DiVinE

0.8.0

Generated by Doxygen 1.5.6

Tue Jul 21 09:09:33 2009

Contents

Chapter 1

Quick Guide Through the DVE Specification Language

DVE specification language was created as easy-to-interpret language with sufficient power to express many problems for model checking. In the directory `divine/examples` you can find a set of case studies already modelled for DiVinE in C language.

Philosophy of DVE language:

You want to model a system. System is composed from processes. Processes can transit from one process state to another through transitions. Transitions can be guarded by a condition - this condition says when the transition can be activated.

Transitions can be synchronized through (named) channels. There are always just 2 processes which can be synchronized in one moment. When more than 2 processes can synchronize through the same channel, there are possible all combinations of such processes to synchronize, but always just 2 of them in one moment. During the synchronization through the channel the value can be optionally transmitted between processes.

The transitions have co called "effects". Effects are assignments to local or global variables. Two synchronized processes should not be able to assign the same variable - it is an error in your model!

The system can be synchronous or asynchronous.

Uniqueness of identifiers

The namespace is common for channels, variables, processes and states. Identifier has to be unique in the current scope of visibility. It means that e. g. when variable **A** is declared in the process **P1** then there cannot be more variables **A** and states **A** in that process and more global variables **A**, channels **A** and processes **A**. But there may be another variable called **A** in the process **P2**.

Syntax elements:

Processes:

Declaration - short example:

```
process My_really_nice_process
{ <code of a process> }
```

Variables:

Variables can be global (declared at the beginning of DVE source) or local (declared at the beginning of a process). They can be of `byte` or `int` type. E.g.:

```
byte A[9];
int i,j;
```

Constants:

Constants can be declared identically as variables using the keyword `const`:

```
const byte k = 3;
```

This time constants cannot be used as parameters in delarations of arrays. For example this construct is erroneous:

```
byte field[k];
```

Process states:

Declared after declaration of variables. You also have to write which of declared process states is the initial one. You can also optionally write which of them are accepting (but you probably will not need this feature at this moment). E.g.:

```
state start, run, reading_input, writing_output;
init start;
accept reading_input, writing_output;
```

For purposes of modelling atomic actions, there are so called *commited states*.
Commited state of a process = state declared as commited - e. g. in the following code states `reading_input` and `writing_output` are commited:

```
state start, run, reading_input, writing_output;
init start;
commit reading_input, writing_output;
```

Commited state of a system = state of a system, where at least one process is in a commited state. If the system is in a commited state, then only processes in commited states can transit to another state. It means that the sequence of transitions beginning with a transition leading to a commited state and ending with a transition leading to a non-commited state cannot be interlaced with other transitions leading from non-commited states.
Synchronization between processes in commited states and non-commited states is ignored. Synchronization between processes (both) in commited states is permitted (but it is probably very rare).

Transitions:

Transitions are written as a trantitions from one process state to another (e. g. `run -> writing_output`). The transition can be executed only if the process in the initial process state of a transition (in the above mentioned example in a state `run`). You should also define an additional condition when the transition can be executed (keyword `guard`) and sometimes also a channel to synchonize through (keyword `sync` with followed by the channel name and `!` or `?`). There can synchronize only transitions with the same channel name and opposite operators `!` and `?`. When you want to transmit a value through the channel, you can write a value after `!` and a variable (where the value will be transmitted) after `?`. The last but not least element of transitions are effects - they are simple assignments to the variables. Example:

```
process Sender {
byte value, sab, retry;
state ready, sending, wait_ack, failed;
init ready;
trans
 ready -> sending {sync send?value; effect sab = 1 -sab; },
 sending -> wait_ack {sync toK!(value*2+sab); effect retry = 1;},
 wait_ack -> wait_ack {guard retry <2; sync toK!(value*2+sab); effect retry = retry+1;},
 wait_ack -> ready {sync fromL?;},
 wait_ack -> failed { guard retry == 2;};
}
```

Expressions:

In assignments, guards and synchronization values you can write arithmetic expressions. They can contain:

- constants: numbers, `true` , `false`
- parenthesis: (,)
- variable identifiers
- unary operators $-$, $\sim$ (= negation of bits) and `not` (= boolean negation)
- binary operators (ordered by precedence - higher line means a lower precedence):

  `imply`,
  `or`,
  `and`,
  `|`,
  `^`,
  `&`,
  `==`, `!=`,
  `<`, `<=`, `>`, `>=`,
  `<<`, `>>`,
  `-`, `+`
  `/`, `*`, `%`

  their semantics is the same as in C programming language except for boolean operators `and`, `or` and `imply` (but their meaning is obvious).

- question on a state of some process (e.g. `Sender.ready` is equal to `1` , iff process `Sender` is in a state `ready`. Otherwise it is equal to `0`).

Channels

Declarations of channels follow declarations of global variables. For example:

```
byte i_am_a_variable;
channel send, receive, toK, fromK, toL, fromL; //untyped unbuffered channels
channel {byte} b_send[0], b_receive[0]; //typed unbuffered channels
channel {byte,int} bi_send[0], bi_receive[0]; //typed unbuffered channels (transmitting 2
channel {byte,int} buf_bi_send[4], buf_bi_receive[1]; //typed buffered channels
```

There is a big difference between buffered and unbuffered channels:

- unbuffered channels can be untyped and they do not need to transmit values (they can play a role of handshake communication)
- buffered channels have to be always typed
- value trasmission or handshake using unbuffered channel is synchronous - i. e. both sending and receiving processes execute their transitions in the same transition of a system
- value trasmission using buffered channel is asynchronous - i. e. If the buffer is not full, value can be sent. If the buffer is not empty, value can be received. But this happens always in different transitions of a system.

Be aware: `channel xxx` and `channel {byte} xxx[0]` are both unbuffered channels and they behave almost the same way, but the second declaration is typed and the transmitted value is type casted (in this case to `byte`) before its transmission.

Type of a system:

Synchronous:

```
system sync;
```

or asynchronous

```
system async;
```

This declarations should be written at the end of DVE source.

Assertions:

Assertions can be written in every process just before transitions definitions. Assertion can be understood as an expression, which should be valid (evaluated to non-zero value) in a given process state. Explamle:

```
process My_really_nice_process
{
 byte hello = 1;
 state one, two, three;
 init one
 assert one: hello >= 1,
        two: hello <  1,
one: hello <  6;
 trans
   ...
}
```