TVT tutorial
Juha Nieminen, Terhi Kilamo,
Timo Kivelä, Jaco Geldenhuys and Timo Erkkilä
Contents

I TVT toolkit

1 Introduction 5
  1.1 General about TVT ................................. 5
  1.2 This document .................................. 5
  1.3 History of the tool ............................. 5

2 Theoretical concepts 7
  2.1 LSTS ........................................... 7
  2.2 Parallel composition of LSTs .................... 7

3 Modelling issues 8
  3.1 .................................................. 8

II Program syntax reference 9

4 Using TVT 10
  4.1 The Basics ....................................... 10
  4.2 Command line of the programs ................... 10

5 Input language compiler (tvt.compiler) 12
  5.1 Syntax ........................................... 12
  5.2 Examples ....................................... 15

6 Parallel composition (tvt.parallel) 17
  6.1 Syntax ........................................... 17
  6.2 Examples ....................................... 18
  6.3 The parallel composition rules file ............... 18
    6.3.1 Action synchronization rules ................. 19
    6.3.2 State proposition parallel composition rules . 19
      Types of state propositions ...................... 19
      State propositions in an LSTS file .............. 19
      Syntax of state proposition rules .............. 20
      Using on-the-fly verification ................... 21
      How permanent and cutting propositions work .. 22
      Defining a prefix for permanent and cutting propositions ... 22
      Hiding permanent propositions .................. 23
7 Classic rules file creation (tvt.createrules) 24
   7.1 Syntax ................................................. 24
   7.1.1 Renaming/hiding ..................................... 25
   7.1.2 Using regular expressions in renaming/hiding .......... 25
   7.2 Examples .............................................. 26

8 Extended rules file 28
   8.1 The extended rules file .................................. 28
   8.2 Aliases ................................................... 30
   8.3 Action name parameters and wildcards ...................... 30
   8.4 State proposition parallel composition rules ............. 31
   8.5 Syntax .................................................. 31
   8.6 Example ................................................ 32

9 Error tracing an interrupted LSTS (tvt.errortracer) 33
   9.1 Syntax .................................................. 33
   9.2 Example ................................................. 33

10 Creating an acceptance graph (tvt.lsts2ag) 34
   10.1 Syntax .................................................. 34
   10.2 Examples ................................................ 34

11 Determinizing (tvt.detlsts) 35
   11.1 Syntax .................................................. 35
   11.2 Examples ................................................ 35

12 Strong bisimulation reduction (tvt.SBreduduction) 36
   12.1 Syntax .................................................. 36
   12.2 Examples ................................................ 36

13 Strong bisimulation comparison (tvt.SBcompare) 37
   13.1 Syntax .................................................. 37
   13.2 Examples ................................................ 37

14 Strong bisimulation reduction (tvt.ReduceKsnp) 38
   14.1 Syntax .................................................. 38
   14.2 Examples ................................................ 38

15 Converting acceptance graphs to tester processes (tvt.ag2tp) 40

16 Tester rules creation (tvt.testerrules) 41
   16.1 Syntax .................................................. 41

17 Converting back to LSTS (tvt.ag2lsts) 42
   17.1 Syntax .................................................. 42
   17.2 Examples ................................................ 42

18 The file format tool (tvt.fftool) 43
   18.1 Syntax .................................................. 43
19 The LSTS simulator (tvt.simulate) 44
  19.1 Syntax 44

20 Illustrator (tvt.illux) 45
  20.1 Syntax 45
     20.1.1 Syntax of the style file 45
  20.2 Examples 46
  20.3 The graphical user interface 46

21 Shell scripts 47
  21.1 tvt.CFFDcompare 47
  21.2 tvt.CFFD_normalize 47

22 Scripting language of input files 48
  22.1 Preface 48
  22.2 Command syntax 48
  22.3 The syntax of the mathematical expressions 50
  22.4 Making loops 50
  22.5 Example 51

Bibliography 52
Part I

TVT toolkit
1. Introduction

1.1 General about TVT

Tampere Verification Tool or TVT for short is a collection of programs for automated verification of concurrent and reactive systems. TVT has its roots in process algebras and explicit state space exploration, but in addition to actions, our formalism allows use of state-based information in the form of truth-valued state-propositions.

The core concept of the tool is the labelled state transition system i.e LSTS. LSTS is used to represent the behaviour of a complete system or the behaviour of one of the system’s components.

The tool supports several different semantics for the LSTS. The most important ones to be mentioned here are the strong bisimulation equivalence and especially the CFFD equivalence. A more in-depth look at the different semantics supported by the tool can be found in [Val00] (unfortunately in Finnish).

1.2 This document

The tool consists of many batch processed programs that can be run from the command line. The first part of this document contains an overview of theory, history and key features of TVT.

The second part is meant as a common tutorial for the programs that are fully functional at present. Presented are the common form of the command line used in the TVT programs. Additionally, since each program has its own characteristics, the purpose and usage of all the programs is described separately.

1.3 History of the tool

The story of Tampere Verification Tool (TVT) started at the beginning of 1990’s, when CFFD semantics was first introduced [?]. CFFD describes an abstraction of the behaviour of a system. It is the weakest congruence relation with respect to the composition operators of process algebras that preserves both stuttering-insensitive properties specified in linear temporal logic and deadlocks [?]. Based on CFFD, the “Advanced Reachability Analysis” (ARA) tool [?] was developed. ARA supports LOTOS as the modelling language, compositional construction of a system, visual verification, and CFFD equivalence comparison. ARA was used in the industry even a decade later [?]. Eventually, ARA became difficult to maintain, LOTOS proved ill-suited for verification applications [?], and many new verification ideas emerged. When Nokia offered us a contract for developing a new verification tool in 1999, we started the development of TVT. TVT was intended to be used both in Nokia Research Center and as a platform for developing and testing new verification ideas in Tampere University of Technology.
TVT has been made freely available for academic use under the Nokia Open Source licence and can be downloaded from [Val00].
2. Theoretical concepts

2.1 LSTS

The formalism used as a model of the behaviour of a process is a labelled state transition system or an LSTS. An LSTS is a labelled transition system (LTS), where the states can be labelled with truth-valued propositions.

Definition 1 (Labelled state transition system) A labelled state transition system or an LSTS is the tuple $\langle S, \Sigma, \Delta, s, \Pi, \Upsilon, \text{val} \rangle$, where LTS $\langle S, \Sigma, \Delta, s \rangle$ is augmented with the set of propositions $\Pi$, evaluation function $\text{val} : S \rightarrow 2^{\Pi}$, and the set $\Upsilon$ of permanent propositions for which $\Upsilon \subseteq \Pi$.

The actions of an LSTS are used to talk about how the components in a system interact with each other and the environment. In other words, the LTS-part of an LSTS describes the behaviour of the system. The propositions have been added to exploit the fact that in state-based models it is easier to express properties that depend on the global state of the system [?].

2.2 Parallel composition of LSTSs

To define parallel composition of LSTSs, synchronization rules are needed. The rules state which actions are executed together.

Definition 2 (Synchronisation rule) Let $L_1, \ldots, L_n$ be LSTSs, with alphabets $\Sigma_1, \ldots, \Sigma_n$. We assume that “−” $\not\in \Sigma_i$. Let $\Psi$ be any set of symbols with $\tau \not\in \Psi$. A synchronisation rule for $L_1, \ldots, L_n$ and $\Psi$ is a $(n + 1)$-dimensional vector, written as $\langle a_1, \ldots, a_n; a \rangle$, for which

- $\forall i \in \{1, \ldots, n\} : a_i \in \Sigma_i \cup \{“−"\}$
- $\exists i \in \{1, \ldots, n\} : a_i \neq “−”$
- $a \in \Psi \cup \{\tau\}$

Definition 3 (Proposition rule) Let $L_1, \ldots, L_n$ be LSTSs with proposition sets $\Pi_1, \ldots, \Pi_n$. Then a proposition rule for them is a pair $(\phi, \pi)$, where $\pi$ is a name, and $\phi$ is a boolean expression formed with boolean variables of the form $(p, i)$, where $p \in \Pi_i$, connectives $\{\vee, \wedge, \neg\}$ and constants True and False. In a proposition rule $\phi$ is called the expression part and $\pi$ the result.

For a collection of states $(s_1, \ldots, s_n)$ we assign truth values to each $(p, i)$ as True iff $p \in \text{val}_i(s_i)$ and False iff $p \not\in \text{val}_i(s_i)$. If $\phi$ then evaluates as True, we write $(s_1, \ldots, s_n) \models \phi$. 

instructions.tex(1.2) 15.4.2012 7/52
3. Modelling issues

3.1
Part II

Program syntax reference
4. Using TVT

4.1 The Basics

Typically the usage of the TVT tool begins with describing some processes with the input language and using the compiler to unfold them into LSTSs. After that, using the tool is based on running different programs in succession in order to reduce the LSTS state space. The basic cycle is that first a parallel composition of two or more LSTSs is composed in one or more steps. Second before the parallel compositions, in between them and after the composition the size of the result LSTSs is reduced by reduction programs. By doing this the state space is kept as small as possible but the original behaviour of the system is still maintained in the result. Finally if the result is small enough for a human viewer to understand a picture of it can be drawn by using the illustrator tool.

