CS-Note* 6 Andrew Ireland and Bill J. Ellis
December 7, 2001

NuSPADE: Proof Planning for SPARK Verification Proofs

1 Introduction

Following on from CSnotes 4 and 5, we now outline our initial ideas for adding a proof planning capability to the SPADE Proof Checker[1]. The name given to the enhanced system will be NuSPADE[2], pronounced "new spade". The development of NuSPADE will be under-pinned by 3 core principles:

Integrity: NuSPADE will only modify the SPADE proof state via the SPADE command level.

Compatibility: NuSPADE will preserve SPADE at its core, *i.e.* no modifications will be made to SPADE.

Transparency: NuSPADE will provide users with the look-and-feel of a SPADE proof session.

While SPADE will remain unchanged by us, we will design NuSPADE so that future incremental changes to SPADE will be accommodated easily within NuSPADE.

The structure of the note is as follows. In §2 the major components of NuSPADE are outlined while in §3 a more operational perspective is presented. In order to emphasize the differences between SPADE and NuSPADE a simple verification example is presented in §4. Some potential longer term issues are raised in §5 while a work plan and conclusions are presented in §6 and §7 respectively.

2 Component perspective

The aim of NuSPADE is to offer additional proof automation capabilities to SPADE users. At an operational level, users of NuSPADE will have direct access to SPADE. In addition, however, users will also be able to invoke a proof planner at any point during a proof session. The proof planner supports proof automation through the generation of command logs that can then be executed within SPADE. Figure 1 shows the major components of NuSPADE. An explanation of each component is given below along with a description of how the overall systems works.

2.1 Interface

The NuSPADE interface will be based around the current SPADE interface[3]. The interfaces will differ only in the commands they offer, *i.e.* NuSPADE will contain a number of additional commands relating to its planning capabilities. Note that the interface will provide a NuSPADE user with direct access to all the planning capabilities in addition to the SPADE command set. Experience has shown that this level of access will prove valuable during the development and initial analysis of new proof plans. However, we envisage that the typical NuSPADE user will have minimum interaction with the planning component, *i.e.* the proof planner will be treated as a "push-button" technology by the user.

2.2 SPADE

As mentioned in §1, the development of NuSPADE will not result in any modifications to the SPADE code itself. Our planning component will require access to data structures within SPADE, *e.g.* parts of the proof state in order to setup planning goals. However, this access will be strictly on a "read-only" basis.

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the Clam-Spark project (EPSRC GR/R24081).

[1] Henceforth, when we refer to SPADE we will be referring to the Proof Check.

[2] We are following the acronym convention started by the NuPRL project [3] and also adopted by the NuSMV project [2].

[3] Our implementation will be developed in the first instance on Solaris therefore the interface component will be terminal based. However, we intend to develop a Windows version at some point.

