e., the residual at its leaf is computed as $\Re_o(P, \Gamma)$. If Γ represents an "ask" test $Z_p \bullet Z_q$ then branches are labeled 'yes' and 'no' with leaves computed as $\Re(P, Z_p \bullet Z_q)$ and $\Re(P, \neg(Z_p \bullet Z_q))$ respectively. If Γ represents a switch test $\mathtt{val}(Z_p)$ then a branch is labeled with a ground value α and its leaf computed as $\Re(P, Z_p = \alpha)$.

For a single-clause residual, tree generation terminates with code committing to the clause. For an empty residual, tree generation terminates with a control transfer to the current continuation.[5]

Control follows the otherwise branches whenever a test fails or cannot be evaluated because of unbound variables. Note that failure will occur at the **suspend** instruction if the suspension stack is empty. For instance, if Z_1 and Z_2 are bound with different values, then the test at T2 will fail, taking the otherwise branch to the **suspend**, which will fail because of the empty suspension stack.

The code space required by this example procedure is grossly estimated as three test nodes and four leaves. The actual code generated for each node will of course differ, but in general nodes require more instructions than simple leaves. We model **suspend** and `fail` as traps, i.e., their code bodies are not expanded in-line.

5 Determinism Testing

This section introduces an algorithm to generate decision graphs for don't-know procedures in a nondeterminate concurrent logic programming language. Specifically we generate an abstract code graph for flat Pandora based on FGHC. The code graph can easily be flattened into a linear sequence of triples suitable for code generation. Qualitatively, the algorithm presented has space complexity comparable to Kliger's method. The expected path length through the code obviously depends on how procedure arguments are dynamically bound, but it also depends on the sophistication of the unspecified indexer at choosing critical constraints near the root, as in Kliger's method. Similarly, the completeness of the algorithm, i.e., the percentage of determinate invocations that commit immediately, is dependent on the strength of the unspecified guard inference mechanism. For the simplest inference mechanism over arbitrarily complex unifications, which is the common case, the code generated is complete. More discussion about completeness over other domains is given in Section 5.1.

[5] In this example, the bottom right subtree is optimized by combining the 'no' branch (terminating in a transfer to the suspend continuation) with the otherwise branch.

Definition: Given that A is the set of guard goals for clause i, then the *don't-know residual* of a clause $C =< i, A >$ with respect to a constraint g is denoted as $\Re_{dk}(C, g)$. Let $G = \{g' \in A \mid g \Rightarrow g'\}$ and $G' = \{g' \in A \mid g \Rightarrow \neg g'\}$.

$$\Re_{dk}(C, g) = \begin{cases} \emptyset & G' \neq \emptyset \\ < i, A \setminus G > & otherwise \end{cases}$$

□

This definition means that only if g disproves the clause, will the residual be empty. Otherwise the clause is retained, even if it is not implied by g. The essence of this inclusion is the construction of a full decision tree rather than a decision graph. However, using the code-sharing optimizations described later, effectively a graph is built.

Definition: The *unbound residual* of a clause $C =< i, A >$ with respect to a constraint g is denoted as $\Re_u(C, g)$. Let $G = \{g' \in A \mid g \geq_\psi g'\}$.

$$\Re_u(C, g) = < i, A \setminus G >$$

□

This definition means that the original clause C is retained except for those guards g' that test variables dependent on the unbound variables in g. The unbound residual of clause with respect to a constraint set Γ is defined similarly.

We now describe the decision-graph construction algorithm for don't-know procedure determinacy testing. The algorithm (outlined in Figures 2, 3, and 4) has two arguments: the input procedure P and a pointer to a global residual table *Table*, which is initially empty. Unlike Kliger's algorithm, a continuation is no longer necessary. Code-sharing optimization exploits the depth-first generation of code by using the residual table. This table is indexed by either a guard test, such as `Z1=Z2`, or a special key, `leaf`, for those entries which have a residual containing a single clause.

When the algorithm is down to a leaf (only a single clause is left in the residual,(1)), the *residual table* is tested for a matching entry, using `leaf` as the search key. If a match is found (2), we can either generate a `go(T)` instruction or an `execute(i,G)` instruction.[6] For code-size optimization, the go is better because the `execute` instruction includes code for checking the suspension stack and possibly forcing the bindings specified in G. Returning a `go(T)` instruction, however, maximally shares code. If no match is found, an `execute(i,G)` instruction is returned (3).

If multiple clauses are left in the residual, the indexer is invoked to select a test Γ from P to index on. We first check if the clauses all have empty guards. This situation is detected by the indexer returning $\Gamma = \emptyset$ and a `suspend` instruction is returned (4).

[6]The exact semantics of `execute` is given in Section 5.3. For now it suffices to say that we commit to clause i.

```
decision-graph(P, Table)
    if (P = ∅) then return(fail);
    if (P = {<i, G>}) then                                              (1)
        — we can generate code for a single clause
        if (∃ [leaf, P, T] ∈ Table) then
            return(go(T));                                              (2)
        else
            Table := [leaf, P, Label] ∪ Table;
            return(Label:execute(i,G));                                 (3)
    else
        — multiple clauses
        — first choose indexing variable and collect constraints
        Γ := index(P);
        if (Γ = ∅) then return(suspend);                                (4)
        if (∃ [Γ, Q, T] ∈ Table | P ⊇ Q) then                           (5)
            — matching residual table entry so code sharing possible
            if (P = Q) then
                return(go(T));                                          (6)
            else
                — not exact match, so partial code sharing
                Table := [Γ, P, Label] ∪ Table;
                return(share-node(P, Q, Label, Γ, Table));              (7)
        else
            — miss in residual table, so no code sharing
            Table := [Γ, P, Label] ∪ Table;
            return(generate-node(P, Γ, Table));                         (8)
```

Figure 2: Decision-Graph Algorithm for Don't-Know Procedures.

share-node($P, Q,$ `Label`, $\Gamma, Table$)

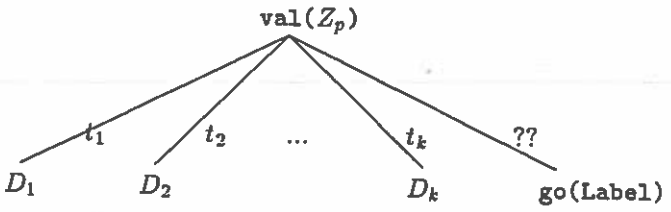
if ($\Gamma = \{Z_p = t_1, ..., Z_p = t_n\}$) then

$\gamma := \{g \in \Gamma, <i, A> \in P \setminus Q \mid g \in A\}$;

— thus $\gamma = \{Z_p = t_1, ..., Z_p = t_k\}$, where $k < n$

— other values of Z_p are covered at shared code `Label`

— return the following code tree:



$\{ \forall i \in 1 \ldots k \mid D_i :=$ decision-graph$(\Re_{dk}(P, Z_p = t_i), Table)$ $\}$

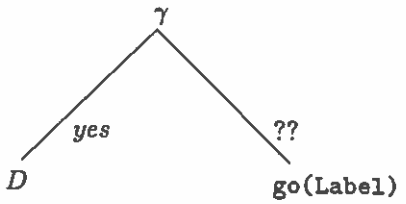
else

if ($\Gamma = \{Z_p \bullet Z_q, \neg(Z_p \bullet Z_q)\}$) then

$\gamma := \{g \in \Gamma, <i, A> \in P \setminus Q, g' \in A \mid g' \Rightarrow g\}$;

— thus γ is "ask" test

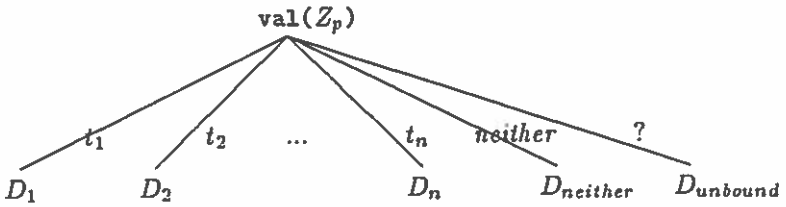
— return the following code tree:



$D :=$ decision-graph$(\Re_{dk}(P, \gamma), Table)$;

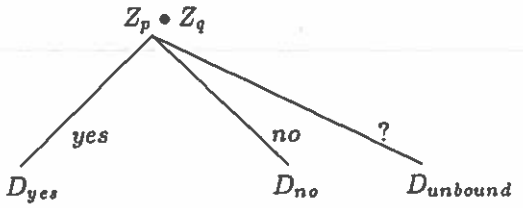
Figure 3: Shared Node Generation for Don't-Know Procedure.

generate-node($P, \Gamma, Table$)
if ($\Gamma = \{Z_p = t_1, ..., Z_p = t_n\}$) then
— return the following code tree:



$\{ \forall\, i \in 1 \ldots n \mid D_i :=$ decision-graph$(\Re_{dk}(P, Z_p = t_i), Table)\ \}$
$D_{neither} :=$ decision-graph$(\Re_o(P, \Gamma), Table)$;
$D_{unbound} :=$ decision-graph$(\Re_u(P, \Gamma), Table)$;

else
if ($\Gamma = \{Z_p \bullet Z_q, \neg(Z_p \bullet Z_q)\}$) then
— return the following code tree:



$D_{yes} :=$ decision-graph$(\Re_{dk}(P, Z_p \bullet Z_q), Table)$;
$D_{no} :=$ decision-graph$(\Re_{dk}(P, \neg(Z_p \bullet Z_q)), Table)$;
$D_{unbound} :=$ decision-graph$(\Re_u(P, Z_p \bullet Z_q), Table)$;

Figure 4: New Node Generation for Don't-Know Procedure.

To check if code sharing is possible, the residual table is accessed with Γ, returning a set of residuals corresponding to nodes in the tree that have the same test (5). The residual P is compared with each candidate Q from the table until one is found such that $P \supseteq Q$. If no such entry exists, then code cannot be shared and the function `generate-node()` is invoked ((8), see also Figure 4). If the stronger condition of equality exists, then the entire node can be shared with a simple control transfer and a `go(T)` instruction is returned (6). In the general case, part of the new node must be built with an otherwise-continuation transferring control to the shared node. The new node is then constructed by invoking the function `share-node()` ((7), see also Figure 3).

As an example of this algorithm, consider the previous `f/2` procedure, now declared as don't-know. Its code tree is given in Figure 5. The residuals are written above each node. Branches labeled '?' are unbound-residual branches and branches labeled with

'neither' are otherwise-residual branches. Control follows the '?' branches whenever a test cannot be evaluated because of unbound variables. Control follows the neither branches whenever a test fails (for example, when a variable is bound, but none of the tests succeed). Control transfers with go represent code sharing. In this simple example, entire subtrees can be shared because the corresponding residuals are identical. Note that this code is *complete* in the sense that it is guaranteed to detect determinacy.

