DVE and meanDVE Language Specification

January 3, 2011

Contents

Chapter 1

The DVE Modeling Language

DiVinE provides support and tools for enumerative model checking (see Sections ?? and ??) which is proper especially for verification of software and models of communication protocols.

Every modeling language has to correspond to systems which are supposed to be modeled in it. Both software and communication protocols can consist of several processes running in parallel which communicate using various types of synchronization and shared memory. Different platforms and levels of abstraction can vary substantially in atomicity of instructions, therefore it has to be possible to model an arbitrarily complex operation as atomic.

The language also has to respect types used in modeled systems. Usually, only integer types and vectors of integers are taken into account because it is often necessary to abstract from any more complex data to keep the state space reasonably small. If some of more complex types are needed (e. g. real numbers), it has to solved using specialized verification tools or tool extensions [2].

The DVE modeling language is designed to model concurrent systems composed from processes. It provides communication by channels (special named elements for sending data between processes) and shared variables. Using so called *committed states* it is possible to create complex atomic operations.

The language has partially been derived from the modeling language of Uppaal [1], but DVE is focused more on the expressibility of a model than on comfort of modeling, therefore most of complex constructs and syntactic sugar have been omitted. Neither time properties needed for modeling of timed systems can be expressed in the language. The language has been designed as an intermediate language, hence writing models in it can be laborious sometimes. Nonetheless, the language is sufficiently strong to represent most of the models considered proper for our kind of verification.

The need for easier modeling in DVE (until a translation from a more congenial language is made) caused a temporary solution in the form of combination of DVE with the `m4` preprocessor allowing designers to write succinct codes using macro definitions. It also allows to define macros externally (from command line of `m4`) and thus, it is possible to instantiate models with different parameters. We often use this parametrization to get models with selected size of the state space.

The language also contains constructs supporting LTL model checking. It is possible to define a process of special kind – *property process*. It differs syntactically only in two things:

- usage of channels is forbidden,
- usage of local variables of different processes is permitted.

It affects also the semantics – each transition of an ordinary process is synchronized with one of transitions of *property process*. The synchronization implements the product of a modeled system and a never claim automaton as described in Section **??**

1.1 Example Model

Before a formal syntax and semantics is given, it is good to show the language on an intuitive level. The easiest way to explain basic language constructs is to demonstrate them on a simple example. The example is given here first as an description in words, then it is modeled as a set of finite-state automata with variables and communication channels and finally, the transcription to DVE is shown.

The example system can be described intuitively as an interaction of two objects: a man and a drink dispenser. The man can work, get a money for his work and spend the money for tea or coffee. The drink dispenser is composed from an electronic control unit and mechanic parts which can break down. Altogether the system consists of three main parallel parts: the man, the control unit and the mechanic parts.

The man can do following actions:

- be working (for money) – initial status
- go with earned money to the dispenser
- put money into the dispenser
- choose a drink and wait for its preparation
- take a drink, when it is prepared
- be happy, if the prepared drink is the chosen one
- be sad, if the prepared drink is different from the chosen one
- return to work, if he is happy

The control unit can do these actions:

- be ready and waiting for money and requests – initial status
- take money from the man
- order mechanic parts to prepare a drink, if one if chosen and money is paid

Finally, the mechanic parts are able to do the following:

- be ready for orders – initial status
- produce an ordered drink
- become ready again, if drink is taken away

The system is modelled using finite state automata with variables and communication channels (the extension of finite automata by variables is obvious, a synchronization using communication channels is described below). The model consists of three finite state machines running in parallel depicted in Figure 1.1. Nodes stand for states of machines and transitions for actions. Each action can consist from at most three parts:

- guard – the action can be executed only if guard is satisfied
- synchronization using channels (e. g. make!0) – the action is executed in parallel with another action synchronized correspondingly (e. g. with make?product); the transmission of value is optional
- effects – assignments to variables

The DVE code is a simple transcription of these automata to the text format.

The state space of the model has 26 states (see Figure 1.2). It is possible to verify that the model satisfies that the man can never be sad, which is certainly the positive property of the system (depending on the point of view of course). In spite of this good attribute the system can get to the deadlock state (none of processes can do a transition), when the man does not put in a money and requests a drink. This is surely unwanted hole in the specification and subsequently also an unwanted property of the model. To repair the model it would suffice to synchronize the transition of the control unit guarded by *not money* with the new transition of the man allowing him to return from the waiting state. This change is intuitively the same as reminding to the man to insert the money.

1.2 Short Explanation of Advanced Constructs

The previous section shows basic syntactic elements of the DVE language and the more advanced ones are omitted there. For this reason they are briefly explained here. The detailed semantics is given in Section 1.4.

1.2.1 Typed Channels

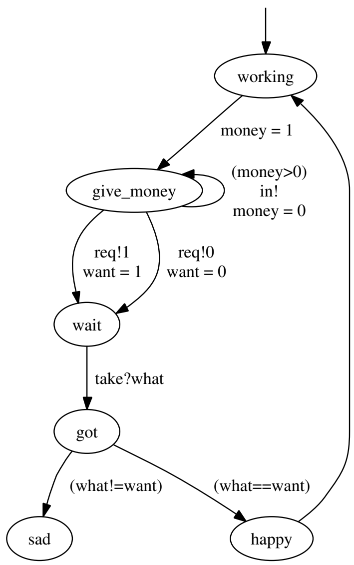
Typed channel is similar to the ordinary untyped channel (which is demonstrated in the Section 1.1). The difference is two-fold:

- typed channel can have a (potentially compound) type,
- it can also be buffered (then the value transmission is asynchronous).