4.2 Command line of the programs

All the programs in the tool are named uniformly. All the names follow the form tvt.program.

The command line of all the programs has the following structure:

```
<program> [ <options> ] [ <input file> [ <output file> ] ]
```

where

- `<program>` is the command that starts the program.
- `<input file>` is the name of the input file.
- `<output file>` is the name of the output file.

Programs always have at least two file parameters. The first tells the name of the input file and the second the name of the file where the result is written. If no file names are given the input is by default read from the standard input stream and the result is written to the standard output stream. If the user wants the program to read the input from the standard input stream but still write the result into a file the name of the input file must be replaced by `-`.

 `<options>` refer to the option flags of a TVT program. Common options for all the TVT programs are:

- `--help` ( `-h` for short) prints a short instructions on how to use the program. Other parameters are ignored.
- `--force` ( `-f` ) allows overwriting an output file. The output file isn’t overwritten by default if the file in question already exists.
- `--extract-section <section> <file>` extracts the section from the result of a TVT program and writes it to the given file.
--copy-section <section> copies the extracted section to the output file.

--log <file> (-l) adds messages and error notifications at the end of the file given as parameter. By default the errors are printed to stderr.

--comment <comment string> (-c) adds the given comment to the history section of the output file.

--version prints the version of the tool the program belongs to. The version number for the development versions is 0.X and distributed versions 2.X, 3.X and so on.

In addition to these basic flags the programs may have flags of their own. The usage of these flags is described in the following chapters.
5. Input language compiler (tvt.compiler)

The TVT input language is used to describe a LSTS with variables in a form that is comfortable to read and write. The tvt.compiler program converts this language to the LSTS file format, in other words, unfolds the process.

The TVT script language described in this document should work in input language files when using tvt.compiler. Everything after a # character in a line is treated as a comment. You can also use include_file "filename" in input language files. This is usually used for including common type definitions at the beginning of the file.

5.1 Syntax

Usage of the tvt.compiler tool itself is pretty straightforward:

tvt.compiler input file output file

Writing the actual input language files is a bit trickier. The BNF specification of the language can be found below. It’s useful as a quick reference to the language. The different parts are explained in text later.

```plaintext
<PROCESS> ::= "TYPEDEF" <type_def>*
  "PROCESS" proname
  ["GATES" gatename*]
  ["STATE_PROPS"
    (<state_prop_type><state_prop>)*]
  "IS"
  ["VARIABLES" <variable_def>]*
  ["EVENT_PARAMS" <param_def>]*
  <init_part>
  <state>*
  "ENDPROC"

<state_prop_type> ::= "%" | "/"

<state_prop> ::= protype

?type_def> ::= typename "="
  [" value= "] |
  [" number= "] |
  [" number ".." number "]

<variable_def> ::= varname * ":
A file can currently contain exactly one process. The process consists of its interface, ie. gates and state propositions, and the internal representation, which may contain some variables, event parameter definitions, initialization and any number of states.

Before defining the actual process, any number of named types can be introduced with a \texttt{TYPEDEF}. The only kind of type available here is enumeration. Types are defined by giving the name of the type followed by a colon and finally, values of the type are listed inside a pair of square brackets, \texttt{[ ]}. The values can be any identifiers or numbers. In the case of numbers, a shortcut can be used, e.g., \texttt{[3 .. 8]} is the same as \texttt{[3 4 5 6 7 8]}.

Gates are the basic names of the actions in the system. They can get parameters. All gates that are going to be used in the transitions of the process have to be introduced in the \texttt{GATES} part of the process definition.

State propositions must also be introduced before they can be used. This can be done in the \texttt{STATE_PROPS} part by simply listing them. It is possible to specify that a state proposition is of one of the two special types: cutting or permanent. Putting a \% before the name of a proposition will make it cutting, and similarly, a / implies stickiness.

Variables and their types must be introduced before they can be used. The type of the variables can be either one of the types that were introduced earlier by \texttt{TYPEDEF} or a new, unnamed type, defined with the same notation as in the \texttt{TYPEDEF} part. Currently there is also one special type, namely \texttt{BOOLEAN}. It’s values are \texttt{FALSE} and \texttt{TRUE} and variables of that type can be used whenever a logical value is expected. Notice that multiple variables of the same type can be introduced like this: \texttt{variables var1}
var2 : [value1 value2 value3].

When the variables are used later in the conditions and assignments in the transitions, the name of the variable refers to the original value (before the transition). The value after the transition can be referred to by adding an apostrophe after the variable name (e.g. $x$ becomes $x'$).

In the EVENT_PARAMS section, all combinations of parameter types that the events can have must be defined. After a gate name, the parameter types must be listed in the correct order, each type enclosed with angle brackets, $< >$. If nothing is said about a gate, the compiler assumes that it takes no parameters. A gate can be allowed to have several different parameter combinations. The alphabet of the resulting LSTS will contain all possible combinations of variable values of the gates, regardless of whether they are used in the process or not.

In the initialization any number of assignments (or actually arbitrary expressions that evaluate to a boolean value) can be given. The initial state must fulfill all of them. Note that the versions with a $'$ of the variables have to be used, since before the initial "transition" none of the variables exist. The initial state of the system is supplied after an arrow, "$\rightarrow$".

SWITCH_ON means the propositions listed should be turned on initially, before even coming to the first state. This is needed for example when initial state has a SWITCH_OFF.

States consist of a name, possible state proposition changers and any amount of transitions. SWITCH_ON switches the proposition on from this state on, following all transitions until a corresponding SWITCH_OFF is encountered. With this feature, whole areas of states where a certain proposition is on or off can be easily created. SWITCH must always change the value of a proposition, because otherwise it would be easy to forget a SWITCH_OFF and leak the proposition to wrong areas.

HERE_ON and OFF switch the proposition on or off only in the state where they are used. A condition can be given after each property name. The condition must be true in order to the change to take place. Use variable names without $'$ to refer to the state’s variables. Using variables with a $'$ causes an error.

The transitions have two types of conditions: a single precondition and any number of postconditions (assignments). Between them, the action which will trigger this transition, possibly tau, has to be supplied. After the action name, attributes (? and !) can be used to handle input and output. They must comply with the legal parameter type combinations that were defined in EVENT_PARAMS.

The big difference between precondition and postconditions is that it’s not legal to refer to target state variables in the precondition. This enables an optimization, which also means that preconditions compile faster. They should be used whenever it’s possible. Also, pure assignments of the form variable$' = <expression>$, where $<expression>$ has no references to target state variables, are optimized to work effectively.

NO_SWITCH means that all SWITCH_ON/OFFs in the target state should be ignored when coming to the state with this transition. Delete means that the variables listed after it do not exist in the target state, thus slowing down the state explosion. Care should be taken to delete variables whenever they are not needed anymore.

As a final part of the transition, the target state is specified after an arrow.

An assignment is actually a condition. It is evaluated in the transition where it is used and the transition will not be created if the condition evaluates to false. This has the same effect as a conventional assignment. Compare $x := 3$ with $x' = 3$. Notice that a semicolon has to be placed after each condition!

Conditions are just expressions that should evaluate to a boolean value.

The expressions can be built using three levels of operators. The first level consists of three operators: AND, OR and NOT. Their parameters should naturally evaluate to a boolean value. These alternative ways of expressing these operators can also be used:
The second level has the self-explanatory operators $=\, \neq\, >\, <\, \geq\, \leq$, which all have the same precedence, right above \texttt{NOT}. The parameters of most of these operators must be of an enumeration type, but $=\, \neq$ accept also variables of the type \texttt{BOOLEAN}.

The last level has four unary operators, which are \texttt{INC}, \texttt{DEC}, \texttt{INC\_NO\_WRAP} and \texttt{DEC\_NO\_WRAP}. They return the next or the previous value of the enumeration type that their sole parameter is of. They wrap around to the first/last value if the type has no more values. The no\_wrap versions display an error message instead of wrapping.

The values of different enumeration types can be used as constants. Also, \texttt{TRUE} and \texttt{FALSE} can be used as constants that evaluate to a boolean value.

The evaluation order can be controlled with parentheses, ( ).

### 5.2 Examples

The example below describes a bit. The boolean type is not used, in order to make the example a bit more useful.