5.1 Indexing

The decision-graph generation mechanism, as defined in Kliger's work and extended here, hinges on the indexer selecting a test either in the form of a switch on value, or builtin predicate, e.g., >/2. As will be obvious, the indexer plays an important part in this algorithm. Apart from the strength of its inference mechanism, it is important to note that the indexer should be fully determinate. This means that given the same set of residuals, the indexer should always return the same Γ, even if multiple, equally well-suited choices exist. If no choice is possible, which is the case when all guard goals are empty, the indexer returns $\emptyset$.

Our method is only as complete as is its inference mechanism in determining $g \Rightarrow g'$ in the residual definitions. Shared variables cause problems because they transfer constraints indirectly. For example, $\{\mathtt{X} = \mathtt{Y}, \mathtt{Y} = \mathtt{Z}, \mathtt{Z} = \mathtt{W}\}$ implies that $\mathtt{X} = \mathtt{W}$. Similarly, $\{\mathtt{X} > \mathtt{Y}, \mathtt{Y} > \mathtt{Z}\}$ implies that $\mathtt{X} > \mathtt{Z}$. The first case can easily be handled during conversion to canonical form, as mentioned earlier. The latter case can be handled in a similar manner without significant code expansion, since sharing is not frequent. However, this method is limited, and does not easily operate *across* clauses. Furthermore, constraints involving both equalities and comparisons need a strong inference mechanism. Given the constraint $g = \{\mathtt{Z} = 0\}$, the indexer should be able to infer that "$\{\mathtt{Z} > 0\} \Rightarrow \neg g$," which is non-trivial. However, we are confident that complex sets of interacting constraints within the same procedure are rare in most logic programs.

5.2 Optimizations

To produce the minimal decision graph, thus achieving the maximum possible sharing, it is important to consider the following optimizations during or after the code graph generation. These optimizations are an extension to the sharing of code, which was discussed before.

The first and simplest optimization, which can occur either during or after the actual code graph generation, is the case where a node has three branches, one of which is a continuation to the other branch. These two shared branches can then be collapsed into a single otherwise branch. These otherwise-residual branches are labeled '??' and are

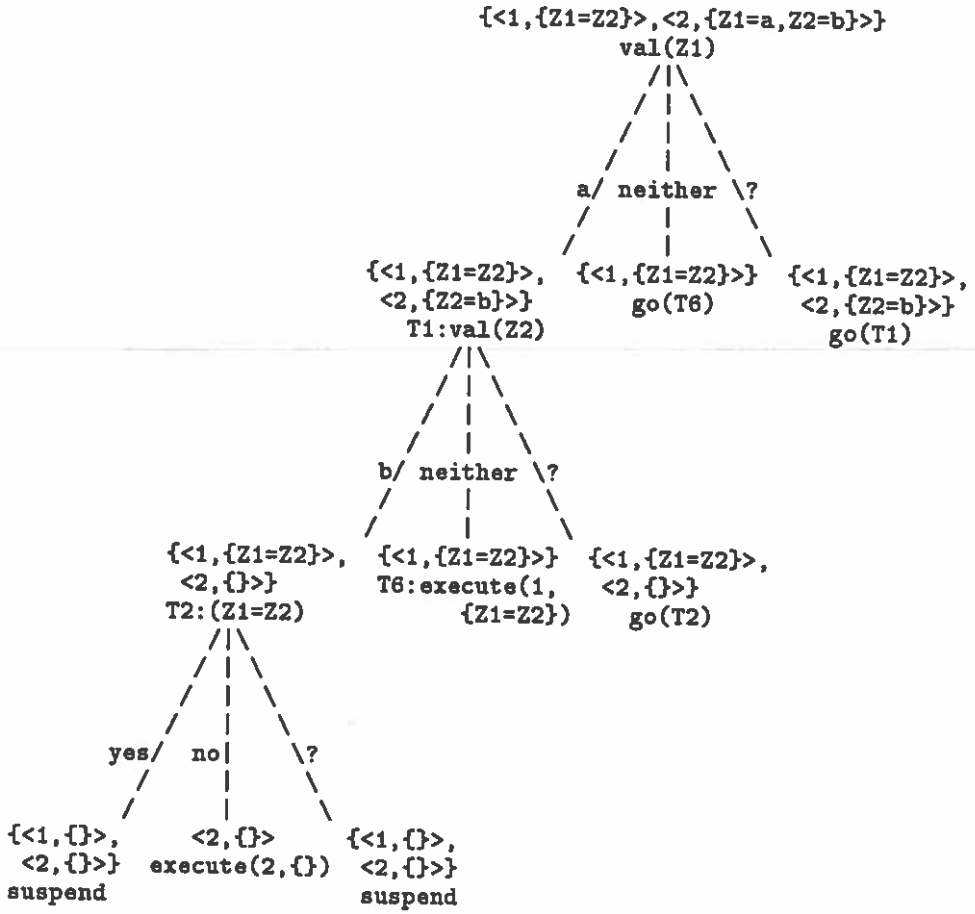
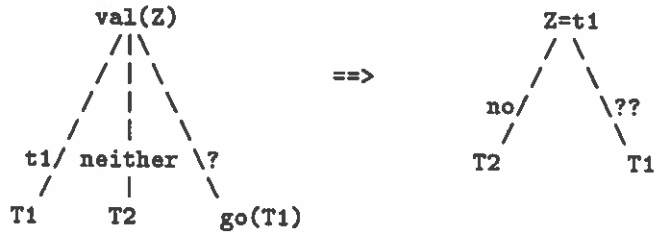


Figure 5: Decision Graph for `f/2` (Compiler's Internal Code Tree).

treated as described for don't-care procedures. If the indexer returns $\Gamma = \{Z = t_1\}$, the normal test would be `val(Z)`; in this case, however, it is replaced by the test `Z = t1`, as illustrated below:



A second optimization, which is a more general version of the previous one, can be applied when the residual contains multiple clauses, each with the same, non-empty, guard, thus $\Re = \{< C_i, G >, \ldots, < C_k, G >\}$, where $G = \{g_1, g_2, \ldots, g_m\}$.[7] The minimal code for this case would be:

```
if (¬g1 ∧ ¬g2 ∧ ... ∧ ¬gm)
     then fail;
     else suspend;
```

It is also possible to replace this entire node with a single **suspend** instruction, thus avoiding the test for G. The code size will be smaller, but this goes against our earlier prerequisite of "fast" failure, i.e., a failure should be detected as quickly as possible, instead of suspending.

5.3 Suspensions

Because suspension and resumption of goals are costly operations, it is necessary to perform them as efficiently as possible. In this section, we describe a possible and efficient implementation of this suspension mechanism, using a combination of an intelligent compile-time code generator and an efficient run-time implementation of this scheme.

At compile time, when the code generator is down to a **suspend** leaf, the residuals have the form $\Re = \{< C_i, \emptyset >, < C_j, \emptyset >, \ldots, < C_k, \emptyset >\}$. For each leaf we can now generate the WAM-like code:

```
try   Ci_H
retry Cj_H
...
trust Ck_H
```

[7] The case for $G = \emptyset$ is discussed in Section 5.3.

where `Ci_H` points to the head of clause i. A continuation in the goal record is set to point to the `try` instruction. This ensures that only the subset of clauses known to be candidates is executed when the goal is resumed.

Another option is to generate a single sequence of `try`, `retry`, and `trust` instructions for *all* clauses, and to give all suspensions the *same* continuation `C1_H`, i.e., the first clause of the procedure. This would obviate the need for individual `try` sequences, at the expense of execution redundancy.

At run time, when a variable in a test is unbound, the variable is pushed onto the suspension stack. A continuation pointing to this suspended test can also be pushed onto the stack. Upon reaching a `suspend` instruction, the stack is popped, and each unbound variable is hooked to the goal. As an optimization, the associated continuations can be attached to each hook, so that resumption continues precisely at the relevant test. Using this optimization makes it possible to restart a resumed goal at exactly the point where the suspension took place, instead of restarting at the root of the graph, thus having to perform previous tests again.

As previously mentioned, residuals with single clauses terminate as code leaves. If no code sharing is possible for residual $\Re = \{< C_i, G >\}$, we generate `execute(i,G)`. The exact semantics for this instruction is:

```
if (suspension_stack_check == ok)
    then G; go(Ci_B);
    else go(Ci_H);
```

where `Ci_B` points to the body of clause i, and `Ci_H` points to head of clause i, as discussed before. The test `suspension_stack_check` tests if there are any variables pushed on the suspension stack which are relevant to clause i, i.e., which occur in the canonical form of clause i.

This scheme can be implemented efficiently with *bit vectors*. At compile time, when expanding each clause into its canonical form, the compiler generates a bit vector for each clause, where for each variable which occurs in the canonical form, the corresponding bit in the vector is set. For example, if the head of clause i is $f(g(X,Y))$, the canonical form would be $f(Z_1) :- Z_1 = g(Z_{1,1}, Z_{1,2})$ and the bit vector would be set to $B_i = [Z_1, Z_{1,1}, Z_{1,2}] = [100]$.

The original residual is extended to be a triplet, where the third part contains the bit vector B_i, as defined above. When a new residual is calculated, a new bit vector B_i' is calculated with the following rules: for all branches labeled with '?' or '??', the bit vector is unaffected. For all other branches,[8] the bits corresponding to the variables in Γ are zeroed.

[8] This corresponds to those branches for which Γ can be fully evaluated, implying that all variables in Γ are ground.

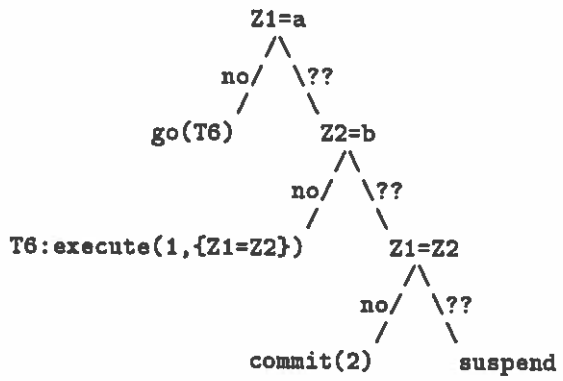


Figure 6: Decision Graph for `f/2` (Representation of Final Code).

When we are down to a leaf with a single residual $< i, G, B >$, the bit vector B is changed with respect to G. If all instances of a variable Z, which appear in the original residual for clause i, still appear in G at the leaf, then we have proven (at compile time!) that Z cannot be on the suspension stack when we reach this leaf. Thus the bit for Z can be reset to 0.

When the entire compile-time bit vector B at a leaf is 0, and no shared nodes (targets of go instructions) appear on the path from the root to the leaf, then we can replace the `execute` instruction with the simpler code sequence "`G; commit(i)`." The semantics for the `commit` instruction is simply: "`go(Ci_B)`." The second condition can be removed by combining bit vectors at shared nodes, although we do not pursue this here.

