Image Cover Sheet

CLASSIFICATION	SYSTEM NUMBER
UNCLASSIFIED	497166

TITLE

PROOF LOGS

System Number:

Patron Number:

Requester:

Notes:

DSIS Use only:

Deliver to: FF

Proof Logs

TR-95-5471-03

Sentot Kromodimoeljo
Bill Pase

Release date: JUNE 1995

ORA Canada
267 Richmond Road, Suite 100
Ottawa, Ontario K1Z 6X3
CANADA

Contents

1 Introduction

This report describes proof logs in NEVER as they appear in EVES, version 2.4.2.

The description of proof logs in this report is informal. (A formal description of proof logs with a reduced set of inference rules appears in [5].) In the next section, we present the background behind the proof logging effort. It includes the objectives of the effort from proof checking and proof browsing perpectives. We then introduce the notion of a proof as a sequence of equivalence preserving transformations on a formula. We follow this with a section on the basic framework for proof logs. We then describe the inference rules, grouped in sections according to their classification. Finally, we conclude with an assessment of our work with respect to the objectives.

With the abstract proof description that is already provided by NEVER, the proof browser will have no trouble in displaying abstract proofs. The proof logs will add detailed descriptions of every proof allowing the proof browser to display detailed proofs. However, it is also desirable to be able to view proofs at levels of detail in between the abstract proof and the detailed proof. Enabling displays of proofs at these intermediate levels places special demands upon proof logs. For example, if a proof applies a conditional rewrite rule then the proof log will contain all of the inferences required to prove the condition. It is likely that a user browsing such a proof will want to skip the details for the condition on a first pass of proof browsing. Later, the user may want to come back and examine the details for the condition without regard for the rest of the proof. This requires that the proof browsing tool be able to identify the proof structure and its various levels, and be able to use the identified structure and its levels to control the level of detail presented to the user. The information regarding the proof structure and its levels can be added to the proof log explicitly (for example, in the form of annotations) during the proof, and then later used by the proof browsing tool.

4 Basic Framework

This section describes the basic framework in which proof logs are recorded. Proof logs are generated by the NEVER theorem prover whenever proof logging is enabled (it is disabled by default). These logs are retained in memory as a component of the proof. With the *proof browser*, a user will be able to examine any proof log within the system. The *proof checker* will check any proof log.

We envision the proof browser as an integrated component of NEVER, while the proof checker is expected to be a stand-alone tool. Both the browser and the checker are to utilize the same proof log, though they are free to ignore any part of the log. Thus, proof logs may contain entries specific to either of the tools in addition to entries common to both tools. Since the proof checker will be a stand-alone tool, there will be a file-based version of the proof log that is written by NEVER for use by the tool. Proof logs are not meant to be read by users without the use of the proof browser.

Before the addition of proof logs, NEVER already maintained proof objects. A proof object consisted of a sequence of *proof displays*, each corresponding to an effective proof command. A proof display consists of the proof command name, optional arguments to the command, an abstract description of the transformation performed by the proof command, and the result of the transformation. Proof logging simply augments proof displays by adding detailed inferences performed by the proof command.

When proof logging is enabled, NEVER records a proof log segment for every successful proof command (a proof command that changes the current formula). The proof log segment is added to the proof display and consists of a sequence of inferences that transform the formula from before the command to the formula that is the result of the command. The proof log for the entire proof is simply the concatenation of the proof log segments. For a completed proof, the proof log describes a transformation from the theorem to `(true)`. For a partial proof, the proof log describes a transformation from the conjecture to some intermediate formula. In either case, the proof log can be handled by the proof browser and by the proof checker.

For simplicity, we view the effect of the `(CASES)` command as zooming the display to a case (a subexpression) after possibly some propositional and quantifier manipulation. The zooming to a subexpression in itself is not an inference since it does not change the formula. Subexpression indexing is with respect to the entire formula, which is not necessarily the displayed formula.

The following example describes a simple proof and the corresponding proof log generated by NEVER. First, we introduce two function stubs and a rewrite rule.

```
(function-stub p (x))
(function-stub q (x))

(rule p-is-q (x) (= (p x) (q x)))
```

Now we perform a simple proof of a proposition involving `P` and `Q`. The proof consists of the two commands: `(SIMPLIFY)` and `(REWRITE)`. The first eliminates the `(P X)` case of the theorem, while the second eliminates the `(Q X)` by applying the rewrite rule `P-IS-Q`. This is the level of detail that is normally displayed by NEVER for successful commands. Note that although the proof summary reports the use of some axioms as forward chains and assumptions, in this example, these axioms do not play any role in the proof.

```
> (try (implies (p x) (and (p x) (q x))))

Beginning proof of ...
(IMPLIES (P X)
         (AND (P X)
              (Q X)))
> (simplify)
```

```
(LOOKUP-TRUE (2 2 1))
(IF-TRUE (2 2))
(IF-EQUAL (2))
(REMOVE-UNIVERSAL ())
(REMOVE-BOOL-COERCION ())
```

The proof log segment for a command records the inferences that the command performed to transform the formula before the command to the formula after the command. It does not include the inferences that occurred during subproofs that failed, such as during subgoaling to prove rule conditions. Proof logs do not support analyses of failed proofs.