```
TYPEDEF BitValueType : [is0 is1]

PROCESS Bit
GATES set reset get

# the propositions are permanent
# (to set an example)
STATE_PROPS /on /off
IS
VARIABLES data : BitValueType

# Get always has a boolean parameter.
# The other gates are not mentioned here,
# so they don’t have any parameters.
EVENT_PARAMS get<BitValueType>

# We’ll start with the bit unset.
INIT [data’ = is0;]
   -> thestate

# The only state of this process
* thestate

   # set the correct state proposition on
   HERE_ON
   off [data = is0]
   on [data = is1]

# The bit can only be set if was unset.
[data = is0] -> set [data’ = is1;] -> thestate

# The bit can only be reset if was set.
[data = is1] -> reset [data’ = is0;] -> thestate
```
# Give the value of the bit out as the first parameter of the action.
get!data -> thestate

ENDPROC

Assuming that the file above is called bit.tvti, it can be converted into a LSTS file with the command
tvt.compiler bit.tvti bit.lsts

Here’s another example. It demonstrates how to use the boolean type.

PROCESS BooleanExample
GATES a
  # STATE_PROPS can be left out if no propositions are used
  IS
  # one variable, type boolean.
  VARIABLES var : BOOLEAN
  # a always takes a parameter of the type boolean
  EVENT_PARAMS a<BOOLEAN>

  # value of var is already boolean, so there’s no need to write var’ = FALSE;
  INIT [-var’;] -> statel
  
  # the precondition is true if var is true
  *statel: [var'] -> a!var -> varWasTrue
     [NOT var'] -> a?var’ -> varWasFalse
  
  # the postcondition toggles var’s value. () are needed, because NOT is less binding than =
  *varWasTrue: a!FALSE [var’ = (NOT var)]; -> statel

  # the postcondition sets var to true in target state
  *varWasFalse: a!TRUE [var’;] -> statel

ENDPROC
6. Parallel composition (tvt.parallel)

The parallel composition program \texttt{(tvt.parallel)} takes one or more LSTS files and a rules file as input and creates an output LSTS.

Unlike other TVT programs, the parallel composition program doesn’t take input LSTS files directly from the command line. Instead, input LSTS files are specified in the parallel composition rules file. [Erk02]

This rules file specifies the input LSTS files and which of their actions synchronize. That is, with the rules file it’s possible to tell which processes synchronize and with which actions. This method is more sophisticated than the classical parallel composition where only actions with same names synchronize and every component which has this action in its alphabet has to participate.

Of course it’s possible to achieve the classical parallel composition with the rules file, and a specialized program for creating such file exists \texttt{(tvt.createrules)}. This program is discussed in the chapter 7.

6.1 Syntax

The parallel composition program differs from other TVT programs in that it doesn’t take an LSTS file as input, but a rules file. The rules file is given in the command line in the usual way as the input file. The resulting file is given as second parameter (or if omitted, the result is outputted to \texttt{stdout}). The syntax is:

\begin{verbatim}
tvt.parallel [<options>] <parallel_rules_file> [<output file>]
\end{verbatim}

Besides the default options, the program supports the following options:

\begin{verbatim}
--trout=<filename>
\end{verbatim}

This tells the program to write the transitions section in a separate file (which name is given to the parameter). The advantage of using this option is that transitions are written to this file as they are created, and thus they don’t take any memory. This frees memory for state information, which allows processing bigger systems (this is mainly useful for on-the-fly verification as the resulting LSTS can be so big that no other program can read it anymore).

\begin{verbatim}
--use-ss=<type>
\end{verbatim}

Tells the program to use the stubborn sets algorithm to produce a reduced LSTS which is equivalent to the regular parallel composition. The type of equivalence is given as the \texttt{<type>} parameter. Valid values for this parameter are: \texttt{deadlocks}, \texttt{csp}, \texttt{cffd}

Note: State propositions do not yet work when using stubborn sets.
6.2 Examples

The simplest way of running the program is:

```
tvt.parallel system.rules result.lsts
```

The input LSTS file names and their synchronization rules are read from `system.rules` and the result is written to `result.lsts`.

If it suffices to make the classical parallel composition, the easiest way of doing this is to use the `tvt.createrules` program for creating the rules. For example:

```
tvt.createrules system.rules process1.lsts process2.lsts
tvt.parallel system.rules result.lsts
```

As all TVT programs can read their input from `stdin` and write to `stdout`, the previous example can be written so that the unnecessary temporary file is avoided:

```
tvt.createrules - process1.lsts process2.lsts |
    tvt.parallel - result.lsts
```

The next chapter contains more examples of this.

6.3 The parallel composition rules file

An example of a very simple (and imaginary) rules file:

```
Parallel_Composition_Rules_File

Begin Header
    Action_cnt = 2
End Header

Begin Action_Names
    1 = "Action 1"
    2 = "Action 2"
End Action_Names

Begin Parcomponents
    1 = "process1.lsts"
    2 = "process2.lsts"
End Parcomponents

Begin Rules
    Amount = 2
    (1,1)(2,1) -> 1
    (1,2)(2,2) -> 2
End Rules

Begin State_prop_rules
    1."Reserved" | 2."Reserved" -> "Critical section";
    1."Reserved" & 2."Reserved" -> rej;
End State_prop_rules

End_Parallel_Composition_Rules
```
6.3.1 Action synchronization rules

The Rules section tells the program how to synchronize the input LSTS files. In the example above this section was:

```
Begin Rules
  Amount = 2
  (1,1) (2,1) -> 1
  (1,2) (2,2) -> 2
End Rules
```

The first line indicates how many rules are there in the section. The rules come after this. Their syntax is:

```
(<input file number>, <action number>) + -> <resulting action number>
```

That is, every rule says which input files synchronize and with which actions, as well as the result of the synchronization. If an input file does not appear in the rule, then it doesn’t participate in the synchronization.

As the program handles only numbers and doesn’t care about the action names, it means that any combination of actions can synchronize regardless of their name. It’s also possible to make hiding and renaming to the result of the synchronization by specifying the proper resulting action number (0 for hiding).

These rules are not intended to be written by hand because using actions only by their numbers is difficult. There’s a much simpler version of this syntax available. It’s described in the chapter 8.

6.3.2 State proposition parallel composition rules

Types of state propositions

There are three types of state propositions: Regular, permanent and cutting propositions. These three types are differentiated by the first character of the name of the proposition.

A regular state proposition is any proposition which name does not begin with the characters ‘/’ or ‘%’. They do not have any special properties and always need state proposition rules in order to affect the result of the parallel composition (ie. if they are not specified in any rule, then they simply have no effect whatsoever on the resulting LSTS).

The name of a permanent proposition starts with the character ‘/’. Permanent propositions have the special property that the parallel composition program automatically generates internally default rules for them. Thus permanent propositions are always transferred to the result. (The user can specify a prefix for the name of the resulting permanent proposition depending on which input file it was in. This is described later in this chapter.) It is possible, however, to specify a list of permanent propositions for which no automatic rules are generated (syntax later in this chapter).

The name of a cutting proposition starts with the character ‘%’. A cutting proposition is a permanent proposition with the additional property of acting as cutting. That is, if the result of any state proposition rule is a cutting proposition, and the rule evaluates to true for a state of the result LSTS, then the parallel composition is not continued from that state forward (ie. the state becomes a deadlock state).

State propositions in an LSTS file

State propositions of an LSTS can be defined making a State Props section to the LSTS file. For example:
Begin State_props
  "Active": 1 5 6;
  "Reserved": 3 4;
  "/Permanent proposition": 10;
  "%Cut here": 15;
End State_props

The state proposition names are just listed and the state numbers where they are on are listed after them. Two state propositions must not have the same name.

Syntax of state proposition rules

If the input LSTS files have regular state propositions, some rules have to be given about how they are generated to the result. The state proposition rules section in the example above was:

Begin State_prop_rules
  1."Reserved" | 2."Reserved" -> "Critical section";
  1."Reserved" & 2."Reserved" -> rej;
End State_prop_rules

The state proposition rules are very versatile. Their syntax is the following:

<rule> ::= <expression> -> <state proposition> | <expression> -> <OTFVI keyword>
<expression> ::= <input proposition> | ( <expression> ) | <expression> <operator> <expression> | ! <expression>
<operator> ::= & | |
<input proposition> ::= <input number>.<state proposition> | true | false
<state proposition> ::= "<proposition name>"
<OTFVI keyword> ::= rej | dl_rej | ll_rej | inf_rej
<input number> ::= <natural number>

That is, the rules have two parts: A logical expression (before the \rightarrow) and the result (after the \rightarrow).

All white spaces in the rules (except inside the proposition names) are skipped and they have no effect whatsoever.

- A logical expression is any combination of input state propositions grouped with parentheses and the logical operators \& (logical and) and | (logical or). The negation of a proposition (or combination of them) can be specified with the operator !. The negation operator has the highest precedence, followed by \& and then |.

- An input proposition has two parts: The input LSTS number (which is given in the Parcomponents section) and the name of the state proposition (as a string), separated with a dot. It’s also possible to use the keywords true and false as input propositions (the value of these keywords is constant independently of anything else). Examples:
  1."State proposition"
  true

If a state proposition name is given (as in the first example), a state proposition with that name must exist in the given input LSTS or else an error message is issued.
• In the result part of the rule it’s possible to specify a state proposition name (a state proposition with this name is always created into the result) or an OTFVI keyword. It’s valid to use the same result in more than one rule or make several rules where the same logical expression generates different results; the program can handle these situations without problems.

• State proposition rules can also be used for on-the-fly verification. This is done by putting an OTFVI keyword as the result of a rule. If this rule ever evaluates to true, then the parallel composition is immediately stopped and the state where the rule became true will be marked with the given rejection. The LSTS calculated so far will be written to the result, but it will be marked as invalid.

  Note: inf_rej is not yet supported. The program will issue an error if it is used in this way.

Examples of valid state proposition rules (valid if the correspondent state proposition names are found in the input files):

1."Sleeping" -> "Process 1 sleeping";
1."Reserved" | 2."Reserved" -> "Critical section";
!(1."Active" & (2."Sleeping" | !3."Active")) -> dl_rej;

The keywords true and false can be used for special purposes. For example:

true -> "Prop1";
false -> "Prop2";

The first rule means that the proposition “Prop1” is on in every state of the result (as it always evaluates to true). The second rule means that the proposition “Prop2” does not hold in any resulting state (however, it’s still written to the resulting LSTS because all resulting state proposition names are always written to the result even if they aren’t on in any state).

Using on-the-fly verification

On-the-fly verification can be used by using the on-the-fly keywords as the result of state proposition rules. This means that for all the result states for which the rule evaluates to true the correspondent on-the-fly verification property will be checked.

If the result fulfills the property to be checked, the parallel composition will be interrupted, the current result which has been calculated so far will be written on the destination and marked as interrupted. This is an invalid LSTS and other programs will refuse to read it except for the errortracer, which can be used to examine the interrupted LSTS. See the chapter 9 for more details.

The keywords have the following meaning:

• rej: If the rule evaluates to true, the parallel composition will be interrupted.

• dl_rej: If the rule evaluates to true and the result state is a deadlock state, the parallel composition will be interrupted.

• ll_rej: If the rule evaluates to true and the guard process does not synchronize with the rest of the system while the latter makes a loop, the parallel composition will be interrupted.

The guard process mentioned above is the process which is used in the state proposition rule in question.

There are certain limitations for the use of ll_rej:

• The rule which result is ll_rej must use at least one input LSTS number.
• Only one input LSTS number can be used in the rule (this will be the guard process). You can use it several times, though.
• Only one state proposition rule can have ll_rej as its result.

Examples of the ll_rej rule:

```
1."Prop" -> ll_rej;   # Ok
!1."Prop1" | !1."Prop2" -> ll_rej;   # Ok
1."Prop1" & 2."Prop2" -> ll_rej;   # Error
true -> ll_rej;       # Error
```

How permanent and cutting propositions work

If the result of any state proposition rule is a cutting proposition, then it will affect the parallel composition if the rule evaluates to true in any state. For example:

```
1."Terminate" & 2."Sleep" -> "%Ignore";
```

This rule means that if in any state the “Terminate” proposition of the first component and the “Sleep” proposition of the second component are true at the same time, the resulting state will be marked with the “%Ignore” proposition and the parallel composition is not continued from that state forward (ie. the state becomes a deadlock state).

When permanent or cutting propositions appear in an LSTS file, they have the special property that state proposition rules are automatically generated for them inside the parallel composition program. Thus they don’t need user-defined rules (even though they can be used in user-defined rules in the exact same way as any other proposition).

Normally the rule is generated so that the proposition is simply preserved. For example, suppose that the first component has a permanent proposition named as "/Permanent" and a cutting proposition named as "%Cut". Also let’s suppose that the third component contains these same propositions as well. By default the parallel composition program will internally generate these rules for them:

```
1."/Permanent" -> "/Permanent";
1."%Cut" -> "%Cut";
3."/Permanent" -> "/Permanent";
3."%Cut" -> "%Cut";
```

Defining a prefix for permanent and cutting propositions

Since the same result appears more than once in the previous example, they will be merged into one (exactly has if the rules which generate them would be joined with the logical or-operator). Sometimes, however, it’s necessary to distinguish which component generated which permanent or cutting proposition.

It’s possible to affect the automatic generation of rules by specifying a prefix string in a per-component basis. This is achieved with a special syntax in the Parcomponents block. For example:

```
Begin Parcomponents
  1 = <"P1."> "process1.lsts"
  2 = "process2.lsts"
  3 = <"P3."> "process3.lsts"
End Parcomponents
```
An optional prefix can be attached to a component by specifying it as a string inside <> characters, as shown above. This prefix will be (blindly) copied to the name of the result of the rule of any permanent and cutting proposition contained in that process. The prefix is copied between the `/` or `%` character and the rest of the name.

So supposing that we had defined the Parcomponents section like this, the previous example would thus become:

1."/Permanent" -> "/P1.Permanent";
1."%Cut" -> "%P1.Cut";
3."/Permanent" -> "/P3.Permanent";
3."%Cut" -> "%P3.Cut";

The prefix does not affect any other state proposition.

If no prefix is specified for the component, then the proposition name is used as is.

**Hiding permanent propositions**

It is possible to stop the program from automatically making default rules for certain permanent propositions. This is achieved by including a special line in the State_prop_rules section. This line begins with the keyword hide followed by a comma-separated list of permanent propositions. The line is terminated with a semi-colon. For example:

hide "/prop1", "/prop5", "/prop7";

This will cause the program to not to create default rules for those propositions. (Default rules are still created for all the other permanent propositions which appear in input files, though.)

Note: Only permanent propositions (ie. propositions starting with a `/` character) can be listed in the hide command. Regular and cutting propositions are not allowed (the former because it has no rational meaning and the latter because it would break semantics).
7. Classic parallel composition rules file creation (tvt.createrules)

Writing a parallel composition rules file can be tedious, and in the case of the classic parallel composition it’s needless. The rules file is not intended to be written by hand, but created with a program.

In the classic parallel composition only actions with the same name synchronize and all components which have that action in their alphabet have to participate.

This is a very common way of making the parallel composition and the rules for this can be generated automatically. Thus a program has been made for this: tvt.createrules

However, this program goes even further as it also supports hiding and renaming of the rules file, so this doesn’t have to be made as a separate step.

7.1 Syntax

It’s important to remember that tvt.createrules is different from other TVT programs with respect to the command line syntax. Usually TVT programs take the input file name as the first parameter and the output file name as second parameter. However, as tvt.createrules takes several input files and outputs one file, the command line syntax has been made so that the output file is specified first and after that all the input files.

The syntax of the program is:

```bash
tvt.createrules [<options>] [<output rules file> <input files>]
```

The correct order of the parameters can be remembered with the rule “create the rule x.rules using the files a.lsts, b.lsts and c.lsts”.

Besides the default options, the program also supports the following:

```
--rename <file>
--use-ere
```

More help about these options can be seen by starting the program as “tvt.createrules --help rename” and “tvt.createrules --help use-ere”. Their functionality is explained in detail in this document as well.
7.1.1 Renaming/hiding

The parallel composition program supports implicitly multiple renaming and hiding in a very simple way and thus this support has been added to tvt.createrules. The option --rename gives the program a file which contains the names of the actions to be renamed or hidden. This file contains string groups, with each group ended with a semicolon. The syntax is:

\[(<\text{string}>+ ; )+\]

If a group has just one string, it means that the action with that name will be hidden. If the group has more than one string then it means that the action with that name will be renamed to the latter action or actions.

Example:

"Action 1";
"Action 2" "Better";
"Action 3" "Triple1" "Triple2" "Triple3";

This example shows all three usages of the file. The lines of the example mean the following:

1. The action named “Action 1” will be hidden in the result.
2. The action named “Action 2” will be renamed to “Better” in the result.
3. The action named “Action 3” will be multiple-renamed to “Triple1”, “Triple2” and “Triple3” in the result.

7.1.2 Using regular expressions in renaming/hiding

If the number of actions to be renamed or hidden is very large, it can be a very tedious job to have to write them all by hand (and frustrating if the names have a common part). For this reason support for Extended Regular Expressions (ERE) has been added to the program.

The regular expressions are activated by using the command-line option --use-ere. When this has been given, then the strings in the renaming/hiding file are interpreted as regular expressions. (If the --rename option is not specified, then --use-ere has no effect at all.)

The first string in a group is an ERE and the next strings in the group are substitution patterns. The idea is pretty much the same as the s/// command in programs like sed or perl.

NOTE: In regular expressions some characters have to be escaped (“escaping” a character means that a character, which has a meaning in a regular expression, needs to be interpreted as a regular character instead, it has to be preceded with a backslash, ie. “\”). Also in the substitution strings the backslash is used in a special meaning. Due to the way the files are parsed in the TVT programs this backslash has to be escaped as well. In TVT programs a backslash is escaped as “\\”.

There is a lot of material about regular expressions in the net and in many Unix man pages so it’s unnecessary to repeat all that here. The basic ERE syntax is also specified in the help of the program.
Renaming/hiding examples:

"Action.*";
"SomeAction\([0-9]+\)" "Test\!1";

This hides all actions which name begins with “Action”. It also renames all actions which begin with “SomeAction” which is followed by a number so that the renamed action will be named “Test” followed by the same number (for example the action “SomeAction321” will be renamed to “Test321”).

Renaming and hiding can be as meticulous as the regular expressions allow. For example the following:

"Process \([0-9]+\): Action \([A-Za-z1-9]\))" "\!2: \!1";

will perform a renaming such that an action which is named, for example “Process 5: Action abc” will be renamed to “abc: 5: Action abc”.

The following hides all actions:

".*";

The following adds “Proc1: ” to the names of all actions:

".*" "Proc1: \!0";

7.2 Examples

Note: In the following examples some commands have been split to several lines due to their length; however they are always just one command which is supposed to be in one line. A line which should belong to the previous line is indented a bit to show this.

One of the simplest ways of using the program is just creating the parallel composition rules for a set of LSTS files:

```
tvt.createrules system.rules proc1.lsts proc2.lsts proc3.lsts
```

It’s possible to pipe the command directly to the parallel composition program and thus the temporary rules file is avoided:

```
tvt.createrules - proc1.lsts proc2.lsts proc3.lsts |
  tvt.parallel - result.lsts
```

A separate file can be used for the renaming and hiding information:

```
tvt.createrules --rename ren.txt - proc1.lsts proc2.lsts proc3.lsts |
  tvt.parallel - result.lsts
```

This file isn’t absolutely necessary either, although avoiding it makes the command rather obfuscated:

```
echo "\"Send\"; \"Ack\";" |
  tvt.createrules --rename - - proc1.lsts proc2.lsts proc3.lsts |
  tvt.parallel - result.lsts
```

This command works like this:
• The `echo` command prints: "Send"; "Ack"; (ie. we want to hide the actions with those names).

• This is piped to the `tvt.createrules` program.

• The option `--rename` takes "-" as parameter, that is, it reads its input from the `stdin` (which is the output of the `echo` command).

• The next "-" in the command line means that the rules file is outputted to `stdout`.

• After this the input files are specified.

• The result is piped to the `tvt.parallel` program which takes the input from `stdin` and writes the result to `result.lsts`.

This example works practically like the Lotos command:

```
hide Send, Ack in (proc1 || proc2 || proc3)
```

It is also possible to create an extended rules file (see next chapter) with the aid of the `tvt.extendedrules` program, like this:

```
tvt.createrules - proc*.lsts | tvt.extendedrules -x - system.erules
```
8. Extended rules file converter
(tvt.extendedrules)

Even though \texttt{tvt.createrules} creates the classic parallel composition rules file in an easy way, it is quite limited in functionality and it doesn’t allow using the full potential of the rules file. There are many things that would be very easy to do with the rules file but are not possible to achieve with the \texttt{tvt.createrules} program.

At the moment the only option is to write the rules file by hand. However, as the rules file is quite difficult to write by hand (and it isn’t even designed for that), a more advanced and easier to use file format has been developed and the proper conversion program (\texttt{tvt.extendedrules}) which converts from this extended format to the regular format. In theory the parallel composition program could read this file directly, but it was decided that it shouldn’t get too bloated so just a small converter program was made instead.

This format is still a “low-level” parallel composition rules file, but its syntax is much easier to understand and write than the regular rules file.

8.1 The extended rules file

This rules file format is both extended in functionality as well as simplified in syntax in order to be more user-friendly. Everything unnecessary has been left out and only the essential things have been implemented.

The following example demonstrates the syntax of the file:

\begin{verbatim}
1 = "philosopher.lsts"
2 = "fork.lsts"
3 = "philosopher.lsts"
4 = "fork.lsts"

(1, "Take left") (4, "Take") \rightarrow "Fill take left"
(1, "Take right") (2, "Take") \rightarrow "Fill take right"
(1, "Release left") (4, "Release") \rightarrow "Fill rel left"
(1, "Release right") (2, "Release") \rightarrow "Fill rel right"

(3, "Take left") (2, "Take") \rightarrow 0
(3, "Take right") (4, "Take") \rightarrow 0
(3, "Release left") (2, "Release") \rightarrow 0
(3, "Release right") (4, "Release") \rightarrow 0
\end{verbatim}

Firstly the input files are specified and numbered. These numbers are used for the same purpose as in the regular rules file, that is, they are used in the rules to specify the input file.
Prefixes for permanent and cutting propositions are supported with the same syntax as described in subsection 6.3.2. That is, you can specify an input filename like this:

$$1 = <"Phil1:" \text{"philosopher.lsts"}$$

After this come the parallel composition rules. The syntax is similar to the one in the regular rules file, but instead of using action numbers it’s possible to use action names directly. This way it’s not necessary to know the number of the action in order to use it.

It is still possible to use action numbers, that is, the following is valid:

$$(1,1) \rightarrow 1$$

This feature is mostly useful when hiding the result of a rule. This can be done by specifying 0 (which means $\tau$) as the result of the synchronization in the same way as in the regular rules file.

Note: Using action numbers instead of their names, although supported, is not recommended because the program does not check whether the input has an action with that number. The only purpose of this feature is to allow raw copies of regular rules to be used here (for example for adding new rules in the easier way at the end).

The parallel composition rules file for dining philosophers example above can be written in a generic way for any number of philosophers by using the scripting language feature of all input files, which is described in the chapter 22, in the following way:

```latex
#{ SET n=2 } # Number of philosophers
#{ SET f=0 }
#{ LABEL loop1 }
#{ EVAL f*2+1 } = "philosopher.lsts"
#{ EVAL f*2+2 } = "fork.lsts"
#{ SET f=f+1 }
#{ CB loop1: n-f }

(1, "Take left") (#{ EVAL n*2 }, "Take") \rightarrow "Fil1 take left"
(1, "Take right") (2, "Take") \rightarrow "Fil1 take right"
(1, "Release left") (#{ EVAL n*2 }, "Release") \rightarrow "Fil1 rel left"
(1, "Release right") (2, "Release") \rightarrow "Fil1 rel right"

#{ SET i=1 }
#{ LABEL loop2 }

(#{ EVAL i*2+1 }, "Take left")
(#{ EVAL i*2 }, "Take") \rightarrow "Fil#{ EVAL i+1 } take left"

(#{ EVAL i*2+1 }, "Take right")
(#{ EVAL i*2+2 }, "Take") \rightarrow "Fil#{ EVAL i+1 } take right"

(#{ EVAL i*2+1 }, "Release left")
(#{ EVAL i*2 }, "Release") \rightarrow "Fil#{ EVAL i+1 } rel left"

(#{ EVAL i*2+1 }, "Release right")
(#{ EVAL i*2+2 }, "Release") \rightarrow "Fil#{ EVAL i+1 } rel right"

#{ SET i=i+1 }
#{ CB loop2: n-i }
```

instructions.tex(1.2) 15.4.2012 29/52
8.2 Aliases

Since writing long action names over and over again can be laborious and error-prone, the file supports an aliasing mechanism. That is, an identifier can be declared as being an alias to a string.

These aliases are declared after the input file names and before the parallel composition rules. An alias is a name written without quotation marks.

In the parallel composition rules any combination of aliases and strings can be used anywhere an action name is expected. When several aliases and/or strings are used, they are internally concatenated. That is, suppose we have defined the following aliases and rule:

\[
\text{Tr1} = "\text{Transition 1}\"
\]
\[
\text{Tr2} = "\text{Transition 2}\"
\]
\[
(1, \text{Tr1}) (2, \text{Tr2} "<\text{param2}>") \rightarrow "\text{Process 1 " Tr1 " done}\"
\]

The rule will be interpreted exactly as if it had been written as:

\[
(1, "\text{Transition 1}\") (2, "\text{Transition 2<param2>}\") \rightarrow
"\text{Process 1 Transition 1 done}\"
\]

The extended rules file for the dining philosophers at the beginning of section 8.1 can thus be written, for example, as the following:

\[
1 = "\text{philosopher.lsts}\"
\]
\[
2 = "\text{fork.lsts}\"
\]
\[
3 = "\text{philosopher.lsts}\"
\]
\[
4 = "\text{fork.lsts}\"
\]
\[
\text{T} = "\text{Take}\"
\]
\[
\text{R} = "\text{Release}\"
\]
\[
\text{TL} = "\text{Take left}\"
\]
\[
\text{TR} = "\text{Take right}\"
\]
\[
\text{RL} = "\text{Release left}\"
\]
\[
\text{RR} = "\text{Release right}\"
\]
\[
(1, \text{TL}) (4, \text{T}) \rightarrow "\text{Fil1 " TL}
\]
\[
(1, \text{TR}) (2, \text{T}) \rightarrow "\text{Fil1 " TR}
\]
\[
(1, \text{RL}) (4, \text{R}) \rightarrow "\text{Fil1 " RL}
\]
\[
(1, \text{RR}) (2, \text{R}) \rightarrow "\text{Fil1 " RR}
\]
\[
(3, \text{TL}) (2, \text{T}) \rightarrow 0
\]
\[
(3, \text{TR}) (4, \text{T}) \rightarrow 0
\]
\[
(3, \text{RL}) (2, \text{R}) \rightarrow 0
\]
\[
(3, \text{RR}) (4, \text{R}) \rightarrow 0
\]

Note: Alias names are case insensitive. This means that the alias name \text{Ack} will be interpreted as the same as the alias name \text{ack} or \text{ACK}.

8.3 Action name parameters and wildcards

In TVT action names can have parameters. The syntax of these action names is like this:
That is, an action name can have one or more parameters after the actual base name of the action, each parameter being surrounded by <>. These parameters are usually generated by the input language compiler.

Since the number of action names with parameters and with the same base name can be large, writing rules for all of them can be prohibitively tedious. For this reason tvt.extendedrules supports a wildcard syntax which automatizes the generation of rules for a set of actions with the same base name.

The name of a parameter can be replaced with a $ symbol followed by a number. This means that any parameter is allowed in that position. The number in question gives an ID for that parameter. If the same number is used again in the same rule, it means that both of those parameters must be the same.

For example:

\[(1, "a<$1>") \ (2, "b<x5><$1>") \ (3, "c<$2><$3>") \rightarrow \ "res<$1><$2><$3>"\]

This will create rules for all action combinations where the parameter of the action 'a' of the first LSTS is the same as the second parameter of the action 'b' of the second LSTS (the first parameter being "x5"), and all these combinations are combined with every action 'c' with two parameters of the third LSTS.

As seen above, the wildcard symbols can be used in the result of the rule. They are replaced with the parameter they matched in the rule itself.

Note: This feature should be used with care. Since the amount of possible combinations can grow exponentially, the resulting file can be enormous (and the running time will be quite long).

### 8.4 State proposition parallel composition rules

The extended rules file supports state proposition rules as well. They can be used by just including a `State_prop_rules` section at the end of the file. This section is exactly the same as in the regular rules file (section 6.3.2). The program just copies this section to the result (ie. no checks are done).

(In fact, the section can be located anywhere after the input file names; it doesn’t really matter where it is; when the program sees the section it reads it to memory and then continues normally.)

### 8.5 Syntax

The syntax of the program does not differ from the usual syntax of the TVT programs:

```
tvt.extendedrules [options] [input file] [output file]
```

Besides the default options, the program supports these command line options:

- `--check` tells the program to issue an error when an action name is not found in the corresponding input file. By default the program ignores the rules containing inexistent action names.

- `-x` makes the program to perform the reverse conversion. That is, it takes a regular rules file as input and outputs an extended rules file (with all the action numbers replaced with their respective names).
8.6 Example

The program can be used smoothly by piping it with the parallel composition program like this:

```
tvt.extendedrules philosophers.erules | tvt.parallel - result.lsts
```
9. Errortracing an interrupted LSTS (tvt.errortracer)

An interrupted LSTS is an LSTS generated by the parallel composition tool when an on-the-fly verification check interrupted the parallel composition. This is not a valid LSTS (because it has no CFFD relationship with the input) and thus all other TVT program intentionally refuse to read it. However, the \texttt{tvt.errortracer} tool can be used to examine this interrupted LSTS.

What the program does is to create an LSTS which shows the (shortest) path from the initial state to the interrupted state from the point of view of one of the input LSTS components. That is, it creates an LSTS which performs the actions performed by one of the input LSTS components until the interruption happened. This can be used to see what the input LSTS did before the interruption.

9.1 Syntax

The syntax of the program is slightly different from the other TVT tools (note: the syntax line has been split here because of page width limitations):

\texttt{tvt.errortracer [\langle options\rangle] \langle component number\rangle} \\
\texttt{\langle interrupted LSTS\rangle [\langle output file\rangle]}

The first parameter to the program (besides the standard command-line options) is an integer number between 1 and the amount of input components given to the parallel composition which created this interrupted LSTS. The error trace will be done to the component with this number (in the parallel composition rules file).

The second parameter is the name if the interrupted LSTS itself. Only an interrupted LSTS created by \texttt{tvt.parallel} is valid here, and the program will refuse to read anything else.

Note that \texttt{tvt.errortracer} will read the parallel composition rules file and the input LSTS files in the same way as \texttt{tvt.parallel} did, and thus they must be located in the same place as when given to \texttt{tvt.parallel}, and \texttt{tvt.errortracer} must be run from the same directory as \texttt{tvt.parallel} was run.

9.2 Example

\texttt{tvt.errortracer 2 interrupted.lsts trace2.lsts}

This will create an error trace from the second input component.
10. Creating an acceptance graph (tvt.lsts2ag)

The program `tvt.lsts2ag` converts an input LSTS into an acceptance graph (AG). An acceptance graph is a special type of an LSTS where the stable failures of an ordinary LSTS are represented by minimal acceptance sets added to each state. The divergence traces in an LSTS are represented by divergence bits in states in an AG. Furthermore there is a boolean variable in the initial state of the AG that describes the stability of the initial state of the LSTS. At any rate since all these characteristics of an LSTS can be described by invisible transitions these transitions aren’t present in the result AG.

The program needs one input LSTS file and produces one output file out of the input which is an acceptance graph of the input LSTS.

10.1 Syntax

`tvt.lsts2ag` doesn’t have any additional parameters to the default parameters of the tool. Therefore the program’s command line is:

```
tvt.lsts2ag [<options>] [input file][<output file>]
```

10.2 Examples

Using the program is rather simple and straightforward. For example:

```
tvt.lsts2ag start.lsts startag.lsts
```
11. Determinizing (tvt.detlsts)

This is a fairly simple program which calculates the deterministic version of an acceptance graph LSTS. This program is intended to be used after the tvt.lsts2ag program (although some reductions could be done between these two as well).

The input for this program has to be a true acceptance graph, which means in practice that the input must not contain any $\tau$ transitions. The result of the program is also this kind of acceptance graph.

11.1 Syntax

Besides the default command line options, the program supports the following option:

\[-m<n>\]

where $<n>$ can be any positive real number.

If this option is given then the program will stop if the result would be larger than $<n>$ times the size of the input. For example if the option $-m2.5$ is given, then program will stop if the size of the result would grow larger than 2.5 times the size of the input. The size of an LSTS is here measured by the number of states.

The purpose of this parameter is to avoid the result from exploding (as the size of the LSTS can grow exponentially). If the result would grow too much, then some reduction could be applied to the input before trying again.

11.2 Examples

The usage of the program is very simple. For example:

```
tvt.detlsts ag.lsts detag.lsts
```

The execution of the program can be stopped if the result would grow too big, as explained above. For example:

```
tvt.detlsts -m100 ag.lsts detag.lsts
```

This stops the program if the result would be more than 100 times the size of the input.
12. Strong bisimulation reduction (tvt.SBreduction)

The program attempts to reduce the number of states in an LSTS by combining strongly bisimilar states into one. The program gets an LSTS as an input and produces a reduced LSTS as a result. There is a strong bisimulation between the resulting LSTS and the input.

The reduction algorithm used in the program is based on the strong bisimulation reduction algorithm described by Jean-Claude Fernandez in [Fer89]. However the implemented algorithm goes through the arcs in the LSTS backwards. 

tvt.SBreduction is able to reduce both deterministic and non-deterministic input LSTSs.

12.1 Syntax

The program uses only the standard parameters.

12.2 Examples

Using the program is simple.

    tvt.SBreduction large.lsts small.lsts
13. Strong bisimulation comparison (tvt.SBcompare)

This program compares two deterministic LSTSs for strong bisimilarity. It takes two LSTS files and if they are deterministic, it will simply tell if they are strongly bisimilar, or if not, the first dissimilarity it finds.

This program is used to test for CFFD equivalence. Two deterministic LSTSs are CFFD equivalent if and only if they are strongly bisimilar.

The program does not support comparing non-deterministic LSTSs, and this limitation is intentional. The main purpose of this program is to test CFFD equivalence, and if the program would tell that two non-deterministic LSTSs are not strongly bisimilar, that would not only be useless for this purpose (non-deterministic LSTSs can be CFFD-equivalent even when they aren’t strongly bisimilar), but it could also be misleading (one could erroneously think that two LSTSs are not CFFD-equivalent just because this program told that they are not strongly bisimilar).

In order to compare two non-deterministic LSTSs for CFFD equivalence, they have to be determinized first. This is done by running the files through the tvt.lsts2ag and tvt.detlsts programs.

13.1 Syntax

The program uses only the standard parameters (even though some of them are not used because the program does not generate any output LSTS). However, instead of taking an input file and an output file, like most of the other programs, it takes just two input files.

13.2 Examples

```
tvt.SBcompare example1.lsts example2.lsts
```

If the LSTSs are equivalent, it will output the following:

```
The LSTSs are strongly bisimilar.
```

If they are not equivalent, it will report the first difference it finds.
14. Strong bisimulation reduction (tvt.ReduceKsnp)

This program—like tvt.SBReduction—reduces the number of states and transitions in an LSTS by computing the smallest LSTS that is strongly bisimilar to its input. While tvt.SBReduction uses the $O(N \log N)$ Fernandez algorithm published in [Fer89], the tvt.ReduceKsnp program uses the $O(N^2)$ naive algorithm described by Kanellakis and Smolka in [KS83] and [KS90]. However, there are some cases where it may be faster and consume less memory than its rival.

There are some restrictions on the input that tvt.ReduceKsnp accepts: it does not handle state names, elusive actions, acceptance sets, divergence bits, or any other sections apart from the action names, transitions and state propositions.

The state propositions (if any) are used to form the initial partition of the resulting LSTS. When they are present, the state propositions are written to the output. Also, only those actions that actually occur as transition labels are written to the output; the rest are discarded. This may not seem appropriate, but the primary objective of tvt.ReduceKsnp is reduction of information. Note that this means that the output of tvt.ReduceKsnp and tvt.SBReduction is not always identical.

### 14.1 Syntax

Apart from the standard parameters, the program accepts the command line option

