AAREADME Reference Manual

Generated by Doxygen 1.4.6

Fri Oct 6 09:19:51 2006

Contents

1 Supervisory Control Problem: An overview of the examples **1**

1.1 Directory contents 1

1.2 How to rebuild the complete demo 2

1.3 Principles of the technique 3

Chapter 1

Supervisory Control Problem: An overview of the examples

Contents:

- **Directory contents**(p. 1)
- **How to rebuild the complete demo**(p. 2)
- **Principles of the technique**(p. 3)

1.1 Directory contents

All the examples are built on the same scheme. In each directory (CAT_AND_MOUSE, AGV, ...) you will find the following content:

- **AAREADME** The directory of the documentation.

- **vt.gpk** is the main Signal file. It has to be loaded by the graphical user interface "polychrony". This Signal program contains different processes.

  |

  |- CONTEXT: used to perform Simulation

  |

  |- VT_*Foo:* your application

  |

  |- vt: This process is simply the process VT_*Foo* to which you add some SIGALI functions and/or some assertions that have to be checked by the inputs. The name is the same for technical reasons than the main program. This is this one that you have to compile in order to obtain the corresponding polynomial dynamical system on which synthesis will be performed.

- **vt.sim**, **vt.res** are file generated by Sigali in order to perform simulation. They contains in a internal format the result of the sigali computations (i.e. the controller). They will be used by the polychrony tool to encapsulate the controller within the global Signal program (see below).

  These files have been imported here from vt/ directory (see below).

- **vt** Directory used by the Polychrony compiler (code generation: C code, z3z code).
- **vt.PAR** parameters of the Signal program.
- **Spec_Liaison.dir** Directory that contains the C/JAVA inteface for the simulation.
- **Demo** Directory that contains some specific java programs for the simulation.

Unix:

- **makeLib** script that
  - compiles the Signal program automatically produced after the "resolver importation" under polychrony Graphical user Interface,
  - then produces the dynamic library for the simulation.
  - then generates a dynamic library for simulation (libVTAGVLIB.$*$ file).
- **Makefile**, **Makefile_MacOs** makefile description, referenced by makeLib command.
- **run_demo** script used for launching the demo.

Windows:

- **makeLib.bat** Similar to makeLib for Windows. Generated library: VTAGVLIB.dll
- **Makefile.win** Similar to Makefile for Windows.
- **run_demo.bat** script used for launching the demo.

1.2 How to rebuild the complete demo

1. Launch "polychrony" GUI and load vt.gpk file.
2. Export le "internal" vt process as a textual file (for example vt.SIG as name).
3. Compile the previous vt.SIG program with "z3z" option

   ```
   signal vt.SIG -z3z
   ```

   It generates in the sub-directory vt/ the vt.z3z and vt_CMD.z3z files. (Sometimes, some modifications of vt.z3z file is indicated).

   - vt.z3z contains the description of the synchronisations of the application. (i.e the polynomial dynmical system encoding the application)
   - vt_CMD.z3z contains all the Sigali commands that have been written in the SIGNAL program.
4. Go to the vt subdirectory and call sigali tool execute (under sigali) the following commands

   ```
   ----------------
   set_reorder(2);
           read("vt_CMD.z3z");
           quit();
   ----------------
   ```

-> set_reorder(?) perform an automatic reordering of the underlying BDD. For some applications it is better to use set_reorder(1); (another kind of reordering).

5. At this point, the files vt.sim and vt.res must have been generated.
6. Goto the root directory of the example (i.e. up directory)
7. Copy the vt.sim and vt.res (generated in 4) in the "current" directory.
8. Under polychrony GUI, goto the vt_*Foo* process and load the resolver by the following command

   "Tools → prove → build_resolv " command.

   After this command, the file vt.SIG.SIG has been generated.

   IMPORTANT: Do not save the program (vt.gpk) after this action (as this program, contains some hidden lines of Signal code that have been automatically added)
9. Use the "makeLib" command (see above). This compilation will basically produce a library that will be used further for the JAVA simulation. (The compiler will produce some c files in vt directory...)
10. For simulate, execute the command

```
run_demo
```

1.3 Principles of the technique

This section is a complement to the `file:J-DEDS.pdf` publication.

First, remember that the Signal program contains different processes.

|

|- CONTEXT: used to perform Simulation

|

|- VT_*Foo:* your application

|

|- vt: This process is simply the process VT_*Foo* to which you add some SIGALI functions and/or some assertions that have to be checked by the inputs. The name is the same for technical reasons than the main program. This is this one that you have to compile in order to obtain the corresponding polynomial dynamical system on which synthesis will be performed.

The last one (**vt**) is exported in vt.SIG program. This process **must** have **vt** as name (see below). It contains SIGALI functions and in particular a call to **Simul()** SIGALI function.

For example, the following text is extracted from an example:

```
| (| SIGALI(Controllable(DoorState_Cat_1))
     | SIGALI(Controllable(DoorState_Cat_2))
     | SIGALI(Controllable(DoorState_Cat_3))
     | SIGALI(Controllable(DoorState_Cat_4))
     ...
     | SIGALI(Controllable(DoorState_Mouse_6))
     | SIGALI(S_Security(B_False(Error)))
     | SIGALI(S_Reachable(B_True(Initial_States)))
     | (| b := Simul()
        | SIGALI(b)
        | b ^= DoorState_Cat_1
        |)
     |)
```

The SIGALI function Simul() is specified by the following declaration

```
process Simul =
     ( ! boolean RESULT;
     )
 pragmas
 SIGALI ""
 COMMENT "simul(S,nom_fichier1,nom_fichier2)        "
         " creates a controller at the right format so that it can be "
         " read by the C resolver  function. The result is given by two files"
         " nom_fichier1.sim/nom_fichier2.res (Cf. Sigali User-manual for more details)"
 end pragmas
%Simul%;
```

When the compiler is called using the command *(signal -tra -z3z vt.SIG)* , the file vt_CMD.z3z is created. It contains the following code:

```
read("vt.z3z");
read("Creat_SDP.lib");
read("Bibli.lib");
PROP:B_False(S,Error);
S : S_Security(S,PROP);
PROP_721:B_True(S,Initial_States);
S : S_Reachable(S,PROP_721);
simul(S,"vt.res","vt.sim");
```

So, the name (**vt**) used in simul(S,"vt.res","vt.sim"); is the name of the model of the program vt.SIG.

The implementation of the resolver must solve the problem of the connexion between the symbolic variables of the polynomous used to represent the equations and the values of the variables in the C code (during the simulation). The solution consists in (See **here**(p. 2)) the generation of files vt.sim and vt.res:

- vt.sim : it contains data for the generating of the simulator (see below): in this file, the symbolic variables are encoded by identifiers.
- vt.res : it contains the data for the resolver and also the TDDs that implement functions and equations of the specification. In this file, the symbolic variables are encoded by integers using the same order than in vt.sim.

In these files, set of variables are defined

```
$E following by the list of the states
$Y following by the list of the uncontrolable inputs
$C following by the list of the conditions
$O following by the outputs of the controller
```

```
Example: in the vt.sim file, you can have the following line

$E Cat_Room_4 Mouse_Room_4 Cat_Room_3 Mouse_Room_3 Cat_Room_2 Mouse_Room_2 Cat_Room_1 Mouse_Room_1 Cat_Roo
states_1 states_2 states_3 states_4 states_6 states_8 states_9 states_10 states_11 states_12

and in the vt.res file, you can have the following line

$E 15 34 9 41 14 24 8 35 0 31 1 3 18 5 16 25 36 32 44 40
```

When the user executes the command (under the GUI of Polychrony) *Tools -> prove -> build_resolv* on the model called VT*_Foo*, Polychrony integrates automatically some SIGNAL code in this model by **analyzing** the file vt.sim (*this name is predefined in the software, it is the reason why this name (vt) is important*).

In this model (VT*_Foo*) there is a model RESOLVER that references the external resolver (resolver model). The Signal code of the RESOLVER model is :

```
process RESOLVER =
      { integer ncond, nx, nu, ny; }
      ( ? [ncond]integer cod_cond;
          [nx]integer cod_x;
          event TTick;
        ! [nu]integer cod_u;
          [ny]integer cod_y;
          event Tick;
      )
    (| (| (| S_cod_cond := cod_cond cell TTick
           | S_cod_x := cod_x cell TTick
           | resolver{}
           |)
       | (| Z_S_cod_u := S_cod_u$1 init [{i to nu}:0]
          | cod_u := Z_S_cod_u when Tick
          | Z_S_cod_y := S_cod_y$1 init [{i to ny}:0]
          | cod_y := Z_S_cod_y when Tick
          |)
       | (| (| b := (when fin_resolver) default false
             | z_b := b$1
             | b ^= TTick
             |)
          | Tick := when z_b
          |)
       |) |)
    where
    boolean z_b init true, b;
    [nx]integer S_cod_x;
    [nu]integer S_cod_u;
    [nu]integer Z_S_cod_u;
    [ny]integer S_cod_y;
    [ny]integer Z_S_cod_y;
    [ncond]integer S_cod_cond;
    boolean fin_resolver;
    process resolver =
         ( ? [ncond]integer S_cod_cond;
             [nx]integer S_cod_x;
           ! [nu]integer S_cod_u;
             [ny]integer S_cod_y;
             boolean fin_resolver;
         )
         spec (| S_cod_cond ^= S_cod_x ^= S_cod_u ^= S_cod_y ^= fin_resolver |)

    ;
    end ;
```

The generation consists in

- the calling of the RESOLVER model by fixing the values of the parameters (ncond, nx, nu, ny) and the definition of the inputs (cod_cond, cod_x). TTick is the master clock of the program.
- the definition of the outputs from the returned values by the RESOLVER (cod_u, cod_y). Tick is the clock at which the outputs are available.

All these informations are extracted from the vt.sim files. For examples,

- For a variable Mvt_Mouse_1 that appears in the set ($Y) of vt.sim , the following definition is produced

```
Mvt_Mouse_1 := (true when (cod_y[0]=1)) default (false when (cod_y[0]=(-1))) default false
```

- For a variable DoorState_Cat_3 that appears in the set ($O) of vt.sim, the following definition is produced

```
DoorState_Cat_3 := (true when (cod_u[2]=1)) default (false when (cod_u[2]=(-1))) default fal
```

- For a state variable Ci that appears in the set ($E) at the i-th rank of vt.sim, the following definition is produced INTERMi := (1 when Ci default (2 when (not Ci)) default (0 when Tick) and you will find INTERMi as the i-th element in the definition of cod_x

```
cod_x := [[0] : INTERM0, ... [i] : INTERMi, ...]
```