Gypsy and the GVE

John McHugh
Portland State University
mchugh@cs.pdx.edu

(re)Sources

- Gypsy is installed in the class directory /home/comp527/bin/gve
- Documentation is in /home/comp527/gve-dir/documentation
- The manuals that are most useful are:
  - report-001-c-gypsy-205.ps
  - This is the Gypsy language reference manual
  - report-002-gypsy-methodology.ps
  - This provides a tutorial with several examples showing how the gve is used.
  - report-012-giftmanual.ps
  - This is the manual for the Gypsy Information Flow Tool (GIFT)
- The directory contains numerous other manuals and internal notes from CLI

Overview and History

- The Gypsy Verification Environment was built at UT Austin by the Institute for Computing Science and Computing Applications under the direction of Don Good with funding from NSA.
- Work started in the late 1970s and continued until about 1986 when the Gypsy group left UT to form Computational Logic, Inc.
- Gve work at CLI continued into the mid 1990s when NSA discontinued support.
- Early versions ran on DEC 10s and 20s. Ports to LISP machines occurred in the mid 80s, to Suns in the late 80s.
- During the 80s and early 90s, the GVE was widely used as a modeling and verification tool for secure systems design.

The gve

- The Gypsy verification environment integrated parsing, compiling, and proof functions with semantically based configuration management. The incremental methods allowed preservation of proof effort in the face of code or specification changes.
- Major components include:
  - Parser (syntax and semantic passes)
  - Verification Condition generator
  - Symbolic Evaluator and interactive theorem prover
  - Database
  - Code generator (common Bliss output)
  - Code optimizer
  - Information flow tool
  - Incremental analysis

What good was it.

- Gypsy was used on a variety of research and production projects:
  - The driving example at UT was a cryptographic interface to the ARPANET, first with NCP then TCP/IP
  - The GVE was used to verify non kernel portions of the SCOMP trusted computing base. The SCOMP was certified at A1 by NSA.
  - Begun as a research project, the message flow modulator, a guard processor, was redone as a fully verified product but never deployed
  - Multinet gateway, deployed during desert storm, was specified in Gypsy. Model code written in gypsy was hand translated to C using the Gypsy optimizer to justify exception suppression.
  - The Secure Computing Corp. LOCK processor was specified in Gypsy and a covert channel analysis used the GIFT.

More applications

- Andy Moore (now at NRL) and I developed a security policy model for what became the VSLAN (Verdix Secure LAN). This was Andy's MS thesis.
- I modeled parts of the X windows protocols while working on a B3 certifiable windowing system project with TRW and TIS. A derivative of this was used in TMACH.

Other systems

- I use Gypsy because it is of historical interest and still represents the only extant integrated environment for code verification. Nontheless, it is crude and a bit clunky. If you want to use a more up to date system for manipulating specifications of the same general sort, try
- PVS from SRI International.
- ACL2 from UT is a verifiable LISP based on the work of Bob Boyer and J Moore.
- For state based systems, there are several model checking systems that are used for both hardware and software (especially protocols) verification.

The Gypsy Language

- The original charge to the UT group was to extract a verifiable subset of Pascal. This was harder than expected and Gypsy is based on Pascal, but with differences.
  - No pointers
  - No global variables
  - No direct IO (buffers to environment)
  - processes communicating via message passing (buffers)
  - set, sequence, and mapping types
  - scoped declarations (packaging)
  - exception semantics
  - specification statements

A simple example

- The next slides intersperse fragments of a program to find the smallest element in an array with observations on the language. Note that this program does not have I/O or concurrency.
- It also has nothing to do with security.
- Friday's example will show us a simple secure file system.

Gypsy Example

```
 scope minimum_procedure =
begin
    procedure loc_of_min (var l : index; a : int_array; i, j : index) =
    begin
        var k : index := i;
        l := i;
        loop
            if k = j then leave end;
            k := k + 1;
            if a[k] < a[l] then l := k end;
        end;
    end;
    name {type} index, int_array from integer_array_types;
end; {scope minimum_procedure}
```

Observations

- The scope statement restricts the scope of potentially global declarations of proceedures, types, constants, etc.
- The name statement allows explicit control over importing names from other scopes (but not over exporting - misfeature?)
- Parameters declared as “var” may be modified
- A general purpose loop is provided with a “leave” statement for exit.

Gypsy Example

```
scope integer_array_types =
begin
    type int_array = array (index) of some_int;
    type index = integer[lo_index..hi_index];
    const lo_index : integer := 0;
    const hi_index : integer := 63;
    type some_int = integer[lo_int..hi_int];
    const lo_int : integer := - 1000;
    const hi_int : integer := 1000;
end; {scope integer_array_types}
```

Observations

- Types and constants may be declared.
- Note that constants are given values.
- This can be avoided by using the construct “pending” as in

  const lo_index : integer := pending;

  const hi_index : integer := pending;
- This will cause proof problems as we will see later because it is possible to refine these definitions in such a way as to produce an ill formed program.
- The developers of the GVE became more than a little paranoid about allowing a user to prove “false” and this results in some (IMHO) silly restrictions.

