DIVINE

Distributed and Parallel Verification Environment

Table Of Contents

Quick Guide Through the DVE Specification Language

DVE specification language has been created as a language that is both easy to interpret and sufficiently powerful to express interesting problems for the purpose of model checking.

A set of "case studies" is available, in the form of DVE source code, in the `examples` subdirectory of the DiVinE distribution.

Philosophy of DVE language

The basic modelling unit in DVE is a system, which is composed of processes. Processes can go from one process state to another through transitions, which can be guarded by a condition – this condition, also called a "guard" determines whether the transition can be activated.

Transitions can be synchronised through (named) channels. Only exactly 2 processes can be synchronised in one "moment" (in a single step). When more than 2 processes would be able to synchronise at the same time on a single channel, any combination of pairs can be picked at that time nondeterministically, however always just 2 of them in one moment. Through the synchronisation on a channel, a value can be optionally transmitted from one process to the other.

The transitions have so-called "effects". Effects are generally assignments to local or global variables. Two processes undergoing a synchronisation should not be able to assign to the same variable – that would be a modelling error.

A system may be synchronous or asynchronous.

Uniqueness of identifiers

There is a single common namespace for channels, variables, processes and states. Identifiers have to be unique in any given scope of visibility. It means that e.g. when variable **A** is declared in the process **P1**, then there cannot be a variable **A**, a state **A** in that process or a global variable **A**, nor a channel or a process **A**. The only other **A** variable may occur in a different process, eg. **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 declarations 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 *committed states*.

Committed state of a process = state declared as committed – e.g. in the following code states `reading_input` and `writing_output` are committed:

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

Committed state of a system = state of a system, where at least one process is in a committed state. If the system is in a committed state, then only processes in committed states can transit to another state. It means that the sequence of transitions beginning with a transition leading to a committed state and ending with a transition leading to a non-committed state cannot be interlaced with other transitions leading from non-committed states.

Synchronization between processes in committed states and non-committed states is ignored. Synchronization between processes (both) in committed states is permitted (but it is probably very rare).

Transitions

Transitions are written as a transitions 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 synchronize 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`
- parentheses: `(`, `)`
- variable identifiers
- unary operators `-`, `~` (= 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 values simultaneously)
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 transmission 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 transmission 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 declaration should be written at the end of a 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. Example:

```
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
        ...
    ;
}
```

DVE Modelling Tutorial

This tutorial will guide you through the creation of a model of the Peterson's mutual exclusion algorithm for `n` processes. This will help you understand the following concepts:

- processes, transitions,
- global and local variables,
- guards, effects.

The Peterson's algorithm

First, let us show you the algorithm itself as a C code snippet:

```
volatile int q[n] = { 0 };
volatile int turn[n] = { 0 };

void enter_critical_section(int i) {
    int j, k;

    for (j = 1; j < n; j++) {
        q[i] = j;
        turn[j] = i;

        for (k = 0; k < n; k++)
            while (k != i && (q[k] >= j && turn[j] == i));
    }

    q[i] = n;
}

void leave_critical_section(int i) {
    q[i] = 0;
}

void process(int process_id) {
    for (;;) {
        enter_critical_section(process_id);
        /* do work */
        leave_critical_section(process_id);
    }
}
```

We will create a model of the `process` procedure and, using *m4* macro language, make the model parametric.

Model, Processes, Transitions

A model in DVE consists of several processes, each of which is an extended finite state automaton. These are (usually) asynchronously composed to become an automaton of the model. The last line of an asynchronous model in DVE reads:

```
system async;
```

Let's just put this line at the end of each model, for now. Synchronous models aren't well supported yet.

A process declaration looks like this (the order of things is important):

```
process P1 {
    variables
    states
    assertions
    transitions
}
```

As an example, we declare a process which alternates between two states: inside a critical section, and outside it.

```
process P1 {
    state outCS, inCS;
    init outCS;
    trans
        outCS -> inCS {},
        inCS -> outCS {};
}
```

The process starts in state `outCS` and there's one transition from `outCS` to `inCS` and one from `inCS` to `outCS`, neither of them having any guards or effects. Indeed, performing a reachability analysis using DiVinE tells us that two states are reachable.

If we omit one of the transitions, we create a *deadlock* state – the automaton is stuck in it, not able to proceed. DiVinE is able to detect and describe such states.

Next, we'd like to have multiple similar processes, all described by one automaton but each having a different name and, possibly, some process number constant. In order to do this, we use the *m4* macro language like this:

```
define(P, `process P_$1 {
    state outCS, inCS;
    init outCS;
    trans
        outCS -> inCS {},
        inCS -> outCS {};
} ')

P(0)
P(1)
P(2)
```

If we process this definition with the `m4` command, we get a model with three processes. Reachability analysis would tell us that we have 8 states, by the way.

To make the model parametric, we shall use some predefined *m4* macros. Filename of the source of such model should end with the `.mdve` extension; usage of such files is described in the Tool guide. Finally, we get this parametric source:

```
default(N,3)

define(P, `process P_$1 {
    state outCS, inCS;
    init outCS;
    trans
        outCS -> inCS {},
        inCS -> outCS {};
} ')

forloop(i,0,N - 1, `P(i)')

system async;
```

Variables, Guards, Effects

In DVE, you may define global and process-local *variables*. There are two types: `byte` and `int`, and you may define one-dimensional arrays using the C syntax. These variables can be modified using transition *effects* and transitions are enabled/disabled based on variable values using *guards*.

The variable declarations are similar to the C syntax, you'll easily understand that from the following examples. We've already shown where process-local variable declarations go. Global variables are declared at the beginning of a DVE source.

Guards and effects go into the curly braces after a transition, like this:

```
trans
    q0 -> q1 { guard (x == 1) && (y == 2);
               effect x = x + 1, y = y - 1; };
```

For a quick overview, let's say the two functions are a bit simpler (and wrong indeed):

```
volatile int mutex = 0;

void enter_critical_section(int i) {
    while (lock == 1) /* wait */;
    lock = 1;
}

void leave_critical_section(int i) {
    lock = 0;
}
```

We shall declare a global variable `mutex` in the model as well (a `byte` type is enough, though).

```
byte lock = 0;
```

The `leave_critical_section` function is simple, it just resets the `lock` variable when leaving the critical section. We'll model this as an effect on the `inCS -> outCS` edge:

```
trans
    ...
    inCS -> outCS { effect lock = 0; };
```

To model the `enter_critical_section` function, we'll have to add a couple of states: a "waiting for `lock` becoming 0" state and a "`lock` has just become 0" state. We'll model the while loop as a cycle around the waiting state, and the assignment as a transition from the other state to `inCS`. It is very important to distinguish these two states since two processes may both reach the "`lock = 1`" line at the same time; if we modelled the assignment as a transition from the waiting state, the model would not allow such behaviour.

The whole model of this faulty lock looks like this:

```
default(N,3)

byte lock = 0;

define(P, `process P_$1 {
    state outCS, inCS, waiting, lockit;
    init outCS;
    trans
        outCS -> waiting {},
        waiting -> waiting { guard lock == 1; },
        waiting -> lockit { guard lock == 0; },
        lockit -> inCS { effect lock = 1; },
        inCS -> outCS { effect lock = 0; };
} ')

forloop(i,0,N - 1, `P(i)')

system async;
```

Now, you may want DiVinE to tell you if it is possible to get to a state where two processes are both in a critical section. An LTL property claiming that it is not is this:

```
#define c0 (P_0.inCS)
#define c1 (P_1.inCS)
#property !F (c0 && c1)
```

If we find a counterexample to this property then it is possible to reach such state and the lock is faulty. DiVinE finds such counterexample indeed. For information about using `divine-mc.combine` and `divine-mc owcty` to perform the check, please see the Tool guide.

Arrays, back to Peterson

We've shown how to create a DVE model of a simple lock in C. Let's go back to our example model of Peterson's algorithm now. You already know the needed syntax; let me repeat the way we transform a C source into a DVE model:

- while loops in C are modelled as cycles around a state; for loops may be transformed to while loops easily,
- each memory read and write should have its own transition – this way we model the possibility of being interrupted by some other process,
- global and local variables are often used without change (we usually use the smallest type that's enough, though, to save memory).

Now we will carefully rewrite the C source to DVE. Let's start by defining the global variables:

```
byte q[N];
byte turn[N];
```

These arrays contain zeros in the initial state, you may count on that. To start with simple things, we shall model the `leave_critical_section` function:

```
define(P, `process P_$1 {
    state outCS, inCS;
    init outCS;
    trans
        inCS -> outCS { effect q[$1] = 0; };
} ')
```

Next, we'll create an outline of the `enter_critical_section`. We will need to add some local variables as well.

```
byte j, k;
...
trans
    outCS -> for_j { effect j = 1; },
    for_j -> inCS { guard j == N; effect q[$1] = N; },
    ...
```

We have just modelled the initialization of the outer for loop, the end of it and the *locking* itself (`q[i] = n` means that process `i` is in the critical section). Let's go to the inside of the loop:

```
trans
    ...
    for_j -> j1 { guard j < N; effect q[$1] = j; },
    j1 -> for_k { effect turn[j] = $1, k = 0; },
    /* for_k ... */
    for_k -> for_j { guard k == N; effect j = j + 1; };
```

We combined two memory writes into one transition: `turn[j] = $1` and `k = 0` even though we said we should not. It's ok here because the `k` variable is not shared. Okay, let's finish it by defining the `for_k` transitions as well:

```
trans
    ...
    for_k -> while { guard k < N; },
    while -> while { guard k != $1 && q[k] >= j && turn[j] == $1; },
    while -> for_k { guard k == $1 || q[k] <  j || turn[j] != $1;
                     effect k = k + 1; };
```

Note that there are two transitions going from a loop state, the guards of which are negations of each other (in a for loop with +1 increments, we usually have `j < N` and `j == N`, since that works too).

Okay, we have completed the model. Let me repeat the whole thing for you:

```
default(N,3)

byte q[N];
byte turn[N];

define(P, `process P_$1 {
    byte j, k;
    state outCS, inCS, for_j, j1, for_k, while;
    init outCS;

    trans
        /* enter_critical_section */
        outCS -> for_j { effect j = 1; },
        for_j -> j1 { guard j < N; effect q[$1] = j; },
        j1 -> for_k { effect turn[j] = $1, k = 0; },
        for_k -> while { guard k < N; },
        while -> while { guard k != $1 && q[k] >= j && turn[j] == $1; },
        while -> for_k { guard k == $1 || q[k] <  j || turn[j] != $1;
                         effect k = k + 1; },
        for_k -> for_j { guard k == N; effect j = j + 1; },
        for_j -> inCS { guard j == N; effect q[$1] = N; },
        /* leave_critical_section: */
        inCS -> outCS { effect q[$1] = 0; };
} ')

forloop(i,0,N - 1, `P(i)')

system async;
```

This model satisfies the before mentioned property indeed. That means a state where two processes are both in a critical section is not reachable. Now, proceed to the Tool guide to see how to combine the model with the property and how to run the model checker on the result.

PROBDVE extension

PROBDVE extends the DVE language with probabilistic transitions, making it possible to model Markov Decision Processes. Let us describe the syntax of such transitions on an example process:

```
process P {
    byte x;
    state q0, p1, p2, q3, q4;
    init q0;

    trans
```

```
        q0 => { p1:1, p2:2 },               /* (0) */
        q0 -> q3 { guard x == 0; },         /* (1) */
        q0 -> q4 { effect x = x + 1; };     /* (2) */
}
```

Suppose this process is in state `q0`. First, we nondeterministically choose which of the three – (0), (1) or (2) – transitions we use. If it is (1) or (2), the behaviour is the same as in DVE. If it is (0), the process gets to state `p1` with probability 1/3 and to state `p2` with probability 2/3.

DVE Language Reference and Semantics

The complete DVE language reference and semantics is given in Chapter 2 of the Master Thesis of Pavel Simecek, which is available in PDF format here.

Updated »Sunday, 5-Oct-2008 15:17 CEST