SVA Manual

David Hopkins

Contents

1 Script Contents

An SVL script contains a series of declarations and assertions. Any line starting // will be ignored as a comment.

1.1 Variable Declarations

All variables must be declared before they can be used. Variables declared at the global level are shared between all processes. There are two basic data-types, integers and booleans and we also allow arrays of either.

```
int i, j;
bool b;
int [] is;
bool [] choosing;
int%10 [4] js;
```

In the last declaration above, the size of the array and the size of the integers in it are both specified. It declares j to be an array of length 4 containing integers from the set 0..9. If this mechanism is not used to provide size information, the defaults (see section 1.4) will be used. Non-default initial values can also be provided.

```
int count = 0, init = 1, slot;
bool[] flags = { true, false, false };
```

Additionally variables can be declared as dirty.

```
dirty int reading;
dirty bool[] okread;
```

Dirty variables are explained in section 19.1 of [1]. Model the fact that variable writes may not be atomic. If one process tries to read from a variable while another is writing to it, the outcome is non-deterministic. As well as variables, it is also sometimes useful to declare global constants.

```
const N = 5;
const maxTurns = 10;
```

1.2 Processes

The individual thread processes must be named and defined before they can be combined together.

```
P(i,j,k) = { body }
```

The identifiers `i`,`j` and `k` will be integer parameters to the process `P`. In the body they can be used as constants, but they cannot be assigned to or modified. The body itself will be made up of a sequence of imperative commands.

```
skip;
if b then c;
if b then c else d;
while b do c;
iter c;
a := v;
atomic c;
sig(channel);
isig(ichannel, i);
```

The meanings of most of these commands should be clear. The use of signals is discussed in section 1.3. `iter` iterates a command forever. It is equivalent to `while true do`. `atomic` executes a section of code atomically, blocking all other threads from proceeding.

Any boolean or integer expressions are built up from variables and constants using common operators.

```
true
false
! b1
b1 && b2
b1 || b2
i1 = 12
i1 != 12
i1 > i2
i1 >= i2
i1 < i2
i1 <= i2

i1 + i2
i1 - i2
i1 * i2
i1 / i2
i1 % i2
-i1
max(i1,i2)
min(i1,i2)
a[i1]
```

1.2.1 Local Variables

Processes can also declare their own local variables within their definition. The syntax for doing so is the same as for declaring global variables, except that they are not allowed to take non-default sizes or initial values, or be dirty. A process cannot declare two variables with the same name, even if they are in different scopes âĂŞ once a variable has been declared it is visible throughout that process. It should be noted that the compiler handles local variables by creating fresh global variables for each instantiation of each process used in the script. This may create unexpected results if a process is placed in parallel with itself using the same arguments - `< P(1),P(2)>` is fine, but `<P(1),P(1)>` may cause problems. There is also the potential for large numbers of unwanted variables to be created if a single script contains a large number of different programs.

1.2.2 Macros

Within the body of a process, we can "call" another process.

```
P(i) = a[i] := 0
Q(i) = while b do P(i)
```

This is handled by copying the body of the called process into the calling process. This can cause scoping issues if the macro uses local variables. As mentioned in section 1.2.1, a process cannot contain multiple local variables with the same name, even in different scopes. Hence, if the caller and callee both have local variables with the same name, or if a process needs to call another process in multiple places, this can cause a parse error.

It should also be noted that the arguments passed in to a process have to be constants.

1.2.3 Variable Assignments

Section 18.2 of [1] explains that in order to improve compression, the user may wish to specify that certain variables should be associated with a particular process. We allow this using the syntax below.

```
P(i) = { body } with { a[i],a[i + 1], b}
```

1.3 Signals

In order to make specification easier, processes can send CSP-like communications to the environment. We differentiate *Signals*, in which the process simply sends a prespecified event, and *ISignals* in which the process first evaluates an integer-valued expression and the sends that on a prespecified channel.

It is required that the set of signals each process uses must be disjoint. To prevent this being an awkward restriction, both Signals and ISignals can be parameterised with a number of indices. These indices must be constant

Before they can be used, signals must be declared in the global scope

```
sig wrongval;
isig result;

sig css : int, cse : int;
isig send : int.int;
```

Note that the specification of a type defines how many constant indices each channel has, not how many integer expressions can be communicated along it.

Sending a signal then takes the form

```
sig(css.i);
isig(send.i.j, a[i] + b[j]);
```

The reason we syntactically differentiate between Signals and ISignals is due to the restriction that the signals used by different processes must be disjoint. Using the channels described above, if `P()` contains `sig(css.1)`, then no other process can use the event `css.1`, but they are free to use say, `css.2`. Conversely, if `P()` contains `isig(result,1)`, then the entire channel `result` is claimed by `P()` so another process would not be allowed to communicate `result.2`.

1.4 CSP Statements

Any line which starts `%%`, will be copied straight into the CSP file. This allows specifications to be constructed in CSP which the SVL programs can be compared to.

This mechanism also need to be used to define a number of constants used to configure the compiler. These are

- `MinI` and `MaxI`, the default minimum and maximum values of the integer type.
- `ditype` and `dctype`, the default sets from which the indices and contents of integer arrays come from.
- `InitB` and `InitI`, the default initial values for booleans and integers.
- `ext_atomic`, used during refinement checking to determine whether the external thread can go into atomic sections. See section 1.6.1 (or section 19.3 of [1]).

1.5 Programs

Individual processes are combined in parallel to form *programs*.

```
Prog = <P(1),P(2),Q() >
```

As explained in section 18.2 of [1], we may wish to apply either a leaf compression or a hierarchical compression. To take full advantage of this we may need to be able to combine our processes together in a tree like manner.

```
Struct =
  hierarchCompress <<P(0),P(1)>,<P(2),P(3)>,<P(4),P(5)>,P(6)>
WideStruct = leafCompress < PP(2),PP(4),Dummy()>
```

To simplify constructing large parameterised networks, we allow the following syntax.

```
WideStruct = hierarchCompress < PP(i) |  i from 1 to N>
```

1.5.1 Overseers

Overseers are described in section 19.5 of [1]. The are special processes which run with priority whenever any of their variables are written to. This can be used to maintain data invariants in order to model more complex data-types than those present in SVL.

```
WideStructOS =
  hierarchCompress < PP(2),PP(4),Dummy(),overseer LO()>
```

1.6 Assertions

Once we have combined processes together in parallel, we will wish to prove or disprove properties of the resulting system. We do this by making assertions which we then get FDR to check. Assertions can take a number of forms. One of the simplest is to assert that some boolean property always holds (or never holds) on any possible execution path.

```
assert always count <= 5 in Struct1
assert never cs[1] && cs[2] in Struct2
```

We can also assert that a particular signal never occurs.

```
assert nosignal {error} in Struct1
assert nosignal{ result.1, result.2} in Struct2
```

Often we may wish to use the %% notation to define a CSP specification and then compare our programs to it.

```
assert %-SPEC [T= WideStructOS \{|error,verror|}-% in WideStructOS
```

Everything between %- and -% will be copied straight into the CSP script. Since we do not attempt to parse it, this allows all sorts of assertions to be made. However, because the contents is not parsed, the annotation `in WideStructOS` is required so that the GUI can correctly interpret any resulting counterexample.

1.6.1 Refinement

Refinement between processes is discussed in detail in section 19.3 of [1]. There are three kinds of refinement considered, depending on the context a process is to be used in: sequential, parallel, and general. Sequential contexts are those when a process `P`is to be run atomically. No other process will be able to read or write to any variables while `P` runs, but we assume that they could set the values beforehand and detect any modifications `P` makes. Parallel contexts are when `P` is to be run at the level of final parallel composition. In this case, nothing runs before or after `P`, but while it is running other processes may be able to read or write to some of its variables. Finally, a general context is a combination of the two. This represents the case when `P` is part of a larger thread but is not forced to be atomic. Some of its variables may be accessed before and after by the preceding or subsequent code in its thread, while others may be accessed by other processes.

In order to use this feature, the user has to define which variables can be read or written to. We differentiate between accesses arising from sequential and parallel contexts.

```
SeqWrites = {turn[3]}
SeqReads = {turn[3]}
ParReads = {turn}
ParWrites = {turn[1],turn[2],turn[4]}
```

The boolean parameter `ext_atomic`, (which the user must set using %%) also affects refinements in parallel or general contexts. If it is true, then the parallel processes are allowed to enter atomic sections, blocking `P` and allowing them to get a snapshot of the current state of all variables.

Having set these parameters, refinement assertions have the following form:

```
assert MaxA(3) [S= MaxB(3)
assert MaxB(3) [P= MaxC(3)
assert MaxD(3) [G= MaxA(3)
```

2 Using the GUI

From SVA's directory, the GUI can be compiled by running `ant`, after which it can be launched with the command `ant run`.

Once running, an SVL file can be loaded. The buttons "Check Assertion" and "Check All Assertions" can be used to run FDR and interpret the results. If FDR finds a counterexample to an assertion, this will be displayed in the lower part of the window. The checkboxes can be used to hide some of the events occurring in a trace — while these events can be crucial to why an assertion fails, but often just clutter up the trace and make it harder to read.

To prevent having to repeat long runs of FDR, once results have been obtained they can be saved and loaded again later.

In order to allow the CSP generated to be analysed separately in FDR, there is an option to save the CSP file. This should be saved to a directory containing svacomp.csp, refsva.csp and compression09.csp.

The status window shows everything with FDR outputs on its error stream. This can be useful for monitoring the status of FDR and debugging. However, to get more useful information, it may be necessary to modify fdrDirect.tcl to read either `config report medium` or `config report full`.

3 Requirements

SVA requires at least Java 1.6, FDR2.90 and ant. Binaries for these must be in the PATH.

References

[1] A. W. Roscoe, *Understanding Concurrent Systems*. Springer, 1st Edition, 2010.