In addition, a proof log does not necessarily reflect the way the theorem prover arrived at the proof. We only require that a proof log segment describes a way to perform the transformation performed by the corresponding proof command.

4.1 Indexing

As mentioned earlier, each proof log entry consists of an inference name, followed by an index to the subexpression on which the inference is to apply, followed by parameters if required. The index to the subexpression is a sequence of natural numbers that indicates the position of the subexpression relative to the entire formula. An empty sequence indicates that the subexpression is the entire formula. Each natural number in the sequence selects a component of an *s-expression* at some level. The selection is zero-origin with 0 meaning the first component, 1 meaning the second component, etc.

Thus, given that the entire formula is:

```
(F (G A B) (C D F))
```

the index `(2 1)` selects the subexpression `D` (the 2 selects the third component of the top-level s-expression, giving `(C D F)`, and the 1 selects the second component of `(C D F)`, giving `D`).

Since the language Verdi is first-order, the first component of an s-expression is never selected. Thus, 0 never appears in an index. For quantification, we can only select the third component, which is the quantified expression.

4.2 Context

The simplification process in NEVER may transform a subexpression of the current formula using information provided by the context. For example, the `(SIMPLIFY)` command above involves the transformation of:

```
(ALL (X)
  (IF (P X)
      (IF (P X)
          (IF (IF (Q X) (TRUE) (FALSE))
              (TRUE)
              (FALSE))
          (IF (FALSE) (TRUE) (FALSE)))
      (TRUE)))
```

to

```
(ALL (X)
  (IF (P X)
      (IF (TRUE)
```

```
PATTERN
```

to

```
(IF CONDITION RESULT PATTERN)
```

The subproof simply becomes the transformation of the subexpression `CONDITION` to `(TRUE)` giving:

```
(IF (TRUE) RESULT PATTERN)
```

which can be transformed to

```
RESULT
```

using the inference rule *IF-TRUE*.

4.4 Boolean Contexts

Some of the inference rules for NEVER make use of the notion of *boolean context*.[1] A boolean context is a position in a formula where any subexpression occupying that position can be replaced by its coercion to boolean, and the resulting formula is equivalent to the original formula. For an expression

```
EXPR
```

the coercion to boolean of that expression is

```
(IF EXPR (TRUE) (FALSE)).
```

The rules for boolean context are as follows:

- The test position of an *if* expression is a boolean context.
- The quantified expression position of a quantification is a boolean context.
- The argument positions of boolean connectives are boolean contexts. (This follows from the definitions of the boolean connectives in terms of *if* expressions.)
- For an *if* expression `(IF T L R)` that is in a boolean context, the positions for `L` and `R` are boolean contexts.

In a boolean context, boolean coercion can be removed (i.e., `(IF EXPR (TRUE) (FALSE))` can be replaced by `EXPR`).

4.5 Accessibility

In the development paradigm of EVES, starting with the initial theory, one extends the theory using declarations. Normally one would discharge a proof obligation for a declaration immediately after entering the declaration. However, EVES allows the proof to be deferred and free movement among proofs are allowed (using the `(TRY)` command). A notion of accessibility is required since otherwise one can use axioms in a circular fashion.

In EVES, the notion of accessibility is rather simple. At the top level, it is simply declaration before use. When discharging a proof obligation for a declaration, one can only use axioms added by earlier (textually) declarations. The current theory for that declaration is simply the initial theory extended by declarations up to, but excluding, itself.[2]

[1] We are considering simplifying the inference rules to not require the notion of boolean context.

[2] For a recursive declaration, the vocabulary extension that results from the declaration is available to a proof obligation for that declaration. However, this is irrelevant to accessibility which deals strictly with axioms.

4.6 Checking Tools

The Verdi source defined in the previous subsection provides a common basis for all checking tools.[3] A proof checker would also need proof logs in conjunction with the Verdi source. NEVER generates separate source and log files. The log file contains a sequence of proof logs, each corresponding to a proof obligation in the source file, in the same order of appearance.

A stand-alone proof checker inputs a source file and the corresponding log file. As it checks the discharging of proof obligations, the proof checker would need to maintain a database of vocabulary and axioms representing the current theory, and determine accessibility of axioms. For simplicity, the proof checker ought to be a batch-oriented program. Accessibility can then simply become a question of whether an axiom is in the database.

Proof obligation checking, theory extension checking, and library load checking tools need only examine parts of a source file. A proof obligation checking tool need only examine the `decl` and corresponding `(obligation*)` of regular extensions. A theory extension checking tool need only examine the `decl` and corresponding `(theory_extension*)` of regular extensions. A library load checking tool need only examine the `(LOAD identifier)` and corresponding `(load_extension*)` of regular and library load extensions. None of these tools needs to examine a log file.

[3] One is, of course, free to implement a single tool to do all the checking.

5.2 IF-FALSE

PATTERN: (IF (FALSE) x y)

REPLACEMENT: y

5.3 SPLIT

PARAMETER: a

PATTERN: b

REPLACEMENT: (IF a b b)

5.4 CASE-ANALYSIS

PARAMETER: n

PATTERN: (f b1 (IF a b2 c2) ...)