```
--info
```

which instructs it to write information about the performance of the algorithm to the log file or standard output.

### 14.2 Examples

Using the program is quite simple:

```
tvt.ReduceKsnp --info large.lsts small.lsts
```

produces the following output:

```
reading time: 410 10 420
processing time: 30 0 30
writing time: 0 10 10
total time: 440 20 460
static memory: 85 bytes
dynamic memory: 525190 bytes
```
size before: S 400 T 63050 A 5 
size after: S 1 T 6 A 5

This gives the user, system and total times (in milliseconds) for the reading of the
LSTS (line 1), the application of the algorithm (line 2), the writing of the resulting
LSTS (line 3) and the total elapsed time (line 4). It also gives the amount of static and
dynamic memory used by the algorithm. This excludes memory used by libraries and
by the program stack. The stack should not be very expensive: since the program is
not recursive, it should be a fixed cost. The last two lines indicates the size of the input
and output LSTSs in terms of states (S), transitions (T) and actions (A).

Without the –info option, tvt.ReduceKsnp produces no output at all.
15. Converting acceptance graphs to tester processes (tvt.ag2tp)

This program converts an acceptance graph to an equivalent tester process as described in [HV00].

Once a tester process has been generated, other LSTSs can be checked against it using the parallel composition tool. For example, to check that Impl $\prec_{\text{CFFD}}$ Spec, do the following:

- Generate the acceptance graph that corresponds to Spec:
  \[
  \text{tvt.lsts2ag spec.lsts ag.lsts}
  \]

- Determinize the acceptance graph:
  \[
  \text{tvt.detlsts ag.lsts det.lsts}
  \]

- Generate the corresponding tester process:
  \[
  \text{tvt.ag2tp det.lsts tp.lsts}
  \]

- Now generate the parallel composition rules, making sure that the tester process is the first component in the parallel composition:
  \[
  \text{tvt.createrules rules tp.lsts impl.lsts}
  \]

- Filter the rules through the rejection adding shell-script:
  \[
  \text{tvt.testall < rules > tp.rules}
  \]

- And finally run the parallel composition tool:
  \[
  \text{tvt.parallel tp.rules result.lsts}
  \]

If Impl $\prec_{\text{CFFD}}$ the parallel composition tool will not report anything. Otherwise, it will report that either deadlock, livelock, or normal rejection has been triggered, the specific kind depending on in which way the preorder violation has taken place.

Naturally, the above steps can be combined somewhat:

\[
\text{tvt.lsts2ag S | tvt.detlsts - | tvt.ag2tp - T}
\]
\[
\text{tvt.createrules - T I | tvt.testall | tvt.parallel - R}
\]

where $S = \text{spec.lsts}$, $T = \text{tp.lsts}$, $I = \text{impl.lsts}$, and $R = \text{result.lsts}$. The \text{tvt.testall} script relies on the fact that the tester process is the first component described by the parallel composition rules. If this is not true, the rules could be edited by hand; see the script for more details.

The program accepts no command line options other than the defaults.
16. Tester rules creation (tvt.testerrules)

Tester processes created using tvt.ag2tp are easily added to any parallel composition using this tool. This tool allows plugging a tester process in a system without having to calculate the parallel composition of the system. This means that the tests performed by the tester process can be done during the parallel composition step (which means that if there’s an error, it’s detected before the whole system is built). This potentially saves significant amounts of resources and time if errors are found.

The program takes a tester LSTS and a parallel composition rule file and creates a new rule file that adds the tester as one of the processes in the parallel composition. An action of the tester synchronizes with such actions that have the same name in the result, e.g. if the tester has an action "a", any rule of the form

\[(p_1, "x_1"), \ldots, (p_n, "x_n") \rightarrow "a"\]

will produce a rule

\[(p_1, "x_1"), \ldots, (p_n, "x_n"), (p_{n+1}, "a") \rightarrow "a"\]

to denote that the tester synchronizes.

(Note: The program works with regular rule files, not extended ones. The syntax used above was used only for clarity.)

16.1 Syntax

The syntax of the program differs slightly from the other TVT tools in that it takes two input files and an output file. The first input file is the tester LSTS and the second input file is the original rules file:

\[ \text{tvt.testerrules [\langle options\rangle]} <\text{tester}> <\text{rules}> [\langle output rules\rangle]\]
17. Converting back to LSTS (tvt.ag2lsts)

This is a fairly simple program which converts an acceptance graph back to a regular LSTS.

The input of the program has to be a true acceptance graph. Giving it a regular LSTS with no acceptance graph information usually causes an error message (like "Internal error: State 1 has empty accsets and divbit off. (Input is most probably not an AG)").

The result of the program is an LSTS which corresponds to the acceptance graph.

17.1 Syntax

The program has no extra parameters besides the default ones.

17.2 Examples

The usage of the program is extremely simple:

tvt.ag2lsts ag.lsts result.lsts
18. The file format tool (tvt.fftool)

The tool program `tvt.fftool` allows converting *ARA files* (files having either ARA LTS or ARA AG format) into *TVT LSTS files*. Also, the same tool makes it possible to convert LSTS files into ARA files, if need be. Another task that `tvt.fftool` can be used to, is to fix defective LSTS files. If a problem in an LSTS file is small enough, `tvt.fftool` is able to read the file and to produce a fixed output LSTS.

18.1 Syntax

`tvt.fftool` makes use of the default command line options and the following special options:

- `--tvt` – writes output in the TVT LSTS file format (default).
- `--ara` – writes output in the ARA LTS file format.
- `--ara-ag` – writes output in the ARA AG file format.
19. The LSTS simulator (tvt.simulate)

The TVT tool box also includes an LSTS simulator with a simple ASCII user interface. The simulator starts from an initial state of a given LSTS and provides the user a set of commands for moving from state to state and viewing information present in the LSTS.

19.1 Syntax

The commands available in the simulator are:

- `h, help` – prints help.
- `q, quit` – exits the simulator.
- `v, view` – views the current state.
- `1, 2,..., n` – chooses the transition the option number represents.
- `b, back` – moves one step back in the current path.
- `p, path` – shows the path from the initial state to the current state.
- `pi, pathinfo` – shows the path from the initial state to the current state with full information.
- `"<action name>"` – chooses a transition labelled with "<action name>".
- `tau` – chooses a transition labelled with tau.
20. Illustrator (tvt.illux)

The illustrator consists of three programs, tvt.illus, tvt.illux_view and tvt.illux, but only the third one is the one which is intended for the user.

- **tvt.illus** is the program which calculates a layout for the states and transitions of the LSTS. Its output is intended exclusively for the tvt.illux_view program, which is the graphical viewer. The user usually doesn’t need to use these programs directly.

- **tvt.illux** is the program which the user should use to view the LSTS. It is just a shell-script which calls the previous programs and pipes them together.

### 20.1 Syntax

The syntax of the program is:

```
tvt.illux [<options>] <input file> [style file]
```

The input can be any LSTS file (but preferably not too big). The program calculates a layout for the LSTS and opens an X-window to show it.

- If the --comment (or -c) option is given, the string following it will be printed in the text window of the viewer.

- An optional style file can be given as second parameter. This file specifies the colors and dash patterns of the transition arrows. By default a set of 16 different color and dash combinations are used.

- The style file is very handy for differentiating between actions. For example actions which are related to each other could be drawn with shades of the same color and different dash patterns. The dash patterns can be used to denote some property of the action (for example a dash pattern consisting of a long line and a short line could mean “message 1” and a pattern consisting of a long line and two short lines could mean “message 2”).

#### 20.1.1 Syntax of the style file

The style file consists of lines where each line contains two or three items. The lines have the following syntax:

```
"<Action name>" <color> ["<dash pattern>"
```

The meaning of the items is the following:

1. A string which contains the name of the action. τ is written as "tau".

2. A Unix-style color definition (without quotes). The color can be specified either by its name (eg. green) or by its rgb value in hexadecimal (eg. #00FF00).
3. The optional dash pattern string. Inside the string the character “x” denotes a solid part and a whitespace denotes a space. If this parameter is not given, then a solid line is used.

Example:

"tau" #80A0FF
"send msg" green
"send ack" green "xx "
"rec msg" red
"rec ack" red "xx x 

20.2 Examples

The usage of the program is very simple:

`tvt.illux process.lsts`

A style file can be provided to the program:

`tvt.illux process.lsts process.style`

20.3 The graphical user interface

The graphical user interface has the following features which might not be obvious:

- Left-clicking on a state will list the state propositions of that state in the text window (at the bottom). If the state has no propositions, the text window is cleared.

- If the LSTS has state propositions their names will appear on the bottom left corner of the window. Clicking on a proposition name will highlight all the states where that proposition appears. Clicking on the empty space below the names will remove the highlights.

- Clicking on a transition arrow head (or close to it) will turn the name of that action red in the list of action names at the left. If there are several arrow heads near the clicking point, the closest one is chosen. Clicking somewhere far from any arrow head will clear the name highlighting.

- Right-clicking on the graph window will zoom in (but without making the states or arrow heads larger). While zoomed, the view can be scrolled by middle-clicking. Pressing the key ‘1’ on the keyboard will reset the zoom.

- Pressing the key ‘n’ on the keyboard will show the state numbers of the states in the graph window. In this mode clicking on the states will print the state number of the clicked state on the text window (clicking on another state will print its number after the previous ones). Pressing ‘n’ again turns off this mode.
21. Shell scripts

Some shell scripts are included in order to make some often used tasks easier. These scripts run the input files through several TVT programs to achieve the desired result.

21.1 tvt.CFFDcompare

This script takes two LSTS files as parameter and tells whether they are CFFD-equivalent or not. If they aren’t, it will print the first difference found and the shortest path to it (the output of tvt.SBcompare).

It does not accept acceptance graph information as input.

21.2 tvt.CFFD_normalize

This script takes a LSTS file as input and outputs a normalized acceptance graph.
22. Scripting language of input files

22.1 Preface

All the input file formats of the TVT-programs support a simple scripting language which allows automatizing many things and make, for example, generic input files (which work, for example, for a differing number of components depending on a single value at the beginning of the file).

Although the input files of TVT programs are seldom intended to be written by hand, if there’s a situation where this must be done, this scripting language can be a great aid.

Note: If there are scripting language commands in an input file, they are “read out” of the input stream before the TVT parser gets them. For this reason the column counter of the parser will not work right in the lines where there are scripting commands (which can cause error messages to show the wrong column). The scripting language parser can’t get the correct column number if there’s an error in the command but only the line number (thus scripting error messages will only have a line number in them).

22.2 Command syntax

Scripting language commands can appear anywhere in the input file (they are read and interpreted independently of where they appear). The commands have the following syntax:

#{<command> <parameters>}

That is, a command starts with the character sequence #{ and ends with the character }). A command should reside in one line (i.e. it can’t be split into several lines). White spaces between commands and parameters are completely optional.

The scripting language supports variables and mathematical expressions. A variable can be any combination of letters (variables are case sensitive, which means that eg. “var”, “VAR” and “Var” are different variable names). The variable type is floating point. The syntax of the mathematical expressions is explained later.

The interpreter supports these six commands:

#{SET <variable>=<expression>}
#{LABEL <label>}
#{CB <label>;<expression>}
#{SCAN}
#{EVAL <expression>}
#{PRINT <expression>}
The meaning of the commands is the following:

**SET**  Evaluates the given expression and sets the result as the value of the variable. If the variable didn’t exist previously, it’s created.
Example: `{SET n=ind*4}`

**LABEL**  Creates a label at the current location in the file. After this a CB command can jump to this point. Any characters can be used in the label name except `};` note that white spaces are ignored (as if they didn’t appear at all). There’s no limit how many labels can be created; if the same label is defined again, the latest definition is the one which holds.
Example: `{LABEL loop1}`

**CB**  Conditional Branch. It evaluates the given expression, rounds it to the closest integer and if the result was anything else than 0, it jumps to the given label. If the result is 0 it does nothing.
Note 1: See the command SCAN in order to enable jumping forward in a file.
Note 2: The jump is done using the C++ fseek() command, which can fail if the input comes from the stdin stream. For this reason files with scripting commands should not be fed to the program through the stdin stream. In some systems it is possible to seek the stdin stream inside a limited range, but this is very platform-specific.
Example: `{CB loop1: ind<10}`

**SCAN**  The CB command can only jump to labels which have been seen. This means that normally jumping forward in a file is not possible. The `{SCAN}` command makes the parser to scan the entire file from the current position forward, searching for labels.
If you want to be able to jump forward in a file, you should put a `{SCAN}` command once at the beginning of the file (or somewhere before the scripts start).
Even though this command can appear anywhere, there’s usually no reason to put it anywhere else than the beginning of the file. It is enough to have this command executed once; there’s no need to execute it several times (it would only slow down the parsing).
The reason for this command is that this scanning can be slow and unnecessary (specially with huge files with no scripting or no forward jumps) and thus it’s only performed if specifically ordered.

**EVAL**  Evaluates the given expression, rounds it to the closest integer and passes the result as if it was in the input file. That is, the program just sees this number instead of the command. As scripting commands can appear anywhere in the input file, this makes it a very powerful command as it’s possible to generate numerical values anywhere in the input.
Example: `{EVAL ind*2} = "Action#{EVAL n}"`

The example could be seen by the program for example as: 4 = "Action2"

**PRINT**  This command is mainly for debugging purposes. It evaluates the given expression and outputs it (in floating point format) to stderr. This can be used to examine what values a variable gets, etc.
Example: `{PRINT ind}`
22.3 The syntax of the mathematical expressions

The mathematical expressions understood by the scripting language are very similar in syntax to the ones in the C++ language.

These are the supported mathematical operators (in order of precedence):

- () For grouping expressions.
- -A Unary minus.
- A^B A to the power of B.
- A*B A/B A%B Multiplication, division and modulo.
- A+B A-B Plus and minus.
- A=B A<B A>B Comparison between A and B (result is either 0 or 1).
- A&B Result is 1 if int(A) and int(B) differ from 0, else 0.
- A|B result is 1 if int(A) or int(B) differ from 0, else 0.

Operators with the same precedence are evaluated from left to right. As the unary minus has higher precedence than any other operator, then for example the following is valid: x*-y

Note that the ‘=’ comparison can be inaccurate due to floating point precision problems (eg. “sqrt(100)=10” probably returns 0, not 1).

Moreover, the following functions are supported:

- abs(A) If A is negative, returns -A, else A.
- acos(A) Arcus-cosine of A.
- asin(A) Arcus-sine of A.
- atan(A) Arcus-tangent of A.
- ceil(A) Rounds A up.
- cos(A) Cosine of A.
- cosh(A) Hyperbolic cosine of A.
- exp(A) e^A
- floor(A) Rounds A down.
- if(A,B,C) If int(A) differs from 0, the return value of this function is B, else C.
- int(A) Rounds A to the closest integer.
- log(A) Natural logarithm of A.
- max(A,B) If A>B, the result is A, else B.
- min(A,B) If A<B, the result is A, else B.
- sin(A) Sine of A.
- sinh(A) Hyperbolic sine of A.
- sqrt(A) Square root of A.
- tan(A) Tangent of A.
- tanh(A) Hyperbolic tangent of A.

The expression can contain variables created with the SET command (if an undefined variable is used, an error is occurs).

Examples of valid expressions:

1+2
x*2+1
m*c/sqrt(1-v^2/c^2)
sqrt(CoordX*CoordX + CoordY*CoordY)

22.4 Making loops

A loop where the variable ind gets values from 0 to 9 can be made like this:
This works pretty much like a do ... while() loop. Sometimes, however, it is necessary to make loops that don’t execute even once (i.e., we want to make a regular while loop). For this we need forward jumping, and thus a #{SCAN} command has to appear somewhere at the beginning of the file. Then we can do the loop like this (with ind going from 0 to n-1, where n can be also 0):

\[
\begin{align*}
\text{#{SET ind=0}} \\
\text{#{LABEL loop}} \\
\text{whatever} \\
\text{#{SET ind=ind+1}} \\
\text{#{CB loop: ind<10}} \\
\end{align*}
\]

22.5 Example

The following example is a part of a generic parallel composition rules file for the dining philosophers system. It creates the parallel composition rules section for any number of philosophers:

\[
\begin{align*}
\text{#{SET n=2} # Number of philosophers} \\
\text{Begin Rules} \\
\text{Amount} &= \text{#{EVAL 4*}n} \\
\text{#{SET i=1}} \\
\text{#{LABEL loop2}} \\
\text{({#{EVAL i*2}}, 1) ({#{EVAL i*2-1}}, 1) \rightarrow {#{EVAL i*4-3}}} \\
\text{({#{EVAL i*2}}, 2) ({#{EVAL (i*2+1)\%(n*2)}}, 1) \rightarrow {#{EVAL i*4-2}}} \\
\text{({#{EVAL i*2}}, 3) ({#{EVAL i*2-1}}, 2) \rightarrow {#{EVAL i*4-1}}} \\
\text{({#{EVAL i*2}}, 4) ({#{EVAL (i*2+1)\%(n*2)}}, 2) \rightarrow {#{EVAL i*4}}} \\
\text{#{SET i=i+1}} \\
\text{#{CB loop2: i<n+1}} \\
\text{End Rules}
\end{align*}
\]

Another similar example can be found in the chapter 8.
Bibliography

[Erk02] Timo Erkkilä. The LSTS file format and file processing library. Published as a part of the TVT documentation, 2002.