The final code graph for `f/2` is shown in Figure 6. This graph represents the linear code generated from the previous compiler code graph. This polished graph need not be explicitly generated, but is implicitly used while generating the actual code. We show it here to illustrate the optimizations previously discussed. For example, only one of the `execute` instructions in Figure 5 can be converted to `commit` using the method outlined before.

At run time, a single bit vector *SS* is used to represent the suspension stack. Each time a variable is pushed on the suspension stack, the corresponding bit is set in *SS*. The `suspension_stack_check` is then reduced to testing the logical AND of two bit vectors:

```
if ((SS AND Bi) == 0)
    then G; go(Ci_B);
    else go(Ci_H);
```

6 Code Space Evaluation

This section presents empirical measurements of code size generated by the previous algorithms. As an example of the benchmarks, the following `cell/10` procedure is taken from an active-constraints program for solving N-Queens:

```
:- dontknow cell/10.

cell(I, J, J, I, I, I, begin, end, begin, end) :- C1.
cell(_, _, _, _, _, _, Hc, Hc, Vc, Vc)         :- C2.
```

In the above clauses, `C1` and `C2` represent unspecified clause bodies. Procedure `cell/10` can be translated into a don't-care procedure with mutually exclusive claues, as first indicated by R. Bahgat [2].

```
cell(I,J,H,V,L,R,Left1,Right1,Left2,Right2) :- H\==J |
    Left1=Right1, Left2=Right2, C2.
cell(I,J,H,V,L,R,Left1,Right1,Left2,Right2) :- V\==I |
    Left1=Right1, Left2=Right2, C2.
cell(I,J,H,V,L,R,Left1,Right1,Left2,Right2) :- L\==I |
    Left1=Right1, Left2=Right2, C2.
cell(I,J,H,V,L,R,Left1,Right1,Left2,Right2) :- R\==I |
    Left1=Right1, Left2=Right2, C2.
cell(I,J,H,V,L,R,Left1,Right1,Left2,Right2) :- V\==L |
    Left1=Right1, Left2=Right2, C2.
cell(I,J,H,V,L,R,Left1,Right1,Left2,Right2) :- L\==R |
    Left1=Right1, Left2=Right2, C2.
cell(I,J,H,V,L,R,Left1,Right1,Left2,Right2) :- R\==V |
    Left1=Right1, Left2=Right2, C2.
cell(I,J,H,V,L,R,begin,end,Left,Right):-
    H=J,V=I,L=I,R=I,Left=begin,Right=end, C1.
cell(I,J,H,V,L,R,Left,Right,begin,end):-
    H=J,V=I,L=I,R=I,Left=begin,Right=end, C1.
```

Note the additional body goals added to force unification once the procedure has been found to be determinate. Although the number of clauses has increased as a function of the number of constraints, the resulting clauses each have only a small number of constraints.

Table 1 gives measurements of compiled benchmark procedures (see Appendix A for source listings and Appendix B for the actual code graphs). The size measurement is written as number of tests and complex code leaves + number of continuations (i.e., `fail`, `suspend`, `go`, and `commit`). Three program sizes are given. **Kliger** represents the code size if compiled as a don't-care procedure. Such a program does not have the semantics of the next two don't-know versions, and is given simply as a baseline. **Bahgat** represents the code size if first hand-translated into a don't-care procedure with equivalent don't-know semantics, and then compiled. **K&T** represents directly compiling the don't-know procedures with our method.

procedure name	# clauses	Kliger size	Bahgat size	K&T size
f/2	2	3+4=7	4+6=10	3+4=7
a/3	4	6+10=16	4+7=11	9+10=19
cell/5	2	3+5=8	5+7=12	5+4=9
cell/10	2	13+14=27	11+12=23	16+11=27
omerge/3	4	4+7=11	4+7=11	6+10=16

Table 1: Empirical Measurements of Code Size.

Consider `f/2` as a simple example. When converted into a don't-care procedure and then translated into a decision graph, there are four trivial code leaves, i.e., simple commits. We count these as control transfers. Directly compiling `f/2` as a don't-know procedure gives only two code leaves. Code leaves in don't-know procedures are considered complex `execute` sequences that check the suspension stack bit vector, unless it can be proven at compile time that one of the vectors is zero. In addition, if committing, extra guards must be executed (these guards have been implicitly added to the new don't-care clauses in the translated version).

Table 1 indicates that compiling into don't-know semantics requires a larger tree than for don't-care semantics because of the added power of rapidly detecting determinism and forcing execution of unbound constraints. Depending on the procedure, the relative code size varies between our method and hand-translation. Although these examples are rather small, they allow us to determine the causes of why our method sometimes does not achieve the size of the hand-translation:

- In some cases, hand-translation into don't-care equivalents removes the need for some constraints. The effective action of testing for these constraints is done by body failure in the don't-care equivalent. An example of this are `cell_/5` clauses 3 and 4 given in the Appendix.

- In direct-compilation of don't-know procedures, some code leaves are complex `executes`, whereas in don't-care procedures, all code leaves are trivial `commits`.

- The method outlined generates "neither" branches causing "fast failure," whereas in Kliger's method failure is as slow as possible, propagating through all otherwise branches until the initial suspend continuation is reached. Thus our faster failure has a cost in additional nodes.

To our knowledge, no algorithm has yet been found for Bahgat's method of don't-know into don't-care translation [2]. The complexity of this hand-translation increases

dramatically with the number of clauses. We feel comfortable that our algorithm has approximately the same space complexity, at low compile-time cost.

7 Conclusions

This paper introduced a decision-graph construction algorithm for code generation of determinacy testing in nondeterminate flat concurrent logic programming languages. The code generated is complete over unification, i.e., determinacy is guaranteed to be detected no matter how complex the data structures and shared variables are. Completeness over other domains, such as arithmetic comparison, is the responsibility of a component of the system, the indexer, for which an algorithm has not yet been specified.

Our algorithm is formulated in the context of committed-choice compilation techniques given by Kliger and Shapiro [8]. For simple procedures, the two are shown to have comparable code-size complexity. Thus we have shown that complete determinacy testing over unification need not significantly increase code size. For more complex procedures the don't-know code size can be significantly larger than the don't-care code size. This size increase is due to all the interacting constraints that must be checked to detect determinacy. Comparing our algorithm to an elegant method of hand-translation from don't-know into don't-care code [2], the code size complexity is more equal. Our algorithm is however more general because it does not require that the clauses be mutually exclusive.

8 Acknowledgements

M. Korsloot was supported by a grant from the Delft University of Technology. E. Tick was supported by an NSF Presidential Young Investigator Award.

References

[1] R. Bahgat. The Pandora Abstract Machine. Technical Report DOC 90/1, Imperial College, Department of Computing, January 1990.

[2] R. Bahgat. *Pandora: Non-deterministic Parallel Logic Programming.* PhD thesis, Imperial College, Department of Computing, 1991. Draft.

[3] R. Bahgat and S. Gregory. Pandora: Non-deterministic Parallel Logic Programming. In *Sixth International Conference on Logic Programming*, pages 471–486. Lisbon, MIT Press, June 1989.

[4] V. S. Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. Technical report, University of Bristol, September 1990.

[5] W. Gibson. *Burning Chrome.* Grafton Books, 1988.

[6] S. Haridi and P. Brand. Andorra Prolog—An Integration of Prolog and Committed Choice Languages. In *International Conference on Fifth Generation Computer Systems*, pages 745–754, Tokyo, November 1988. ICOT.

[7] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *International Symposium on Logic Programming*, pages 468–477. San Francisco, IEEE Computer Society, August 1987.

[8] S. Kliger and E. Shapiro. From Decision Trees to Decision Graphs. In *North American Conference on Logic Programming.* Austin, MIT Press, October 1990.

[9] S. Taylor. *Parallel Logic Programming Techniques.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[10] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 140–156. MIT Press, Cambridge MA, 1987.

[11] R. Yang and V. S. Costa. Andorra-I: A System Integrating Dependent And-Parallelism and Or-parallelism. Technical Report TR-90-03, University of Bristol, March 1990.

A Benchmarks: Source Code

```
:- dontknow omerge/3, a/3, f/2, cell/5.

omerge([],Y,Z) :- C1.              % don't-care equivalent is same!
omerge(X,[],Z) :- C2.
omerge([X|Xs],[Y|Ys],Z) :- X <= Y | C3.
omerge([X|Xs],[Y|Ys],Z) :- X >  Y | C4.

a(1,1,1) :- C1.
a(2,1,1) :- C2.
a(2,2,1) :- C3.
a(2,2,2) :- C4.

a_(1,Y,Z) :- Y=1, Z=1, C1.         % don't-care equivalent to a/3
a_(2,1,Z) :- Z=1, C2.
a_(X,2,1) :- X=1, C3.
a_(X,Y,2) :- X=2, Y=2, C4.

f(X,X) :- C1.
f(a,b) :- C2.

f_(X,X) :- C1.                     % don't-care equivalent to f/2
f_(X,Y) :- X \== a | X=Y, C1.
f_(X,Y) :- Y \== b | X=Y, C1.
f_(a,b) :- C2.

cell(on, Val, Val, _, _) :- C1.
cell(off, _, _, Chain, Chain) :- C2.

cell_(on,A,B,_,_) :- A=B, C1.      % don't-care equivalent to cell/5
cell_(off,_,_,C,D) :- C=D, C2.
cell_(X,A,B,C,D) :- C \== D | X=on, A=B, C1.
cell_(X,A,B,C,D) :- A \== B | X=off, C=D, C2.
```