REPLACEMENT: (IF a (f b1 b2 ...) (f b1 c2 ...))

where n is 2 in the example.

5.5 LOOK-UP-TRUE

PATTERN: e

CONDITION: e is in the set of assumptions part of the context.

REPLACEMENT: (TRUE)

5.6 LOOK-UP-FALSE

PATTERN: e

CONDITIONS: e is in the set of denials part of the context.
e is a boolean expression or e is in a boolean context.

REPLACEMENT: (FALSE)

5.7 LOOK-UP-EQUAL

PARAMETER: e1

PATTERN: e

CONDITION: (= e e1) or (= e1 e) is in the set of assumptions part of the context.

REPLACEMENT: e1

5.12 IF-EQUAL

PATTERN: (IF a b b)

REPLACEMENT: b

This is simply the *SPLIT* inference rule applied in reverse.

5.13 IF-NEGATE-TEST

PATTERN: (IF (IF a (FALSE) (TRUE)) b c)

REPLACEMENT: (IF a c b)

This inference rule can be expanded in terms of *CASE-ANALYSIS*, *IF-TRUE* and *IF-FALSE* inference rules.

5.14 REMOVE-BOOL-COERCION

PATTERN: (IF a (TRUE) (FALSE))

CONDITION: a is boolean or the pattern is in a boolean context.

REPLACEMENT: a

If the pattern is in a boolean context, then the inference rule follows from the definition of boolean context.

If a is boolean, then the inference rule can be derived as follows. Starting with

(IF a (TRUE) (FALSE)),

since a is a boolean expression, we can transform the above to

```
(IF (IF (= a (TRUE)) (TRUE) (= a (FALSE)))
    (IF a (TRUE) (FALSE))
    (TRUE)).
```

The above transformation does not involve a special inference rule for boolean expressions but, instead, involves the use of axioms from the current theory (see Section 7). Using *CASE-ANALYSIS* we get

```
(IF (= a (TRUE))
    (IF (TRUE) (IF a (TRUE) (FALSE)) (TRUE))
    (IF (= a (FALSE)) (IF a (TRUE) (FALSE)) (TRUE))).
```

We then use *LOOK-UP-EQUAL*, *IF-TRUE* and *LOOK-UP-EQUAL* again to get

```
(IF (= a (TRUE))
    (IF (TRUE) a (TRUE))
    (IF (= a (FALSE)) (IF a (TRUE) (FALSE)) (TRUE))).
```

Use *LOOK-UP-EQUAL*, *IF-FALSE* and *LOOK-UP-EQUAL* again to get

```
(IF (= a (TRUE))
    (IF (TRUE) a (TRUE))
    (IF (= a (FALSE)) a (TRUE))).
```

where n is 3 in the example. This inference rule follows from the basic propositional inference rules, the syntactic abbreviation rules in Section 11, and the definition of OR.

5.19 CASES-IF

```
PATTERN: (IF a b c)

CONDITION: pattern is in boolean context or
           b and c are boolean expressions.

REPLACEMENT: (AND (IMPLIES x y) (IMPLIES (NOT x) z))
```

This inference rule follows from the definitions of the boolean connectives, the basic propositional inference rules, the definition of boolean context, and the definition of boolean expressions.

5.20 REMOVE-BOOL-COERCION-FROM-CONJUNCT

```
PARAMETER: n

PATTERN: (AND a b (IF c (TRUE) (FALSE)) ...)

REPLACEMENT: (AND a b c ...)
```

where n is 3 in the example. This inference rule follows from the syntactic abbreviation rules in Section 11, the basic propositional rules, and the definition of AND.

5.21 NOT-TRUE

```
PATTERN: (NOT (TRUE))

REPLACEMENT: (FALSE)
```

This inference rule follows from the definition of NOT and the *IF-TRUE* inference rule.

5.22 NOT-FALSE

```
PATTERN: (NOT (FALSE))

REPLACEMENT: (TRUE)
```

This inference rule follows from the definition of NOT and the *IF-FALSE* inference rule.

5.23 NOT-NOT

```
PATTERN: (NOT (NOT a))

REPLACEMENT: (IF a (TRUE) (FALSE))
```

This inference rule follows from the definition of NOT, and the *CASE-ANALYSIS*, *IF-FALSE* and *IF-TRUE* inference rules.

5.29 DISTRIBUTE-AND-OR-LEFT

```
PATTERN: (AND (OR a b) c)

REPLACEMENT: (OR (AND a c) (AND b c))
```

This inference rule follows from the definition of the boolean connectives and the basic propositional inference rules.

5.30 DISTRIBUTE-AND-OR-RIGHT

```
PATTERN: (AND a (OR b c))

REPLACEMENT: (OR (AND a b) (AND a c))
```

This inference rule follows from the definition of the boolean connectives and the basic propositional inference rules.

5.31 DISTRIBUTE-OR-AND-LEFT

```
PATTERN: (OR (AND a b) c)

REPLACEMENT: (AND (OR a c) (OR b c))
```

This inference rule follows from the definition of the boolean connectives and the basic propositional inference rules.

