MASARYKOVA UNIVERZITA V BRNĚ
FAKULTA INFORMATIKY

DIVINE – Distributed Verification Environment

MASTER THESIS

Pavel Šimeček

Brno, 2006

Declaration

Advisor: doc. RNDr. Luboš Brim, CSc.

Thanks

At the first place, I would like to thank my advisor and the head of the Parallel and Distributed Systems Laboratory, Luboš Brim, who lead me during almost my entire study at this faculty and provided me a lot of opportunities to broaden my horizons.

My special thanks go to Jiří Barnat, the leader of the DiVinE project, who was always ready to discuss problems in development of the project.

I also have to mention all the staff of the Parallel and Distributed Systems Laboratory who participated more of less in development of DiVinE and had patience to find and report many bugs in it.

Last but not least, I would like to thank my family who always gave me support and the best environment for my study.

Abstract

Automated verification of practical systems is computationally very hard. One of possibilities to verify extensive systems is to use the accumulated power of computer clusters. Recently, many distributed algorithms for model checking have been developed. The objectives of the DiVinE project are to create an environment for development of distributed algorithms and simultaneously to provide a set of distributed algorithms in the form of a verification tool. The aims of the Master thesis are to design a modeling language for DiVinE and to implement its interpreter. The thesis also involves development of the main modules of the tool in collaboration with other members of the development team. The work on the thesis also comprises the building of publicly available software package including a basic documentation. The other aims of the thesis include researching possible directions of future development of the tool and initial study of implementation of such extensions.

Keywords

model checking, model checker, DiVinE, modeling language, DVE, temporal logic, LTL, distributed

Contents

List of Figures

List of Tables

Introduction

During the twentieth century the massive expansion of computers came to everyday life. Programmable electronic devices spread to various branches of science and industry including medicine, armament industry, space exploration, where the single failure of a machine can cause loss of lives or big amounts of money. In 1999, NASA lost its $125 million Mars Climate Orbiter because of an error in a navigation software.

Nowadays needs of mass industry to verify correctness of their designs grow because even a small error in a cheap component may cause enormous loss if millions of them have been produced. In 1994, Intel, which was the biggest producer of microprocessors for personal computers, lost approximately $500 million due to a hardware design error. As electronics in cars becomes more and more complex more and more mass withdrawals of cars with an erroneous software appear – e. g. in 2005 Toyota had to call 160,000 Toyota Prius gas-electric hybrid cars to services because of an software error causing their motors to stall suddenly.

For these reasons, industry seeks efficient ways to search for errors in software and hardware designs before the mass production begins or a machine for billions of dollars is irretrievably lost. In early stages of development the traditional verification methods such as *debugging* and *simulation* help to remove most of errors. Unfortunately, these techniques cannot guarantee the correctness of a system. Even if developers know a property which should be satisfied by the system, these methods cannot prove validity of the property, they can only disprove it – they are not *complete*.

To provide better evidence of correctness *formal verification* has been developed. It uses mathematical formalisms to prove or disprove consistency of a system with its specification. *model checking* is the well-known representative of formal verification. In contrast to the traditional methods *model checking* is complete, which means that for a given property it can decide whether the property holds in the system or not. Moreover, if the property does not hold, it provides the developer with a prove of an error (e. g. a run of the system, where the property is clearly invalid). Model checking explores the entire state space of the system and if it does not find any counterexample, it validates the property.

Because it is easy to imagine even small systems having enormous state spaces, model checking is very expensive method in both time and memory. Every additional one-bit variable can potentially double the state space of the system – this is called the *state explosion* problem. Procedures exploring a behavior of a model state by state, called *enumerative*, are the most affected by the state explosion. There are many heuristics helping enumerative methods to avoid the generation of the entire state space, while preserving the completeness of these methods. *Symbolic model checking*, based on storage of sets of states (represented by Boolean formulas) instead of single states, is an alternative approach to enumerative model checking [JEK$^+$90]. Although it is very efficient especially in verification of hardware designs, it can fail in case of software systems which have essentially different structure. Moreover, reduction techniques developed for enumerative model checking (e. g.

partial order reduction) are not applicable in the symbolic approach.

Another way to cope with large state spaces is the usage of more computation power. Because the performance of a single computer is always limited, it is better to use a more universal approach – the distribution. It allows to overcome the lack of memory and CPU speed by addition of more machines. Of course, the distribution combined with state space reduction techniques becomes yet more powerful. Although there are many widespread tools for model checking (model checkers) such as Uppaal [BDL04], SPIN [SPI], SMV [SMV], Java PathFinder [HP98], Bogor [DHHR05], etc., none of them is distributed. Several distributed model checkers were developed, but they did not become as well-known as older sequential tools. It is often caused by the way of implementation and software distribution. E. g. the distributed version of Uppaal is publicly available neither as open source application nor as ready-to-install package. Many other tools exist only as experimental software for testing new techniques (e. g. [HLL04]), which means that they are not well-documented, source codes are often hidden or unmaintained, their interface is not user-friendly and input languages are unusual. Most of tools are also hard to extend because they are not modular, i. e. a single change on one place of the source code can imply many modifications on other places. The interesting exception is CADP [GMS01] which contains a collection of various verification tools and libraries including support for distributed state space generation. Nevertheless this tool does not focus on distributed computations much and support for distributed model checking is primitive compared to the latest research results. For instance the implementation in CADP needs to generate the entire state space before the verification begins.

To fill the gap in the field of distributed model checking, our group at the Faculty of Informatics started the DiVinE project with the aim to develop a distributed LTL model checking verification tool and at the same time to provide a public platform for development and comparison of distributed enumerative model checking algorithms. Thus, the project offers both the model checker and the uniform experimental environment for researchers.

Author's Contribution to DiVinE

The DiVinE project is a product of three years' team work. Many people were contributing to it and working together on the same parts. Thus, the portions of work made by single persons are bounded very unshapely. This section tries to pick up the main contribution of the author of this thesis to the project.

The author was (historically) the first developer of DiVinE. His main contribution to the project is a design of the DVE modeling language and an implementation of its interpreter – in fact, for purposes of model checking, a generator of states of a model was created, where each state represents a complete content of all variables and process counters of the modeled system during its run. The state generator also provides a programmer with a complete structure of the source code, which is useful for state space reduction techniques.

During development of the project the author was adapting the state generation module to the needs of other developers of DiVinE and helping them with the design of other modules. He also developed several auxiliary C++ classes (e. g. error handling unit) and created a system of documentation and source code compilation. He implemented the model checking algorithm based on negative cycle detection described in [BvKP01]. Last but not least, he created a web site of the project and the system of publication of DiVinE as a software package.

Thesis Structure

Chapter 1 introduces the enumerative model checking as a verification method. Then, the DVE modeling language is demonstrated and formally described in Chapter 2. Chapter 3 describes the structure of DIVINE and formulates basic goals of the project. The DVE language is used as a native modeling language of DIVINE. First, a library for support of distributed enumerative model checking (Section 3.2) has been developed and then also a set of tools and model checkers based on the library (Section 3.3). A number of model checkers has been developed. The experimental comparison of them can be found at Chapter 4. First, scalability of distributed BFS-based state space exploration is demonstrated in Section 4.2 and then, a comparison of run times on a set of models is done in Section 4.3. Chapter 5 shows the performance of the state generator and measures influence of various changes on the speed of state space generation. Finally, Chapter 6 proposes changes and extensions of the current version of DIVINE with a special attention given to the state generation module and distribution on heterogeneous clusters.

Chapter 1

Preliminaries

In this chapter the way of modeling systems and verification used in DIVINE is described. First, notions *modeling language* and *Kripke structure* are explained. Second the linear temporal logic (LTL) is introduced. Third a relationship of LTL and models of programs with automata over infinite words is described. Finally, the way of model checking used in DIVINE is depicted. More details and related topics can be found in [CGP99].

1.1 Modeling Systems

Systems are usually given by their source codes. There are many languages for a system description. Because of this diversity a unifying formalism is needed to represent (concurrent) systems of any type. Of course, we can use first order logic for this purpose [CGP99], but it is rare modeling language in practice. Many universal modeling languages, e. g. Promela [SpL] or native modeling languages of model checkers SMV [SMV], Bandera [Rob00] and Uppaal [BDL04], containing high-level constructs have been designed to keep models short and easy to understand.

The set of all possible configurations of the system forms its *state space*. The configurations are often called simply *states* or *system states*.

Independently on chosen modeling language the model corresponds unambiguously to a labeled graph of system states. Vertices of the graph are states of the modeled system and each edge (u, v) stand for possible execution of command in u determining the change of u to v. This graph is called *Kripke structure* a it is defined as follows:

Definition 1.1.1. Let AP be a set of atomic propositions. A *Kripke structure* M over AP is a four tuple $M = (S, S_0, R, L)$ where:

- S is a finite set of states - the *state space*.
- $S_0 \subseteq S$ is the *initial set of states*.
- $R \subseteq S \times S$ is the total *transition relation* (R is *total* means $\forall s_1 \in S : \exists s_2 \in S : s_1 R s_2$).
- $L : S \to 2^{AP}$ is the *labeling function*. It labels each state with the set of atomic propositions true in that state.

The *run* of the modeled system is then a labeling of a path in the state space of Kripke structure, i. e. $w = L(s_0)L(s_1)L(s_2)L(s_3)\dots$ is the *run* of (S, S_0, R, L) iff $s_0 \in S_0$ and $\forall i \geq 0 : s_i R s_{i+1}$.

1.2 Linear Temporal Logic

The linear temporal logic (LTL) is currently the only logic for property specification explicitly supported in DIVINE. It has been chosen for several reasons:

- This logic is popular and easy to understand,
- There are effective on-the-fly sequential algorithms for verification of properties specified in LTL,
- There are several distributed algorithms for model checking LTL properties developed in ParaDiSe Laboratory.

The set of LTL formulas is defined inductively:

$$\Phi \ ::= \ a \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \mathbf{X}\Phi \mid \Phi_1\mathbf{U}\Phi_2$$

Where $a \in AP$, AP is a countable set of atomic propositions, $\neg$ and $\wedge$ are Boolean operators and **X** and **U** are temporal operators called *NEXT* and *UNTIL*.

Formulas of LTL are interpreted over runs. The semantics of LTL is defined as follows:

$w \models a$	iff	$a \in w(0)$, where $a \in AP$
$w \models \neg a$	iff	$a \notin w(0)$, where $a \in AP$
$w \models \phi \wedge \psi$	iff	$w \models \phi$ and $w \models \psi$
$w \models \mathbf{X}\phi$	iff	$w_1 \models \phi$
$w \models \phi\mathbf{U}\psi$	iff	$\exists k, k \geq 0 : (w_k \models \psi \ \wedge \ \forall i, 0 \leq i < k : w_i \models \phi)$

1.3 Finite Automata over Infinite Words

A finite automaton over infinite words is a model, that takes an advantage of state space independent of the size of input word and simultaneously is able to accept or refute infinite words.

Definition 1.3.1. *Finite automaton* $\mathcal{A}$ is a five tuple $(\Sigma, Q, \delta, Q_0, \mathcal{F})$ such that

- Σ is the finite *alphabet*.
- Q is the finite set of *states*.
- δ is the *transition relation*.
- $Q_0 \subseteq Q$ is the set of *initial states*.
- $\mathcal{F}$ is the *acceptance condition*.

If $\delta \in Q \times \Sigma \times Q$, then the automaton is *deterministic*.
If $\delta \in Q \times \Sigma \times 2^Q$, then the automaton *non-deterministic*.
According to acceptance condition several types of automata over infinite words are recognized (here are defined only two of them):

- *Büchi automaton*: Let $F \subseteq Q$ be a set of *final states*. Then $\mathcal{F}(w) = accept \Leftrightarrow \exists$ run w of $\mathcal{A}$: $\exists s \in F : s$ appears infinitely often in w.

- *Generalized Büchi automaton*: Let $F \subseteq 2^Q$ be an accepting component. Let $F = (P_1, P_2, \dots, P_n)$. Then $\mathcal{F}(w) = accept \Leftrightarrow \exists$ run w of $\mathcal{A}$: $\forall i, 1 \leq i \leq n : \exists s \in P_i : s$ appears infinitely often in w.

Generalized non-deterministic *Büchi automaton* $(\Sigma, Q, \delta, Q_0, (P_1, \dots, P_n))$ can be transformed to the ordinary non-deterministic *Büchi automaton* with $|Q| \times (n+1)$ states (see [CGP99]).

1.4 Model Checking

Definition 1.4.1. Let Kripke structure $M = (S, S_0, R, L)$ represent a finite-state system and a temporal logic formula ϕ express a specification. Then *model checking problem* is finding the set of all states in S which satisfy ϕ, i. e. the set:

$$SAT = \{s \in S | M, s \vDash \phi\}$$

The system *satisfies the specification* if $S_0 \subseteq SAT$.

Remark 1.4.2. In context of LTL model checking the state is usually omitted and $M \vDash \phi \Leftrightarrow \forall s \in S_0 : M, s \vDash \phi$.

An automata based approach to LTL was discussed in [PW83] for the first time. But the current method of LTL model checking was not given until the publication of [Var96]. The detailed description can be also found in [CGP99]. The method is based on the fact, that given a Kripke structure and an LTL formula, it is possible to create a Büchi automaton, that accepts language of all infinite runs of a system modeled by the Kripke structure not satisfying a formula.

First, the LTL formula is translated to the Büchi automata that accepts a language of all infinite runs (not necessarily runs of modeled system) not satisfying a formula. Then this automaton is combined with a Kripke structure (in fact it is combined with automata accepting the language of all runs of the given Kripke structure). Checking for non-emptiness of a language accepted by the resulting automata is equivalent to searching for a counterexample to the given formula. If the language of the resulting automata is empty, then the formula is true in the given system.

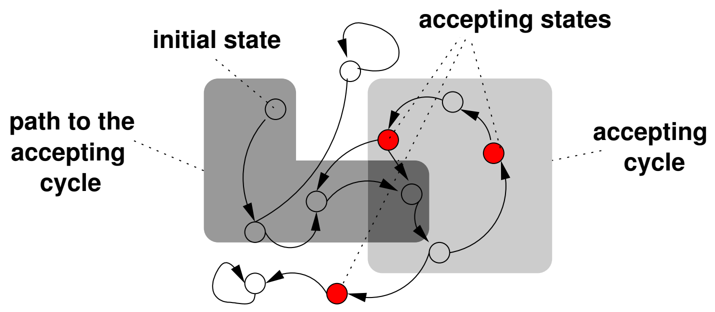
Definition 1.4.3. Let $M = (S, S_0, R, L)$ be a Kripke structure and $\mathcal{A} = (2^{AP}, Q, \delta, Q_0, F)$ a non-deterministic Büchi automaton. Then the *product automaton* of M and $\mathcal{A}$ is defined as non-deterministic Büchi automaton

$$M \| \mathcal{A} = (2^{AP}, S \times Q, (s_0, q_0), \delta', S \times F), \text{where}$$
$$\delta'((s,q),a) = \{(s',q') | a \in L(s), (s,s') \in R \wedge q' \in \delta(q,a)\}$$

Theorem 1.4.4. *Let $M = (S, S_0, R, L)$ be a Kripke structure ϕ be an LTL formula. Let further $\mathcal{A}_{\neg\phi}$ be an automaton accepting all runs which do not satisfy ϕ. Then $M \| \mathcal{A}_{\neg\phi}$ accepts empty language precisely if ϕ is true in M.*

If $M \| \mathcal{A}_{\neg\phi}$ accepts non-empty language, then words of that language are counterexamples to ϕ in M.

Definition 1.4.5. Let $\mathcal{A} = (\Sigma, Q, \delta, Q_0, F)$ be a non-deterministic Büchi automaton. Then the *accepting cycle* is a sequence of states



Graph in the picture shows an accepting cycle, a path to it and the intersection of them. There is yet another accepting cycle in the graph consisting of the depicted accepting cycle and the third state of the path to it.

Figure 1.1: Accepting cycle and a path to it

$$q_1, \dots, q_n \in Q :(\forall i, 1 \leq i < n : \exists a \in \Sigma : q_{i+1} \in \delta(q_i, a)) \wedge (\exists a \in \Sigma : q_1 \in \delta(q_n, a)) \\ \text{and } \exists j : q_j \in F.$$

Accepting cycle $q_1, \dots, q_n$ *is reachable* from state p_1, if there is a sequence of states

$$p_1, \dots p_m \in Q : (\forall i, 1 \leq i < m : \exists a \in \Sigma : p_{i+1} \in \delta(p_i, a)) \wedge (\exists j, 1 \leq j \leq n : p_m = q_j)$$

$p_1, \dots p_m$ is then called the *path to the accepting cycle*.

Definition 1.4.6. Let $\mathcal{A} = (\Sigma, Q, \delta, Q_0, F)$ be a non-deterministic Büchi automaton and let $q_1, \dots, q_n \in Q$ be a sequence of states of $\mathcal{A}$. Then *trace* of this sequence is defined as follows:

$$Trace(q_1, \dots, q_n) = \{w \in \Sigma^* \mid \forall i, 1 \leq i < n : q_{i+1} \in \delta(q_i, w(i))\}$$

Theorem 1.4.7. *Let $\mathcal{A} = (\Sigma, Q, \delta, Q_0, F)$ be a non-deterministic Büchi automaton. Then the language accepted by $\mathcal{A}$ is non-empty if and only if there is an accepting cycle $q_1, \dots, q_n$ reachable from some $p_1 \in Q_0$ along the path $p_1, \dots, p_m$ and $p_m = q_1$.*

Moreover, if $u \in Trace(p_1, \dots, p_m)$ and $v \in Trace(q_1, \dots, q_m, q_1)$, then $u \cdot v^\omega$ is accepted by $\mathcal{A}$.

In practice the method proceeds as follows:

1. Let $M = (S, S_0, R, L)$ be a Kripke structure of the verified system and ϕ is an LTL formula representing a specification.
2. Translate an LTL formula $\neg\phi$ to the non-deterministic Büchi automaton $\mathcal{A}_{\neg\phi} = (2^{AP}, Q, \delta, Q_0, F)$.
3. Make a product automaton $M \| \mathcal{A}_{\neg\phi}$.
4. Generate a state space of $M \| \mathcal{A}_{\neg\phi}$ and search for accepting cycles.
5. (a) Accepting cycle is found: Reconstruct path to the accepting cycle and return counterexample using theorem 1.4.7.

(b) Accepting cycle is not found and the entire state space is generated: Return, that ϕ is true.

This method has a time and memory complexity in $\mathcal{O}(|M| \cdot 2^{|\phi|})$, where M is the Kripke structure of verified model and ϕ is an LTL formula. $|M| \cdot 2^{|\phi|}$ is also the lower bound of time complexity, but the lower bound of space complexity is given by non-deterministic algorithm of logarithmic complexity. The exponential time complexity to $|\phi|$ follows from the equal complexity of transformation of LTL formula to non-deterministic Büchi automaton.

The procedure described above is usually preserved in all enumerative model checkers of LTL properties and they differ mostly in the implementation of point 4 – search for the accepting cycles.

There are several efficient sequential algorithms for accepting cycle detection. The time complexity of them is in $\theta(N + M)$, where N is a size of a state space and M is a size of transition relation of the product automaton.

Unfortunately, these algorithms are hard to distribute. Existing distributed algorithms have worse time complexity (e. g. cubic with respect to the size of a state space) and various smart heuristics are used to lower the time costs in practise.

Chapter 2

The DVE Modeling Language

DIVINE provides support and tools for enumerative model checking (see Sections 3.3.3 and 3.2) 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 [CB97].

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 [BDL04], 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 1.4

2.1 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 2.1. 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 2.2). 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.

2.2 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 2.4.

2.2.1 Typed Channels

Typed channel is similar to the ordinary untyped channel (which is demonstrated in the Section 2.1). 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:

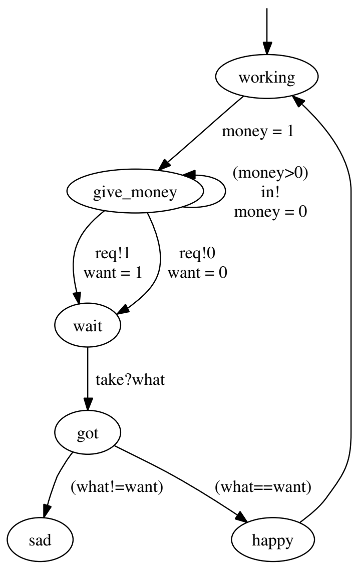
```
channel ordinary;
channel {byte} simple[0];
channel {byte,int} quick[0];
channel {byte,int} deep[4];
```

Channel `ordinary` is the same as channels in Section 2.1. 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`).

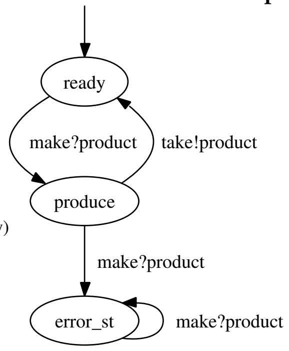
Channel `quick` is very similar to `simple`, however it can transmit 2 values at one moment (in this case the first value is casted to `byte` and the second to `int`).

Man:

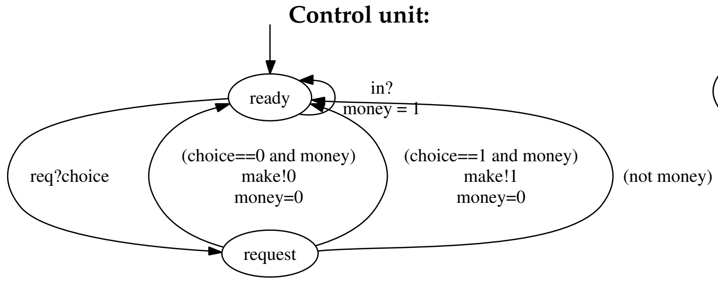


```
process man {
byte what, want, money;
state working, give_money, wait, got, happy, sad;
init working;
trans
 working   ->give_money { effect money = 1; },
 give_money->give_money { guard money>0; sync in!;
                          effect money = 0; },
 give_money->wait { sync req!0; effect want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

Mechanic parts:



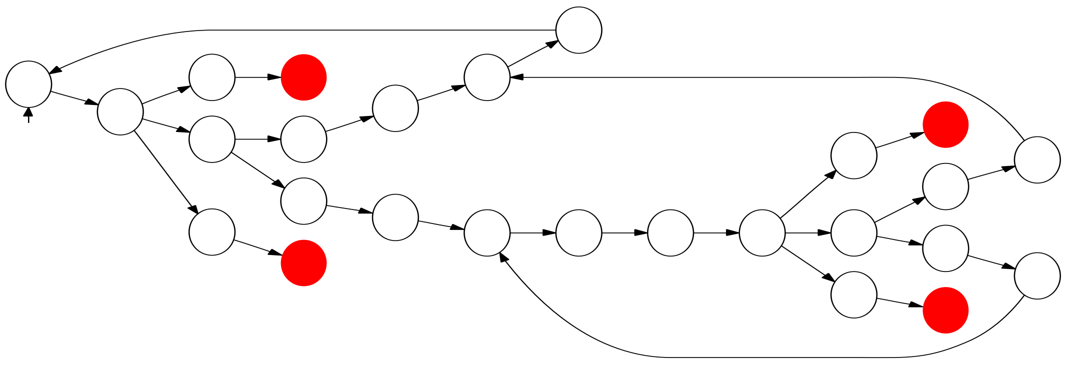
Control unit:



```
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; };
}
```

```
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 2.1: Finite state machines of a drink dispenser and their transcriptions to DVE



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

Figure 2.2: State space of the model of a drink dispenser

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.

2.2.2 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:

```
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

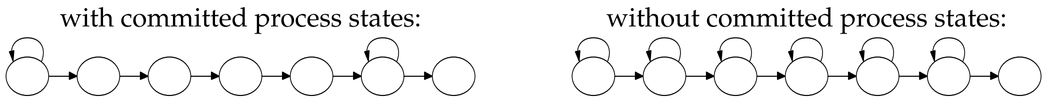

with committed process states:
without committed process states:

Figure 2.3: State spaces of the example with and without committed states

with the third transition of `set_parameters` which is otherwise always executable.

This observation is demonstrated in Figure 2.3, 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.

2.3 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 2.1). Equations marked with ◇ have a dynamic semantics defined in 2.4, 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
$\{a, \dots, z, A \dots, Z, \_\} \cdot \{0, \dots, 9, a, \dots, z, A \dots, Z, \_\}^*$,
$number$ stand for terminal symbols from $\{0, \dots, 9\}^+$.

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 $\{value_1, value_2, \dots, value_n\}$.

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.

$$\begin{array}{rcll}\text{TransitionOpt} & ::= & -\!\!>\ id\ \{\ \text{Guard Sync Effect}\ \}\ |\ \text{Transition} & \diamond\end{array}$$

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

$$\begin{array}{rcll}\text{Guard} & ::= & \varepsilon\ |\ \texttt{guard}\ \text{Expr}\ \texttt{;} & \diamond\end{array}$$

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.

$$\begin{array}{rcll}\text{Sync} & ::= & \varepsilon\ |\ \texttt{sync}\ \text{SyncExpr}\ \texttt{;} & \diamond\\ \text{SyncExpr} & ::= & id\ \texttt{!}\ \text{SyncValue}\ |\ id\ \texttt{?}\ \text{SyncValue} & \diamond\\ \text{SyncValue} & ::= & \varepsilon\ |\ \text{Expr}\ |\ \{\ \text{ExprList}\ \} & \diamond\\ \text{ExprList} & ::= & \text{Expr}\ |\ \text{ExprList}\ \texttt{,}\ \text{Expr} & \diamond\end{array}$$

Effects consist of lists of assignments.

$$\begin{array}{rcll}\text{Effect} & ::= & \texttt{effect}\ \text{EffList}\ \texttt{;} & \diamond\\ \text{EffList} & ::= & \text{Assignment}\ |\ \text{EffList}\ \texttt{,}\ \text{Assignment} & \diamond\\ \text{Assignment} & ::= & \text{Expr} = \text{Expr} & \diamond\end{array}$$

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 2.1 is needed.

$$\begin{array}{rcll}\text{Expr} & ::= & \texttt{false}\ |\ \texttt{true}\ |\ number\ |\ id\ |\ id\ [\ \text{Expr}\ ]\ | & \diamond\\ & & \texttt{(}\ \text{Expr}\ \texttt{)}\ |\ \text{UnaryOp Expr}\ | & \diamond\\ & & \text{Expr}_1 < \text{Expr}_2\ |\ \text{Expr}_1 <= \text{Expr}_2\ | & \diamond\\ & & \text{Expr}_1 == \text{Expr}_2\ |\ \text{Expr}_1\ != \text{Expr}_2\ | & \diamond\\ & & \text{Expr}_1 > \text{Expr}_2\ |\ \text{Expr}_1 >= \text{Expr}_2\ | & \diamond\\ & & \text{Expr}_1 + \text{Expr}_2\ |\ \text{Expr}_1 - \text{Expr}_2\ | & \diamond\\ & & \text{Expr}_1 * \text{Expr}_2\ |\ \text{Expr}_1\ /\ \text{Expr}_2\ | & \diamond\\ & & \text{Expr}_1\ \%\ \text{Expr}_2\ | & \diamond\\ & & \text{Expr}_1\ \&\ \text{Expr}_2\ |\ \text{Expr}_1\ |\ \text{Expr}_2\ | & \diamond\\ & & \text{Expr}_1\ \hat{}\ \text{Expr}_2\ | & \diamond\\ & & \text{Expr}_1 << \text{Expr}_2\ |\ \text{Expr}_1 >> \text{Expr}_2\ | & \diamond\\ & & \text{Expr}_1\ \texttt{or}\ \text{Expr}_2\ |\ \text{Expr}_1\ \texttt{and}\ \text{Expr}_2\ | & \diamond\\ & & \text{Expr}_1\ \texttt{imply}\ \text{Expr}_2\ | & \diamond\\ & & id_1\ \texttt{.}\ id_2\ |\ id_1 -\!\!> id_2\ |\ id_1 -\!\!> id_2\ [\ \text{Expr}\ ] & \diamond\\ \text{UnaryOp} & ::= & \texttt{-}\ |\ \tilde{}\ |\ \texttt{not} & \diamond\end{array}$$

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).

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

There are also additional constraints put on the source code:

Table 2.1: 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

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 $\rightarrow id$, id must be a scalar variable
   (c) In context of Expr $\rightarrow id$ [Expr], id must be a vector variable
   (d) In context of Expr $\rightarrow id_1$. id_1, id_1 must be a process and id_2 must be a process state
   (e) In context of Expr $\rightarrow id_1$ -> 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 $\rightarrow id$? SyncValue and SyncValue $\rightarrow$ Expr, Expr has to be a scalar variable or an element of a vector variable (and similarly for SyncValue $\rightarrow$ { 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

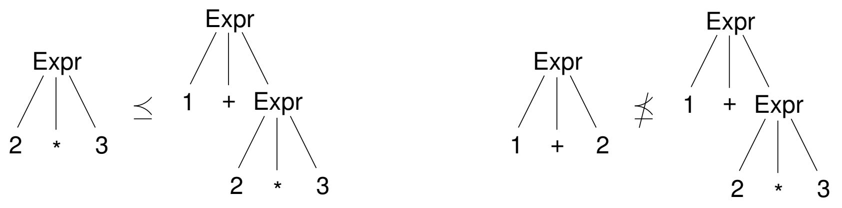


Figure 2.4: Example of relation $\preceq$ between expressions

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)

2.4 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 2.1. Equations of the concrete syntax in Section 2.3 needed for dynamic semantics are marked with $\diamond$.

First, several common notions are defined in 2.4.1. The dynamic semantics is given almost exclusively by transitions of processes. Therefore denotational semantics of transitions is given in Section 2.4.3. Transitions contain a lot of expressions. For this reason denotational semantics of expressions is defined in Section 2.4.2. Finally, the small step dynamic operational semantics of the entire DVE system is described in Section 2.4.4 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.

2.4.1 Common Definitions and Conventions

Definition 2.4.1. $\mathcal{L}(N)$ is the language of all terms derivable from non-terminal N.

Remark 2.4.2. *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 $\mathcal{L}$(Guard).

Definition 2.4.3. 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 2.4.4. E. g. $2 * 3 \preceq 1 + 2 * 3$, but $1 + 2 \npreceq 1 + 2 * 3$ (see Figure 2.4) because multiplication has a higher priority than addition.

Definition 2.4.5. *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 2.4.6. In the following text $\sigma, \sigma', \ldots$ stand for a system state.

2.4.2 Expressions

The denotation semantics of all terms from $\mathcal{L}$(Expr) is defined as follows (the semantics of used operator symbols can be found in Table 2.1):

$[\![\texttt{false}]\!](\sigma) = 0$
$[\![\texttt{true}]\!](\sigma) = 1$
$[\![\mathit{number}]\!](\sigma) = number$
$[\![\mathit{id}]\!](\sigma) = \sigma(id)$
$[\![\mathit{id}\ [\ \mathit{expr}\ ]]\!](\sigma) = \sigma(id)([\![\mathit{expr}]\!](\sigma))$ ... id is vector variable and $[\![\mathit{expr}]\!](\sigma)$ is an index to it
$[\![\ (\ \mathit{expr}\ )\ ]\!](\sigma) = [\![\mathit{expr}]\!](\sigma)$
$[\![\mathit{unary\_op}\ \mathit{expr}]\!](\sigma) = \mathit{unary\_op}\ [\![\mathit{expr}]\!](\sigma)$ ... $\mathit{unary\_op} \in \{-, \tilde{}, \mathrm{not}\}$
$[\![\mathit{expr}_1\ \mathit{binary\_op}\ \mathit{expr}_2]\!](\sigma) = [\![\mathit{expr}_1]\!](\sigma)\ \mathit{binary\_op}\ [\![\mathit{expr}_2]\!](\sigma)$
... $\mathit{binary\_op} \in \{<, <=, ==, !=, >, >=, +, -, *, /, \%, \&, |, \hat{}, <<, >>, \mathrm{or}, \mathrm{and}, \mathrm{imply}\}$,
relational and Boolean operators (e. g. $==$ or `or`) return always 0 or 1.

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

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:
$[\![id_1 \texttt{->} id_2]\!](\sigma) = \sigma(id_1 :: id_2)$
$[\![id_1 \texttt{->} id_2\ [\ expr\ ]\ ]\!](\sigma) = \sigma(id_1 :: id_2)([\![expr]\!](\sigma))$ ... id_2 is vector variable and $[\![expr]\!](\sigma)$ is an index to it

Remark 2.4.7. 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 6).

Remark 2.4.8. 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.

2.4.3 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 $\mathit{transition} \equiv \mathit{id}_1$ `->` id_2 { *guard sync effect* } and $\mathit{effect} \equiv \mathit{assignment}_1, \ldots, \mathit{assignment}_n$.

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

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

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

Remark 2.4.9. The definition of $[\![\mathit{sync}]\!](\sigma)$ 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,

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

- $\sigma' = [\![\mathit{assignment}_n]\!]([\![\mathit{assignment}_{n-1}]\!](\ldots [\![\mathit{assignment}_1]\!](\sigma_2) \ldots))$
  (if $n = 0$, then $\sigma' = \sigma_2$)
- $\sigma_2 = [\![\mathit{sync}]\!](\sigma_1)$
- $\sigma_1 = [\![id_1 \texttt{ -> } id_2]\!](\sigma)$

2.4.4 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

$parent(\mathit{transition}) = \langle$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:

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

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

Then $PartEnabled(transition, \sigma) =$

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

Function $SyncReceiving$ is defined separately for 2 cases:

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

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

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

Definition 2.4.10. Let function $Prioritized : \mathcal{L}(\text{Transition}) \times States \rightarrow Boolean$ is defined as follows:

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.

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

Furthermore, transitions can require synchronization or not.

Definition 2.4.11. Let function $SyncReq : \mathcal{L}(\text{Transition}) \rightarrow Boolean$ is defined as follows:

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

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:

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

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

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

   `system async property` *id*$_{prop}$ $\preceq$ *dve*,

   then its semantics is defined in the following way:

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

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

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

`system sync` *procproperty* $\preceq dve$,

then its semantics is defined as follows:

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

Chapter 3

DIVINE

The main goals of DIVINE – *Distributed Verification Environment* can be summarized as follows:

- distributed enumerative LTL model checker,
- platform for development of model checking algorithms,
- platform for experimental evaluation and comparison.

This chapter introduces a structure of DIVINE and describes single parts of it. A standalone section is dedicated to the history of the project, where the way to the current status is explained and impasses of development are discussed.

3.1 Structure

As new tasks and challenges are coming, the structure of DIVINE project is changing. Originally there was only the library for easy distributed model checker development. Later a lot of tools were implemented using this library and added to the project. In this section the current structure is depicted. Single parts of DIVINE are described in next chapters in more detail.

The project is divided in following sections:

- DIVINE LIBRARY – the library for a verification tool development. Needed by developers of distributed model checking tools.
- DIVINE TOOLSET – the collection of implemented model checking algorithms and auxiliary programs (testing, simulation, modeling languages translations,...)
- DIVINE TESTSUITE – the catalog of models, their properties and a benchmark support
- DIVINE GUI – the graphical user interface for a control of computer clusters

A top-level structure of DIVINE used in practice is illustrated in Figure 3.1. From the user point of view it behaves as an application with a graphical user interface. The user assigns a verification task and chooses the tool (algorithm) to solve it. The chosen tool based on the DIVINE LIBRARY is run on a cluster of computers and provides a runtime information to the user waiting for finish of the computation.

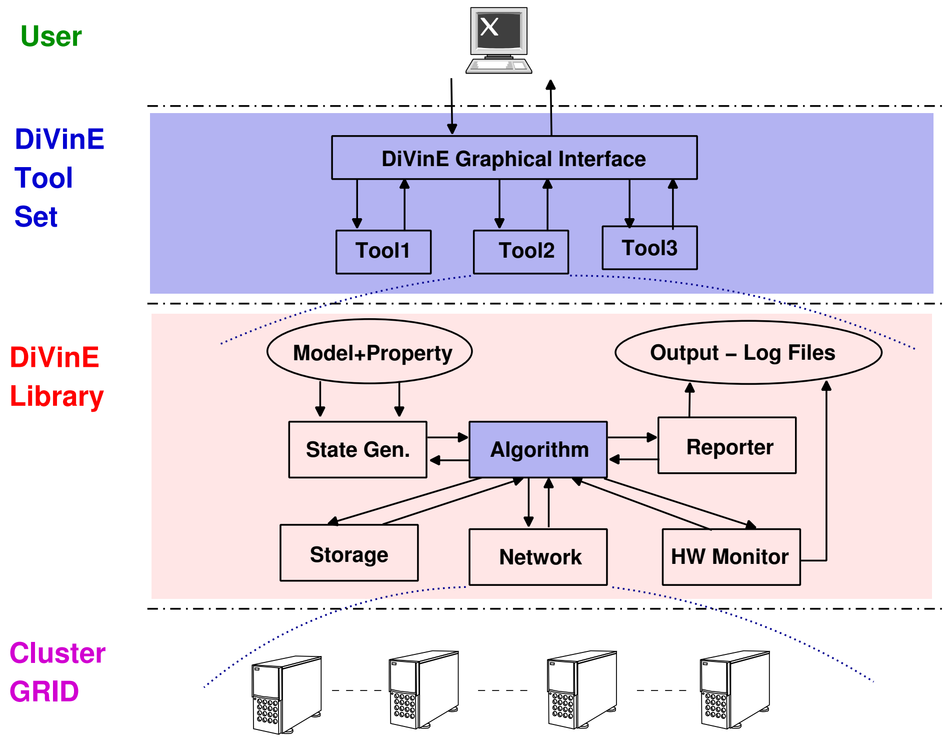


Figure 3.1: The top-level structure of DIVINE project

3.2 The DIVINE LIBRARY

The DIVINE LIBRARY is the central part of DIVINE. It is the first completed part of it. Its role in the project is dual:

- to allow internal developers to build the DIVINE TOOLSET
- to allow external developers to easily implement their own distributed model checking algorithm (and compare it to existing ones)

The rough structure of the library is depicted in the middle part of Figure 3.1. The graph of classes corresponding to it can be found in the Figure D.1.

3.2.1 State generator

Enumerative model checkers process a state space of a verified model state by state. Therefore they need a module which gives them the initial state of the model and which is able to generate successors of a given state (state t is a successor of state s, if t is a state of the model after a single execution step beginning in s). And this is the purpose of a state generator.

In Chapter 2 the DVE modeling language is demonstrated and formally described. According to the formal definition an interpreter of the language has been created, which works in DIVINE as a state generator (see Figure 3.2). The equations of concrete syntax (see Section 2.3) are used for generation of LALR(1) syntax analyzer using GNU Bison [Bis]. In Bison grammar files it is possible to assign a *hook* to each equation. The hook is a call of an external function after successful parsing of a word matching the equation. In this case hooks

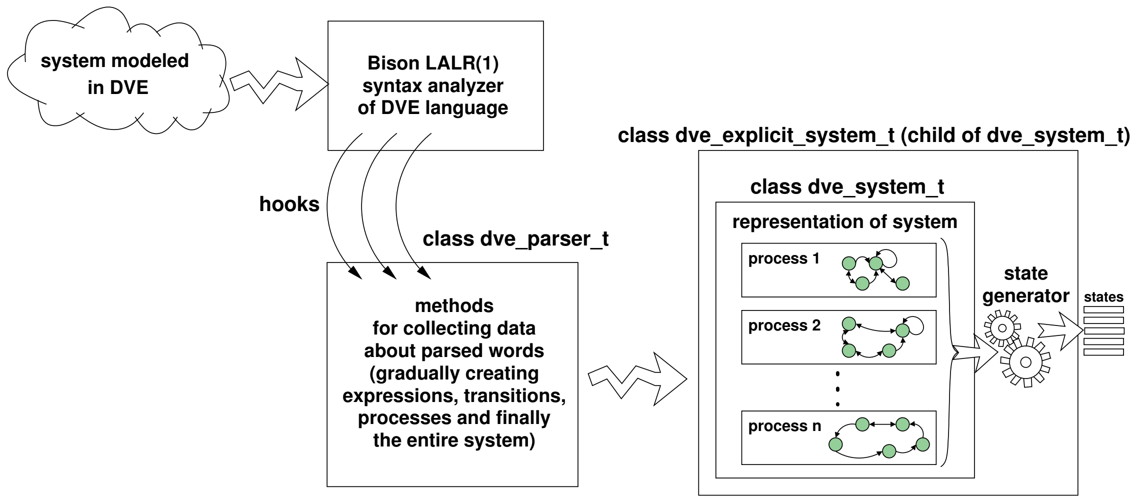

system modeled in DVE
Bison LALR(1) syntax analyzer of DVE language
hooks
class dve_parser_t
methods for collecting data about parsed words (gradually creating expressions, transitions, processes and finally the entire system)
class dve_explicit_system_t (child of dve_system_t)
class dve_system_t
representation of system
process 1
process 2
process n
state generator
states

Figure 3.2: State generation of models in the DVE language

call methods of a class `dve_parser_t` which stores information about parsed words. From collected data the class `dve_parser_t` gradually constructs an entire representation of the model. Because the language is very simple, the representation copies the model precisely and a class hierarchy of the representation is the same as hierarchy of objects in DVE:

- *system* (class `dve_system_t`) – a modeled system to be interpreted; contains some declarations (channels, global variables) and processes,
- *process* (class `dve_process_t`) – an extended finite automaton; contains declarations of local variables and transitions,
- *transition* (class `dve_transition_t`) – a part of a transition relation of a process owning the transition; defines a change of a process state and contains a guard, a synchronization and effects – and these parts contain expressions
- *expression* (class `dve_expression_t`) – a representation of expressions containing various operators, constants and variables

The DVE state generator allows an access to all syntactic entities of the model and thus facilitates its static analysis.

Because it was needed to incorporate also the Promela state generator developed in the RWTH Aachen [NIP] into DIVINE, an abstract interface derived from the structure of the DVE state generator has been created. It is formed of a set of classes shown in Figure D.2. The abstract interface determines that the model can be accessed using class `system_t` and states are generated by its child class `explicit_system_t`. Implementation of other classes of the abstract interface is optional and it depends on a concrete state generator. For instance, the current implementation of the Promela state generator does not allow any static analysis of the model and it implements only a part of the interface enabling state generation.

The process of state generation is depicted in Figure 3.3. An instance of the proper implementation of `explicit_system_t` is created depending on a format of the model, a source code is read and an a main program gets states without a care of the actual implementation

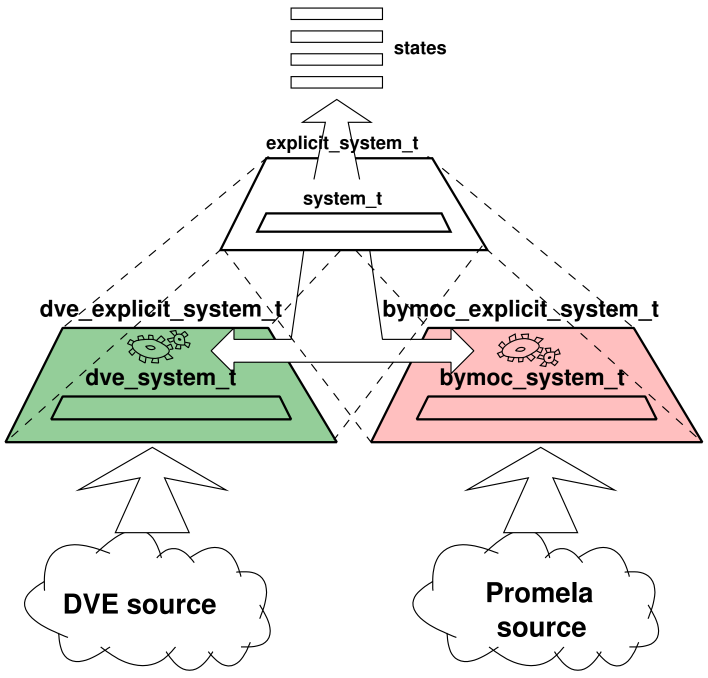


Figure 3.3: Current generation of states using the abstract interface

of the state generator hidden behind the abstract interface. The idea of abstract interfaces is worked out in detail in Section 3.6.4.

3.2.2 Storage

For a cycle detection model checkers need to store states generated by the state generator to the memory. This way they recognize revisits of states.

Model checking algorithms usually need to:

- store a state
- find, whether a state has been stored

Therefore the storage module implements a set with fast implementations of operations listed above. Currently, the set is implemented using hash tables with a default or custom hash function. For efficiency reasons, this module also provides possibility to:

- get a short constantly sized identifiers of states called *references to states*; these identifiers are independent of a real position of states in memory
- retrieve a state given by the reference to it.
- store a constantly sized piece of information called *appendix* for each state

All operations listed above have a constant time complexity depending on a quality of a hash function. Figure 3.4 shows schematically a usage of the storage module in a model checker.

The storage module is implemented in DiVinE using class `explicit_storage_t`.

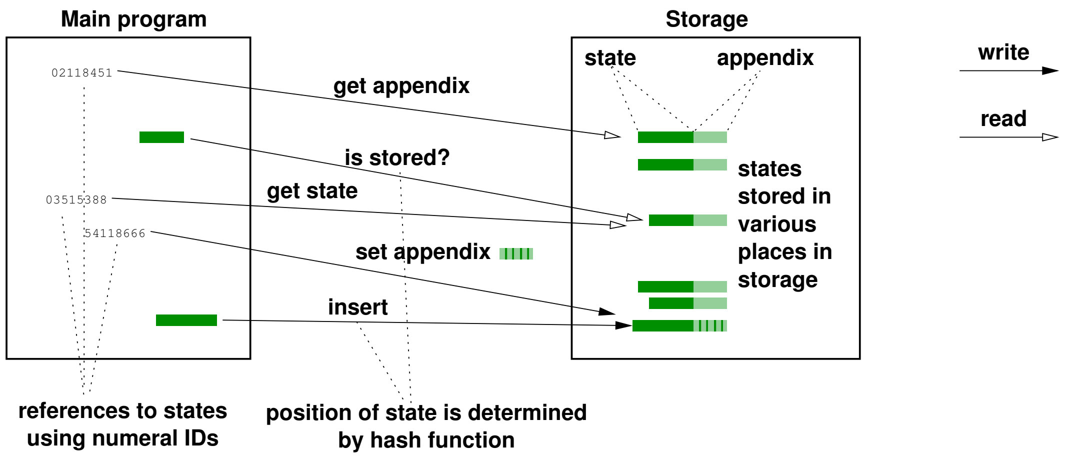

Main program
Storage
state
appendix
write
read
02118451
get appendix
is stored?
get state
03515388
54118666
set appendix
insert
states stored in various places in storage
references to states using numeral IDs
position of state is determined by hash function

Figure 3.4: Usage of the storage module

3.2.3 Network

Distributed algorithms need to communicate through a network. To provide a sufficiently powerful and relatively simple interface, a network module has been constructed.

It is built on the standard MPI [MPI] providing various types of basic communication primitives. Normal and urgent (unbuffered) messages can be sent using a dedicated functions (`send_message`, etc.), for receiving of messages there is a special callback function (`process_message`) called each time an arbitrary message is received.

Besides the common communication primitives for sending and receiving of messages the network module also provides advanced features:

- buffering of messages - message is not sent until the buffer is full,
- barrier synchronization - workstation waits until the others reach the same point,
- distributed termination detection (DTD) - computation is finished, if all workstations are idle and there is no pending message in the network.

Especially DTD is very useful in practice because it is often needed not only at the end of the entire computation, but also during a computation after a finish of an individual phase of a computation. Ordinary run of a model checker can be divided into phases as follows:

$$\boxed{\text{PHASE 1}} \rightarrow \boxed{\text{DTD}} \rightarrow \boxed{\text{PHASE 2}} \rightarrow \boxed{\text{DTD}} \rightarrow \ldots \rightarrow \boxed{\text{PHASE } n}$$

$\boxed{\text{PHASE } i}$ stands for a phase of computation and $\boxed{\text{DTD}}$ for the distributed termination detection. Usually model checking algorithms work in several computation phases and they need to be sure, that all workstations are working on a same task in a same phase and they cannot be disturbed by messages from previous computation phases.

3.2.4 Reporter, Hardware Monitor and Logger

Several model checking algorithms are currently implemented in DiVinE. Both user and developer of model checkers are usually interested not only in validity of a verified property, but also in a time and memory consumption and sometimes also other runtime information.

This information can be reported using a standardized interface of the reporter module. Memory and CPU usage informations are provided by the hardware monitor module.

The logger module is used for a standardized output during a runtime, giving to user an information about a progress of computation (it uses the hardware monitor to get most of information automatically).

3.3 The DIVINE TOOLSET

The DIVINE TOOLSET is a collection of programs based on the DIVINE LIBRARY. All tools described in this section are non-interactive and they can be controlled only by command line parameters, except for simulator which has an interactive text interface. On one hand, the usage from the command line is uncomfortable, on the other hand, it makes the development of tools easier. The graphical interface wrapping all these tools is under intensive development (see section 3.5).

Programs contained in the DIVINE TOOLSET are presented here very briefly, the detailed description of their usage can be found in the documentation.

Tools can be divided into three basic branches according to their purpose.

3.3.1 Model Manipulation Tools

Developers of interpreters and modules for a model analysis often wish to have an interpreted language as simple as possible. Opposite to this tendency, model writers need the modeling language as powerful and concise as possible.

A compromise between them can be made as follows:

- interpreted language is very simple, but sufficiently powerful,
- modeling language is more concise and there is a translator from it to the interpreted language

This is also true in DIVINE. The DVE modeling language is very simple and it can be laborious to write a model in it. Furthermore, if user wants to have a parametrized model, he has to create the new model for each tested value of a parameter. E. g. if user wanted to test his model containing buffered channels for the lengths of buffers 1 and 2, he would have to write two models – one containing buffers of length 1 and the other containing buffers of length 2.

No really concise high-level language translatable to DVE has been created till now, for this reason DIVINE provides a user with possibility to use the m4 preprocessor, widespread especially on Unix and related platforms. The preprocessing is automatically made during the creation of product automaton using the script combine as described below.

In case of Promela the situation is getting yet more complicated. This modeling language has its native cpp preprocessor from a standard distribution of C compiler. But neither preprocessed Promela is possible to interpret in DIVINE. The interpreter developed in the RWTH Aachen is only able to execute a special byte code. Therefore first, the translator (pml2s) to the special kind of assembler has to be used and then the compiler (nips_asm.pl) produces the corresponding byte code. Both the translator and the compiler are developed in the RWTH Aachen and they are included in the distribution of DIVINE.

Preprocessing of both DVE and Promela is wrapped to the script preprocessor.

$$\left.\begin{array}{lcl} \text{model.mdve} & \overset{(preprocessor)}{\longrightarrow} & \text{model.dve} \\ \text{model.mltl} & \overset{(preprocessor)}{\longrightarrow} & \text{model.ltl} \end{array}\right\} \overset{combine}{\longrightarrow} \text{model.prop1.dve, model.prop2.dve, \ldots}$$

model.prop$i$.dve is a DVE input for model checkers

$$\text{model.cpp.pml} \overset{preprocessor}{\longrightarrow} \text{model.pml} \overset{pml2s}{\longrightarrow} \text{model.pml.s} \overset{nips\_asm.pl}{\longrightarrow} \text{model.pml.b}$$

model.pml.b created from an original Promela model model.cpp.pml containing manually added never claim automaton is a byte code input for model checkers

Figure 3.5: Sequences of translations of DVE and Promela.

Model checkers in the Section 3.3.3 expect an input model to be a product automaton of an original model and a verified LTL property (see definition 1.4.3). Both supported modeling languages allow to include so called never claim automaton (i. e. the automaton $\mathcal{A}_{\neg\phi}$ from theorem 1.4.4) directly to the source code of the model. For DVE there is a utility combine which runs a preprocessor and then combines a model with an LTL formula using auxiliary tool ltl2ba. ltl2ba translates LTL to the never claim automaton (in a syntax of DVE). Unfortunately, it is still not able to produce a Promela output, therefore combine does not work for Promela and the never claim automaton has to be added to the Promela sources manually (or using the model checker SPIN [SPI])

The process of translation is briefly depicted in Figure 3.5.

3.3.2 Basic Tools - Simulation and Testing

Basic tools consist of programs for a model simulation, generation of a state space (on a single workstation) and small utilities for graphical state space printing and statistics graphs printing.

The most important tool is simulator because it is necessary for modeling of systems and tracking of counterexamples. By default, simulator runs the model step by step and in each step the user interactively chooses a transition to proceed.

Unfortunately, simulator currently works only on DVE inputs, Promela is not supported yet (but there is a simulator of Promela included in the distribution of SPIN [SPI] model checker).

The next described tool is generator. It only generates the state space and prints it in various formats. It is also able to detect a basic version of deadlock - a state, where it is not possible to execute any transition. It can detect errors (like division by zero,...), which would degrade a process of model checking. At the end of the run it optionally reports a statistics about memory consumption, number of states of a model and collisions in a hash table. This tool is especially useful for premature testing of small models or small instances of parametrized models. It is not distributed and thus its usage is limited by a physical memory of a single computer. For similar purposes without these memory limitations, it is possible to use the tool for distributed reachability analysis.

3.3.3 Model Checkers

Except for the tool doing the reachability analysis, all model checkers in DIVINE currently do the LTL model checking. All of them are also distributed, i. e. the model checking task can be distributed to an arbitrary number of workstations.

As stated in Section 3.3.1, model checkers in DIVINE expect the input model in a product form with an LTL formula. Thus the model checking task reduces to the detection of accepting cycles (see Section 1.4).

At this time eight distributed cycle detection algorithms are implemented, all of them work on DVE and most of them on Promela inputs.

These algorithms differ in the memory and time complexity and in the real performance (as shown in 4). The description and proofs of correctness of used algorithms can be found in papers published by members of ParaDiSe Laboratory: [BB03], [BBv02], [BvKP01], [BBC03], [BBC05], [Bar04] and [BvMv05].

3.4 The DIVINE TESTSUITE

This part is probably the least evolved part of DIVINE. On one hand, there is a collection of models written in DVE and their LTL properties, which is necessary for any testing and comparison of algorithms. This collection is very useful because it provides many models with large state spaces which do not fit in a memory of a single workstation.

On the other hand, the DIVINE TESTSUITE should originally have contained more functionality:

- a system of evaluation of model characteristics (e. g. number of states, transitions, cycles or back level edges)
- a system of selecting models with given properties (e. g. models without accepting cycles with at least 10,000,000 states)
- a system of automatic comparison of tools and easy visualization of results

From all these features only the visualization has been worked out a little. Utility plotlog has been developed as a part of the DIVINE TOOLSET. It can plot graphs showing various data about a computation runtime. Its example outputs can be found in appendix B.

3.5 Graphical User Interface

The DIVINE TOOLSET is a collection of purely command line applications (except for simulator). The usage of them from the command line can be very uncomfortable for unexperienced users. Furthermore, a need to compile and install DIVINE can be very frustrating for some people. Although we try to write a portable code, the compatibility can be guaranteed at most for 32-bit Linux platforms with proper compiler and an implementation of MPI.

For this reason it was decided to create a highly portable GUI using client/server architecture. We plan to run the server side of GUI permanently and thus offer a cluster of ParaDiSe Laboratory machines available to public. The only thing the user needs to do is to download a GUI client written in Java (and thus compatible with all platforms running Java Runtime Environment). This way both software and hardware are provided by DIVINE to perform a distributed verification. We expect users to build their own clusters in future.

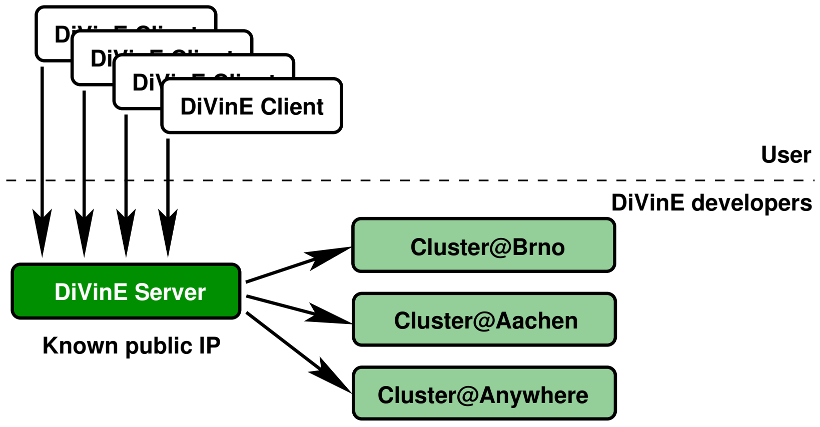


Figure 3.6: GUI client/server architecture with more than 1 cluster

The client allows to choose a cluster for computations. Such a cluster has to be registered in the server part of GUI. Architecture of such a system is depicted in Figure 3.6. In case the user wants to work independently of ParaDiSe Laboratory, he can build his own server with absolutely different clusters – the DIVINE GUI server is a part of a distribution of DIVINE.

Figure 3.7 shows, how the client currently looks. The user accesses a cluster of computers using his own account. On this account the user can store models and properties to be verified. Using the main window he can choose a verification task and a tool to solve it. He can also choose a cluster to use for his task. The smaller window shows a status of a cluster in ParaDiSe Laboratory – more precisely it shows a CPU load of all workstations in the cluster.

We believe that GUI will make DIVINE more popular and easier to present. Nevertheless it adds no functionality to model checkers and all tools can be successfully used without the graphical interface after a short training.

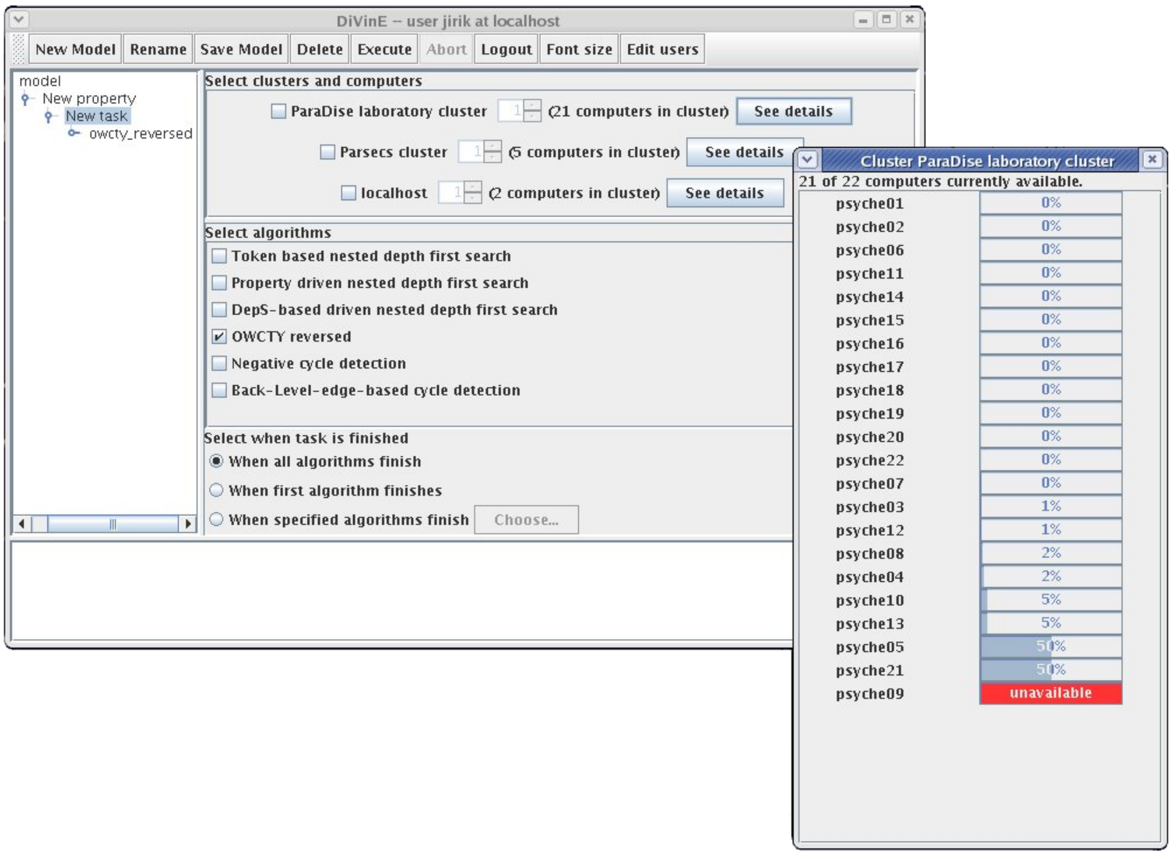
3.6 History and Ways to Proceed Further

Because a process of development has been quite complicated, it we believe that this section can help to better understand the current status of the project and to avoid some mistakes we did in past.

3.6.1 Prehistory of DIVINE

At the very beginnings there was only one developer and few people contributing to a design of the tool and a specification of a modeling language (later called DVE). The priority was to create an environment for easy implementation and comparison of distributed model checking algorithms. Because ParaDiSe Laboratory published several distributed LTL model checking algorithms, it was decided to add support for LTL model checking directly into the modeling language in a form of *never claim automaton* (syntactically written as one special processes in a model). C++ was chosen as a development language.

In few months the first version of an interpreter was created. Unfortunately, because of the lack of experiences with a software design and unclear concept which features are required, the resulting interpreter had very homogeneous structure with insufficient func-



Select clusters and computers

☐ ParaDise laboratory cluster 1 (21 computers in cluster)

☐ Parsecs cluster 1 (5 computers in cluster)

☐ localhost 1 (2 computers in cluster)

Select algorithms

☐ Token based nested depth first search

☐ Property driven nested depth first search

☐ DepS-based driven nested depth first search

☑ OWCTY reversed

☐ Negative cycle detection

☐ Back-Level-edge-based cycle detection

Select when task is finished

● When all algorithms finish

○ When first algorithm finishes

○ When specified algorithms finish

Cluster ParaDise laboratory cluster

21 of 22 computers currently available.

psyche01	0%
psyche02	0%
psyche06	0%
psyche11	0%
psyche14	0%
psyche15	0%
psyche16	0%
psyche17	0%
psyche18	0%
psyche19	0%
psyche20	0%
psyche22	0%
psyche07	0%
psyche03	1%
psyche12	1%
psyche08	2%
psyche04	2%
psyche10	5%
psyche13	5%
psyche05	50%
psyche21	50%
psyche09	unavailable

Figure 3.7: GUI client

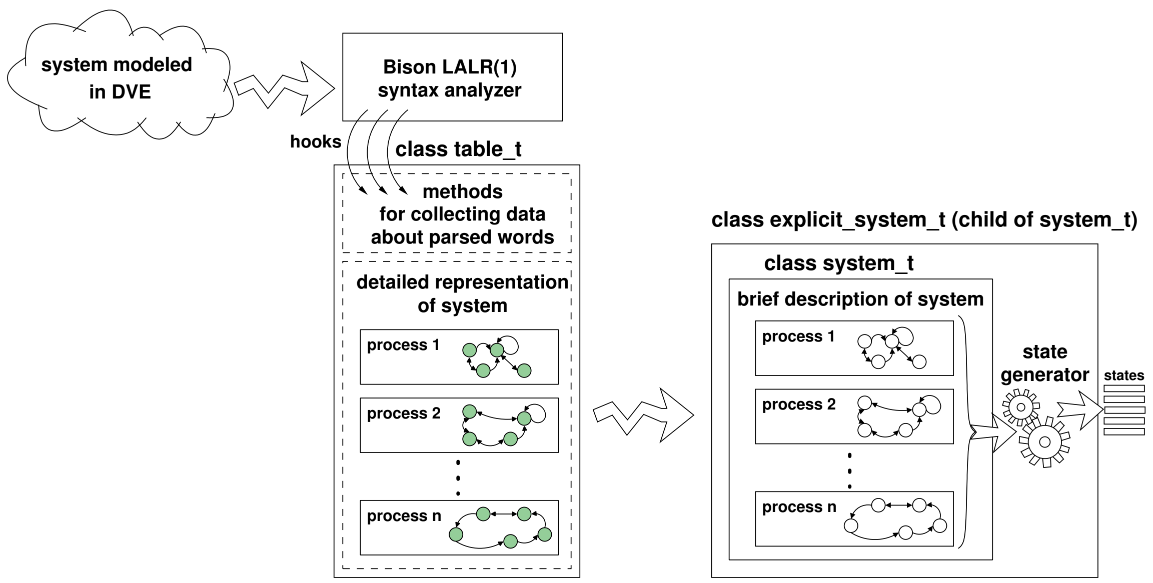
tionality. If fact it was one big C++ class with many methods and hidden internal structure.

In the second half of 2003, the class hierarchy and needed functionality of the interpreter became much clearer. Classes were separated to a representation of a read model and a generation of states (see Figure 3.8 for more details). From this point the interpreter is also called a *state generator* after its most important functionality. The first module for storage of generated states was created and the interactive simulator of DVE models was built using the interpreter. Also the existing translator of LTL formulas to Büchi automata was added to the project and modified properly (Büchi automata was written as a process in DVE source). At the end of 2003 the first sequential model checking algorithm was implemented. At that time the project had several active programmers, mostly Master and PhD students.

The try to create the absolutely efficient and optimized code from the beginning of the development made the progress slow and caused many bugs in a code. Many optimizations did not last out the first changes in a structure and the time spent for them was lost. Later the clarity of the code and the speed of the development got much higher priority, which has shown to be much better approach.

3.6.2 Era of Growth

At the beginning of 2004 the module for distributed computation was finished. It was the most important step to the implementation of distributed model checking algorithms. The


system modeled in DVE
Bison LALR(1) syntax analyzer
hooks
class table_t
methods for collecting data about parsed words
detailed representation of system
process 1
process 2
process n
class explicit_system_t (child of system_t)
class system_t
brief description of system
process 1
process 2
process n
state generator
states

DVE source is read and stored in 1:1 correspondence to the hierarchy of objects contained in class `table_t`. Then the brief representation of the system is extracted. The shortened representation contains only data needed for generation of an initial state and successors. The idea of having briefer copy of model was rejected later because of unclear interface to the developer and duplicity of code. But the concept of classes `system_t` and its child `explicit_system_t`, where the first stores a read model and the second generates states, has been preserved till now.

Figure 3.8: Former structure of the interpreter

partial order support was added too. The library with well defined interface and reference documentation was being commonly used inside ParaDiSe Laboratory. The number of model checking algorithms, various tools and small modules was created. The repository of sources began to grow, but the documentation existed only for the state generator. The project suffered from absence of any internal policy of code and documentation style. It was decided to remedy the situation: clear all sources, delete obsolete ones, restructure the entire repository of codes, revise the class hierarchy, reimplement several tools and prepare DiVinE for the first public release.

3.6.3 The First Release

The preparations for the first release took almost half a year and at the beginning of 2005 the it was finally published. The released version of DiVinE contained system of makefiles based on Automake [AM] and Autoconf [AC]. Also a documentation created using Doxygen [Dox] was present in the published package including the complete reference guide for the library. The code was distributed under the terms of GNU GPL [GPL].

3.6.4 Abstract Interfaces

During the preparation of the first public release, an idea of conversion of current C++ classes without any virtual methods to abstract interfaces and their implementations was invented. This would allow us to have more than one implementation of some modules and thus enhance the library with a possibility to choose and compare them.

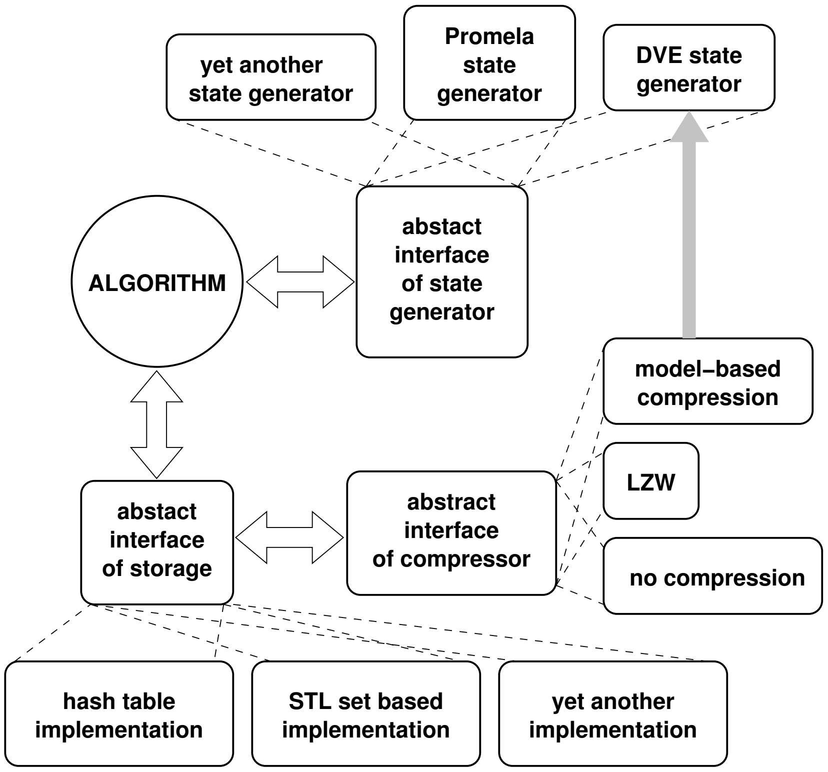

yet another state generator
Promela state generator
DVE state generator
ALGORITHM
abstact interface of state generator
model-based compression
LZW
no compression
abstact interface of storage
abstract interface of compressor
hash table implementation
STL set based implementation
yet another implementation

Figure 3.9: The concept of abstract interfaces

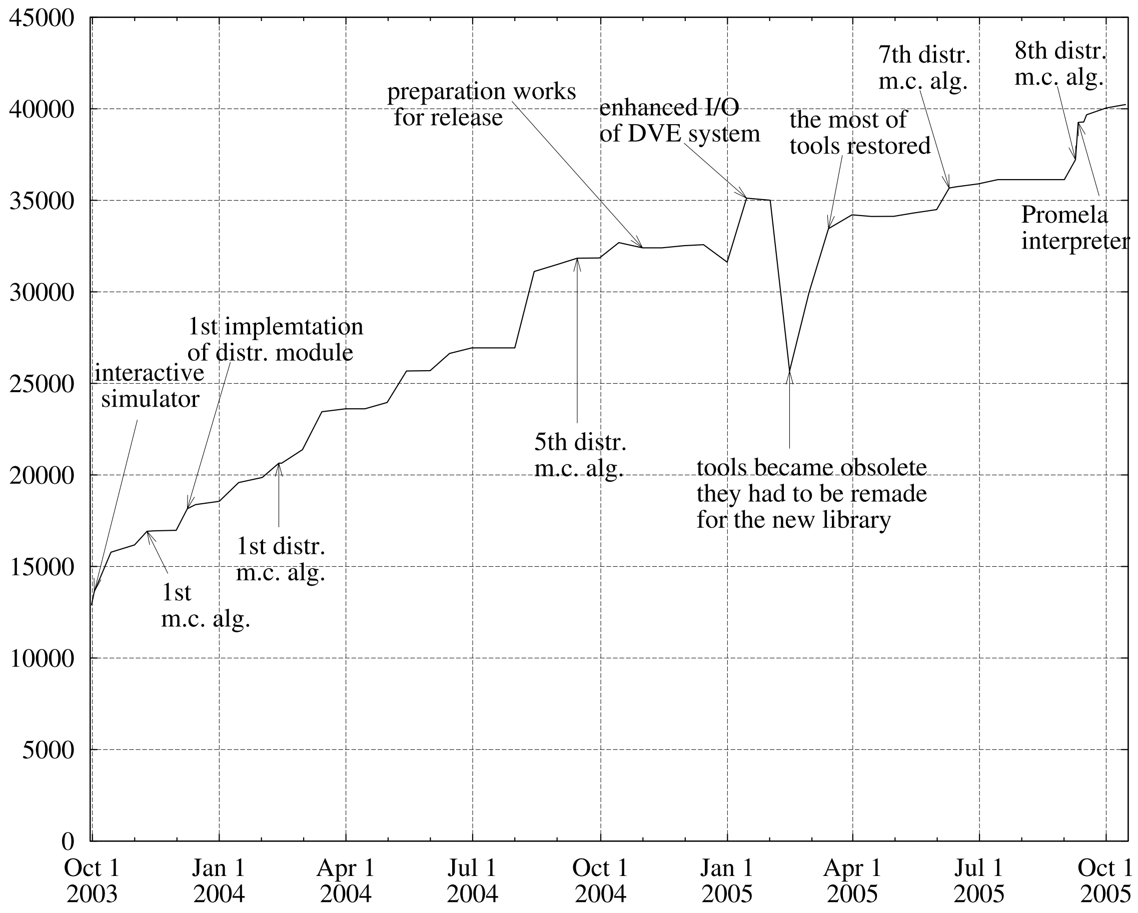
Figure 3.9 shows an example of the concept of abstract interfaces demonstrated on modules for generation, compression and storage of states. Abstract interfaces and their implementation drawn in the picture are only examples and most of them do not exist in DIVINE at all.

An algorithm and single modules communicate using abstract interfaces. A concrete implementations hidden behind abstract interfaces are chosen during a start of a program.

Sometimes some modules can require special features of an implementation of another module and an abstract interface does not suffice to provide it (e. g. model-based compression can require from a state generator a special information about the model – grey arrow in the picture). This feature often cannot be added to the abstract interface because it cannot be provided by the rest of implementations. The module can work around the abstract interface using casting. It is important to notice that this way a dependence between these two modules arises. The system of abstract interfaces then must not allow to break this dependence (e. g. model-based compression can be instantiated only together with the DVE state generator and not with the Promela state generator).

The requirement for a wider interface is solved above. But what if a module cannot implement all features provided by an abstract interface? This problem can be solved using a set of Boolean functions, telling for each feature, whether it is present in the implementation or not. These special functions can be imagined as lights on control panels of some machine – only panels with lights on can be used to control the machine, the rest of panels is broken and it is not a good idea to use them (panels play a role of interface features in this analogy).

The idea has been worked out, but it has not been fully carried out till now because of the



The count of active C/C++ lines in on the vertical axis. The date of source repository snapshot is on the horizontal axis. From 15th March 2005 directory `divine/demo` is not taken into account because sources in it are obsolete

Figure 3.10: The count of active C/C++ code lines in the source repository of DiVinE project.

lack of human resources. The abstract interface has only been created for classes related to the state generator. Thereby we managed to involve the interpreter of the Promela modeling language [NIP] developed in the RWTH Aachen to the library as a new implementation of abstract state generator class and thus made the usage of a model checker easy for people having their models written in Promela [SpL]. The current status of the state generator is described in Section 3.2.1

3.6.5 Gathered Experiences

The main gathered experiences from the field of team work and software design are listed below. They are very DiVinE specific, but some of them can be probably generalized:

- Elaborate specification and structure design is needed before the coding begins.
- The clarity of code and the speed of development are more important than the efficiency of the implementation. Optimization should be one of the last (but not least)

steps.

- Common code and documentation style policy is necessary.
- System of makefiles based on one of standard solutions (e. g. Automake/Autoconf) is very useful.
- Virtual methods are not as computationally expensive as many people are afraid.

Although the big piece of work has been done on revisions and restructurations, DIVINE still suffers from the heritage of a code of earlier versions. Therefore there is still a lot of things to change and we do not guarantee the backward compatibility of the library yet. On one hand, we have spent and we are spending a lot of time on recoding and restructuring, which could be prevented by the better specification at the beginning, on the other hand, this way we got sooner the functioning library and tools and subsequently also the knowledge, what we really need and which ways to proceed further.

DIVINE has been introduced to public in July 2005 [BBvv05].

Chapter 4

Performance Analysis of Distributed Model Checkers

In Chapter 3 it was advertised that DIVINE can be used as a platform for comparison of model checking algorithms. Although eight LTL model checkers have been implemented, no comprehensive comparison of all of them has been made until now because there is only a collection of models and formulas, but no support for *automated* testing and comparison (as mentioned in Section 3.4) and thus, any performance analysis is very time-consuming.

In this chapter, first, performance of the distributed reachability tool is analysed, in order to get better insight into distributed algorithms based on distributed BFS and then a comparison of distributed model checkers follows.

Experiments were run on 10 or 20 workstations with 1 GB RAM and 2.6 GHz Pentium 4 CPU.

4.1 Testing Set

A set of inputs used for experiments consists of models written in DVE contained in the DIVINE TESTSUITE. Because all these models are parametrized, they are used with fixed parameters (ensuring sufficiently large and representative state spaces).

Table 4.1 shows characteristics of all models used in experiments (numbers of states and transitions). More characteristics could be interesting, but DIVINE currently has no tools which would be able to evaluate models deeply (see Section 3.4). Because DIVINE is focused on model checking, the inputs are products of models and never claim automatons (see Section 1.4). The presence of accepting cycle is equivalent to invalidity of LTL formula encoded in a product automaton.

Models Elevator*i* and Phils1 are the largest ones and their role is mostly in a demonstration of computation power of distributed model checking. The comparison is valid on smaller examples too and it does not take so much time.

Models Anders5, Bakery*i*, MCS*i* and Peterson*i* are models of known mutual exclusion

	PLC1	Anders5	Bakery1	Bakery6	MCS2	MCS3
Acc. cycle:	Yes	No	Yes	No	No	Yes
States:	29048151	19647785	372934	108278	10339043	4742223
Transitions:	74516546	90036320	1599194	277338	42665116	29068886

	Peterson7	LUP2	LUP3	Elevator1	Elevator2	Phils1
Acc. cycle:	Yes	No	Yes	No	Yes	Yes
States:	13509021	23512370	16562363	61210594	104660960	28697811
Transitions:	59516927	69117680	33464135	953351322	1340963934	399643576

Table 4.1: Table of used models with their characteristics

algorithms with various LTL properties and sizes of their state spaces differ much. Bakery*i* has the smallest state space. PLC1 is a model of experimental chemical plant and LUP*i* is a model of a shared memory control in a microprocessor.

The testing set contains various models of various sizes. Seven models contain an accepting cycle and five models do not. It is necessary to test algorithms on both types of inputs, because model checking can be used for both bug searching and correctness proving.

4.2 Distribution of BFS

Many distributed algorithms for cycle detection are based rather on breadth first search (BFS) than on depth first search (DFS). For this reason it is good to understand a behavior of distributed BFS. The cycle detection based on DFS does not rely on the fact that a state visited for the second time lies on a cycle, which is not the case of BFS-based cycle detection. Distributed version of BFS has a single queue of waiting vertices for each workstation. Workstations explore only their own states (according to the partitioning of the state space) and foreign states generated as successors of own states are sent to their owners.