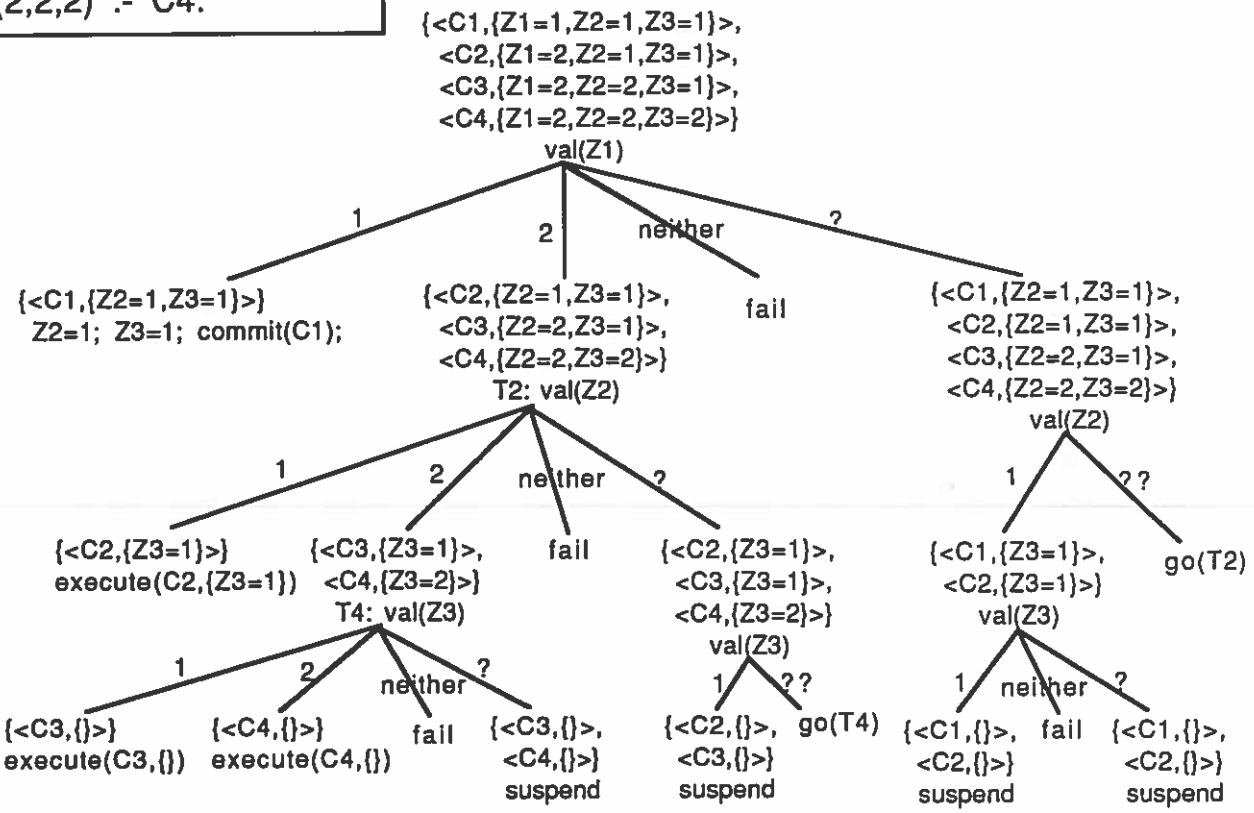
B Benchmarks: Code Graphs

This appendix shows the actual code graphs for the benchmarks, listed in Appendix A. As Baghat's don't care equivalent of `omerge/3` is the same as Kliger's definition, there is only a single don't-care graph for `omerge/3`.

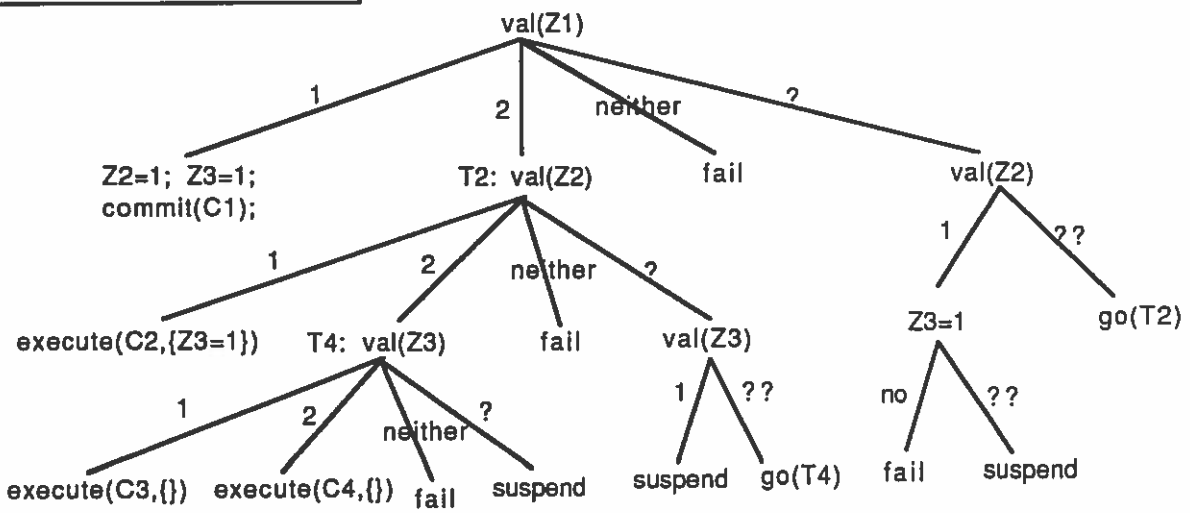
For all other benchmarks, we give the don't-care graph according to Kliger, the don't-know/care graph, using Bahgat's translation method, and the don't-know graph, using our algorithm.

Don't-know procedure:

```
a(1,1,1) :- C1.
a(2,1,1) :- C2.
a(2,2,1) :- C3.
a(2,2,2) :- C4.
```

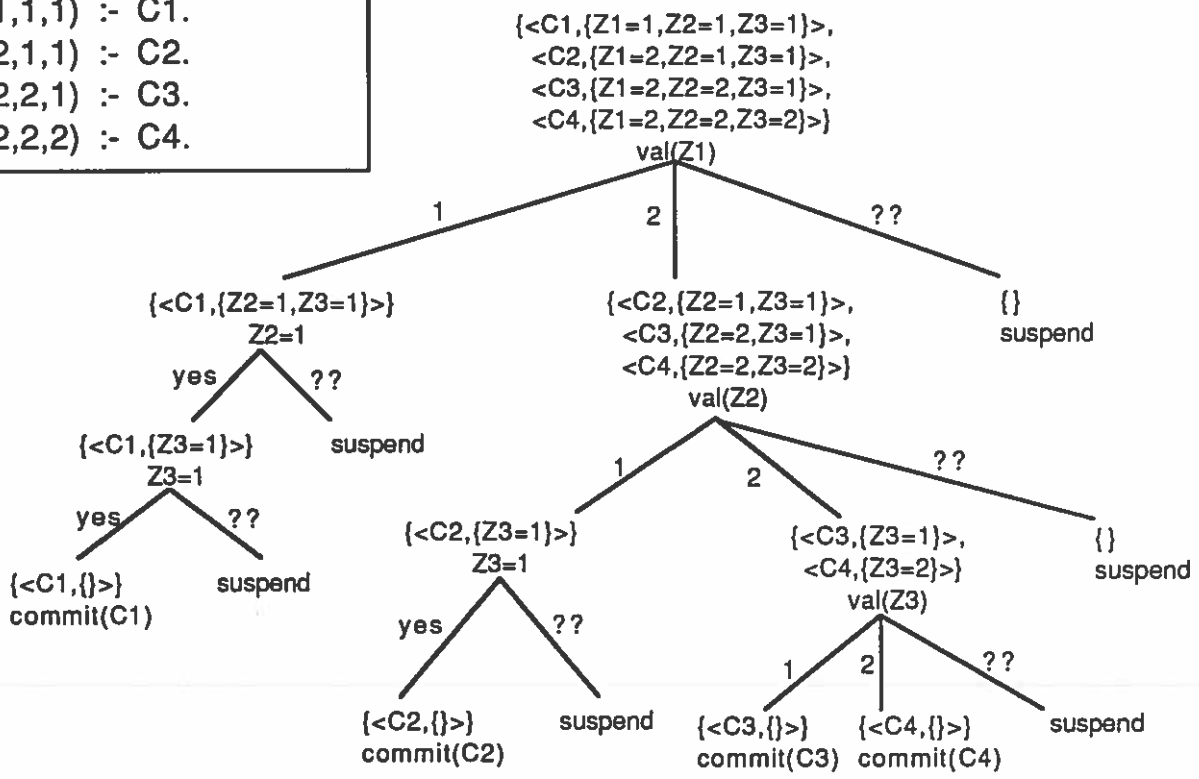


final don't-know graph:

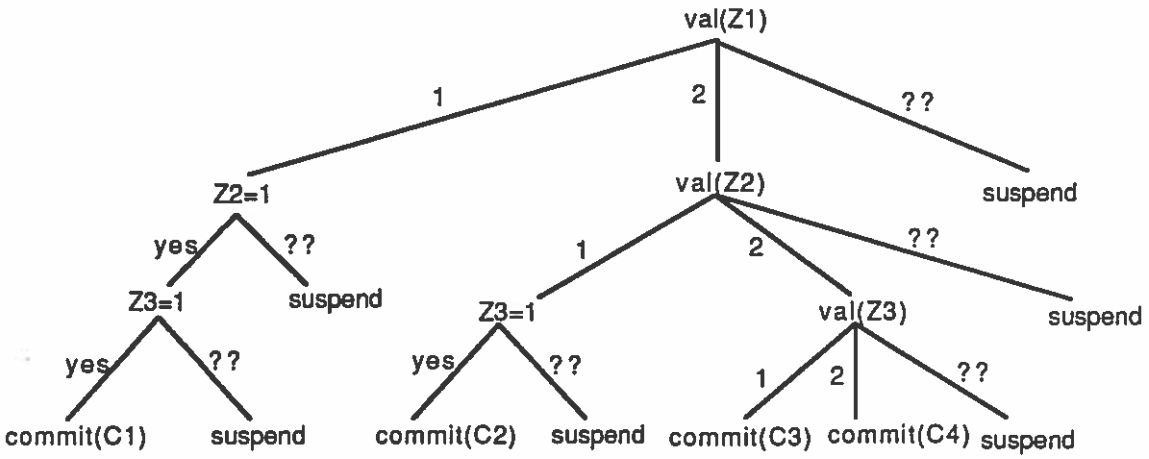


Don't-care procedure:

```
a(1,1,1) :- C1.
a(2,1,1) :- C2.
a(2,2,1) :- C3.
a(2,2,2) :- C4.
```