5.32 DISTRIBUTE-OR-AND-RIGHT

```
PATTERN: (OR a (AND b c))

REPLACEMENT: (AND (OR a b) (OR a c))
```

This inference rule follows from the definition of the boolean connectives and the basic propositional inference rules.

5.33 NOT-TO-IF

```
PATTERN: (NOT a)

REPLACEMENT: (IF a (FALSE) (TRUE))
```

This is the definition of NOT.

5.34 AND-TO-IF

```
PATTERN: (AND a b)

REPLACEMENT: (IF a (IF b (TRUE) (FALSE)) (FALSE))
```

This is the definition of AND.

Use *LOOK-UP-EQUAL* to get

```
(IF (= (TRUE) a)
    (IF a (TRUE) (FALSE))
    (IF a (TRUE) (= (TRUE) a))).
```

Use *LOOK-UP-FALSE* ((= (TRUE) a) is boolean) to get

```
(IF (= (TRUE) a)
    (IF a (TRUE) (FALSE))
    (IF a (TRUE) (FALSE))).
```

Finally, use *SPLIT* in reverse to get

```
(IF a (TRUE) (FALSE)).
```

5.38 =-TRUE-RIGHT-TO-IF

```
PATTERN: (= a (TRUE))

REPLACEMENT: (IF a (TRUE) (FALSE))
```

The derivation of this inference rule is similar to *=-TRUE-LEFT-TO-IF*.

5.39 =-FALSE-LEFT-TO-IF

```
PATTERN: (= (FALSE) a)

CONDITION: a is a boolean expression.

REPLACEMENT: (IF a (FALSE) (TRUE))
```

The derivation of this inference rule makes use of the fact that a is a boolean expression.

5.40 =-FALSE-RIGHT-TO-IF

```
PATTERN: (= a (FALSE))

CONDITION: a is a boolean expression.

REPLACEMENT: (IF a (FALSE) (TRUE))
```

The derivation of this inference rule makes use of the fact that a is a boolean expression.

5.41 IF-TO-NOT

```
PATTERN: (IF a (FALSE) (TRUE))

REPLACEMENT: (NOT a)
```

This inference rule follows from the definition of NOT.

5.48 BOOL-IF-TO-AND-NOT

```
PATTERN: (IF a (FALSE) b)

CONDITION: pattern is in a boolean context or
           b is a boolean expression.

REPLACEMENT: (AND (NOT a) b)
```

This inference rule follows from *IF-TO-AND-NOT* (first coerce b to boolean).

5.49 BOOL-IF-TO-IMPLIES-NOT

```
PATTERN: (IF a (TRUE) b)

CONDITION: pattern is in a boolean context or
           b is a boolean expression.

REPLACEMENT: (IMPLIES (NOT a) b)
```

This inference rule follows from the definitions of boolean context, boolean expressions, and the boolean connectives, and the basic propositional inference rules.

5.50 IF-TO-OR

```
PATTERN: (IF a (TRUE) (IF b (TRUE) (FALSE)))

REPLACEMENT: (OR a b)
```

This inference rule follows from the definition of OR.

5.51 BOOL-IF-TO-OR

```
PATTERN: (IF a (TRUE) b)

CONDITION: pattern is in a boolean context or
           b is a boolean expression.

REPLACEMENT: (OR a b)
```

This inference rule follows from *IF-TO-OR* (first coerce b to boolean).

5.52 BOOL-IF-TO-IMPLIES-OR

```
PATTERN: (IF a b (IMPLIES c b))

CONDITION: pattern is in a boolean context or
           b is a boolean expression.

REPLACEMENT: (IMPLIES (OR a c) b)
```

This inference rule follows from the definitions of boolean context, boolean expressions, and the boolean connectives, and the basic propositional inference rules.

5.60 AND-ASSOCIATE-LEFT

PATTERN: (AND a1 (AND a2 a3))

REPLACEMENT: (AND (AND a1 a2) a3)

This inference rule follows from the definition of AND and the basic propositional inference rules.

5.61 AND-ASSOCIATE-RIGHT

PATTERN: (AND (AND a1 a2) a3)

REPLACEMENT: (AND a1 (AND a2 a3))

This inference rule follows from the definition of AND and the basic propositional inference rules.

5.62 OR-ASSOCIATE-LEFT

PATTERN: (OR a1 (OR a2 a3))

REPLACEMENT: (OR (OR a1 a2) a3)

This inference rule follows from the definition of OR and the basic propositional inference rules.

5.63 OR-ASSOCIATE-RIGHT

PATTERN: (OR (OR a1 a2) a3)

REPLACEMENT: (OR a1 (OR a2 a3))

This inference rule follows from the definition of OR and the basic propositional inference rules.

5.64 TRUE-TO-=

PARAMETER: x

PATTERN: (TRUE)

REPLACEMENT: (= x x)

This inference rule is the reverse of *=-TO-TRUE*.

```
REPLACEMENT: (ALL (var) expr)
```

This is the *REMOVE-UNIVERSAL* inference rule in reverse.

6.6 UNREMOVE-EXISTENTIAL

```
PARAMETER: var

PATTERN: expr

CONDITIONS: var does not occur free in expr, and
            expr is a boolean expression or is in a boolean context.

REPLACEMENT: (SOME (var) expr)
```

This is the *REMOVE-EXISTENTIAL* inference rule in reverse.