```
DISTRIBUTED_BFS()
if own($state_{init}$)
    add $state_{init}$ to $Wait\_queue$
    add $state_{init}$ to $States\_set$

while $Wait\_queue \neq \emptyset$ on any workstation
    if $Wait\_queue \neq \emptyset$
        remove $state_1$ from $Wait\_queue$
        foreach $state_1 \xrightarrow{t_i} state_2$
            if own($state_2$)
                if $state_2 \notin States\_set$
                    add $state_2$ to $Wait\_queue$
                    add $state_2$ to $States\_set$
            else
                send $state_2$ to $Wait\_queue$ and $States\_set$ of an owner of $state_2$
                (if not already present in $States\_set$ of the owner)
```

Queues in BFS tend to grow a lot, which has two consequences:

1. Disadvantage: Queues consume a non-trivial part of a memory.
2. Advantage for distributed computations: All workstations have enough useful work to do.

The second point is the reason, why BFS is used in a distributed environment although the cycle detection based on it is asymptotically slower in sequential case.

An asynchronous send and receive suffice for sending of states between workstations, thus the computation is not delayed by any blocking synchronization.

For better insight into results in the next section a performance of a reachability algorithm based on BFS is measured here. Scalability of BFS with regard to the number of used workstations is studied here.

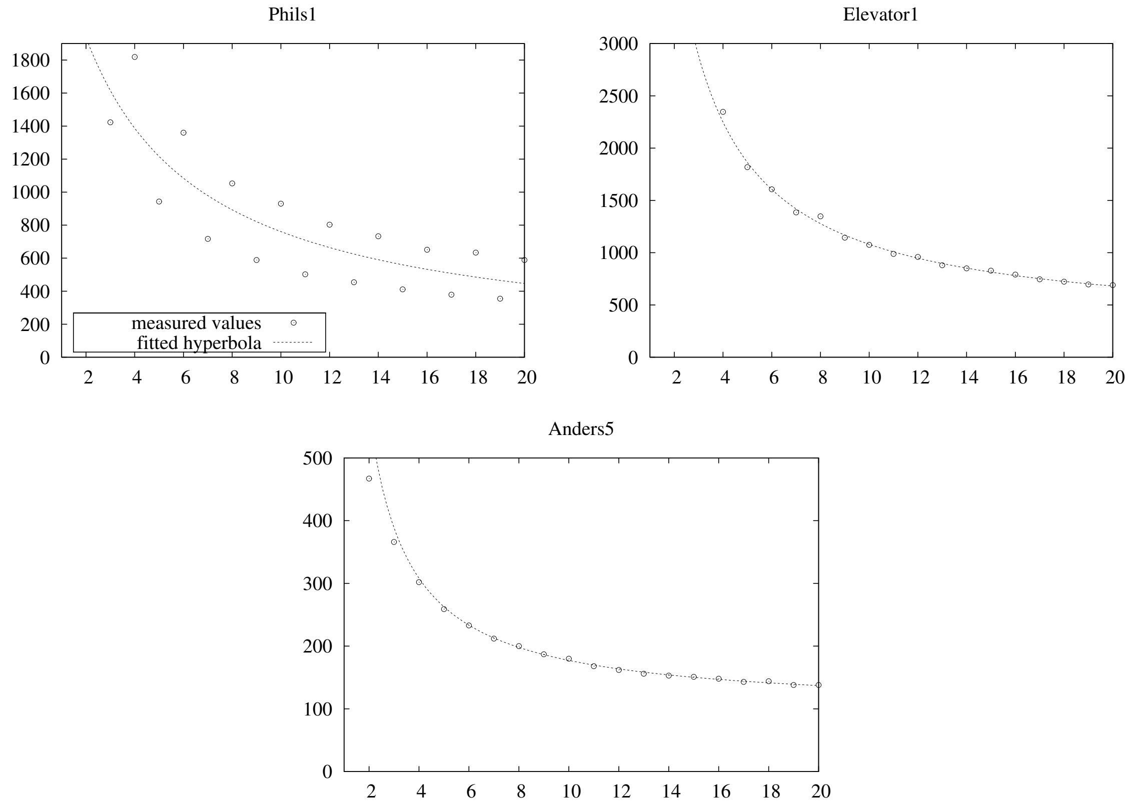


Figure 4.1: Running times of distr_reachability on different counts of workstations

An experiment shown in Figure 4.1 consists of runs of tool distr_reachability on gradually increasing number of workstations. The hyperbola fitted to the points of measured values is there only for illustration of inverse proportionality of a run time and a number of workstations. In case of Phil1 a partitioning of a state space was from unknown reason very unbalanced for even count of workstations. Unbalanced runs made the computation slower. But a tend of decreasing time costs is visible there too.

The experiment confirms a relatively good scalability of a distributed version of BFS. E. g. to double the computation power of 5 workstations approximately 12-13 workstations are needed for Elevator1, 11-13 for Phils1, but more than 20 workstations for of Anders5. The scalability for Anders5 is not catastrophic - e. g. 6 machines suffice to double the speed of 2 machines. But it is much worse than in case of Elevator1 and Phils1 because a state space of Anders5 is much smaller and with smaller data the distribution of computation on them is naturally less useful (a single workstation has less work to do, thus it spends more time by communication and waiting in idle state).

4.3 Comparison of Distributed Accepting Cycle Detection Algorithms

Eight distributed algorithms for accepting cycle detection (i. e. for verification of LTL properties) are currently present in the DIVINE TOOLSET. Until now there has not been any comparison of all of them on a single place. In a table in Table 4.2 model checkers are compared on all inputs from the testing set. For each pair of a model and model checker a run time (is seconds) is measured.

The comparison has shown, that there is a group of algorithms relatively stable in performance and sufficiently fast to cope even with large models. Algorithms NCD, OWCTY, OWCTY reversed, Distributed MAP and Property Driven NDFS belong to this group. It appears better to use OWCTY reversed than OWCTY because of OWCTY appears to be in common case at least twice as slow as OWCTY reversed. However OWCTY needs to store less information to appendix of states and thus it is able to finish even for the largest models. Very surprising is the good performance of NCD, which has an asymptotic time complexity cubic to a size of a state space. In practice NCD seems to be powerful similarly to OWCTY reversed.

All five named algorithms are based on BFS except for Property Driven NDFS which is based on DFS (or more exactly on nested DFS which is DFS modified for accepting cycle detection). This algorithm is very fast at searching for counterexamples, but it is rather slow if a model satisfies a specification because it cannot take a full advantage of distributed computations. In fact it does not scale after reaching a number of workstations given by the algorithm.

Algorithms Token Based NDFS and Back Level Edges are also good at searching for counterexamples, but on valid specifications they are almost not able to finish – their run times are many times higher than run times of other tools. These algorithms are not very useful in practice because model checking is a complete method and it needs to give a result even when a verified property holds. Furthermore, Property Driven NDFS outperforms both algorithms on most of inputs and where it does not, there the difference is little.

The last algorithm, Deps Cycle Detection is very hard to compare because it has so big memory requirements, that it is not able to finish on almost any input. An appendix of states in this algorithm is very large, which is the probable reason of all these memory overflows. We believe that after some optimizations results could be better and the comparison will be repeated, but currently the the tool based on this algorithm is not usable in practice at all.

20 workstations	PLC1	Anders5	Bakery1	Bakery6	MCS2	MCS3
NCD	2.4	142.1	5.3	53.4	75.2	1.0
OWCTY	1408.1	528.0	11.1	1.8	253.4	137.7
OWCTY reversed	868.8	159.2	3.7	2.1	85.2	50.2
Distr. MAP	0.7	411.3	1.2	0.9	126.6	0.4
Prop. Driven NDFS	0.8	919.9	0.2	12.7	465.5	2.7
Token Based NDFS	0.8	TE	0.4	87.5	20408.2	55.1
Back Level Edges	5.1	TE	41.6	4.7	31211.6	1.8
Deps Cycle Det.	1.0	ME	4.2	4.4	ME	1436.7
	Peterson7	LUP2	LUP3	Elevator1	Elevator2	Phils1
NCD	86.7	197.8	52.8	2004.1	1477.4	1.7
OWCTY	291.2	527.8	398.7	1715.3	4138.3	2083.9
OWCTY reversed	113.9	174.6	126.6	ME	ME	ME
Distr. MAP	43.3	481.6	64.9	ME	ME	ME
Prop. Driven NDFS	2.9	1785.4	4.7	ME	0.8	84.0
Token Based NDFS	8.3	TE	10.6	TE	29196.3	68.9
Back Level Edges	98.0	14077.1	514.4	41.6	ME	9.9
Deps Cycle Det.	454.0	ME	338.9	ME	ME	3.7

10 workstations	PLC1	Anders5	Bakery1	Bakery6	MCS2	MCS3
NCD	1.9	180.7	5.0	33.8	90.8	0.3
OWCTY	1627.6	670.6	12.2	1.6	336.0	186.8
OWCTY reversed	955.3	207.3	3.9	1.0	110.8	73.7
Distr. MAP	0.6	693.9	1.0	0.6	239.6	1.0
Prop. Driven NDFS	0.1	1964.6	0.1	9.5	530.5	2.3
Token Based NDFS	0.5	TE	0.1	63.1	15141.8	40.4
Back Level Edges	3.2	TE	41.8	2.7	TE	0.5
Deps Cycle Det.	0.7	ME	5.1	2.9	ME	3100.8
	Peterson7	LUP2	LUP3	Elevator1	Elevator2	Phils1
NCD	274.0	254.4	15.6	3539.1	1934.9	ME
OWCTY	376.2	679.3	508.3	2555.0	ME	3157.9
OWCTY reversed	166.2	227.3	172.2	ME	ME	ME
Distr. MAP	104.2	732.5	106.2	ME	ME	ME
Prop. Driven NDFS	2.2	2155.6	3.7	ME	0.6	45.5
Token Based NDFS	6.4	TE	7.7	TE	20028.2	46.9
Back Level Edges	289.7	16909.1	947.4	ME	ME	2.1
Deps Cycle Det.	464.3	ME	433.5	ME	ME	0.3

underlined models have an accepting cycle
ME = memory limit exceeded
TE = time limit (10 hours) exceeded

Table 4.2: Comparison of model checkers currently implemented in DIVINE

Chapter 5

Performance Analysis of State Generation

The main author's contribution to DiVinE is the state generation module. Therefore it is apposite to compare it with existing state generators contained in other model checkers and to examine the influence of optimizations to a speed of state space generation.

In Section 5.1, the workstation with 4 GB RAM and two 3 GHz Xeon CPUs was used. The rest of experiments were run on the computer with 2 GB RAM and one 3 GHz Pentium 4 CPU.

5.1 Comparison of State Generation with SPIN

A speed of state generation is one of major aspects influencing a performance of model checkers. In the field of LTL model checking of discrete time finite state systems, the model checker SPIN [SPI] plays a dominating role. It is also the reason, why the interpreter of Promela has been integrated into DiVinE. The state generator of SPIN is used here for a comparison with the state generator of DiVinE.

Altogether five state generators were compared:

- DiVinE state generator on DVE inputs
- DiVinE state generator on Promela inputs
- NIPS experimental state generator on Promela inputs [NIP]
- SPIN state generator using BFS without optimizations on Promela inputs[1]
- SPIN state generator using DFS without optimizations on Promela inputs[2]

To get the same conditions for all state generators, same systems were used. For this reason they had to be written in both Promela and DVE modeling languages. The mutual exclusion protocol and the leader election protocol from the SPIN example models collection have been used. The important aspect is, that the size of state space is preserved exactly the same in both modeling languages.

NIPS and SPIN state generators differ slightly in an interpretation of `break` because NIPS regards it to be a standalone command, while SPIN merges a jump from `break` with a previous command. This is not really a bug because the Promela language permit arbitrary number of skips between any two commands, which does not influence a model checking of LTL formulas without the operator **X**. To get the same size of a state space for SPIN it was necessary to replace all commands `break` with a compound command `skip;break`.

1. SPIN was used with parameters -o1 -o3 and a model checker compiled with -O3 -DBFS -DNOREDUCE
2. SPIN was used with parameters -o1 -o3 and a model checker compiled with -O3 -DNOREDUCE

	DIVINE DVE	DIVINE Promela	NIPS	SPIN+BFS	SPIN+DFS
Peterson	103.1	177.9	123.8	37.8	32.0
Leader election	112.6	211.4	139.6	64.1	56.5

Table 5.1: Run times of various state generators on the same model

It was also necessary to switch off all optimizations of model because any such optimization caused the state space reduction, which was not the matter to measure in this experiment.

For fair tests also hash tables storing generated states have to be of approximately same sizes (a number of conflicts in a hash table influences results substantially). For DIVINE and SPIN a number of hash table entries was set to 2^{24}, for NIPS, where hash table is characterized by two values, the number of entries was set to 131072 (maximal possible value) and a number of so called *retries* was set to 500 (default value).

It is not surprising that SPIN+BFS outperforms DIVINE because SPIN converts a model to C++ code and compiles it to a binary executable which generates states of the model. This is very effective method compared to the simple interpretation used in DIVINE. The slower method has been chosen in DIVINE because it fits better into the modular design of the environment and it is much easier to implement and debug. However we do not want to resist a development of a compiled state generator in future.

The difference in speed of DIVINE Promela and NIPS generators is much more surprising – unfortunately in a negative way. They have the same interpreter of Promela inside, therefore the slow-down in DIVINE Promela has to be caused by components of the DIVINE LIBRARY or slow constructs in an implementation of the generator. Because the generator passed a code revision recently, main efficiency losses must be caused by an implementation of BFS queue, by an implementation of storage module or generally by an inefficient memory management.

An origin of the inefficiency is further studied in Section 5.3.1. It is believed that the performance of problematic parts will be improved in future substantially, which would speed up both DIVINE generators. The DVE state generator could then probably run at least at half speed of SPIN+BFS generator.

The difference between SPIN+BFS and SPIN+DFS is probably caused by a better optimized code of DFS (e. g. an implementation of the DFS stack can be more effective due to its fixed length given from a command line). SPIN+DFS run times are provided here mainly because it is a default option in SPIN.

5.2 Comparison of Abstract Interfaces with Non-abstract Ones

The concept of abstract interfaces is presented in Section 3.6.4. Although it has not been fully implemented, it has been used for an incorporation of the Promela state generator into DIVINE (see Section 3.2.1). We had a little concern about the speed of code using C++ virtual methods. To get rid of the fear of virtual methods, a short comparison of two DVE state generators was made. The first, compiled from DIVINE source repository from 10th September 2005 have no virtual functions inside. The second one from 20th September 2005 contains abstract interfaces of classes related to the state generator. A table in Table 5.2 shows a comparison of both generators on three inputs. From results it follows that abstract interfaces

	Peterson7	MCS3	LUP3
old generator	493.9	188.5	1074.5
generator with abs. ifc.	505.8	194.0	1094.7
slow-down	2.4%	2.9%	1.9%

Table 5.2: Comparison of an old state generator with a newer one containing abstract interfaces

cost a little in comparison to flexibility brought to whole library.

5.3 Optimizations

Till now there has not been much time for optimizations. In spite of this fact two basic observations were made helping us to understand better some corners of C++ and Standard Template Library (STL).

5.3.1 Queues

Section 5.1 states that an implementation of the BFS queue can be one of the causes of the state space generation slow-down in DiVinE. generator uses the queue from STL. To check the speed of the queue it was necessary to develop an alternative implementation which could be considered fast. The code of this queue can be found in appendix C. The queue allocates an optionally large block of memory at a time and thus saves time for allocation of small pieces.

Using a simple artificial test consisting of a large number of adding and removing operations on a queue it was found out, that the STL implementation is by 27,5% faster than the experimental implementation. Moreover, experiments in generator show, that the replacement of STL queue by the experimental version do not cause any substantial difference in a speed of state space generation. The deviation of measurement is about 1 s, therefore there is probably almost no difference between these two generators:

	Peterson7	MCS3	LUP3
generator with STL queue	155.9	57.1	302.2
generator with exper. queue	155.3	57.0	305.2

From these run times it is obvious that the implementation of the BFS queue does not slow down the state space generation noticeably.

5.3.2 Simple change with big impacts

Similarly as in case of queues it was also interesting, whether an STL implementation of stack is sufficiently efficient because it is used in an evaluation of DVE expressions which is called usually hundreds or more times during each successors generation (expressions are contained in guards, synchronizations and effects).

During a code inspection of the method evaluating expressions it was found out that stacks are declared locally on the level of the method. They were moved[3] to the global decla-

3. In fact, they had to be changed from `std::stack` to `std::vector`, but this change had no effect on speed,

rations of the class owning the method. This change has probably speeded up an expression evaluation many times because generator currently runs several times as fast as before. The following table shows that the state space generator runs faster now depending on the frequency of calls of the expression evaluation for a given model.

	Peterson7	MCS3	LUP3
generator with local stacks	532.6	201.4	1164.5
generator with global stacks	217.9	58.1	333.7
speedup	2.4×	3.5×	3.5×

The difference in speed is caused by the elimination of creation of stacks each time the evaluation of expression is called. Creation of STL containers is an expensive operation because they usually preallocate a relatively big piece of memory and the repeated allocation and deallocation cost a lot of time. Clearing of stacks takes almost no time because no memory is released by this operation.

as it was verified by several simple tests. The change was necessary because std::stack has not the method `clear()`

Chapter 6

Future Work

Because the project is permanently under development, there are many aims to be fulfilled, many units to be extended and also some bugs to be fixed. This chapter tries to find the most important goals to achieve in the near future and it also criticizes some problematic points.

6.1 Modeling Languages and Interpreters

The type system of the DVE modeling language is currently very poor. It contains only unsigned 8-bit and signed 16-bit integer types. The evaluation of numbers then works over signed 32-bit arguments (8-bit and 16-bit values are casted to it), which is very unusual and it has to be fixed. More integer types and bit vectors have to be added. It will probably cause a little slowdown of the interpreter, but it will also allow to create a greater number of more realistic models.

We believe that collaboration with other projects can bring more modeling languages and their interpreters into DIVINE in the similar way as in case of the Promela interpreter NIPS. We also need to integrate NIPS better because the current level of integration does not allow any syntax-dependent operations – for instance a partial order reduction.

Because experiments in Section 5.1 have shown that interpretation of models slows down state space generation, a translator of DVE models into a C++ code could be created generating states similarly to SPIN. To preserve the modularity of DIVINE the C++ code could be compiled to shared library (plugin) with nice and neat abstract interface of the state generator. It is also possible to imagine doing the same with SPIN state generators and thus, gaining very fast state generator of Promela models. But this can be very hard to implement because SPIN is known to be very complex and it is probably not easy to automatically manipulate C source codes of state generators produced by it.

The long sequence of translations (in case of NIPS interpreter) and preprocessing (in case of both currently present interpreters) blur a backward correspondence of read models with their original sources. It is planed to add the system of back references which enables to assign a position in the original source code to each element of read model.

6.2 GUI

Although the graphical user interface was introduced in July 2005, it has not been given to public yet and a sedulous work is currently made to create a really stable version of it including a user management and counterexample tracing.

We would like to provide an interactive simulator using GUI and generally make the interface more user-friendly. It is also necessary to have a good documentation for it and easy-to-use installation package of the server part of GUI. It should be possible to use the interface naively without a care of its client/server architecture.

6.3 Grids, Heterogeneous Clusters and Load Balancing

Because it is possible that DIVINE will be used on big external grids of computers, a need for cooperation with an external grid middleware may arise. It can be very hard to implement support for it. Moreover, there are so many implementations of grid middleware that a compatibility will probably be guaranteed only for a small subset of them.

Currently, DIVINE allows distribution only to isolated clusters of computers and no inter-cluster computations are allowed. This limitation is given by the currently used MPI implementation, but we hope for breaking it in future using other version of MPI or by our own software solution.

The support for grids and inter-cluster computations will bring new problems and challenges to developers of DIVINE. Till now, all computations have been made on homogeneous clusters, i. e. all computers have had the same memory size and CPU speed. If we allow cooperation of computers from different clusters, the unbalancing of machine loads will surely appear. The unbalanced loads cause necessarily the degradation of performance, therefore load balancing techniques will be studied, created and implemented to prevent it.

6.4 Abstract Interfaces

The concept of abstract interfaces is described in Section 3.6.4. It is already used in the state generator, but the rest of the library has to be remade gradually too. We plan to begin with the storage module and probably also compressions which are closely related. After that, it will be possible to compare different implementations of storage, choose the most effective one and do experiments with them.

6.5 Completion of the DIVINE TESTSUITE

As already stated in Section 3.4, the DIVINE TESTSUITE is not completed at all. Tools for easy model evaluation, selection and presentation of results are almost completely missing. This makes an experimental work on DIVINE as uncomfortable as on any other model checking tool, but DIVINE should be better, especially because the possibility of experimental work is one of its main goals.

After creation of the basic tools, it would be surely useful to set up a public database of models, where everyone could easily add and maintain his/her model. Access to the database and support for experiments should be also incorporated into the graphical interface.

6.6 Support for More Logics

Currently, only LTL is supported in DIVINE. Nowadays, also the support for alternation-free mu-calculus is under development as a student project. DIVINE does not favor any logic, support of LTL follows from the existing scientific work of ParaDiSe Laboratory members. Support for CTL and other well known logics for property specification would be a welcome contribution to the project.

Conclusions

In the first part, the enumerative model checking is introduced and both syntax and semantics of the DVE modeling language are defined. The interpreter of DVE incorporated as a state generator is the main contribution of the author to DIVINE. The proposed modeling language is simple and easy to understand. On one hand, because of omission of all high-level constructs, it is laborious to write more complex models in it, but on the other hand, it is very easy to do any static analysis of models written in it.

The second part introduces DIVINE as a modular project and describes its most important parts. The DIVINE LIBRARY facilitates the implementation of distributed model checkers. In addition to DVE, it also allows to interpret Promela, which is helpful for people having their models written in this widespread modeling language. Although the library is currently quite stable and several moder checkers were implemented using it, some of its parts will have to be restructured according to the concept of abstract interfaces introduced in Section 3.6.4. We also plan to add support for load balancing to the library (as described in Section 6.3). Experiments in Chapter 5 show that the DVE state generator is sufficiently fast compared to the SPIN generator – the difference in speed can be easily compensated by the distribution.

The DIVINE TOOLSET contains model checkers and various tools useful for modeling and visualization. Model checkers are compared in Chapter 4. The test has shown that there is a group of five efficient distributed model checkers and it reveals some weaknesses of them. There is also explained, why the remaining three model checkers can hardly be used in practice. The DIVINE TOOLSET currently provides the set of efficient algorithms for LTL model checking developed according to the latest research on this field.

The DIVINE TESTSUITE contains a collection of models. These models are sufficiently large to be used for distributed model checkers evaluation. An automated evaluation of models and model checkers is not implemented yet, but it will be added in near future and we believe that it can be very useful for experimental work in DIVINE.

The DIVINE GUI is under intensive development and it will be offered to public soon. It adds no functionality to model checkers, but it makes the usage of DIVINE easier and user friendlier. We plan to provide our cluster using the client/server architecture of GUI to external users (they will only need to download the client, the server will run on our cluster). This way users are protected from a need to install and configure DIVINE, when they only need to run a simple task or they only want to try it out. The DIVINE GUI provides an interface expected from a model checking tool.

The DIVINE project is open source and its codes are distributed under the terms of GNU GPL [GPL]. We try to cooperate with other projects – this way we want to make DIVINE more popular and gain new extensions as in case of the Promela state generator from the RWTH Aachen [NIP, Div]. We believe that DIVINE could play a major role on the field of distributed model checking.

Bibliography

[AC] Autoconf.
URL: `http://www.gnu.org/software/autoconf/`.

[AM] Automake.
URL: `http://www.gnu.org/software/automake/`.

[Bar04] Jiří Barnat. *Distributed Memory LTL Model Checking*. PhD thesis, Faculty of Informatics, Masaryk University Brno, 2004.

[BB03] L. Brim and J. Barnat. Distribution of Explicit-State LTL Model-Checking. In Thomas Arts and Wan Fokkink, editors, *Electronic Notes in Theoretical Computer Science*, volume 80, pages 1–6. Elsevier, 2003.

[BBC03] J. Barnat, L. Brim, and J. Chaloupka. Parallel Breadth-First Search LTL Model-Checking. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 106–115. IEEE Computer Society, Oct. 2003.

[BBC05] J. Barnat, L. Brim, and J. Chaloupka. From Distributed Memory Cycle Detection to Parallel LTL Model Checking. *Electronic Notes in Theoretical Computer Science*, 133(1):21–39, May 2005.

[BBv02] J. Barnat, L. Brim, and I. Černá. Property Driven Distribution of Nested DFS. In Michael Leuschel and Ulrich Ultes-Nitsche, editors, *Proceeding of the 3rd International Workshop on Verification and Computational Logic (VCL'2002)*, number DSSE-TR-2002-5 in DSSE Technical Report, pages 1–10, Pittsburgh, PA, USA, October 2002. Dept. of Electronics and Computer Science, University of Southampton, UK.

[BBvv05] J. Barnat, L. Brim, I. Černá, and P. Šimeček. Divine - the distributed verification environment. In M. Leucker and J. van de Pol, editors, *Proc. of 4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC05)*, pages 89–94, 2005.

[BDL04] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on Uppaal. In *SFM*, pages 200–236, 2004.

[Bis] Bison.
URL: `http://www.gnu.org/software/bison/bison.html`.

[BvKP01] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In *Proc. of Foundations of Software Technology and Theoretical Computer Science (FST TCS 2001)*, volume 2245 of *LNCS*, pages 96–107. Springer, 2001.

[BvMv05] L. Brim, I. Černá, P. Moravec, and J. Šimša. How to order vertices for distributed ltl model-checking based on accepting predecessors. In *Proceedings of the 4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2005)*, volume 1M0545, pages 1–12, 2005.

[CB97] 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.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999.

[DHHR05] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, and Robby. Building your own software model checker using the bogor extensible model checking framework. In *CAV*, pages 148–152, 2005.

[Div] DiVSPIN.
URL: `http://www.fi.muni.cz/paradise/divspin.html`.

[Dox] Doxygen.
URL: `http://www.doxygen.org/`.

[GMS01] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proc. SPIN Workshop on Model Checking of Software*, volume 2057 of *LNCS*, pages 215+. Springer, 2001.

[GPL] GNU GPL.
URL: `http://www.gnu.org/copyleft/gpl.html`.

[HLL04] Fredrik Holmén, Martin Leucker, and Marcus Lindström. UppDMC – a distributed model checker for fragments of the μ-calculus. In Lubos Brim and Martin Leucker, editors, *Proceedings of the* $3rd$ *Workshop on Parallel and Distributed Methods for Verification*, volume 128/3 of *Electronic Notes in Computer Science*. Elsevier Science Publishers, 2004.

[HP98] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder, 1998.

[JEK$^+$90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[MPI] The message passing interface (MPI) standard.
URL: `http://www-unix.mcs.anl.gov/mpi/`.

[NIP] A virtual machine for state space generation.
URL:
`http://www-i2.informatik.rwth-aachen.de/Research/vmssg/`.

[PW83] A.P. Sistla P. Wolper, Moshe Y. Vardi. Reasoning about infinite computation paths. In *Proceedings of 24th IEEE symposium on foundation of computer science*, pages 185–194. IEEE Computer Society Press, 1983.

[Rob00] Robby. Bandera specification language: A specification language for software model checking, 2000. Master's thesis, Kansas State University.

[SMV] The SMV System.
URL: `http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.ps`.

[SPI] On-the-fly, LTL model checking with SPIN.
URL: `http://spinroot.com/`.

[SpL] Spin version 4: Language reference.
URL: `http://spinroot.com/spin/Man/promela.html`.

[Var96] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata*, pages 238–266, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

Appendix A

Support for Development

This chapter contains a technical information about support for development, documentation and package distribution.

In development of DiVinE, the following software is used (the used version is in parenthesis):

- C and C++ compiler (GCC 3.2.3 or higher recommended) – compilation of source codes of the DiVinE Library and the DiVinE ToolSet
- MPI (implementations MPICH or LAM/MPI recommended) – support for distributed computations
- Autoconf (2.59) – creation of platform independent software package
- Automake (1.9.5) – automatic Makefile generation
- GPerf (3.0.1) – a perfect hash function generator – needed in syntax analysis
- Flex (2.5.4) – a fast lexical analyser generator – needed in syntax analysis
- Bison (2.0) – a parser generator that converts a grammar description for an LALR context-free grammar into a C program – needed in syntax analysis
- Doxygen (1.4.4) – a documentation system for C++, C, Java and other programming languages
- CVS (1.11.19) – a source codes repository
- Perl (5.8.6) – a cross platform programming language and its interpreter – used for implementation of various scripts
- Bourne-Again Shell and related command line tools (sed, grep, find, ...) – used for implementation of various scripts (GNU implementation recommended)
- Graphviz (2.6) – Graph Visualization Software – used in tools for state space and automata printing
- Gnuplot (4.0) – a data and function plotting utility – used in plotlog
- Java Runtime Environment (1.5) – required for development and execution of GUI
- XWeb (1.0rc3) – a set of tools to generate complete websites out of a set of XML and XHTML input files – needed for generation of DiVinE web site

Autoconf and Automake offer both automatic Makefile generation and platform independent software package creation. It is used for compilation of C and C++ codes of the DIVINE LIBRARY and DIVINE TOOLSET. It is configured to distinguish the compilation during a development phase (debugging information included, no optimization enabled) from the compilation of distributed ready-to-use package (debugging information excluded, the maximal level of optimization enabled). This way a developer gets fast compilation and debugging, while a user gets the maximally fast tools and library. Autoconf and Automake are also used for platform-dependent modifications in DIVINE TOOLSET scripts. The main configuration files are:

- `configure.ac` – configuration of both Autoconf and Automake
- `acinclude.m4` – files included to the configuration of Autoconf
- `Makefile.am.global` – file included into all files `Makefile.am` in the DIVINE LIBRARY and the DIVINE TOOLSET

Both tools are also used for compilation of Java source codes of the graphical interface.

Doxygen in connection with Automake form a flexible documentation system. Documentation is organized in tree-like manner (using file system directories), where each node contains the file `doxyfile.in` setting up Doxygen for the subtree of documentation. A new node of the documentation tree can be created using the script make_documentation_dir.sh. The configuration of the root node (i. e. the global options) can be found in the directory `support/etc/doxygen` in the source repository.

Although using automatically generated Makefile system it is possible to create a distribution package simply using command `make dist`, it was necessary to automatize the package creation yet more. The script publish_distribution.sh checks out the source repository, builds a package from its source codes, publishes the package to the Internet and installs it for local users. Using this script it is possible to publish the current version of DIVINE by a single command. For deletion of published package it is possible to use the script delete_distribution.sh. The script refresh_doc.sh automatically regenerates documentation for the current version of DIVINE in the source repository.

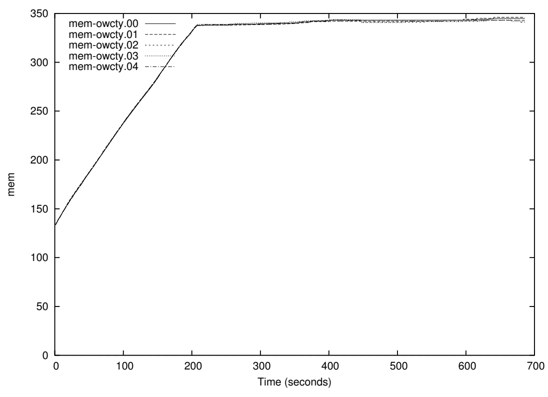
The set of scripts and configuration files described above makes a development easier, allows to publish a new version of DIVINE by a single command and make the distribution package of DIVINE comparable to other open source distributions.

Appendix B

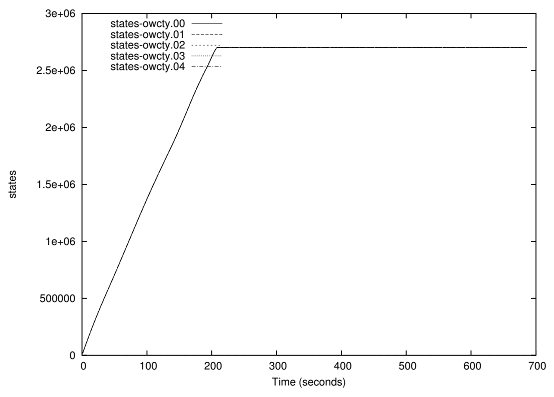
plotlog Outputs

Sample plotlog output on the run of OWCTY (see Section 4.3) with input Peterson7 from the testing set (see Section 4.1).

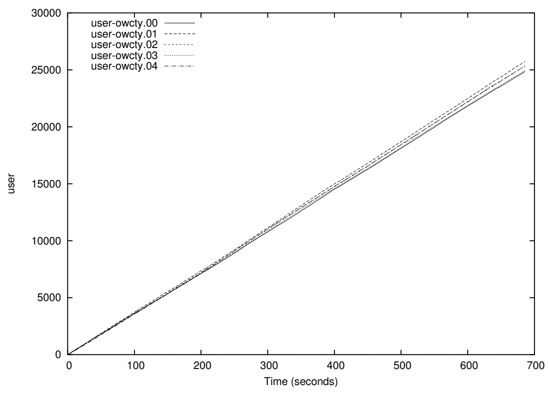
Memory consumption in MB



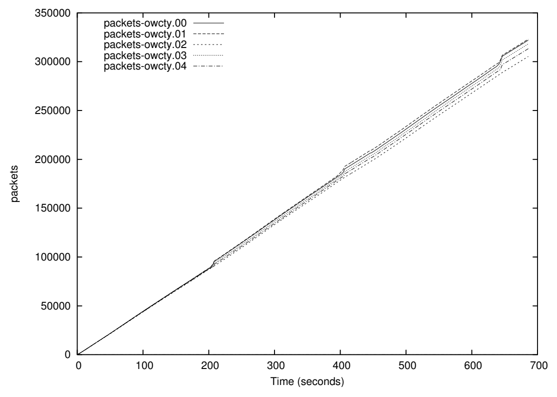
Count of states



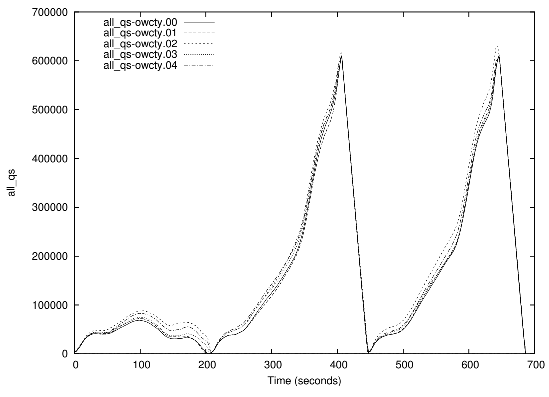
Time spent in user space



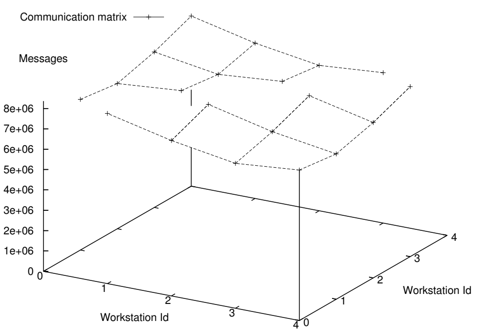
Transmitted packages



Custom output showing the size of queues



Communication matrix



Appendix C

Source code of fast queue

```
template <class T>
class queue_t
{
 const std::size_t seg_size;  //size of a single allocation unit (segment)
 const std::size_t max_elems; //seg_size/sizeof(T)
 std::size_t count;           //count of elements currently stored
 std::queue<T*> seg_pointers; //queue of segments
 T* front_seg_pointer;        //pointer to the segment containing a front of q.
 T* back_seg_pointer;         //pointer to the segment containing a back of q.
 std::size_t front_elems;     //read elements from the front of the queue
 std::size_t back_elems;      //written elements to the back of the queue

 //method for allocation of new segment
 T * new_segment() { return (T*)(malloc(max_elems*sizeof(T))); }

 public:
 //A constructor
 queue_t<T>(std::size_t segment_size = 4096): seg_size(segment_size),
    max_elems(segment_size/sizeof(T)), count(0),
    front_seg_pointer(0), back_seg_pointer(0),
    front_elems(max_elems), back_elems(max_elems)
 { }

 //A destructor
 ~queue_t<T>()
 {
  while (!seg_pointers.empty())
   { free(seg_pointers.front()); seg_pointers.pop(); }
 }

 //Append a new element to the end of queue
 void push(const T to_push) {
   count++;
   if (back_elems>=max_elems) //allocation of new segment if necessary
    {
     back_seg_pointer = new_segment();
     back_elems=0;
     seg_pointers.push(back_seg_pointer);
     if (front_seg_pointer==0) //if it is the first segment update front info
      { front_seg_pointer = back_seg_pointer; front_elems=1; }
    }
   else { back_seg_pointer++; }
   //copy an element into the queue
   memcpy(back_seg_pointer, &to_push, sizeof(to_push));
   back_elems++;
 }

 //Remove an element from the beginning of queue
 void pop() {
   count--;
   if (empty()) std::cerr << "UNDERFLOW" << std::endl;
   if (front_elems>=max_elems) //deallocation of a segment if it is the last
    {                          //element in it
     free(seg_pointers.front());
     seg_pointers.pop();
     if (seg_pointers.size())
      {
       front_seg_pointer = seg_pointers.front();
       front_elems=1;
      }
     else { front_seg_pointer = back_seg_pointer = 0; }
    }
   else { front_seg_pointer++; front_elems++; }
 }

 //Return true iff queue is empty
 bool empty() const
  {return (!front_seg_pointer ||  //queue has no allocated segments or
           //front and back of queue are on the same place in the last segment
          (seg_pointers.size()==1 && front_elems>back_elems));}
 //Returns a count of elements stored in queue
 std::size_t size() const { return count; }
 //Returns a front element of queue
 T & front() { return (*front_seg_pointer); }
 //Returns a back element of queue
 T & back()  { return (*back_seg_pointer); }
 //The same as above, but for constant instances of queue
 const T & front() const { return (*front_seg_pointer); }
 const T & back() const { return (*back_seg_pointer); }
};
```

Appendix D

Class Structure

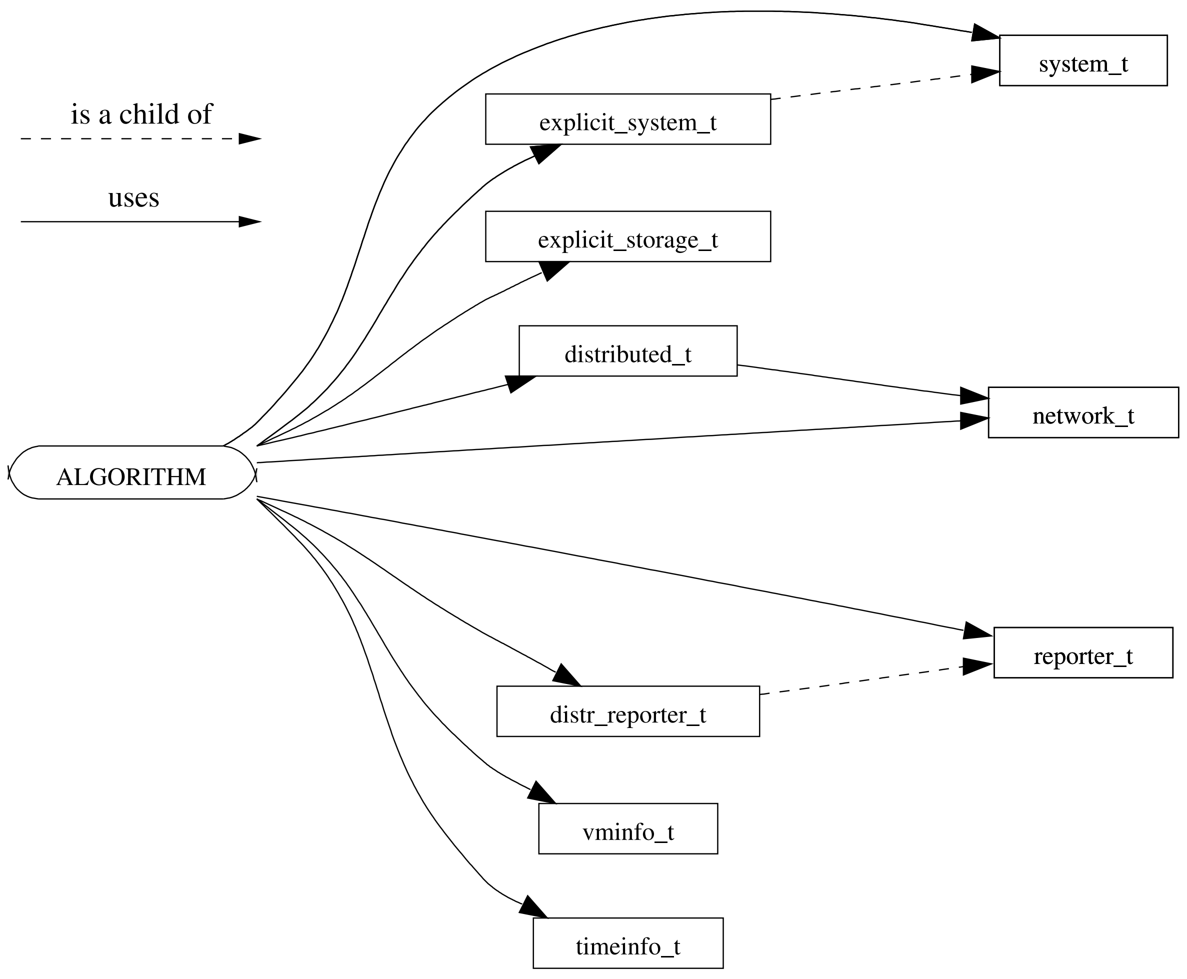


Figure D.1: The structure of a model checker as a graph of classes

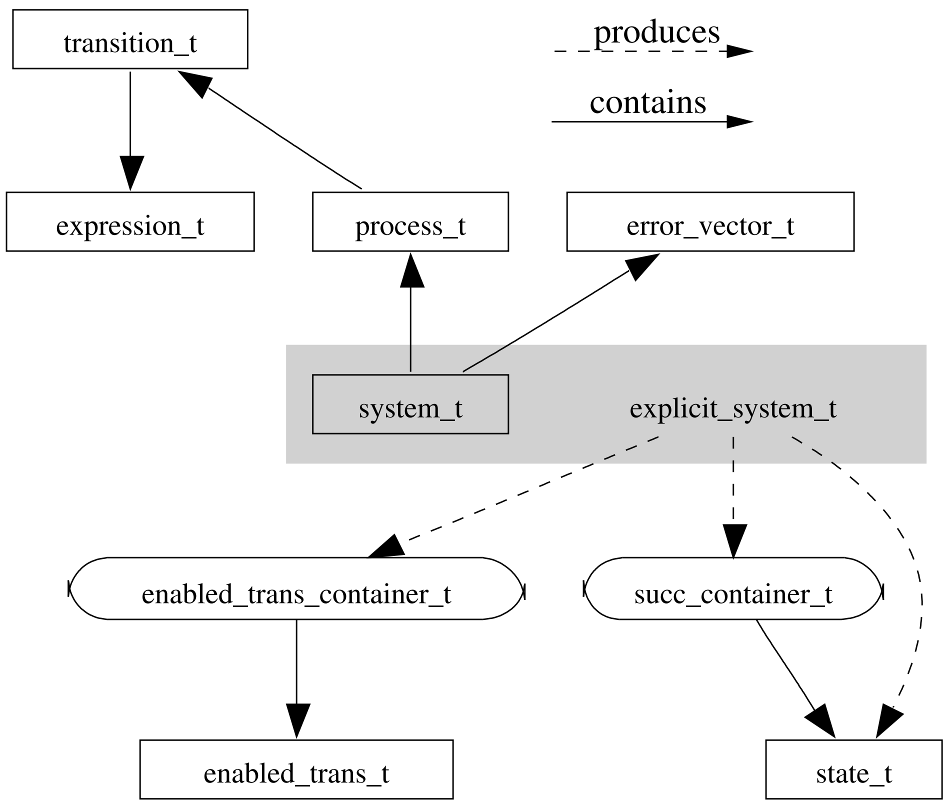


Figure D.2: The structure of a model representation and a state generator