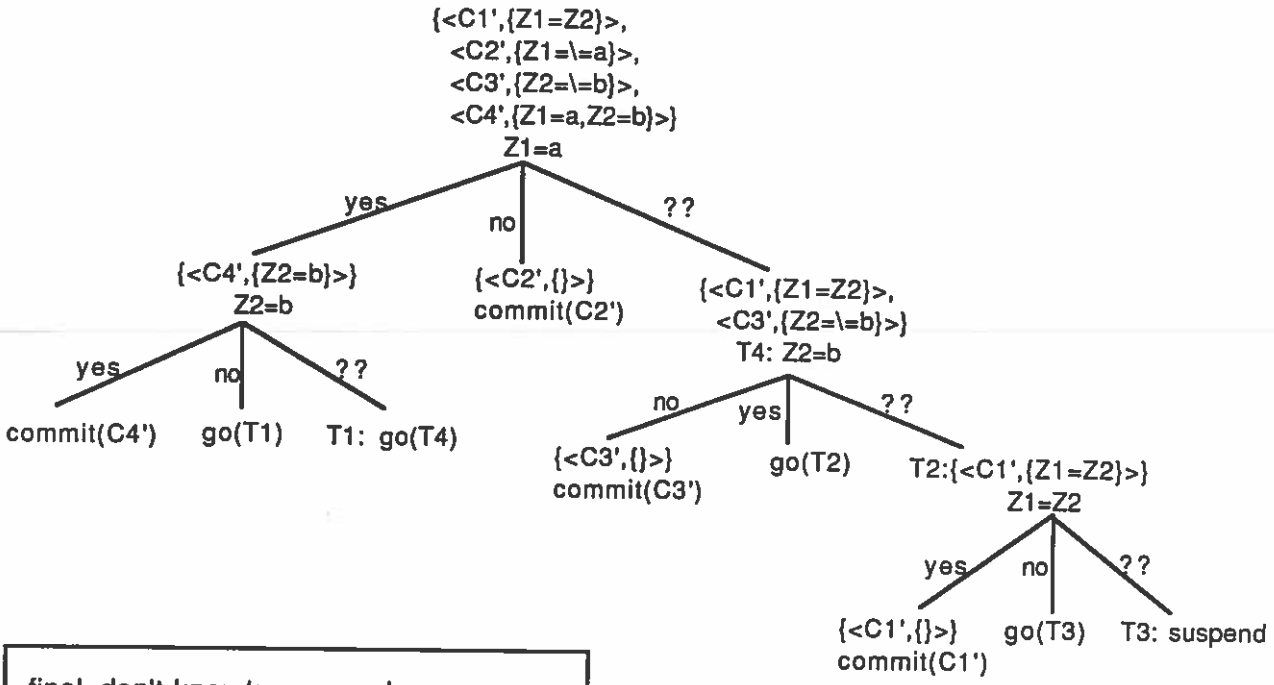


final don't-care graph:

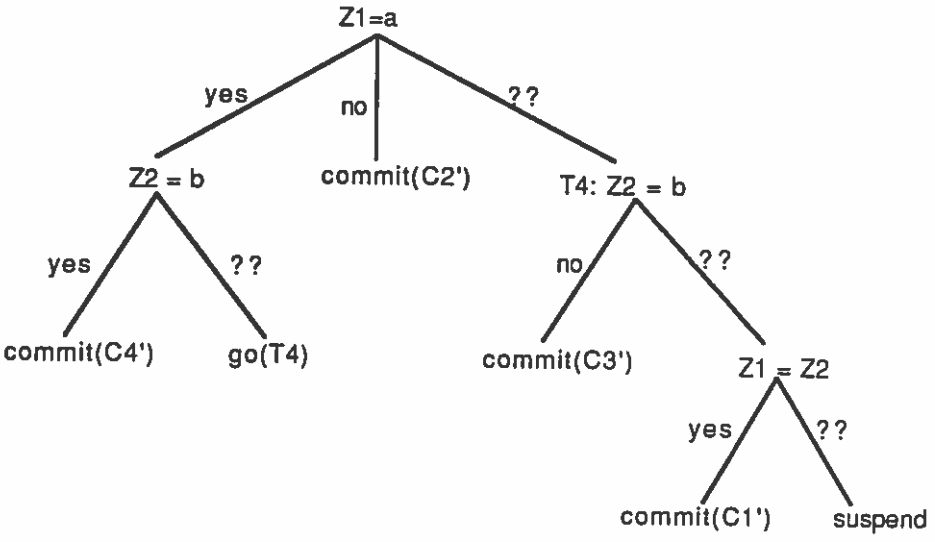


don't-know/care procedure:

```
f_(X,X) :- C1.                    C1'
f_(X,Y) :- X =\= a | X=Y, C1.     C2'
f_(X,Y) :- Y =\= b | X=Y, C1.     C3'
f_(a,b) :- C2.                    C4'
```

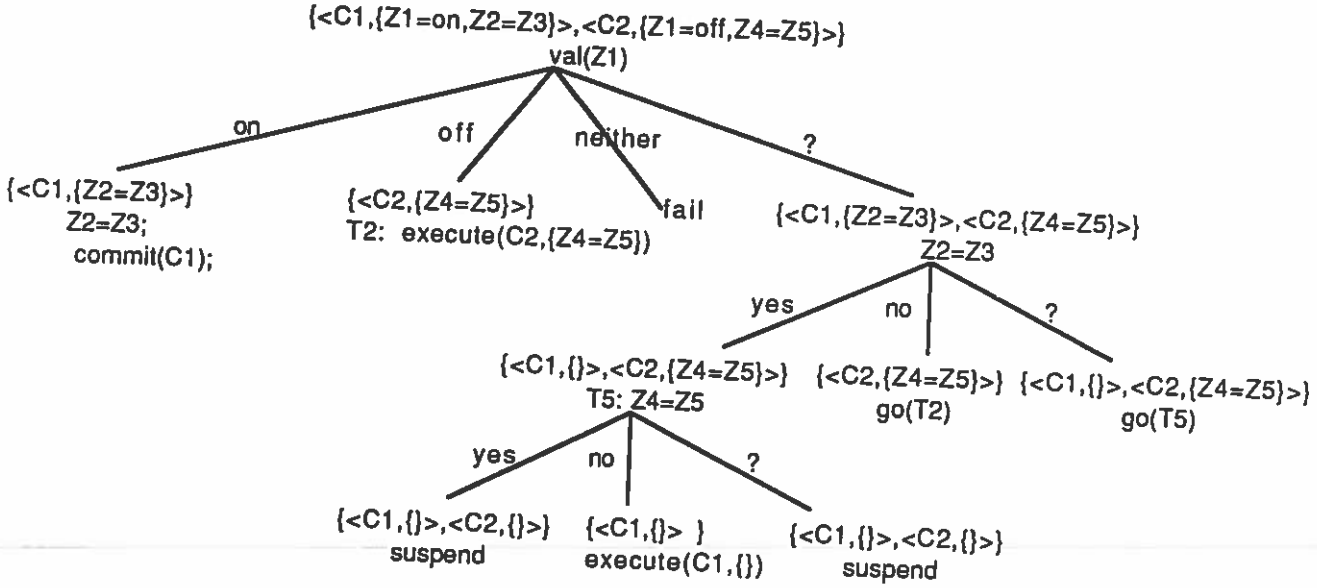


final don't-know/care graph:



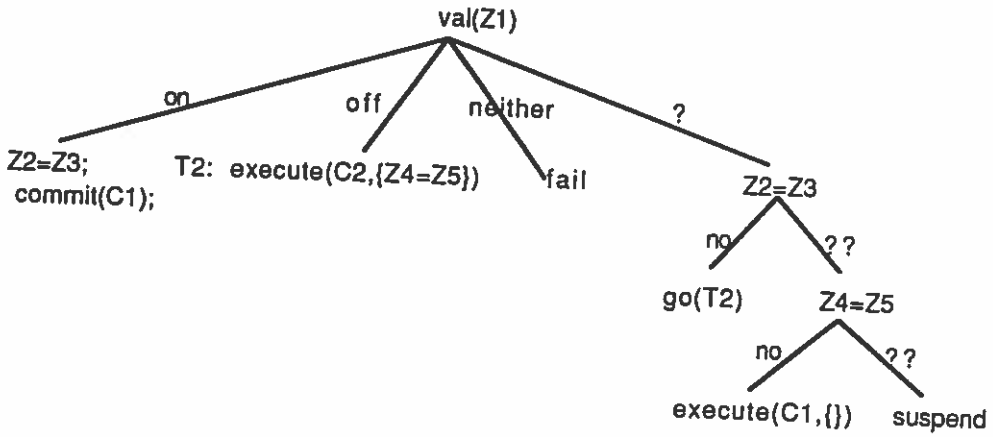
Don't-know procedure:

```
cell(on,X,X,_,_) :- C1.
cell(off,_,_,X,X) :- C2.
```



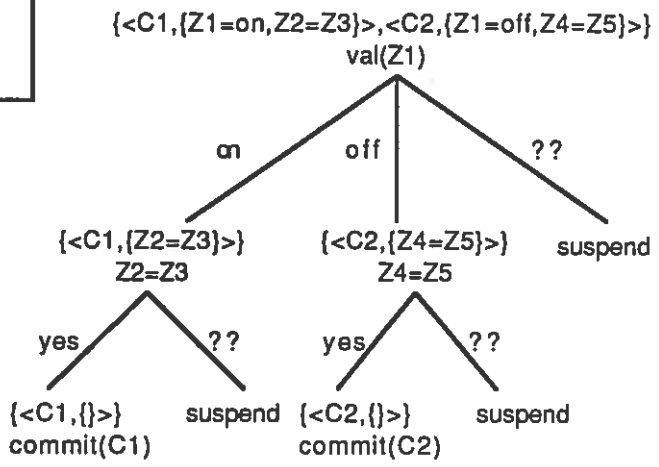
final don't-know graph:

```
cell(on,X,X,_,_) :- C1.
cell(off,_,_,X,X) :- C2.
```



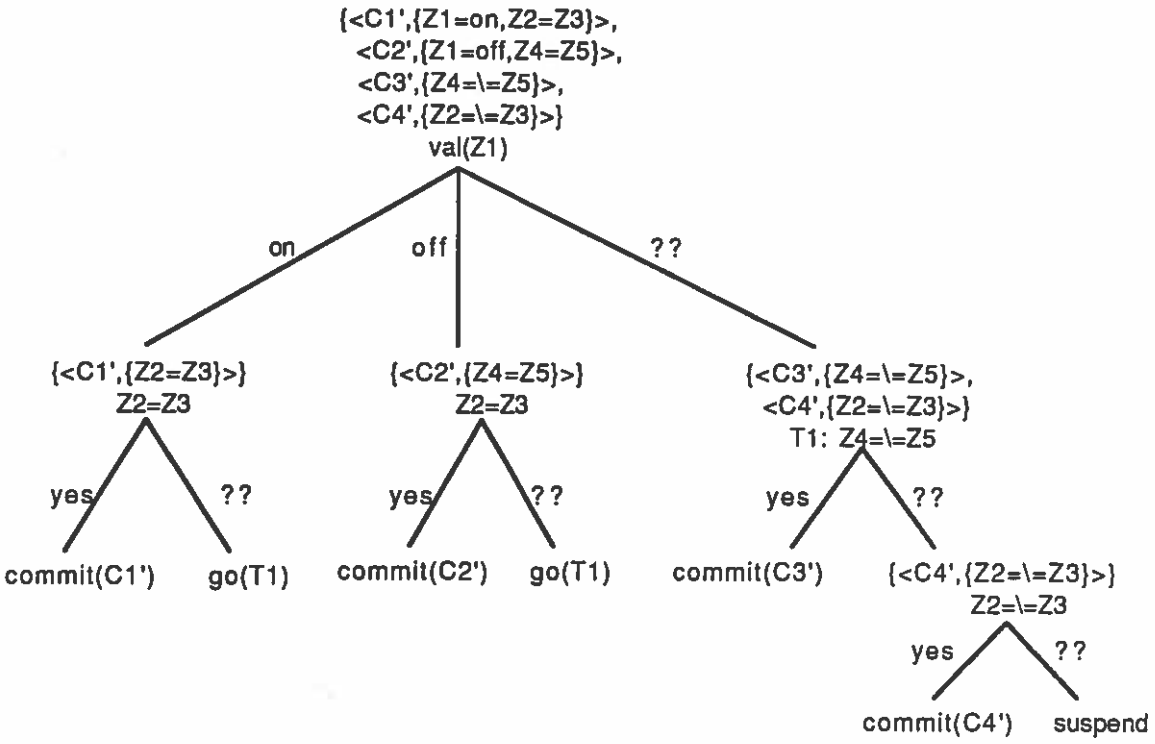
Don't-care procedure:

```
cell(on,X,X,_,_) :- C1.
cell(off,_,_,X,X) :- C2.
```