6.7 INSTANTIATE-UNIVERSAL

```
PARAMETER: (= x exp)

PATTERN: (ALL (x) p)

CONDITION: free variables of exp are not bound in p.

REPLACEMENT: (IF p1 (ALL (x) p) (FALSE))
```

where p1 is p with free occurrences of x replaced by exp.

6.8 INSTANTIATE-EXISTENTIAL

```
PARAMETER: (= x exp)

PATTERN: (SOME (x) p)

CONDITION: free variables of exp are not bound in p.

REPLACEMENT: (IF p1 (TRUE) (SOME (x) p1))
```

where p1 is p with free occurrences of x replaced by exp.

6.9 UNINSTANTIATE-UNIVERSAL

```
PARAMETER: (= x exp)

PATTERN: (IF p1 (ALL (x) p) (FALSE))

CONDITION: free variables of exp are not bound in p.

REPLACEMENT: (ALL (x) p)
```

where p1 is p with free occurrences of x replaced by exp. This is the *INSTANTIATE-UNIVERSAL* inference rule in reverse.

6.15 ALL-CASE-ANALYSIS-1

PATTERN: (ALL (x) (IF a b c))

CONDITION: x does not occur free in a.

REPLACEMENT: (IF a (ALL (x) b) (all (x) c))

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

6.16 ALL-CASE-ANALYSIS-2

PATTERN: (ALL (x) (IF a b (TRUE)))

CONDITION: x does not occur free in b.

REPLACEMENT: (IF (SOME (x) a) (if b (TRUE) (FALSE)) (TRUE))

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

6.17 ALL-CASE-ANALYSIS-3

PATTERN: (ALL (x) (IF a (TRUE) b))

CONDITION: x does not occur free in b.

REPLACEMENT: (IF (ALL (x) a) (TRUE) (if b (TRUE) (FALSE)))

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

6.18 ALL-CASE-ANALYSIS-4

PATTERN: (ALL (x) (IF a b (FALSE)))

REPLACEMENT: (IF (ALL (x) a) (ALL (x) b) (FALSE))

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

6.19 ALL-CASE-ANALYSIS-5

PATTERN: (ALL (x) (IF a (FALSE) b))

REPLACEMENT: (IF (SOME (x) a) (FALSE) (ALL (x) b))

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

6.25 ALL-OUT-TEST

```
PATTERN: (IF (ALL (x) a) (TRUE) b)

CONDITION: x does not occur free in b.

REPLACEMENT: (ALL (x) (IF a (TRUE) b))
```

or

```
PATTERN: (IF (ALL (x) a) b (FALSE))

CONDITION: x does not occur free in b.

REPLACEMENT: (ALL (x) (IF a b (FALSE)))
```

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

6.26 ALL-OUT-TEST-NEGATE

```
PATTERN: (IF (ALL (x) a) (FALSE) b)

CONDITION: x does not occur free in b.

REPLACEMENT: (SOME (x) (IF a (FALSE) b))
```

or

```
PATTERN: (IF (ALL (x) a) b (TRUE))

CONDITION: x does not occur free in b.

REPLACEMENT: (SOME (x) (IF a b (TRUE)))
```

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

6.27 SOME-OUT-TEST

```
PATTERN: (IF (SOME (x) a) (TRUE) b)

CONDITION: x does not occur free in b.

REPLACEMENT: (SOME (x) (IF a (TRUE) b))
```

or

```
PATTERN: (IF (SOME (x) a) b (FALSE))

CONDITION: x does not occur free in b.

REPLACEMENT: (SOME (x) (IF a b (FALSE)))
```

```
REPLACEMENT: (ALL (x) (IF a b c))
```

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

6.32 SOME-OUT-RIGHT

```
PATTERN: (IF a b (SOME (x) c))

CONDITIONS: x does not occur free in a and
            x does not occur free in b.

REPLACEMENT: (SOME (x) (IF a b c))
```

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

6.33 CASES-COLLECT-ALL

```
PARAMETER: y

PATTERN: (AND (ALL (y) p) q (ALL (y) r) ...)

CONDITION: y does not occur free in pattern.

REPLACEMENT: (ALL (y) (AND p q r ...))
```

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

6.34 ALL-IN-AND

```
PATTERN: (ALL (x) (ALL (y) (ALL ... (AND p q ...))))

REPLACEMENT: (AND (ALL (x) (ALL (y) (ALL ... P)))
                  (ALL (x) (ALL (y) (ALL ... q)))
                  ...)
```

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

6.35 ALL-UNCASE-ANALYSIS-1

```
PATTERN: (IF a (ALL (x) b) (ALL (x) c))

CONDITION: x does not occur free in a.

REPLACEMENT: (ALL (x) (IF a b c))
```

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

6.41 SOME-UNCASE-ANALYSIS-2

```
PATTERN: (IF (SOME (x) a) (IF b (TRUE) (FALSE)) (FALSE))

CONDITION: x does not occur in b.

REPLACEMENT: (SOME (x) (IF a b (FALSE)))
```

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

6.42 SOME-UNCASE-ANALYSIS-3

```
PATTERN: (IF (ALL (x) a) (FALSE) (IF b (TRUE) (FALSE)))

CONDITION: x does not occur in b.
REPLACEMENT: (SOME (x) (IF a (FALSE) b))
```

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

6.43 SOME-UNCASE-ANALYSIS-4

```
PATTERN: (IF (ALL (x) a) (SOME (x) b) (TRUE))

REPLACEMENT: (SOME (x) (IF a b (TRUE)))
```

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

6.44 SOME-UNCASE-ANALYSIS-5

```
PATTERN: (IF (SOME (x) a) (TRUE) (SOME (x) b))

REPLACEMENT: (SOME (x) (IF a (TRUE) b))
```

This inference rule is a consequence of propositional reasoning and previous quantifier inference rules.