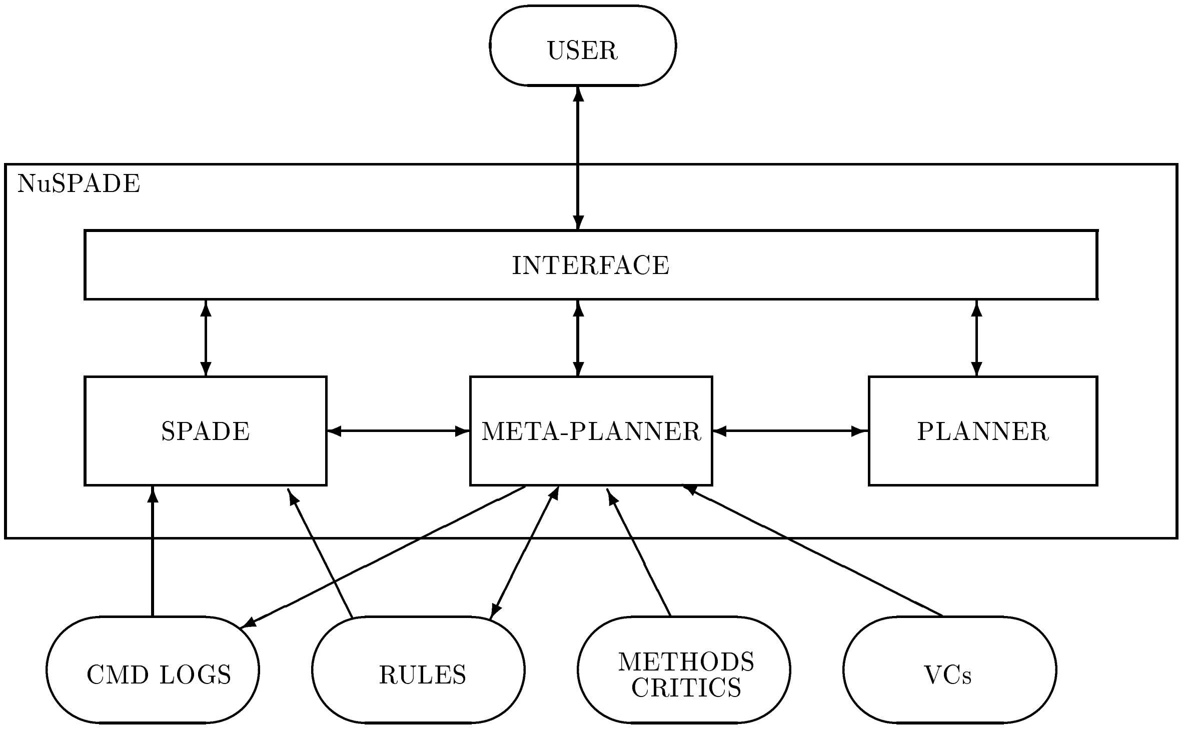

Figure 1: NuSPADE Architecture

2.3 Meta-planner

Meta-planning is concerned with planning about planning. Within the context of proof planning this corresponds to:

- Deciding which proof methods and critics should be considered for the planning of a particular VC.
- Performing the appropriate meta-level analysis of definitions, rules *etc* that relate to the current VC.

Part of the motivation for meta-planning comes from efficiency concerns, *e.g.* methods developed for reasoning about arrays should only be added into the planners working set of methods if the current VC relates to arrays. From an engineering perspective, it also makes sense to decouple the knowledge embodied within methods from the knowledge about which sets of methods are most applicable in particular contexts.

The meta-planner will also manage the process of translating SPADE definitions, rules and state information into the form appropriate for meta-level analysis and planning. In addition, the meta-planner will be responsible for translating the plans generated by the planner into corresponding SPADE command logs. It is envisaged that this translation will, as a side-effect, also generate an "executive summary" of the plan for the user.

As hinted above, the typical NuSPADE user will have minimum interaction with the meta-planner, rather they will issue a "plan command" which will result in the meta-planner automatically setting up the context within which the planner is applied. All the user will see is a plan summary followed by the result of applying the plan (command log) to the current VC.

2.4 Planner

In terms of actual planning proofs of VCs we will adopt the model described in CS-note 5 for the CLAM system. An ordered list of methods will be used to iteratively decompose the current VCs. Where methods are judged to be only partially applicable then the corresponding critics will be invoked.

As with the meta-planner, the planner will support the option of interaction. By default the user will not interact with the planning process. In order to support the development of proof plans, the planner will support debug options for methods and critics as well as providing feedback on the proof search (see appendix of CS-note 5). However, in the longer term we envisage support for a collaborative style of proof patching through interactive proof critics (again see CS-note 5). The key point is that the user will have complete control in determining the level of interaction.

3 Systems perspective

We now consider in more detail the interactions between the NuSPADE components.

3.1 Loading the Problem

Before any proof attempt can be investigated a problem must first be loaded. This problem will contain VCs yet to be discharged. It will also contain definitions and rules, either explicitly or implicitly loaded[4], to help tackle the VCs. A problem is loaded into SPADE, by first specifying the files describing the VCs and then loading each rule file as required. As NuSPADE shares the same problem as SPADE, the mechanism for loading NuSPADE problems is exactly the same.

3.2 SPADE Proof Commands

Inside SPADE, proof commands are performed on the current VC through the SPADE console. As mentioned in §2.1 these commands will continue to be supported in NuSPADE.

3.3 NuSPADE Planning Commands

During the course of a proof effort the user will be able to call upon the planning capability of NuSPADE for assistance in tackling any VC. Once selected, the automatic planning of a VC will follow a general sequence of events:

1. The meta-planner extracts from SPADE proof state the information that is required to set-up a goal for the planner. The SPADE proof state information will also be used by the meta-planner to determine which rules will potentially be required during planning. These rules are loaded and analyzed (meta-level analysis) by the meta-planner. Note that saving the results of the meta-level analysis, *i.e.* "meta-data", might be considered in order to improve efficiency. The selection of the planner's working set of methods and critics will be performed by the meta-planner. Once these initialization activities are complete the meta-planner will invoke the planner.

2. The planner will terminate[5] and generate one of the following:

   - a proof plan or
   - a proof plan with reference to auxiliary rules or
   - a proof plan with reference to auxiliary assertions or
   - a proof plan with reference to auxiliary rules and assertions or
   - no proof plan.

   Note that a plan (command log) generated by the planner may only partially complete the current VC. This raises the issue of how best to assist the user in orienting themselves within the new proof context. Where auxiliary rules are generated we expect to generate supporting proofs. Where auxiliary assertions, *e.g.* loop invariants, are generated then the user will have to manually change their SPARK code annotations appropriately, re-generate the VCs via the Examiner and then restart the proof process. A development environment within which this feedback loop is hidden would be desirable but is out with the scope of the current project.

3. The information generated by the planner will be passed back to the meta-planner which in turn extracts:

   - the command log that will be applied at the SPADE level and
   - the rules that will be used at the SPADE level and
   - the proof plan summary that will be presented to the user.

   Note that both the command log and the auxiliary rules are written to files. Once the proof plan summary has been presented to the user the meta-planner automatically applies the command log at the SPADE level.

4. Control is returned to the user.

[4]The built-in rules are loaded automatically in each SPADE session.
[5]Termination can be guaranteed using a simple resource bound.

4 Example

To illustrate the proposed system a small example shall be considered. A SPARK program has been written that finds positive exponents[6]. The source code is shown in figures 2, 3 and 4 (Appendix A.1). The simplified VCs generated for the exponentiation code are shown in Appendix 5. As a run time-check is being performed a target data file was also created, to give the target machine specific range of integers, shown in figure 5 (Appendix A.5). The related library, fdl and rule definitions are given in figures 6, 7 and 8 respectively (Appendix A.3).

4.1 Proof in SPADE

Proof in SPADE is performed through interaction with the user. There may be a number of ways in which a proof can be constructed. In this case we assume the user has created and proved two extra rules (lemmas), shown in figure 9 (Appendix A.4).

Once these two lemmas have have been saved as rules, the following command log can be used to discharge VCs 5, 6 and 13. Note that we have not considered the remaining VCs (7-12).

```
load 'explib'.
consult 'explemma.rul'.
replace c#1: exponent(x, y) by
        result * exponent(workx, worky) using eq(1).
y.
n.
replace c#1: result * workx * exponent(workx * workx, worky div 2) by
        result * (workx * exponent(workx * workx, worky div 2)) using assoc(3).
y.
n.
replace c#1: workx * exponent(workx * workx, worky div 2) by
        exponent(workx, worky) using explemma(2).
y.
n.
done.
6.
replace c#1: exponent(x, y) by
        result * exponent(workx, worky) using eq(1).
y.
n.
replace h#9: not isodd(worky) by
        iseven(worky) using useeven(1).
y.
n.
replace c#1: exponent(workx * workx, worky div 2) by
        exponent(workx, worky) using explemma(1).
y.
n.
done.
13.
replace h#1: exponent(workx,0) by
             1 using defexp(1).
y.
n.
replace h#1: result * 1 by
             result using arith(1).
y.
n.
replace c#1: result by
             exponent(x,y) using eq(1).
y.
n.
done.
```

[6] X^Y where $Y >= 0$

4.2 Proof in NuSPADE

This section is speculative, it outlines what we aim to achieve with in NuSPADE. In order to illustrate the ideas we will make reference to the exponentiation example introduced above.

4.2.1 Proof automation

Our starting point is modest, planning the partial correctness subgoals that are left once the work of the SPADE Simplifier is complete, *e.g.* VCs 5, 6 and 13. In concrete terms this means that the planner automatically generates a command log equivalent to the one given in §4.1. Here we assume that a suitable loop invariant has been provided by the programmer and that all auxiliary rules are available. In the case of the exponentiation example, this means that the invariant is known:

```
result * exponent(workx, worky) = exponent(x, y)
```

In addition, that the following properties are also known:

```
 explemma(1): exponent(A*A,B div 2) may_be_replaced_by exponent(A,B) if [iseven(B)].
explemma(2): A*exponent(A*A,B div 2) may_be_replaced_by exponent(A,B) if [isodd(B)].
```

4.2.2 Invariant discovery

The next step will be to transfer, adapt and extend our existing techniques for dynamically generating invariants based upon the analysis of failed proof planning attempts. In the exponentiation example this would typically mean starting with a candidate invariant of the form:

```
result  = exponent(x, y)
```

from which a suitable invariant is generated:

```
result * exponent(workx, worky) = exponent(x, y)
```

Note that we are assuming that rules `explemma(1)` and `explemma(2)` are given.

4.2.3 Rule discovery

Our previous work on the dynamic generation of missing lemmas will also provide the foundation for generating missing rules with in the context of SPADE. In the case of the exponentiation example, this translates into generating rules `explemma(1)` and `explemma(2)` automatically. Note that this will be based upon the programmer having provided a suitable invariant.

In [1] (page 262) John Barnes highlights the importance of working with correct rules. He proposes a method for ensuring this in SPADE, by manually composing rules as VCs and conducting proofs in SPADE in the normal manner. Within NuSPADE this manual process should to a large extent become redundant. For instance, an existing proof plan[7] for mathematical induction will automatically verify lemmas such as `explemma(1)` and `explemma(2)`.

4.2.4 Invariant and rule discovery

Our previous work has focused on the automatic discovery of lemmas (rules) given a suitable inductive generalization (inductive invariant) and the automatic discovery of inductive generalizations (inductive invariants) given suitable lemmas (rules). If we have neither the lemmas (rules) or the inductive generalizations (inductive invariants) then we have a Catch-22! One of our aims is therefore to make progress in breaking this Catch-22 by seeking to exploit synergies between the techniques for lemma discovery and generalization. In the exponentiation example this means automatically generating the invariant as well as rules `explemma(1)` and `explemma(2)`.

4.2.5 Automation for run-time checks

Automation for proving the absence of run-time errors is an aspect of program proof that we have not considered before. However, the work on invariant discovery will play an important role here. The run-time proof obligations (upper-bounds) that are generated for the exponentiation example certainly look unprovable as the code stands. This raises the question as to whether we should be providing feedback to the programmer in terms of how they might change the annotations or the code?

[7] With minor changes for the SPADE logic.

5 Some longer term issues

The proposal for NuSPADE outlined above represents a first prototype. As we explore case studies material provided by Praxis we believe that the need for additional functionality will arise. A couple of potential future enhancements are outlined below.

5.1 Tackling multiple VCs

While the discussion above relates to the sequential planning of VCs, the meta-planner may also have a role to play in planning multiple VCs simultaneously[8]. The motivations for such an approach would be to exploit meta-level constraints not present when planning is undertaken sequentially. Time will tell whether such synergies actually arise in practice.

5.2 Using the source code

There is a very close relationship between VCs and the source code they are extracted from. However, it is not the case that the code can be reconstructed directly from the VCs. There is a loss of information.

It is therefore conceivable that the source code may hold information that can be used to assist in proof planning. However, there is no way to access the source code from SPADE[9]. Further, in some cases NuSPADE may propose amendments to the source annotations. If direct access to the source was available this might be achieved without user interaction.

Giving NuSPADE access to the source code that VCs are generated from might require modifications to SPADE, which is against our core principles[10]. There is also some concern at breaking the logical progression of the SPARK approach (Code $\rightarrow$ $\boxed{\text{Examiner}}$ $\rightarrow$ VCs $\rightarrow$ $\boxed{\text{Simplifier}}$ $\rightarrow$ Simplified VCs $\rightarrow$ $\boxed{\text{Proof Checker}}$ $\rightarrow$ Proof) by accessing the top level source during the low level proof.

6 Immediate work plan

Presented above is a proposed architecture of the NuSPADE system. This is a significant first step, building a mental model of the system envisaged. The refinement of this model and the development of an initial prototype are amongst our key objectives for the coming months. A rough work plan is presented below:

- **December 2001**
  - Publish this note with the aim of gaining feedback on the NuSPADE architecture from expert SPADE users.
  - Identify in detail the data-flows associated with the NuSPADE architecture.
  - Revisit Clam with a view to determining what components can be reused within NuSPADE.
- **January 2002**
  - Re-evaluate the NuSPADE architecture, taking into account feedback from expert SPADE users.
  - Begin detailed design phase.
  - Produce a small corpus of examples for use in the development and testing of the initial prototype.
- **February 2002**
  - Complete detailed design phase.
  - Draw-up milestones for implementation phase.
  - Begin implementation phase[11].
- **March 2002**
  - Continue implementation phase.
  - Conduct uniting testing.

[8] A planning strategy based upon coroutining would provide one approach.

[9] It is however possible to access information describing how each VC is generated from the source.

[10] This could also be achieved in a more roundabout way - by adding fake rules that contain path information, enforcing a strict convention for where to access the source, etc.

[11] Establish framework for managing code, *e.g.* CVS.

- **April 2002**
  - Complete initial prototype.
  - Evaluate prototype on example corpus.

7 Conclusion

The major architectural components of our NuSPADE proposal have been presented. Over the coming months the details of this architecture will be fleshed out. Within the next five months we believe that we can deliver an initial prototype based upon this architecture.

References

[1] J. G. P. Barnes. *High Integrity Ada The SPARK Approach.* Addison-Wesley, 1997.

[2] At NuSMV. In *http://nusmv.irst.itc.it/.*

[3] At PRL PRoject. In *http://www.cs.cornell.edu/Info/Projects/NuPrl/.*

A Appendix

A.1 Spark Exponentiation Code

```
--main.adb

with Spark_IO;
with exp;

procedure Main is
        X:Integer;
        Y:Integer;
        Result:Integer;
        Ok:Boolean;
    begin

--      Get the value of X.
        loop
            Spark_IO.Put_String(Spark_IO.Standard_Output,"X = ",0);
            Spark_IO.Get_Integer(Spark_IO.Standard_Input,X,0,Ok);
        exit when (Ok=true);
        end loop;

--      Get the value of Y.
        loop
            Spark_IO.Put_String(Spark_IO.Standard_Output,"Y = ",0);
            Spark_IO.Get_Integer(Spark_IO.Standard_Input,Y,0,Ok);
        exit when (Ok=true);
        end loop;

--      Do the calcuation.
        exp.doExp(X,Y,Result);

--      Display the result.
        Spark_IO.Put_String(Spark_IO.Standard_Output,"exp(X,Y) = ",0);
        Spark_IO.Put_Integer(Spark_IO.Standard_Output,Result,0,10);

    end Main;

--End of file
```

Figure 2: Exponentiation code - main.adb

```
--exp.ads

package exp is
    --#function exponent(A,B: Integer) return Integer;
    --#function iseven(A: Integer) return Boolean;

    function isodd(A: Integer) return Boolean;
    --#return (not (A rem 2 = 0));

    procedure doExp(X,Y: in Integer ; Result: out Integer);
    --#derives Result from X,Y;
    --#pre Y>=0;
    --#post Result=exponent(X,Y);

end exp;

--End of file
```

Figure 3: Exponentiation code - exp.ads

```
--exp.adb

package body exp is

    function isodd(A: Integer) return Boolean is
    begin
        return (not (A rem 2 = 0));
    end isodd;

    procedure doExp(X,Y: in Integer;Result: out Integer) is
WorkX:Integer;
WorkY:Integer;
    begin
--      Can not modify X or Y as they are input only.
--      Use Working copies instead.
        WorkX:=X;
        WorkY:=Y;
        Result:=1;

--      Calcuate exp(WorkX,WorkY).
        loop
    --#assert Result*exponent(WorkX,WorkY)=exponent(X,Y);
            exit when (WorkY=0);
            if isodd(WorkY) then
                  Result:=Result*WorkX;
            end if;
            WorkY:=WorkY/2;
            WorkX:=WorkX*WorkX;
        end loop;
--      Result holds the value exp(X,Y).

    end doExp;
end exp;

--End of file
```

Figure 4: Exponentiation code - exp.adb

A.2 Spark Target file

```
integer'first = -32768
integer'last = 32767
```

Figure 5: Target file - exp.dat

A.3 Spark Rule Library

```
/* Library Build Script */

declarations_file = 'explib.fdl'.
rule_file = 'explib.rul'.
```

Figure 6: Library build script for rule library - explib.lbs

```
{ FDL for rule library }

  function isodd(integer) : boolean;
  function iseven(integer) : boolean;
  function exponent(integer, integer) : integer;
enddeclarations;
```

Figure 7: FDL for rule library - explib.fdl

```
rule_family defexp:
    exponent(X,Y) requires [X:i,Y:i].

defexp(1): exponent(X,0) may_be_replaced_by 1.
defexp(2): exponent(X,Y+1) may_be_replaced_by X * exponent(X,Y).

rule_family useeven:
    iseven(X) requires [X:i] &
    not X requires [X:any].

useeven(1): not (isodd(X)) may_be_replaced_by iseven(X).
useeven(2): iseven(X+1) may_be_replaced_by isodd(X).

rule_family useodd:
    isodd(X) requires [X:i] &
    not X requires [X:any].

useodd(1): not (iseven(X)) may_be_replaced_by isodd(X).
useodd(2): isodd(X+1) may_be_replaced_by iseven(X).
```

Figure 8: RUL for rule library - explib.rul

A.4 Eureka Lemmas

```
rule_family explemma:
    exponent(X,Y) requires [X:i,Y:i] &
    iseven(X) requires [X:i] &
    isodd(X) requires [X:i] &
    X * Y requires [X:ir,Y:ir].

explemma(1): exponent(A*A,B div 2) may_be_replaced_by exponent(A,B) if [iseven(B)].
explemma(2): A*exponent(A*A,B div 2) may_be_replaced_by exponent(A,B) if [isodd(B)].
```

Figure 9: RUL for rule library - explib.rul

A.5 Simplified VCs generated for exponentiation example

```
          *******************************************************
                      Semantic Analysis of SPARK Text
      SPARK95 Examiner with VC and RTC Generator Release 5.01 / 08.00
               For teaching and non-commercial research only
          *******************************************************


      CREATED 26-SEP-2001, 12:17:52  SIMPLIFIED 26-SEP-2001, 12:18:48
               (Simplified by SPADE Simplifier, Version 1.45)

                            procedure exp.doExp




For path(s) from start to run-time check associated with statement of line 16:

procedure_doexp_1.
*** true .          /* all conclusions proved */


For path(s) from start to run-time check associated with statement of line 17:

procedure_doexp_2.
*** true .          /* all conclusions proved */


For path(s) from start to run-time check associated with statement of line 18:

procedure_doexp_3.
*** true .          /* all conclusions proved */


For path(s) from start to assertion of line 22:

procedure_doexp_4.
*** true .          /* all conclusions proved */


For path(s) from assertion of line 22 to assertion of line 22:

procedure_doexp_5.
H1:    result * exponent(workx,worky) = exponent(x,y) .
H2:    x >= - 32768 .
H3:    x <= 32767 .
H4:    y <= 32767 .
```

```
H5:    y >= 0 .
H6:    worky <> 0 .
H7:    worky >= - 32768 .
H8:    worky <= 32767 .
H9:    isodd(worky) .
H10:   result * workx >= - 32768 .
H11:   result * workx <= 32767 .
H12:   worky div 2 >= - 32768 .
H13:   worky div 2 <= 32767 .
H14:   workx * workx >= - 32768 .
H15:   workx * workx <= 32767 .
       ->
C1:    result * workx * exponent(workx * workx,worky div 2) = exponent(x,y) .


procedure_doexp_6.
H1:    result * exponent(workx,worky) = exponent(x,y) .
H2:    x >= - 32768 .
H3:    x <= 32767 .
H4:    y <= 32767 .
H5:    y >= 0 .
H6:    worky <> 0 .
H7:    worky >= - 32768 .
H8:    worky <= 32767 .
H9:    not isodd(worky) .
H10:   worky div 2 >= - 32768 .
H11:   worky div 2 <= 32767 .
H12:   workx * workx >= - 32768 .
H13:   workx * workx <= 32767 .
       ->
C1:    result * exponent(workx * workx,worky div 2) = exponent(x,y) .


For path(s) from assertion of line 22 to run-time check associated with
          statement of line 24:

procedure_doexp_7.
H1:    result * exponent(workx,worky) = exponent(x,y) .
H2:    x >= - 32768 .
H3:    x <= 32767 .
H4:    y <= 32767 .
H5:    y >= 0 .
H6:    worky <> 0 .
       ->
C1:    worky >= - 32768 .
C2:    worky <= 32767 .


For path(s) from assertion of line 22 to run-time check associated with
          statement of line 25:

procedure_doexp_8.
H1:    result * exponent(workx,worky) = exponent(x,y) .
H2:    x >= - 32768 .
H3:    x <= 32767 .
H4:    y <= 32767 .
H5:    y >= 0 .
H6:    worky <> 0 .
H7:    worky >= - 32768 .
H8:    worky <= 32767 .
H9:    isodd(worky) .
       ->
C1:    result * workx >= - 32768 .
C2:    result * workx <= 32767 .
```

For path(s) from assertion of line 22 to run-time check associated with
          statement of line 27:

```
procedure_doexp_9.
H1:    result * exponent(workx,worky) = exponent(x,y) .
H2:    x >= - 32768 .
H3:    x <= 32767 .
H4:    y <= 32767 .
H5:    y >= 0 .
H6:    worky <> 0 .
H7:    worky >= - 32768 .
H8:    worky <= 32767 .
H9:    isodd(worky) .
H10:   result * workx >= - 32768 .
H11:   result * workx <= 32767 .
       ->
C1:    worky div 2 >= - 32768 .
C2:    worky div 2 <= 32767 .


procedure_doexp_10.
H1:    result * exponent(workx,worky) = exponent(x,y) .
H2:    x >= - 32768 .
H3:    x <= 32767 .
H4:    y <= 32767 .
H5:    y >= 0 .
H6:    worky <> 0 .
H7:    worky >= - 32768 .
H8:    worky <= 32767 .
H9:    not isodd(worky) .
       ->
C1:    worky div 2 >= - 32768 .
C2:    worky div 2 <= 32767 .
```

For path(s) from assertion of line 22 to run-time check associated with
          statement of line 28:

```
procedure_doexp_11.
H1:    result * exponent(workx,worky) = exponent(x,y) .
H2:    x >= - 32768 .
H3:    x <= 32767 .
H4:    y <= 32767 .
H5:    y >= 0 .
H6:    worky <> 0 .
H7:    worky >= - 32768 .
H8:    worky <= 32767 .
H9:    isodd(worky) .
H10:   result * workx >= - 32768 .
H11:   result * workx <= 32767 .
H12:   worky div 2 >= - 32768 .
H13:   worky div 2 <= 32767 .
       ->
C1:    workx * workx >= - 32768 .
C2:    workx * workx <= 32767 .


procedure_doexp_12.
H1:    result * exponent(workx,worky) = exponent(x,y) .
H2:    x >= - 32768 .
H3:    x <= 32767 .
H4:    y <= 32767 .
H5:    y >= 0 .
H6:    worky <> 0 .
```

```
H7:    worky >= - 32768 .
H8:    worky <= 32767 .
H9:    not isodd(worky) .
H10:   worky div 2 >= - 32768 .
H11:   worky div 2 <= 32767 .
       ->
C1:    workx * workx >= - 32768 .
C2:    workx * workx <= 32767 .


For path(s) from assertion of line 22 to finish:

procedure_doexp_13.
H1:    result * exponent(workx,0) = exponent(x,y) .
H2:    x >= - 32768 .
H3:    x <= 32767 .
H4:    y <= 32767 .
H5:    y >= 0 .
       ->
C1:    result = exponent(x,y) .
```