Gypsy Example

```
$extending scope minimum_procedure =
begin
    procedure loc_of_min (var l : index; a : int_array; i, j : index) =
    begin
        entry i le j;
        exit  is_minimum (a[l], a, i, j) & l in [i..j];
        var k : index := i;
        l := i;
        loop
            assert   is_minimum (a[l], a, i, k) & l in [i..j] & k in [i..j];
            if k = j then leave end;
            k := k + 1;
            if a[k] < a[l] then l := k end;
        end;
    end;
    name {function} is_minimum from minimum_specs;
end; {scope minimum_procedure}
```

Observations

- The GVE stores intermediate results in an internal database. The "$extending" construct tells the system that a previously defined scope is being modified. This may invalidate proofs as well as change definitions.
- Note the "entry" and "exit" specification statements. Entry specs are assumed true upon entry to the procedure (but must be proven to hold at calling sites). Exit specs must be proven from the entry and the effect of the procedure body.
- The assert statement can appear anywhere in the body of the routine, but an assert statement must lie in each path through a loop. The assertion must be proven from the path to it (which may start with the same assertion).
- Note the specification function "is_minimum."

Gypsy Example

```
scope minimum_specs =
begin
    function is_minimum (
        m : some_int;
        a : int_array;
        p, q : index) : boolean =
            pending;
    name {type} some_int, int_array, index from integer_array_types;
end; {scope minimum_specs}
```

Observation

- Note that the function “is_minimum” has no body and is declared “pending.” Later on, we will add some lemmas to specify its behavior.
- Boolean functions are often used for specification.
- The use of “pending” will be tracked as a reminder that we have not yet finished our job. In this case, we will never give a body to the function.

Gypsy Example

```
$extending scope minimum_specs =
begin
    lemma singleton_min (a : int_array; p : index) =
        (assume     is_minimum (a[p], a, p, p));
    lemma extend_old_min_up (m : some_int; a : int_array; p, q : index) =
        (assume     is_minimum (m, a, p, q - 1) & m le a[q]
                -> is_minimum (m, a, p, q));
    lemma extend_new_min_up (m : some_int; a : int_array; p, q : index) =
        (assume     is_minimum (m, a, p, q - 1) & a[q] le m
                -> is_minimum (a[q], a, p, q));
end; {scope minimum_specs}
```

Observations

- The lemmas define the properties that “is_minimum” is expected to have.
  - The minimum of a subarray of size 1 is the single element in it
  - If we extend the size of the subarray by one element and that element is larger than the previous minimum, the previous minimum is the minimum of the expanded subarray
  - If we extend as above but the added element is smaller than the previous minimum, it becomes the minimum of the expanded subarray.
- The lemmas are not well formed. Why? How can you fix it.?
- We have not shown the extension to “minimum_procedure” required to access the lemmas in the process of proving the program.

Gypsy Example (fixed)

- The other lemma “extend_new_min_up” is modified in a similar fashion. This allows one to prove that q-1 is in the type of index.

```
type index_1 = integer[lo_index_1..hi_index];
const lo_index_1 : integer := lo_index+1;
lemma extend_old_min_up (m : some_int; a : int_array;
                                 p : index; q : index_1) =
    (assume    is_minimum (m, a, p, q - 1) & m le a[q]
            -> is_minimum (m, a, p, q));
```

Starting the gve

- We start by invoking the gve. I run it in an emacs shell buffer. There is actually emacs interface code, but I haven't bothered to load it here.

```
%m:%~> gve
GCL (GNU Common Lisp)  Version(2.2.2) Tue Mar  9
  20:06:58 CST 1999
Licensed under GNU Public Library License
Contains Enhancements by W. Schelter
Do you wish to run the GVE in character mode or line
  mode?
(Character, Line, Help)? l
```

- The "l" for line mode is essential. "c" takes an immediate segmentation fault.

gve initialization

```
Note: System Hacking [CLI=SYSTEM-HACKING] is set.
Checking loaded configuration of the GVE
  ...configuration ok!
Gypsy System version 20.70 (EXPERIMENTAL) of 1-Aug-
  1990 05:39:39 AM
Type 'NEWS' for changes in this version.
Report problems by using vbr at Dockmaster.
gve-init.lisp not found in user directory or default
  directory.
Welcome to the Experimental Gypsy 2.05 GVE ... you
  may begin.
Good luck!
```

- System hacking goes to lisp on error. System integrity is checked. vbr at dockmaster no longer exists. I haven't done an initialization file.

gve setup

- Note that commands may be abbreviated. The full version echos.
  ```
  Gve -> set par ec on
  set parSE ecHO on
  ```
- So we can see what we are doing set parse echo on
  ```
  Gve -> set pa pre off
  set paRSE prePROCESSOR off
  ```
- Gypsy supports cpp like macros and includes but needs to find cpp. Better to turn it off.
  ```
  Gve -> set def /home/comp527/gve-dir/example1/
  set defAULT-FILE-NAME /home/comp527/gve-dir/example1/
  Gve -> parse min1.gyp
  parse min1.gyp

  scope minimum_procedure =
   begin
     ...
  ```


Proceeding

- At this point, we will switch to a live demonstration.
- Remember:
  - Any sufficiently advanced technology is indistinguishable from a rigged demo.