8 Simplification

The NEVER theorem prover has a simplifier component that, among other things, performs equality and arithmetic reasoning. The workhorse of the simplifier is the *deductive database*. It is the deductive database that performs equality and arithmetic reasoning for the simplifier. As the simplifier traverses the current formula, it adds and retracts assertions to be used by the deductive database as context. (Retractions are done strictly on a *last-in-first-out* basis.) The simplifier transforms a subexpression by querying the deductive database for an equivalent replacement for the subexpression given its context.

For the purpose of this discussion, a simplifier step is an inference performed by the deductive database, on a subexpression of the current formula, and making use of the context of the subexpression. It does not include simplification performed by the formula traversal routine such as simplification using the *IF-TRUE* inference rule.

The simplifier interface to the deductive database is a *lookup* routine. During traversal of the current formula, the simplifier queries the deductive database about possible "better" replacements for subexpressions using the lookup routine. Most of the time, the deductive database will not find a replacement expression as a result of a query, and nothing needs to be logged. However, when a replacement expression is found, then the resulting transformation needs to be logged. The replacement is computed using decision procedures and heuristics internal to the deductive database, and may make use of assumptions provided by the context, equality reasoning, integer arithmetic reasoning, computation, and instantiation of quantified variables. Since a single lookup deduction may require complex reasoning, we do not want to justify it in a single step. Instead, we would like the deduction to be logged in terms of simple inference rules for equality reasoning, arithmetic reasoning, and variable instantiations.

Logging the equality and arithmetic reasoning is difficult due to the fact that the deductive database does not perform the reasoning directly in terms of the simple inferences. Rather, the inferences are buried inside the decision procedures that operate on the *e-graph* and *arithmetic tableau* data structures. Hence the decision procedures would have to be modified if they are to generate a log of simple, easily checkable, inferences.

There is already a known technique for generating *equality certificates* for the equality reasoning performed by a congruence closure based algorithm [4]. Our logging of equality reasoning is based on this technique. As an example of an equality certificate, consider the process of simplifying the subexpression `(= C A)` in the following formula:

```
(IF (= A B)
    (IF (= B C)
        (= C A)
        (TRUE))
    (TRUE))
```

The context of the subexpression include the assumptions `(= A B)` and `(= B C)`. When queried for a replacement for `(= C A)`, the deductive database will reply with the expression `(TRUE)`. The certificate for the equality reasoning is the equality chain `A = B, B = C`. If we start with `(= C A)` and follow the equality chain (i.e., replace occurrences of the left side with the corresponding right side), we will derive the identity `(= C C)`, which can be transformed to `(TRUE)`:

```
(= C A)

(= C B)

(= C C)
```

8 Simplification

The NEVER theorem prover has a simplifier component that, among other things, performs equality and arithmetic reasoning. The workhorse of the simplifier is the *deductive database*. It is the deductive database that performs equality and arithmetic reasoning for the simplifier. As the simplifier traverses the current formula, it adds and retracts assertions to be used by the deductive database as context. (Retractions are done strictly on a *last-in-first-out* basis.) The simplifier transforms a subexpression by querying the deductive database for an equivalent replacement for the subexpression given its context.

For the purpose of this discussion, a simplifier step is an inference performed by the deductive database, on a subexpression of the current formula, and making use of the context of the subexpression. It does not include simplification performed by the formula traversal routine such as simplification using the *IF-TRUE* inference rule.

The simplifier interface to the deductive database is a *lookup* routine. During traversal of the current formula, the simplifier queries the deductive database about possible "better" replacements for subexpressions using the lookup routine. Most of the time, the deductive database will not find a replacement expression as a result of a query, and nothing needs to be logged. However, when a replacement expression is found, then the resulting transformation needs to be logged. The replacement is computed using decision procedures and heuristics internal to the deductive database, and may make use of assumptions provided by the context, equality reasoning, integer arithmetic reasoning, computation, and instantiation of quantified variables. Since a single lookup deduction may require complex reasoning, we do not want to justify it in a single step. Instead, we would like the deduction to be logged in terms of simple inference rules for equality reasoning, arithmetic reasoning, and variable instantiations.

Logging the equality and arithmetic reasoning is difficult due to the fact that the deductive database does not perform the reasoning directly in terms of the simple inferences. Rather, the inferences are buried inside the decision procedures that operate on the *e-graph* and *arithmetic tableau* data structures. Hence the decision procedures would have to be modified if they are to generate a log of simple, easily checkable, inferences.