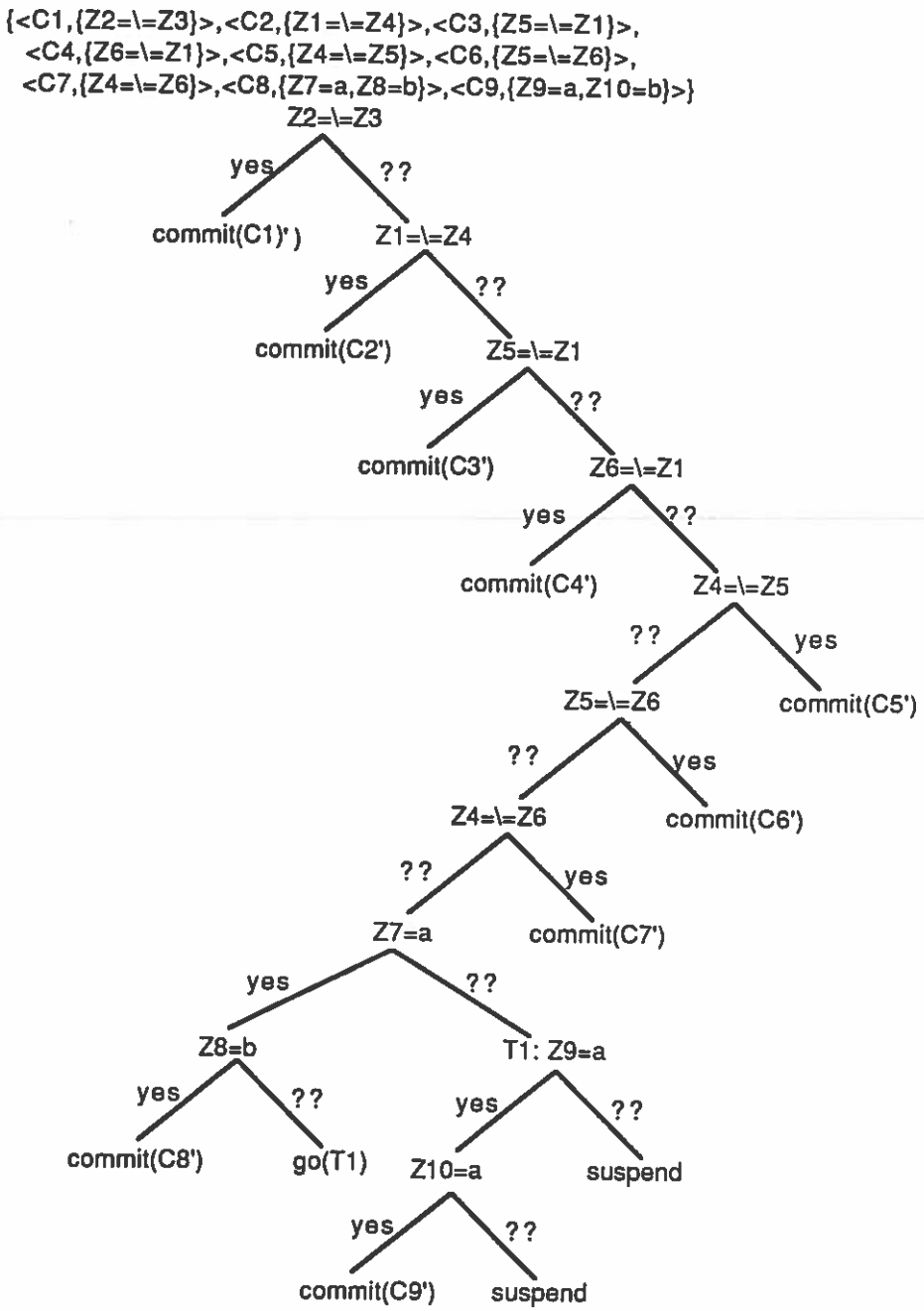


Don't-know/care procedure:

```
cell_(on,A,B,_,_) :- A=B, C1.                          C1'
cell_(off,_,_,C,D) :- C=D, C2.                         C2'
cell_(X,A,B,C,D) :- C=\=D | X=on, A=B, C1.             C3'
cell_(X,A,B,C,D) :- A=\=B | X=off, C=D, C2.            C4'
```

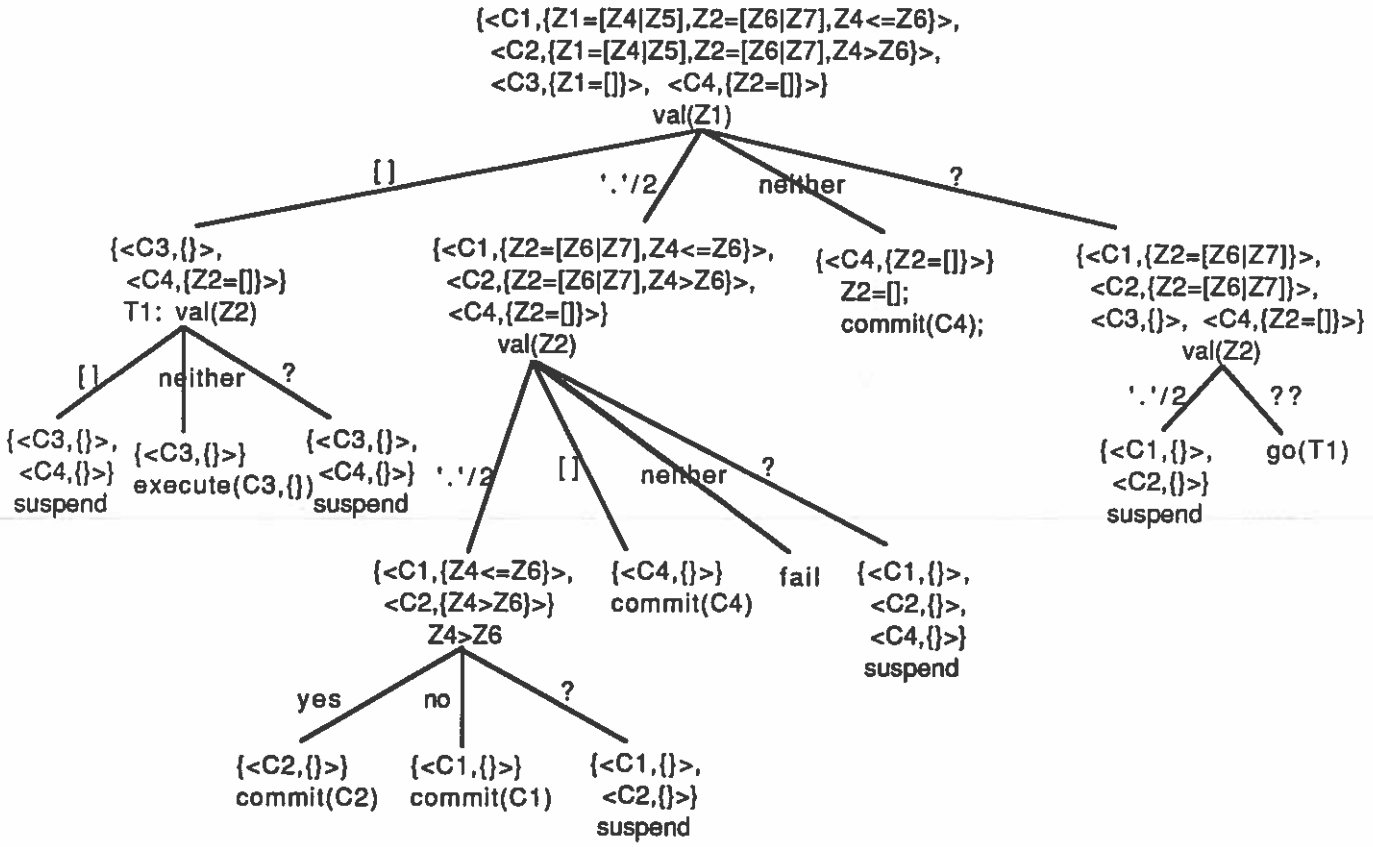


cell/10 don't-know/care procedure:

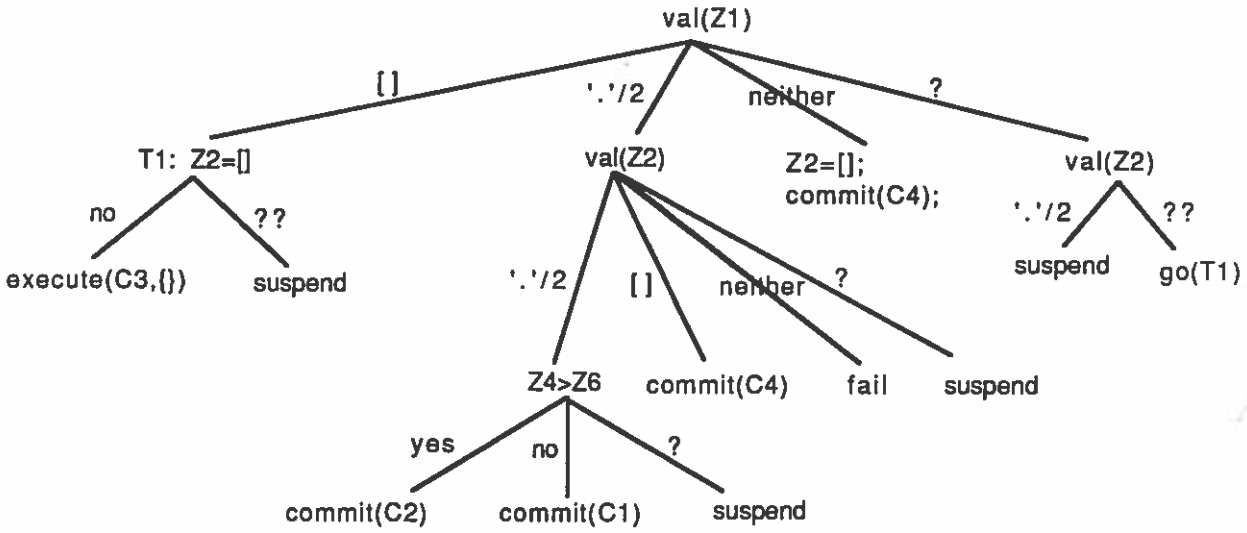


Don't-know procedure:

```
omerge([X|Xs],[Y|Ys],Z) :- X <= Y | C1.
omerge([X|Xs],[Y|Ys],Z) :- X>Y | C2.
omerge([],Y,Z) :- C3.
omerge(X,[],Z) :- C4.
```