Here are four sample channels `ordinary`, `simple`, `quick` and `deep` declared as follows:

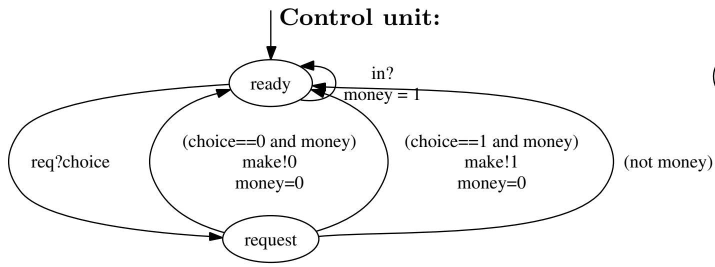
```
channel ordinary;
channel {byte} simple[0];
channel {byte,int} quick[0];
channel {byte,int} deep[4];
```

Man:



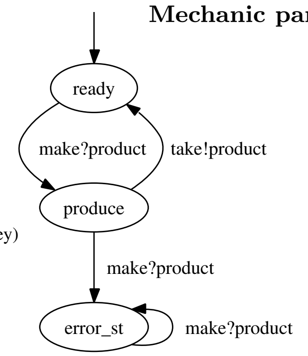

```
process man {
byte what, want, money;
state working, give_money, wait, got, happy, sad;
init working;
trans
 working   ->give_money { effect money = 1; },
 give_money->give_money { guard money>0; sync in!;
                         effect money = 0; },
 give_money->wait { sync req!0; effect want = 0; },
 give_money->wait { sync req!1; effect want = 1; },
 wait      ->got { sync take?what;},
 got       ->happy { guard what == want; },
 got       ->sad { guard what != want; },
 happy     ->working { };
}
```

Control unit:



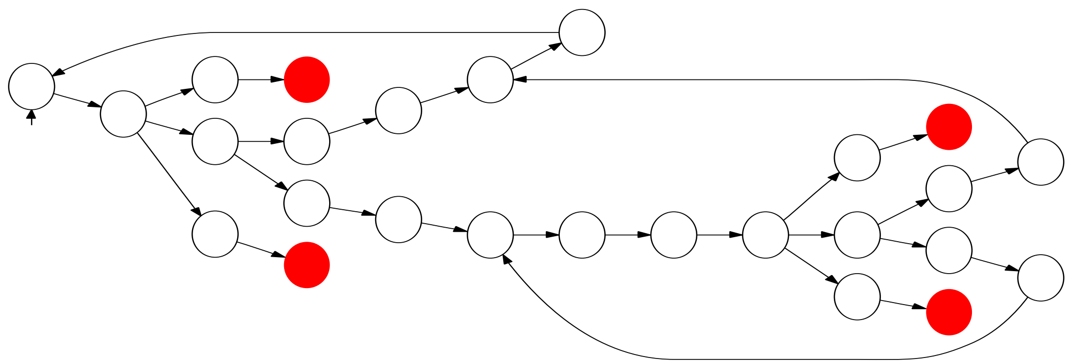

```
process control_unit {
byte money, choice;
state ready, request;
init ready;
trans
 ready  ->ready { sync in?; effect money = 1; },
 ready  ->request { sync req?choice; },
 request->ready { guard choice==0 and money;
                   sync make!0; effect money=0;},
 request->ready { guard choice==1 and money;
                  sync make!1; effect money=0;},
 request->ready { guard not money; };
}
```

Mechanic parts:




```
process mechanic_parts {
byte product;
state ready, produce,
      error_st;
init ready;
trans
 ready    -> produce
     { sync make?product; },
 produce  -> error_st
     { sync make?product; },
 error_st -> error_st
     { sync make?product; },
 produce  -> ready
     { sync take!product; };
}
```

Figure 1.1: Finite state machines of a drink dispenser and their transcriptions to DVE



28 transitions between 26 states including 4 deadlock states (highlighted).

Figure 1.2: State space of the model of a drink dispenser

Channel `ordinary` is the same as channels in Section 1.1. It can transmit a value of arbitrary type, the transmission causes a synchronization of a sender and a receiver of the value.

Channel `simple` is almost the same as `ordinary`, but before the value is transmitted, it is casted to `byte` (the type of `simple`).

Channel `quick` is very similar to `simple`, however it can transmit 2 values at one moment (in this case the first value is casted to `byte` and the second to `int`).

Channel `deep` is of compound type, thus it can transmit two values at once in the same manner as `quick`, but the transmission is asynchronous, which means that the value is inserted to a buffer (if it is not full) and kept until some process picks it up. Elements are stored in the buffer in FIFO order (buffers behave like queues).

Generally if a buffered channel is full, a transition sending a message to it cannot be executed and if the buffer is empty, a transition receiving a message from it cannot be executed. No message losses are permitted.

1.2.2 Committed States

To model more complex atomic operations simple assignments permitted in effects of transitions are not sufficient. Sometimes a repetition is needed or even a synchronization between processes. To protect other processes to interleave the operation with their own actions it is possible to mark process states used in the operation as *committed*.

Example:

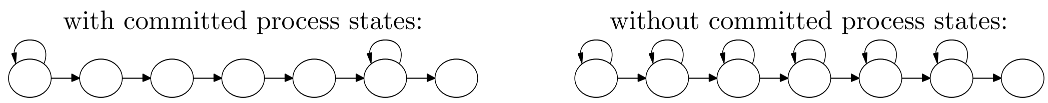


Figure 1.3: State spaces of the example with and without committed states

```
process set_parameters
{
 int result;
 state start, finish;
 init start;

 trans
 start->start
    { sync param!3; },
 start->finish
    { sync return?result;},
 start->start { };
}
```

```
process computing_power_of_2
{
 int result=1;
 int exponent;
 state receive, compute, send;
 commit compute;
 init receive;

 trans
 receive->compute { sync param?exponent; },
 compute->compute { guard (exponent!=0);
                    effect result=2*result,
                           exponent=exponent-1; },
 compute->send { guard (exponent==0); },
 send   ->receive { sync return!result;
                   effect result=1; };
}
```

Process `set_parameters` sends an exponent x to process `computing_power_of_2` and it computes 2^x. In this case $x = 3$ and `computing_power_of_2` sends back number 8. Process `computing_power_of_2` contains committed state `compute`. It means that computation of the power is an atomic action, hence it is not possible to interleave the computation with the third transition of `set_parameters` which is otherwise always executable.

This observation is demonstrated in Figure 1.3, where the left picture is a state space of the model given above and the right one shows a state space of the same model, but without committed states. In the second case in almost all states it is possible to execute the third transition of `set_parameters`, which brings on many self-loops.

Intuitively, committed states are prior to other states. If developer guarantees that always at most one process is in committed state, then a sequence of transitions from committed states of one process cannot be interleaved by actions of other process.

1.3 Concrete Syntax

Concrete syntax of DVE modeling language is given by the following set of recursive equations together with operator precedences making the syntax analysis unambiguous (see Table 1.1). Equations marked with $\diamond$ have a dynamic semantics defined in 1.4, the rest of equations have only static semantics (declarations of variables, channels, etc.):

In the following text:

id, id_1, id_2 stand for terminal symbols from
$\{a, \ldots, z, A \ldots, Z, \_\} \cdot \{0, \ldots, 9, a, \ldots, z, A \ldots, Z, \_\}^*$,
number stand for terminal symbols from $\{0, \ldots, 9\}^+$.

The entire system consists of declarations, definitions of processes and a definition of a type of the system.

DVE ::= Declaration ProcDefList System ◇

It is possible to declare variables or channels.

Declaration ::= ε | Declaration VariableDecl | Declaration Channels

Variables can be of two different integer types. They can also be scalar or vector. Keyword `Const` may be used in a declaration of constant.

VariableDecl	::=	TypeName DeclIdList ;
Const	::=	ε | `const`
TypeName	::=	Const TypeId
TypeId	::=	`int` | `byte`
DeclIdList	::=	DeclId | DeclIdList , DeclId
DeclId	::=	*id* VectorDecl VarInit
VectorDecl	::=	ε | [*number*]

It is possible to initialize variables using operator `=`. Vector variables can be initialized by a vector of values written as $\{value_1, value_2, \ldots, value_n\}$.

VarInit	::=	ε | = Initializer
Initializer	::=	Expr | { VectorInitList }
VectorInitList	::=	VectorInit | VectorInitList , VectorInit
VectorInit	::=	Expr

Channels are typed or untyped. Element sent through a typed channel may consist of several items of different types. Typed channels can also be buffered (the size of buffer is given by a positive integer).

Channels	::=	`channel` ChannelDeclList ; |
		`channel` { TypeList } TypedChannelDeclList ;
ChannelDeclList	::=	ChannelDecl | ChannelDeclList , ChannelDecl
ChannelDecl	::=	*id*
TypedChannelDeclList	::=	TypedChannelDecl |
		TypedChannelDeclList , TypedChannelDecl
TypedChannelDecl	::=	*id* [*number*]
TypeList	::=	TypeId | TypeList , TypeId

Processes are identified by a unique name. They consist of local variable declarations, a list of process states, a list of accepting process states, a list of committed process states, declaration of an initial state and a list of transitions.

ProcDefList	::=	ProcDef ProcDefList	◇
ProcDef	::=	`process` *id* { ProcBody }	◇
ProcBody	::=	ProcLocalDeclList States	◇
		InitAndCommitAndAccept Transitions	◇

First, local variables are declared.

ProcLocalDeclList ::= ε | ProcLocalDeclList VariableDecl

Second process states are declared and some of them are marked to have a special type (initial, committed and accepting).

States	::=	`state` StateDeclList ;
StateDeclList	::=	StateDecl | StateDeclList , StateDecl
StateDecl	::=	*id*
InitCommitAndAccept	::=	Init | Init CommitAndAccept | CommitAndAccept Init
Init	::=	`init` *id* ;
CommitAndAccept	::=	Commit Accept | Accept Commit | Accept | Commit
Accept	::=	`accept` AcceptList
AcceptList	::=	*id* ; | *id* , AcceptList
Commit	::=	`commit` CommitList
CommitList	::=	*id* ; | *id* , CommitList

Third the list of transitions follows. A transition leads from one process state to another and further it may contain a guard, a synchronization and effects.

Transitions	::=	ε | `trans` TransitionList ;	◇
TransitionList	::=	Transition | TransitionList , TransitionOpt	◇
Transition	::=	id_1 -> id_2 { Guard Sync Effect }	◇

A little of syntactic sugar: It is possible to omit the starting process state in a transition, if it is the same as in a transition immediately preceding the transition.

TransitionOpt	::=	-> *id* { Guard Sync Effect } | Transition	◇

A guard is simply an expression (semantically it is the condition to be fulfilled, when a transition is executed).

Guard	::=	ε | `guard` Expr ;	◇

A synchronization can be either plain or it can transmit a value. Synchronization can transmit a value also to the buffer, although the transmission is asynchronous and no synchronization between processes happens in fact.

Sync	::=	ε | `sync` SyncExpr ;	◇
SyncExpr	::=	*id* ! SyncValue | *id* ? SyncValue	◇
SyncValue	::=	ε | Expr | { ExprList }	◇
ExprList	::=	Expr | ExprList , Expr	◇

Effects consist of lists of assignments.

Effect	::=	`effect` EffList ;	◇
EffList	::=	Assignment | EffList , Assignment	◇
Assignment	::=	Expr = Expr	◇

DVE has an universal expressions used in initializations of variables, guards, synchronizations and effects. They may contain nullary operators (constants, variables, etc.), unary operators (unary minus, Boolean negation and bitwise negation) and binary operators (plus, minus, bitwise shifts, Boolean operators, etc.). The expressions are defined recursively and for the unambiguous interpretation table in Table 1.1 is needed.

Table 1.1: Operators sorted by precedence from the lowest to the highest:

1.	`imply`	Boolean implication
2.	`or`, `and`	Boolean *or*, *and*
3.	|, `&`, `^`	bitwise *or*, *and*, *xor*
4.	== !=	integer equality, integer non-equality
5.	< <= >= >	stands for integer relations $<, \leq, \geq, >$
6.	<< >>	left bit shift, right bit shift
7.	− +	subtraction, addition of integers
8.	∗ / %	multiplication, division, modulo of integers
9.	− ˜ `not`	unary *minus*, bitwise *not*, boolean *not*
10.	() [] . ->	parentheses, element of vector selection, ”process at state” test, variable of process

$$
\begin{array}{rcll}
\mathsf{Expr} & ::= & \texttt{false} \mid \texttt{true} \mid \mathit{number} \mid \mathit{id} \mid \mathit{id}\ [\ \mathsf{Expr}\ ] \mid & \diamond \\
 & & (\ \mathsf{Expr}\ ) \mid \mathsf{UnaryOp}\ \mathsf{Expr} \mid & \diamond \\
 & & \mathsf{Expr}_1 < \mathsf{Expr}_2 \mid \mathsf{Expr}_1 <= \mathsf{Expr}_2 \mid & \diamond \\
 & & \mathsf{Expr}_1 == \mathsf{Expr}_2 \mid \mathsf{Expr}_1\ != \mathsf{Expr}_2 \mid & \diamond \\
 & & \mathsf{Expr}_1 > \mathsf{Expr}_2 \mid \mathsf{Expr}_1 >= \mathsf{Expr}_2 \mid & \diamond \\
 & & \mathsf{Expr}_1 + \mathsf{Expr}_2 \mid \mathsf{Expr}_1 - \mathsf{Expr}_2 \mid & \diamond \\
 & & \mathsf{Expr}_1 * \mathsf{Expr}_2 \mid \mathsf{Expr}_1\ /\ \mathsf{Expr}_2 \mid & \diamond \\
 & & \mathsf{Expr}_1\ \%\ \mathsf{Expr}_2 \mid & \diamond \\
 & & \mathsf{Expr}_1\ \&\ \mathsf{Expr}_2 \mid \mathsf{Expr}_1 \mid \mathsf{Expr}_2 \mid & \diamond \\
 & & \mathsf{Expr}_1\ \hat{}\ \mathsf{Expr}_2 \mid & \diamond \\
 & & \mathsf{Expr}_1 << \mathsf{Expr}_2 \mid \mathsf{Expr}_1 >> \mathsf{Expr}_2 \mid & \diamond \\
 & & \mathsf{Expr}_1\ \texttt{or}\ \mathsf{Expr}_2 \mid \mathsf{Expr}_1\ \texttt{and}\ \mathsf{Expr}_2 \mid & \diamond \\
 & & \mathsf{Expr}_1\ \texttt{imply}\ \mathsf{Expr}_2 \mid & \diamond \\
 & & \mathit{id}_1\ .\ \mathit{id}_2 \mid \mathit{id}_1 -> \mathit{id}_2 \mid \mathit{id}_1 -> \mathit{id}_2\ [\ \mathsf{Expr}\ ] & \diamond \\
\mathsf{UnaryOp} & ::= & \texttt{-} \mid \tilde{}\ \mid \texttt{not} & \diamond
\end{array}
$$

A system can be declared as synchronous or asynchronous. One of processes may be marked as a property process (process implementing the never claim automaton).

$$
\begin{array}{rcll}
\mathsf{System} & ::= & \texttt{system}\ \mathsf{SystemType} & \diamond \\
\mathsf{SystemType} & ::= & \texttt{async}\ \mathsf{ProcProperty}\ \texttt{;} \mid \texttt{sync}\ \mathsf{ProcProperty}\ \texttt{;} & \diamond \\
\mathsf{ProcProperty} & ::= & \varepsilon \mid \texttt{property}\ \mathit{id} & \diamond
\end{array}
$$

There are also additional constraints put on the source code:

1. All symbols (processes, variables, channels or process states) must be declared.
2. Symbols (processes, variables, channels or process states) cannot be of the same name in the same scope of view (e. g. local variable `A` is in a conflict with global variable `A`, but it is not in a conflict with another local variable `A` declared in a different process).
3. The type of symbol has to correspond to the usage (e. g. it is not possible to use channel as variable) – there are many restrictions given by this rule:

(a) In context of Init, Accept, Commit, Transition and TransitionOpt there id must be a declared process state

(b) In context of Expr $\to id$, id must be a scalar variable

(c) In context of Expr $\to id$ [Expr], id must be a vector variable

(d) In context of Expr $\to id_1$. id_1, id_1 must be a process and id_2 must be a process state

(e) In context of Expr $\to id_1$ $->$ id_2 or id_1 $->$ id_2 [Expr], id_1 must be a process and id_2 must be a variable (scalar or vector).

4. Scalar variable cannot be initialized with a vector value and vector variable cannot be initialized with a scalar value.

5. Array size has to be at least 1 and at most 2147483647.

6. The left side of an assignment has to be a scalar variable or an element of a vector variable

7. In context of both SyncExpr $\to id$? SyncValue and SyncValue $\to$ Expr, Expr has to be a scalar variable or an element of a vector variable (and similarly for SyncValue $\to$ { ExprList }).

8. Expressions id_1 . id_2, id_1 $->$ id_2 and id_1 $->$ id_2 [Expr] are permitted only in a property process and processes used in these expressions has to be declared before the property process

9. The number of items transmitted simultaneously through a single channel must correspond to the declaration (in case of typed channels) or the first use of the channel (in case of untyped channels)

1.4 Dynamic Semantics

In this section a dynamic semantics of DVE source is set up. A static semantics is omitted for simplicity reasons (e. g. semantics of declarations is not explained) and it is be referred only using the intuition given in the Section 1.1. Equations of the concrete syntax in Section 1.3 needed for dynamic semantics are marked with $\diamond$.

First, several common notions are defined in 1.4.1. The dynamic semantics is given almost exclusively by transitions of processes. Therefore denotational semantics of transitions is given in Section 1.4.3. Transitions contain a lot of expressions. For this reason denotational semantics of expressions is defined in Section 1.4.2. Finally, the small step dynamic operational semantics of the entire DVE system is described in Section 1.4.4 using semantics of transitions and states of processes.

One may notice, that no *abstract syntax* is given here to make the definition of semantics easier. But, as the abstract syntax would be almost precisely the same as the concrete syntax excluding terminal symbols, the semantics is defined directly for the concrete syntax. This way we also avoid the need for definition of correspondence between abstract and concrete syntax.

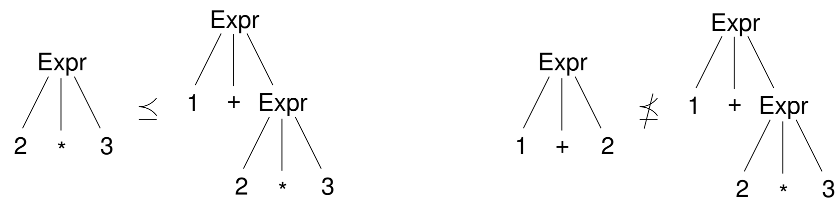


Figure 1.4: Example of relation $\preceq$ between expressions

1.4.1 Common Definitions and Conventions

Definition 1.4.1. $\mathcal{L}(N)$ is the language of all terms derivable from non-terminal N.

Remark 1.4.2. *Convention:* The name of a non-terminal in lower case letters denotes a term from a language given by the non-terminal. E. g. *guard*, $guard_1$, $guard'$ denote terms from $\mathcal{L}(\mathsf{Guard})$.

Definition 1.4.3. Let t_1 and t_2 are trees. Then $t_1 \preceq t_2$, if and only if t_1 is a subtree of t_2.

Let w_1 and w_2 are words. Then $w_1 \preceq w_2$, if and only if $t_1 \preceq t_2$ and t_1, t_2 are syntax trees of w_1, w_2.

Remark 1.4.4. E. g. $2 * 3 \preceq 1 + 2 * 3$, but $1 + 2 \npreceq 1 + 2 * 3$ (see Figure 1.4) because multiplication has a higher priority than addition.

Definition 1.4.5. *System state* σ is a function mapping:

- scalar variable name to its value (names of variables are always understood in context of current scope of their visibility); $pr :: id$ denotes a variable id in context of process pr,
- vector variable name to the vector of values indexable by integers; $pr :: id$ denotes a variable id in context of process pr,
- process name to the name of its current state,
- channel name to the list of vectors of values contained in it (for unbuffered channels too - it is needed for the easy assembling of value transmission to the semantics of DVE system).

Remark 1.4.6. In the following text σ, σ', ... stand for a system state.

1.4.2 Expressions

The denotation semantics of all terms from $\mathcal{L}(\text{Expr})$ is defined as follows (the semantics of used operator symbols can be found in Table 1.1):

$[\![\texttt{false}]\!](\sigma) = 0$
$[\![\texttt{true}]\!](\sigma) = 1$
$[\![number]\!](\sigma) = number$
$[\![id]\!](\sigma) = \sigma(id)$

$[\![id \ [\ expr\ ]]\!](\sigma) = \sigma(id)([\![expr]\!](\sigma))$ ... id is vector variable and $[\![expr]\!](\sigma)$ is an index to it
$[\![\texttt{(}\ \ expr\ \ \texttt{)}]\!](\sigma) = [\![expr]\!](\sigma)$
$[\![unary\_op\ \ expr]\!](\sigma) = unary\_op\ [\![expr]\!](\sigma)$ ... $unary\_op \in \{-,\ \tilde{}\ ,\ \text{not}\}$
$[\![expr_1\ \ binary\_op\ \ expr_2]\!](\sigma) = [\![expr_1]\!](\sigma)\ \ binary\_op\ \ [\![expr_2]\!](\sigma)$
... $binary\_op \in \{<, <=, ==, !=, >, >=, +, -, *, /, \%, \&, |, \hat{}\ , <<, >>, \text{or}, \text{and}, \text{imply}\}$,
relational and Boolean operators (e. g. == or `or`) return always 0 or 1.

$$[\![id_1\ \texttt{.}\ id_2]\!](\sigma) = \begin{cases} 1 & \sigma(id_1) = id_2 \ldots \text{which means: process } id_1 \text{ is in its state } id_2 \\ 0 & \text{otherwise} \end{cases}$$

The following is the same as $[\![id]\!](\sigma)$ and $[\![id\ [\ expr\ ]]\!](\sigma)$ except for the context of process, where the name of variable is interpreted:
$[\![id_1 \texttt{ -> } id_2]\!](\sigma) = \sigma(id_1 :: id_2)$
$[\![id_1 \texttt{ -> } id_2 \texttt{ [} expr \texttt{] }]\!](\sigma) = \sigma(id_1 :: id_2)([\![expr]\!](\sigma))$ ... id_2 is vector variable and $[\![expr]\!](\sigma)$ is an index to it

Remark 1.4.7. Because the current implementation contain only a poor type system (8-bit unsigned and 16-bit signed integer), the evaluation of expressions is made in the following the way: arguments of an operator are first casted to the 32-bit signed integers and then the corresponding standard C++ operator is applied. This will be fixed in the future (see Chapter **??**).

Remark 1.4.8. The usage of variables *unary_op* and *binary_op* is not type correct, but it is used this way because the correspondence between operators and their syntactic representation is obvious.

1.4.3 Transitions

Process transitions change a system state in three ways:

1. changes a process state to the ending process state of the transition,
2. changes a content of buffers of channels.
3. changes values of variables,

The semantics of transitions follows the division depicted above.
Let $transition \equiv id_1 \texttt{ -> } id_2\ \{guard\ sync\ effect\}$ and $effect \equiv assignment_1, \ldots, assignment_n$.

1. $[\![id_1 \texttt{ -> } id_2]\!](\sigma) = \sigma[parent(transition)/id_2]$

2. $$[\![sync]\!](\sigma) = \begin{cases} \sigma \\ \text{if } sync \equiv\ \varepsilon \\ \\ \sigma[id_{ch}/id_{ch} \cdot \sigma(syncvalue)] \\ \text{if } sync \equiv \texttt{sync}\ id_{ch}\texttt{!}\,syncvalue \\ \\ \sigma[id_{ch}/tail(id_{ch}), syncvalue/head(id_{ch})] \\ \text{if } sync \equiv \texttt{sync}\ id_{ch}\texttt{?}syncvalue \end{cases}$$

3. $$[\![assignment]\!](\sigma) = \begin{cases} [\![id_{var}/[\![expr_{value}]\!](\sigma)]\!](\sigma) \\ \text{if } assignment \equiv id_{var} \texttt{ = } expr_{value} \\ [\![id_{var}([\![expr_{index}]\!](\sigma))/[\![expr_{value}]\!](\sigma)]\!](\sigma) \\ \text{if } assignment \equiv id_{var}\,\texttt{[}expr_{index}\texttt{]} \texttt{ = } expr_{value} \end{cases}$$

Remark 1.4.9. The definition of $[\![sync]\!](\sigma)$ is little simplified by ignorance of structure of *syncvalue* because it can be empty or it can be a tuple of values of various types. The transmission of such values through channels is defined in a natural way - item by item - technical details are omitted.

Finally,

$[\![transition]\!](\sigma) = \sigma'$ where

- $\sigma' = [\![assignment_n]\!]([\![assignment_{n-1}]\!](\dots [\![assignment_1]\!](\sigma_2)\dots))$ (if $n = 0$, then $\sigma' = \sigma_2$)
- $\sigma_2 = [\![sync]\!](\sigma_1)$
- $\sigma_1 = [\![id_1 \texttt{ -> } id_2]\!](\sigma)$

1.4.4 System

Small step operational semantics of the DVE system strongly depends on the semantics of transitions. System state changes in each step using a semantics of several transitions.

Transitions can be either *enabled* or *disabled* depending on a state of the system. A transition is understood to be enabled precisely if it is permitted to execute effects of this transition in a given system state (i. e. the process owning the transition is in a proper state, guard of the transition is satisfied and optional synchronization can be performed).

For this purpose functions *PartEnabled* and *SyncReceiving* are defined in the following paragraphs. *PartEnabled* returns true if and only if the transition leads from the current process state and its guard is satisfied. Function *SyncReceiving* returns a set of processes receiving a data from the given transition through a common channel in a single transition of the system. Now formal definitions follow:

Let *States* denote the set of system states of and

$parent(transition) = \langle$name of process, where *transition* is defined$\rangle$.

For the simplicity reasons we abstract from the difference between $\mathcal{L}$(Transition) and $\mathcal{L}$(TransitionOpt). Anyway the only difference is, that transitions described by terms from $\mathcal{L}$(TransitionOpt) have no starting process state. This missing state is then assumed to be equal to the starting state of the preceding transition in the transition list.

Then the types of mentioned functions are:

$PartEnabled : \mathcal{L}(\text{Transition}) \times States \to Boolean$
$SyncReceiving : \mathcal{L}(\text{Transition}) \times States \to 2^{\mathcal{L}(\text{Transition})}$

Let $transition \equiv id_1$ `->` id_2 $\{$ *guard sync effect* $\}$ and let *dve* denote a fixed source of DVE system code, such that $transition \preceq dve$.

Then $PartEnabled(transition, \sigma) =$

- $true$ if ($guard \equiv \varepsilon$ or $[\![guard]\!](\sigma) \neq 0$) and $[\![parent(transition).id_1]\!](\sigma) \neq 0$.
- $false$ otherwise.

Function *SyncReceiving* is defined separately for 2 cases:

1. If $sync = \varepsilon$ or $sync =$ `sync` id`!`$\dots$ [1] , then $SyncReceiving(transition, \sigma) = \emptyset$
2. If $sync =$ `sync` id`?`$\dots$ [1] and id is not a buffered channel, then

$$\begin{aligned} SyncReceiving(transition, \sigma) = \{transition' \mid\ & transition' \preceq dve \wedge \\ & parent(transition') \neq parent(transition) \wedge \\ & PartEnabled(transition') \wedge \\ & sync' \preceq transition' \wedge sync' \equiv \texttt{sync}\ id\texttt{!}\dots\} \end{aligned}$$

Transitions can be also *prioritized* or not, depending on their starting process state.

Definition 1.4.10. Let function $Prioritized : \mathcal{L}(\text{Transition}) \times States \rightarrow Boolean$ is defined as follows:

$$Prioritized(t, \sigma) = \begin{cases} true & \text{if the starting process state } id_1 \text{ of transition } t \text{ is declared} \\ & \text{as } \textit{committed} \text{ and } [\![parent(t).id_1]\!](\sigma) \neq 0, \\ false & \text{otherwise.} \end{cases}$$

Furthermore, transitions can require synchronization or not.

Definition 1.4.11. Let function $SyncReq : \mathcal{L}(\text{Transition}) \rightarrow Boolean$ is defined as follows:

$$SyncReq(t) = \begin{cases} false & t \equiv id_1 \texttt{ -> } id_2\ \{\ guard\ effect\ \} \\ & \dots \text{i. e. part with synchronization is missing} \\ true & \text{otherwise} \end{cases}$$

Dynamic operational semantics of the entire source code is defined as follows:

1. If the system is declared as asynchronous without property - i. e. `system async` $\preceq dve$, then its semantics is defined as follows:

$$\begin{aligned} \sigma_1 \xrightarrow{t} \sigma_2 \Leftrightarrow\ & t \in \mathcal{L}(\text{Transition}), t \preceq dve \wedge PartEnabled(t, \sigma) \wedge \neg SyncReq(t) \wedge \\ & [\![t]\!](\sigma_1) = \sigma_2 \wedge (Prioritized(t, \sigma_1) \vee \\ & \qquad \nexists t' \in \mathcal{L}(\text{Transition}) : Prioritized(t', \sigma_1)) \end{aligned}$$

---

[1] We only care of *id* denoting a channel and a type of synchronization, the value transmission does not matter. Moreover, the matching of counts of transmitted values is guaranteed by the syntax analysis.

$$\begin{aligned}\sigma_1 \xrightarrow{t_1,t_2} \sigma_2 \Leftrightarrow\ & t_1, t_2 \in \mathcal{L}(\text{Transition}), t_1, t_2 \preceq \mathit{dve} \wedge \\ & \mathit{PartEnabled}(t_1, \sigma) \wedge \mathit{PartEnabled}(t_2, \sigma) \wedge \\ & t_2 \in \mathit{SyncReceiving}(t_1, \sigma) \wedge [\![t_1]\!]([\![t_2]\!](\sigma_1)) = \sigma_2 \wedge \\ & ((\mathit{Prioritized}(t_1, \sigma_1) \wedge \mathit{Prioritized}(t_2, \sigma_1)) \vee \\ & \ \nexists t' \in \mathcal{L}(\text{Transition}) : \mathit{Prioritized}(t', \sigma_1))\end{aligned}$$

2. If the system is declared as asynchronous with property - i. e.

$$\texttt{system async property } \mathit{id}_{\mathit{prop}} \preceq \mathit{dve},$$

   then its semantics is defined in the following way:

$$\begin{aligned}\sigma_1 \xrightarrow{t,t_p} \sigma_2 \Leftrightarrow\ & t, t_p \in \mathcal{L}(\text{Transition}), t, t_p \preceq \mathit{dve} \wedge \mathit{parent}(t) \neq \mathit{parent}(t_p) = \mathit{id}_{\mathit{prop}} \wedge \\ & \mathit{PartEnabled}(t, \sigma) \wedge \mathit{PartEnabled}(t_p, \sigma) \wedge \neg \mathit{SyncReq}(t) \wedge \\ & [\![t]\!](\sigma_1) = \sigma_2 \wedge (\mathit{Prioritized}(t, \sigma_1) \vee \\ & \qquad\qquad \nexists t' \in \mathcal{L}(\text{Transition}) : \mathit{Prioritized}(t', \sigma_1))\end{aligned}$$

$$\begin{aligned}\sigma_1 \xrightarrow{t_1,t_2,t_p} \sigma_2 \Leftrightarrow\ & t_1, t_2, t_p \in \mathcal{L}(\text{Transition}), t_1, t_2, t_p \preceq \mathit{dve} \wedge \mathit{parent}(t_p) = \mathit{id}_{\mathit{prop}} \wedge \\ & \mathit{parent}(t_1) \neq \mathit{id}_{\mathit{prop}} \wedge \mathit{parent}(t_2) \neq \mathit{id}_{\mathit{prop}} \\ & \mathit{PartEnabled}(t_1, \sigma) \wedge \mathit{PartEnabled}(t_2, \sigma) \wedge \mathit{PartEnabled}(t_p, \sigma) \\ & t_2 \in \mathit{SyncReceiving}(t_1, \sigma) \wedge [\![t_1]\!]([\![t_2]\!](\sigma_1)) = \sigma_2 \wedge \\ & ((\mathit{Prioritized}(t_1, \sigma_1) \wedge \mathit{Prioritized}(t_2, \sigma_1)) \vee \\ & \ \nexists t' \in \mathcal{L}(\text{Transition}) : \mathit{Prioritized}(t', \sigma_1))\end{aligned}$$

3. Let $p_1, \ldots, p_n \in \mathcal{L}(\text{Process})$ denote all processes of interpreted DVE system dve (i.e. $\forall i : p_i \preceq \mathit{dve}$). If the system is declared as synchronous - i. e.

$$\texttt{system sync } \mathit{procproperty} \preceq \mathit{dve},$$

   then its semantics is defined as follows:

$$\sigma_1 \xrightarrow{t_1,\ldots,t_n} \sigma_2 \Leftrightarrow \forall i, 1 \leq i \leq n : t_i \in \mathcal{L}(\text{Transition}) \wedge t_i \preceq p_i \wedge \mathit{PartEnabled}(\mathit{transition})$$

Chapter 2

The meanDVE Modeling Language

2.1 Concrete Syntax

Concrete syntax of meanDVE Modeling language follows the one of DVE [3] (or Chapter 1) with only the syntax of transitions augmented in the following way:

$$\begin{array}{rcl} \mathsf{Transitions} & ::= & \varepsilon \mid \texttt{trans}\ \mathsf{TransitionList}\ \texttt{;} \\ \mathsf{TransitionList} & ::= & \mathsf{Transition} \mid \mathsf{TransitionList}\ \texttt{,}\ \mathsf{TransitionOpt} \\ \mathsf{Transition} & ::= & id_1\ \texttt{-->}\ id_2\ \{\ \mathsf{Guard\ Sync\ Effect\ Cost\ Time}\ \} \end{array}$$

Cost (or weight) of a given transition, semantically representing the computational complexity of the transition:

$$\mathsf{Cost} \quad ::= \quad \varepsilon \mid \texttt{cost}\ number\ number\ \texttt{;}$$

And finally time, specifying how many time units does the transition last:

$$\mathsf{Time} \quad ::= \quad \varepsilon \mid \texttt{time}\ number\ \texttt{;}$$

2.2 Dynamic Semantics

There are various combinations of model properties that are meaningless, but by specifying them as exceptions we would decrease the readability of syntactic rules, e.g. property process transitions are not allowed to have either cost or time specified.

2.2.1 Common Definitions and Conventions

Definition 2.2.1. $\mathcal{L}(N)$ is the language of all terms derivable from non-terminal N.

Definition 2.2.2. Let t_1 and t_2 are trees. Then $t_1 \preceq t_2$, if and only if t_1 is a subtree of t_2.

Let w_1 and w_2 are words. Then $w_1 \preceq w_2$, if and only if $t_1 \preceq t_2$, where t_1, t_2 are syntax trees of w_1, w_2.

Definition 2.2.3. *System state* σ is a function mapping:

- scalar variable name to its value,
- vector variable name to the vector of values indexable by integers,
- process name to the name of its current state,

- channel name to the list of vectors of values contained in it.

Let Σ be the set of all states.

Definition 2.2.4. Mean Büchi automaton (BA) is $\mathcal{A}_m = (Q, \delta, F, s)$, where

- $Q \subseteq \Sigma$ set of states,
- $\delta \subseteq Q \times (\mathbb{Q}^+ \times \mathbb{N}) \times Q$ transition function,
- $F \subseteq Q$ set of accepting states,
- $s \in Q$ initial state.

Definition 2.2.5. Several type-definition of support functions:

- $\mathit{parent} : \mathcal{L}(\text{Transition}) \to \mathcal{L}(\text{Process})$
- $\mathit{PartEnabled} : \mathcal{L}(\text{Transition}) \times \mathit{States} \to \mathit{Boolean}$
- $\mathit{SyncReceving} : \mathcal{L}(\text{Transition}) \times \mathit{States} \to 2^{\mathcal{L}(\text{Transition})}$
- $\mathit{Prioritised} : \mathcal{L}(\text{Transition}) \times \mathit{States} \to \mathit{Boolean}$
- $\mathit{SyncReq} : \mathcal{L}(\text{Transition}) \to \mathit{Boolean}$

Remark 2.2.6. For $t \in \mathcal{L}(\text{Transition})$ let $t.\mathfrak{c}$ denote $q \in \mathbb{Q} : (\texttt{cost}\ a\ b \preceq t \wedge q = \frac{a}{b}) \vee q = 0$ and $t.\mathtt{t}$ denote $n \in \mathbb{N} : \texttt{time}\ n \preceq t \vee n = 1$.

2.2.2 Model Semantics

In the following we will construct BA $\mathcal{A}_m$ for meanDVE model $\mathcal{M}$ inductively:

1. initial state $s := \sigma \in Q$,
2. if $\sigma \in Q, t \subseteq \mathbb{Q}^+ \times \mathbb{N}, \sigma' \in \delta(Q, t)$, then $\sigma' \in Q$,
3. if $\texttt{property}\ \mathit{id}_{prop} \preceq \mathcal{M}$ and $\texttt{accept}\ \sigma(\mathit{id}_{prop}) \preceq \mathit{id}_{prop}$, then $\sigma \in F$
4. nothing else is in Q and F.

Then $\mathcal{A}_m = (Q, \delta|_{Q\times(\mathbb{Q}^+\times\mathbb{N})\times Q}, F, s)$, where $\delta \subseteq \Sigma \times (\mathbb{Q}^+ \times \mathbb{N}) \times \Sigma$ is a transition function satisfying

1. if $\texttt{system async ;} \preceq \mathcal{M}$

$$
\begin{aligned}
\delta(\sigma_1, q, n, \sigma_2) \Leftrightarrow (\ & \exists t \in \mathcal{L}(\text{Transition}) : t \preceq \mathcal{M}\ \wedge \\
& \mathit{PartEnabled}(t, \sigma_1)\ \wedge \\
& \neg \mathit{SyncReq}(t)\ \wedge \\
& [\![t]\!](\sigma_1) = \sigma_2\ \wedge \\
& (\mathit{Prioritised}(t, \sigma_1) \vee \nexists t' \in \mathcal{L}(\text{Transition}) : \mathit{Prioritised}(t', \sigma_1))\ \wedge \\
& q = t.\mathfrak{c}\ \wedge \\
& n = t.\mathtt{t}\ )\ \vee \\
(\ & \exists t_1, t_2 \in \mathcal{L}(\text{Transition}) : t_1, t_2 \preceq \mathcal{M}\ \wedge
\end{aligned}
$$

$$\begin{aligned}
&\mathit{PartEnabled}(t_1, \sigma_1) \wedge \\
&\mathit{PartEnabled}(t_2, \sigma_1) \wedge \\
&t_2 \in \mathit{SyncReceiving}(t_1, \sigma_1) \wedge \\
&[\![t_1]\!]([\![t_2]\!](\sigma_1)) = \sigma_2 \wedge \\
&(\ (\mathit{Prioritised}(t_1, \sigma_1) \wedge \mathit{Prioritised}(t_2, \sigma_1)) \vee \\
&\ \ \nexists t' \in \mathcal{L}(\text{Transition}) : \mathit{Prioritised}(t', \sigma_1)\ ) \wedge \\
&q = t_1.\mathfrak{c} + t_2.\mathfrak{c} \wedge \\
&n = \max\{t_1.\mathfrak{t}, t_2.\mathfrak{t}\}\ )
\end{aligned}$$

2. if `system async property` id_{prop} ; $\preceq \mathcal{M}$

$$\begin{aligned}
\delta(\sigma_1, q, n, \sigma_2) \Leftrightarrow\ &(\ \exists t, t_p \in \mathcal{L}(\text{Transition}) : t, t_p \preceq \mathcal{M} \wedge \\
&\ \mathit{parent}(t) \neq \mathit{parent}(t_p) = id_{prop} \wedge \\
&\ \mathit{PartEnabled}(t, \sigma_1) \wedge \\
&\ \mathit{PartEnabled}(t_p, \sigma_1) \wedge \\
&\ \neg \mathit{SyncReq}(t) \wedge \\
&\ [\![t]\!](\sigma_1) = \sigma_2 \wedge \\
&\ (\mathit{Prioritised}(t, \sigma_1) \vee \nexists t' \in \mathcal{L}(\text{Transition}) : \mathit{Prioritised}(t', \sigma_1)) \wedge \\
&\ q = t.\mathfrak{c} \wedge \\
&\ n = t.\mathfrak{t}\ ) \vee \\
&(\ \exists t_1, t_2, t_p \in \mathcal{L}(\text{Transition}) : t_1, t_2, t_p \preceq \mathcal{M} \wedge \\
&\ \mathit{parent}(t_p) = id_{prop} \wedge \\
&\ \mathit{parent}(t_1) \neq id_{prop} \wedge \\
&\ \mathit{parent}(t_2) \neq id_{prop} \wedge \\
&\ \mathit{PartEnabled}(t_1, \sigma_1) \wedge \\
&\ \mathit{PartEnabled}(t_2, \sigma_1) \wedge \\
&\ \mathit{PartEnabled}(t_p, \sigma_1) \wedge \\
&\ t_2 \in \mathit{SyncReceiving}(t_1, \sigma_1) \wedge \\
&\ [\![t_1]\!]([\![t_2]\!](\sigma_1)) = \sigma_2 \wedge \\
&\ (\ (\mathit{Prioritised}(t_1, \sigma_1) \wedge \mathit{Prioritised}(t_2, \sigma_1)) \vee \\
&\ \ \ \nexists t' \in \mathcal{L}(\text{Transition}) : \mathit{Prioritised}(t', \sigma_1)\ ) \wedge \\
&\ q = t_1.\mathfrak{c} + t_2.\mathfrak{c} \wedge \\
&\ n = \max\{t_1.\mathfrak{t}, t_2.\mathfrak{t}\}\ )
\end{aligned}$$

3. if `system sync` $\preceq \mathcal{M}$ and $\{p_1, \ldots, p_m\} := \{p | p \in \mathcal{L}(\text{Process}), p \preceq \mathcal{M}\}$

$$\begin{aligned}
\delta(\sigma_1, q, n, \sigma_2) \Leftrightarrow\ &\exists t_i, \ldots, t_m \in \mathcal{L}(\text{Transition}) : (\forall i, 1 \leq i \leq m : t_i \preceq p_i \wedge \\
&\mathit{PartEnabled}(t_i)) \wedge \\
&[\![t_1]\!]([\![t_2]\!](\ldots [\![t_n]\!](\sigma_1) \ldots)) = \sigma_2 \wedge \\
&q = \textstyle\sum_{1 \leq i \leq m} t_i.\mathfrak{c} \wedge \\
&n = \max_{1 \leq i \leq m}\{t_i.\mathfrak{t}\}
\end{aligned}$$

Bibliography

[1] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on Uppaal. In *SFM*, pages 200–236, 2004.

[2] Yirng-An Chen and Randal E. Bryant. Phdd: an efficient graph representation for floating point circuit verification. In *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 2–7, Washington, DC, USA, 1997. IEEE Computer Society.

[3] Pavel Šimeček. *DiVinE: Distributed Verification Environment*. PhD thesis, Faculty of Informatics, Masaryk University Brno, 2006.