There is already a known technique for generating *equality certificates* for the equality reasoning performed by a congruence closure based algorithm [4]. Our logging of equality reasoning is based on this technique. As an example of an equality certificate, consider the process of simplifying the subexpression `(= C A)` in the following formula:

```
(IF (= A B)
    (IF (= B C)
        (= C A)
        (TRUE))
    (TRUE))
```

The context of the subexpression include the assumptions `(= A B)` and `(= B C)`. When queried for a replacement for `(= C A)`, the deductive database will reply with the expression `(TRUE)`. The certificate for the equality reasoning is the equality chain `A = B, B = C`. If we start with `(= C A)` and follow the equality chain (i.e., replace occurrences of the left side with the corresponding right side), we will derive the identity `(= C C)`, which can be transformed to `(TRUE)`:

```
(= C A)

(= C B)

(= C C)
```

```
(IF-TRUE (2 2 2 2 2 1))
```

The subsequence begins with a *SPLIT* followed by *LOOK-UP-EQUAL* giving:

```
(IF (= A B) (IF (= B C) (= (IF (= C B) B C) A) (TRUE)) (TRUE))
```

The entries from *USE-AXIOM* to the *IF-TRUE* after the *DISCARD-AXIOM* simply commutes the equality (= C B) to (= B C) which is then discharged using *LOOK-UP-TRUE* since it is in the context. In this case, the witness information between the node for B and the node for C indicates that the nodes were merged because of the true equality (= B C). Thus, the initial part of the sequence transforms the subformula to a subformula involving (= B C). The proof logger then examines the witness information for the merge of (= B C) with (TRUE), which in this case is simply context, which only requires a single *LOOK-UP-TRUE*. Note that the bulk of the subsequence has to do with commuting the equality, which would not be needed if the subexpression (= B C) was instead (= C B).

```
(SPLIT (2 2 2 2 2 1) (= B A))
(LOOK-UP-EQUAL (2 2 2 2 2 1 2) A)
(USE-AXIOM (2 2 2 2 2 1 1) =.COMMUTES)
(RENAME-UNIVERSAL (2 2 2 2 2 1 1 1) X G2608)
(RENAME-UNIVERSAL (2 2 2 2 2 1 1 1 2) Y G2609)
(INSTANTIATE-UNIVERSAL (2 2 2 2 2 1 1 1) (= G2608 B))
(IF-NORMAL (2 2 2 2 2 1 1))
(IF-FALSE (2 2 2 2 2 1 1 3))
(INSTANTIATE-UNIVERSAL (2 2 2 2 2 1 1 1) (= G2609 A))
(IF-NORMAL (2 2 2 2 2 1 1))
(IF-FALSE (2 2 2 2 2 1 1 3))
(LOOK-UP-EQUAL (2 2 2 2 2 1 1 2 2 2) (= A B))
(INVERSE-IF-FALSE
  (2 2 2 2 2 1 1 3)
  (IF (ALL G2608 (ALL G2609 (= (= G2608 G2609) (= G2609 G2608))))
      (= A B)
      (TRUE)))
(UNNORMALIZE-IF (2 2 2 2 2 1 1))
(UNINSTANTIATE-UNIVERSAL (2 2 2 2 2 1 1 1) (= G2609 A))
(INVERSE-IF-FALSE (2 2 2 2 2 1 1 3) (= A B))
(UNNORMALIZE-IF (2 2 2 2 2 1 1))
(UNINSTANTIATE-UNIVERSAL (2 2 2 2 2 1 1 1) (= G2608 B))
(RENAME-UNIVERSAL (2 2 2 2 2 1 1 1) G2608 X)
(RENAME-UNIVERSAL (2 2 2 2 2 1 1 1 2) G2609 Y)
(DISCARD-AXIOM (2 2 2 2 2 1 1 1) =.COMMUTES)
(IF-TRUE (2 2 2 2 2 1 1))
(LOOK-UP-TRUE (2 2 2 2 2 1 1))
(IF-TRUE (2 2 2 2 2 1))
```

The above subsequence transform the resulting B from the first part to A giving the equality (= A A). The remaining subsequence uses *=-TO-TRUE* to transform the equality to (TRUE) and completes the proof using propositional and quantifier reasoning.

```
(=-TO-TRUE (2 2 2 2 2))
(IF-EQUAL (2 2 2 2))
(IF-EQUAL (2 2 2))
(REMOVE-UNIVERSAL ())
```

9 Induction

The induction performed by NEVER is based on the induction techniques of Boyer and Moore described in [2]. The justification for an induction is the strong induction principle and the termination lemma for the recursive function that suggested the induction.

Prior to our work on proof logging, the termination lemma, which is the proof obligation for the recursive function, was not available for use in NEVER (i.e., it was not an axiom in the system). We have since made all proof obligations available as axioms.

Since one of our goals was simplicity of inference rules, we settled on a single simple inference rule for induction; and it is essentially the strong induction principle. Typically, the log of an `(INDUCT)` command includes the use of the relevant termination lemma (using *USE-AXIOM*), the *INDUCT* inference, instantiations and case analysis using quantifier and propositional inference rules.

9.1 INDUCT

```
PARAMETERS: m1

PATTERN: (ALL (m) p)

CONDITION: m1 does not occur in p

REPLACEMENT: (ALL (m) (IMPLIES (ALL (m1) (IMPLIES (M< m1 m) p1)) p))
```

where `p1` is `p` with free occurrences of `m` replaced by `m1`. `M<` is a predefined well-founded relation in Verdi.

11 Syntactic Abbreviations

One of the goals of proof checking is to check every single transformation to the current formula. Expansion and unexpansion of syntactic abbreviations are no exception. There are three kinds of syntactic abbreviations in Verdi: *n-ary* functions, quantification abbreviations and constructor notations. The n-ary functions in Verdi are +, *, and, and or. The constructors in Verdi are for finite sets, strings and general arrays. For the purpose of proof logging, we also include normalization of arithmetic relations to >= as expansions of syntactic abbreviations.

11.1 EXPAND-FUNCTION

```
PATTERN: (f a1 ... an-1 an)

CONDITIONS: f is one of +, *, and, or.
            n >= 3.

REPLACEMENT: (f a1 ... (f an-1 an))
```

11.2 UNEXPAND-FUNCTION

```
PARAMETER: m

PATTERN: (f a1 ... (f am1 ... amk) ... an)

CONDITIONS: f is one of +, *, and, or.
            n >= m >= 1.

REPLACEMENT: (f a1 ... am1 ... amk ... an)
```

11.3 EXPAND-ALL

```
PATTERN: (ALL (x1 ... xn) expr)

CONDITION: n >= 2.

REPLACEMENT: (ALL (x1) (ALL (... xn) expr))
```

11.4 UNEXPAND-ALL

```
PATTERN: (ALL (x) (ALL (y1 ...) expr))

REPLACEMENT: (ALL (x1 y1) expr)
```

11.5 EXPAND-SOME

```
PATTERN: (SOME (x1 ... xn) expr)

CONDITION: n >= 2.

REPLACEMENT: (SOME (x1) (SOME (... xn) expr))
```

11.15 MAKE-SET-TO-SETADD

PATTERN: (MAKE-SET a1 a2 ... an)

REPLACEMENT: (SETADD a1 (SETADD a2 ... (SETADD an (NULLSET))...))

11.16 STRING-TO-MAKE-STRING

PATTERN: "ab...x"

REPLACEMENT: (ASET ... (ASET (ASET (MAKE-STRING n) 1 'a') 2 'b') ... n 'x')

where n is the length of the string.

11.17 ARRAY-VAL-TO-MAP-PAD

PATTERN: (arr.VAL v1 v2 ...)

CONDITION: arr is an array type name.

REPLACEMENT: (... (ASET (ASET (arr.MAP (arr.PAD)) lo v1) (SUCC lo) v2) ...)

where lo is (ARRAY-TYPE-LOB (arr)).

11.18 <-TO->=

PATTERN: (< x y)

REPLACEMENT: (>= y (SUCC x))

11.19 >-TO->=

PATTERN: (> x y)

REPLACEMENT: (>= x (SUCC y))

11.20 <=-TO->=

PATTERN: (<= x y)

REPLACEMENT: (>= y x)

11.21 >=-TO-<=

PATTERN: (>= x y)

REPLACEMENT: (<= y x)

References

[1] W.W. Bledsoe, The sup-inf method in Presburger arithmetic, Memo ATP-18, Math. Dept., Univ. of Texas at Austin, 1974.

[2] R.S. Boyer, J S. Moore, *A Computational Logic*, Academic Press, NY, 1979.

[3] D. Craigen, Reference Manual for the Language Verdi, TR-91-5429-09a, ORA Canada, September 1991.

[4] D.B. Krafft, A.J. Demers, Determining logical dependency in a decision procedure for equality, TR 81-458, Dept. of Computer Science, Cornell Univ., Ithaca, NY, April 18, 1981.

[5] M. Saaltink, A Formal Description of Verdi, TR-95-5482-02, ORA Canada, March 1995

[6] R.E. Shostak, Deciding linear inequalities by computing loop residues, in *JACM* 28(4):769-779.

NO. OF COPIES NOMBRE DE COPIES	COPY NO. COPIE N°	INFORMATION SCIENTIST'S INITIALS INITIALES DE L'AGENT D'INFORMATION SCIENTIFIQUE
1	1	
AQUISITION ROUTE FOURNI PAR ▶	DSACCIS via CRAD HQ DRP	
DATE ▶	16 Apr 96	
DSIS ACCESSION NO. NUMÉRO DSIS ▶		

DND 1158 (6-87)

National Defence / Défense nationale

Symcom 497166

PLEASE RETURN THIS DOCUMENT TO THE FOLLOWING ADDRESS:

DIRECTOR
SCIENTIFIC INFORMATION SERVICES
NATIONAL DEFENCE
HEADQUARTERS
OTTAWA, ONT. - CANADA K1A 0K2

PRIÈRE DE RETOURNER CE DOCUMENT À L'ADRESSE SUIVANTE:

DIRECTEUR
SERVICES D'INFORMATION SCIENTIFIQUES
QUARTIER GÉNÉRAL
DE LA DÉFENSE NATIONALE
OTTAWA, ONT. - CANADA K1A 0K2