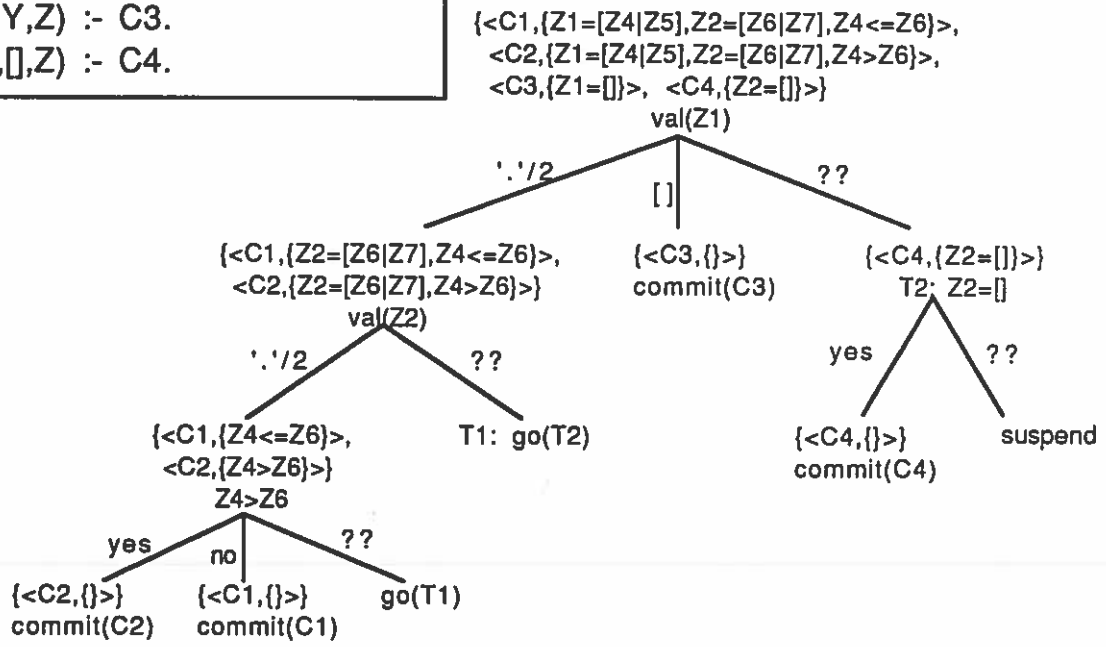


final don't-know graph:

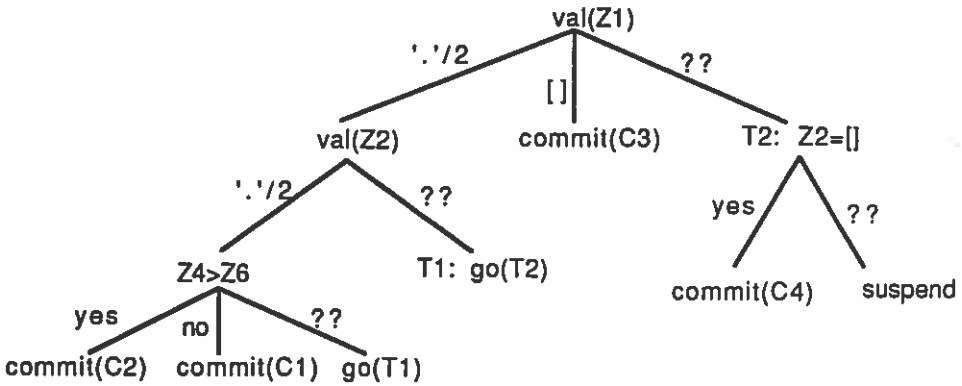


Don't-care procedure:

```
omerge([X|Xs],[Y|Ys],Z) :- X <= Y | C1.
omerge([X|Xs],[Y|Ys],Z) :- X>Y | C2.
omerge([],Y,Z) :- C3.
omerge(X,[],Z) :- C4.
```

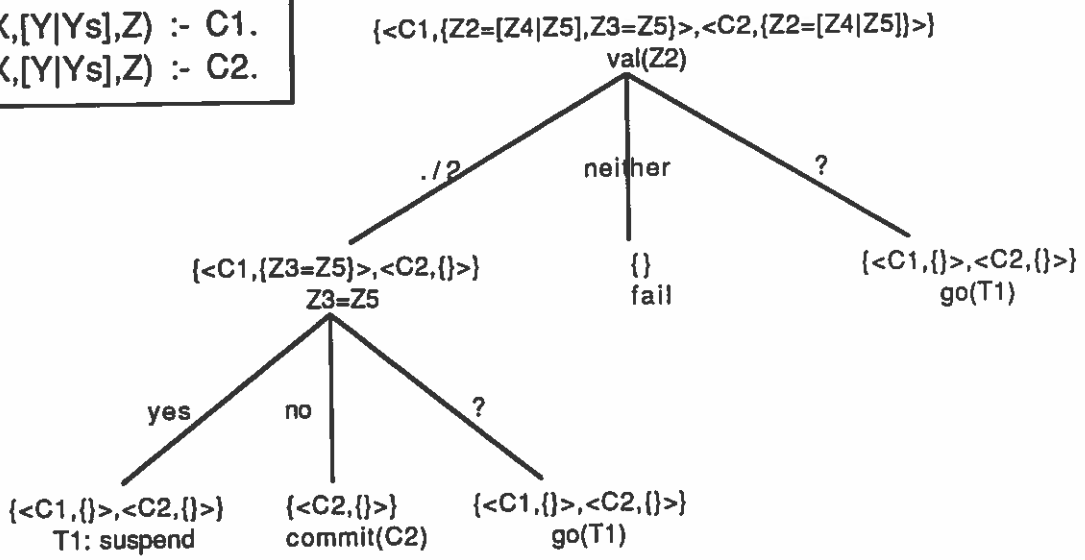


final don't-care graph:



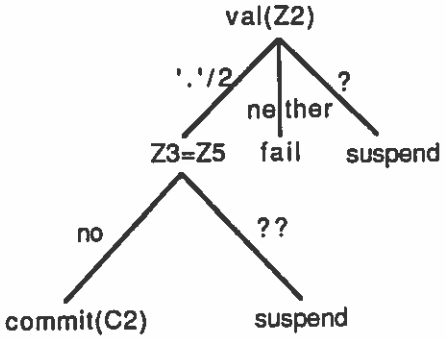
Don't-know procedure:

```
delete(X,[Y|Ys],Z) :- C1.
delete(X,[Y|Ys],Z) :- C2.
```



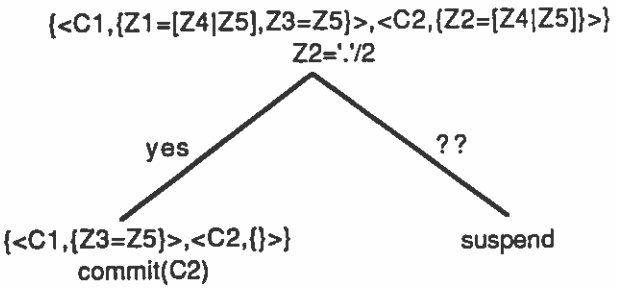
final don't-know graph:

```
delete(X,[Y|Ys],Z) :- C1.
delete(X,[Y|Ys],Z) :- C2.
```



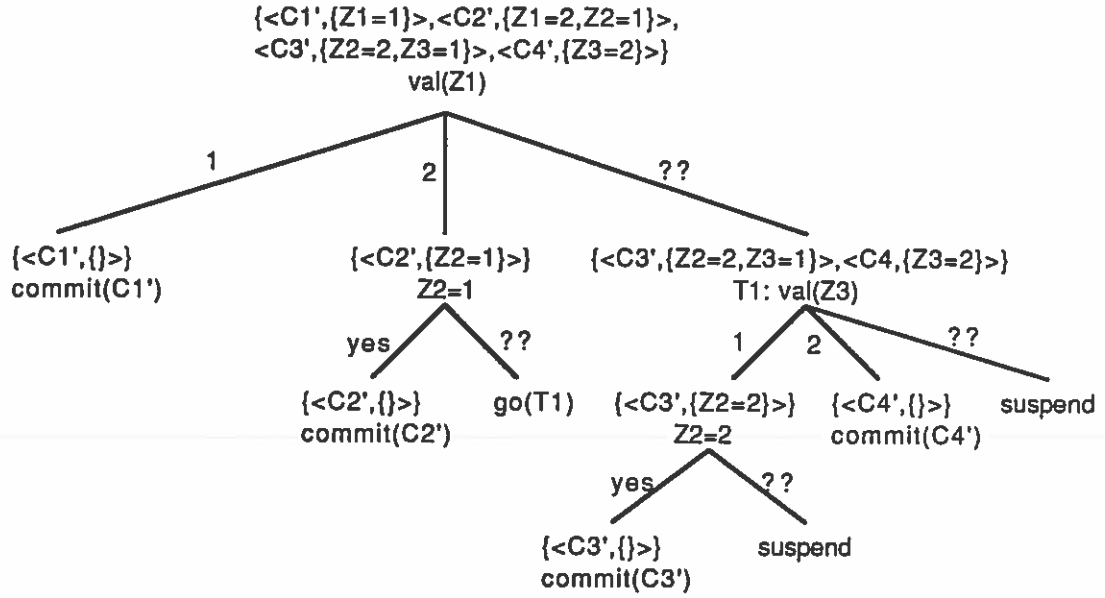
Don't-care procedure:

```
delete(X,[Y|Ys],Z) :- C1.
delete(X,[Y|Ys],Z) :- C2.
```

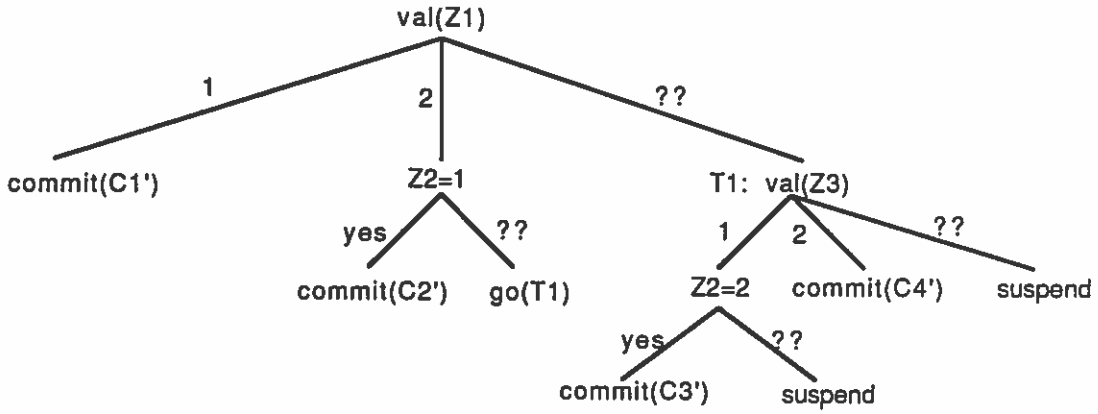


Don't-know/care procedure:

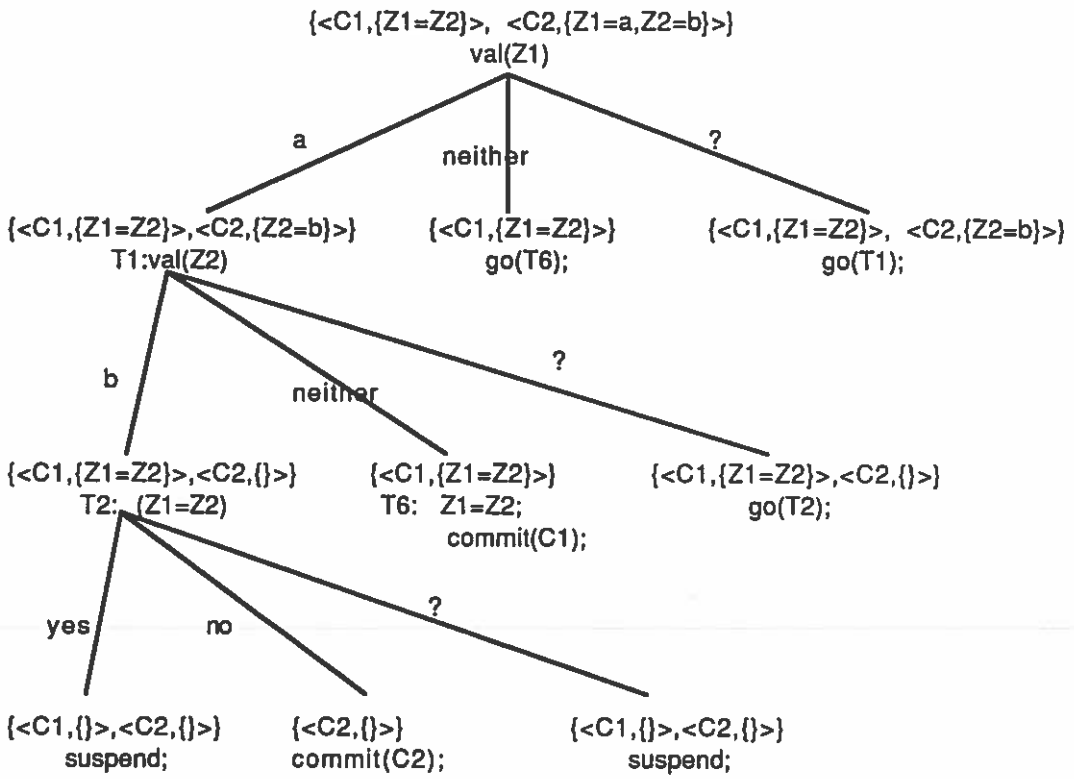
```
a_(1,Y,Z) :- Y=1, Z=1, C1.      C1'
a_(2,1,Z) :- Z=1, C2.           C2'
a_(X,2,1) :- X=1, C3.           C3'
a_(X,Y,2) :-  X=2, Y=2, C4.     C4'
```



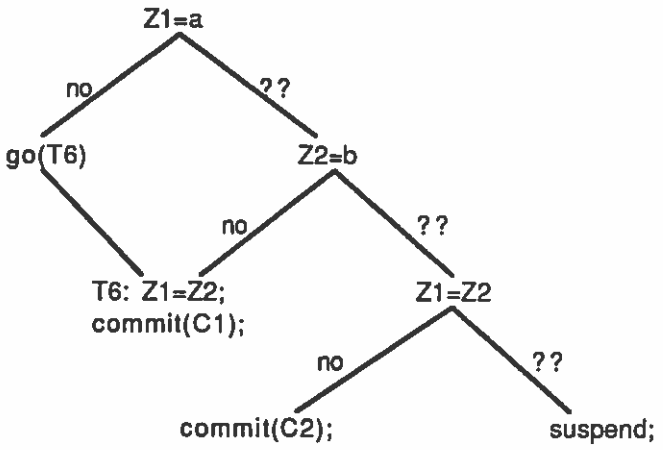
final don't-know/care graph:



don't-know procedure:
f(X,X) :- C1.
f(a,b) :- C2.

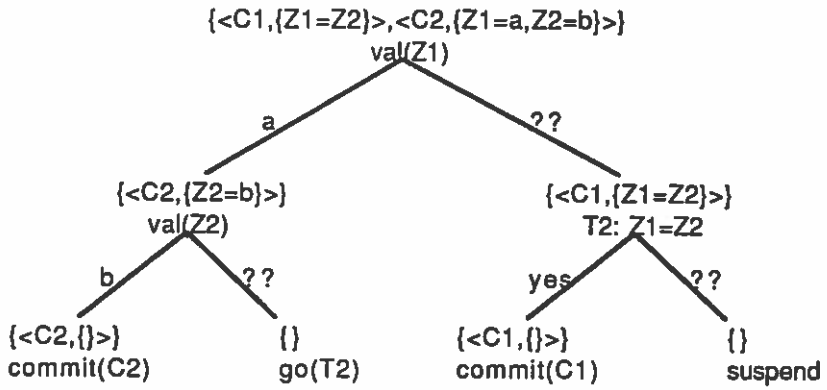


final don't-know graph:
f(X,X) :- C1.
f(a,b) :- C2.

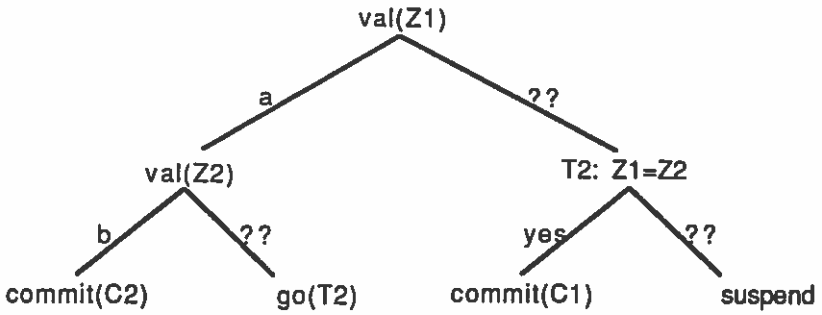


don't care procedure:

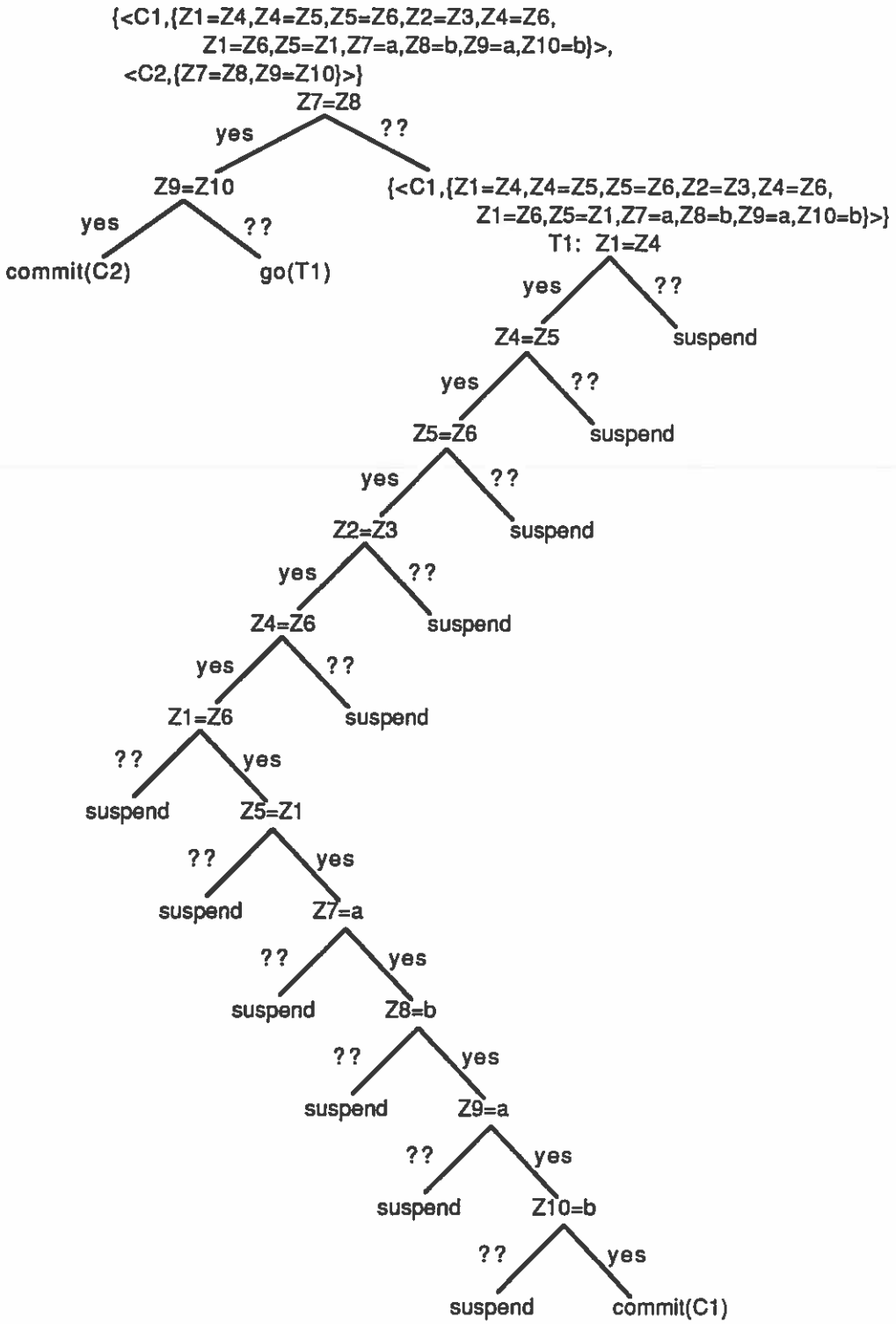
f(X,X) :- C1.
f(a,b) :- C2.



final don't-care graph:
f(X,X) :- C1.
f(a,b) :- C2.



cell/10 don't-care procedure:



cell/10 don't-know procedure: