C

VERIFICATION-RELATED COMMANDS

The man of science has learned to believe in justification, not by faith, but by verification.

—Thomas H. Huxley - Aphorisms and Reflections, 1907

Summary

TRANSYT provides a number of commands for the analysis of properties in transition systems, and in particular for the formal verification of timed systems. This appendix introduces the commands and their different options, which are related with the work presented in this thesis. For more information about other commands of the TRANSYT tool, refer to [PPb] or the on-screen help of the tool.

C.1 Failure analysis

Several commands are provided by TRANSYT in order to specify and analyze failure conditions for verification. A brief introduction to these commands is given in the following sections.

C.1.1 The `add_fail` command

Prior to the verification process, a number of properties must be associated to the system under verification. This can be done inside the `tsif` files that describe the corresponding models for verification. Moreover, the `add_fail` command allows to specify the failure conditions externally to the `tsif` files, such that different properties for the same system can be analyzed without actually modifying the system specification itself. This command allows to assign a boolean failure condition to either a label, an event or the entire transition system. Predefined failure conditions can be also assigned. Examples of the use of this command appeared in Chapter 5.

What follows is the on-screen help provided by TRANSYT for the `add_fail` command:

```
ts > help add_fail
ts:: add_fail [-dbxN][-i][-p][-m][-d] <fail type> ({<name1>}{,<name2>}) {EQN <equation>}

     Adds selected failure conditions to the default system.

     Fails can be added to three types of objects <fail type>:
         TFAIL: to a transition system specified by <name1>
         LFAIL: to a label specified by <name1>
         EFAIL: to the event <name1> of label <name2>

     The equation of the fail condition is specified by <equation>.

     Options:
      -dbxN    Temporal setting of the verbose level to N.
      -p       Adds a persistency fail condition.
      -i       Adds properties to internal labels.
      -m       Adds a conformance fail condition.
      -d       Adds a deadlock fail condition.
```

C.1.2 The `check_fails` command

The failure conditions can be evaluated in a variety of ways during the reachability analysis of the system under verification with the `traverse` command. Such evaluation, however, often makes the traversal process quite costly since, for example, each time an event is fired the failure conditions need to be evaluated. Despite of this mechanism, TRANSYT also allows the evaluation of failure conditions once the reachability analysis has been completed. Thus, the system can be first traversed with no failure analysis (`traverse` command with the `-nF` flag), and afterwards the `check_fails` command can be used to evaluate the failure conditions over the whole reachability space.

What follows is the on-screen help provided by TRANSYT for this command. Currently, no particular options exist for the `check_fails` command.

```
ts > help check_fails
ts:: check_fails [-dbxN]

     Checks the fail conditions of the default system after it has been traversed.

     Options:
      -dbxN    Temporal setting of the verbose level to N.
```

C.1.3 The `print_fails` command

The command `print_fails` allows to show information about what failure conditions are associated to the different objects of a system. Moreover, if the failure conditions have been analyzed either during the traversal or with the `check_fails` command, the resulting failure states can be also shown in the form of boolean characteristic functions. Examples of the use of this command appeared in Chapter 5.

What follows is the on-screen help provided by TRANSYT for the `print_fails` command.

```
ts > help print_fails
ts:: print_fails [-dbxN][-e][-l][-t][-a][-s] <model_name>

     Shows the fail information stored for the specified TS.

     Options:
      -dbxN    Temporal setting of the verbose level to N.
      -e       Prints information only for the events.
      -l       Prints information only for the labels.
      -t       Prints information only for the TSs.
      -a       Prints information for all fail conditions (by default
               only those with failure states are shown).
      -s       Prints the failing states.
```

C.2 Analysis of delay relations

The command `prune_dr` intends to refine the untimed state space of a system by using easy-to-find timing relations between events. Such relations can have nothing to see with the specified failure conditions, but its application can help to refine fake untimed concurrency situations, for example, so that a later verification process can be less costly.

Three options allow to explore different types of timing relations:

`-npp` If this option is set, the global nodal states of the system are identified and the delays of the events than become enabled in them are analyzed. Thus, given two events e_i and e_j newly enabled in a nodal point, if the upper delay bound of e_i is smaller than the lower delay bound of e_j, then event e_i will always fire before e_j in the timed domain. That is, an obvious relative timing relation is found between both events. As a consequence, the firing function of e_j can

be updated accordingly, so that both events become sequential, and the number of untimed states of the system is reduced. The reachability set of the system is not needed at any time.

`-pwp` Consider every pair of event in a system such that: they are simultaneously enabled in some set of states; they are simultaneously not enabled in some other set of states; and the first situation is reachable from the second one in a forward traversal step. Under these conditions, a local nodal point for such pair of events exists in the system. Therefore, if the upper delay bound of one of them is smaller than the lower delay bound of the other, an obvious relative timing relation is found, that can be used to prune the state space of the system.

If this flag is set, a CES containing both events is built, a trivial timing analysis is performed over it, and the resulting LzCES is finally composed with the original system. Since the process only modifies the transition relations of the affected events, the reachability set of the system is not needed.

Notice that is options also covers the previous one. However, in this case LzCESs are used, hence increasing the CPU cost and possibly causing state splitting.

`-gwp` This option extends the type of analysis provided by the previous option but for groups of events, not just pairs.

Finally, what follows is the on-screen output of running the `help prune_dr` command.

```
ts > help prune_dr
ts:: prune_dr [options] <ts name>

     Pruning of untimed concurrency which is actually fake if delays are
     considered in <ts name>.

     Options:
      -dbxN      Temporal setting of the verbose level to N.
      -npp       Perform pairwise pruning without using ESs, only with the
                 events around nodal points, if any.
      -pwp       Perform events pairwise pruning based on their delays. This also
                 covers the previous case, but here timed event structures are
                 used anyway, therefore increasing CPU cost and splitting.
      -gwp       Perform events group-wise pruning based on their delays.
```

C.3 The `uverif` command

In Chapter 5, a verification framework was presented for the verification of a system which must satisfy input-output conformance with respect to a given specification (see Figure 5.8). The (mirrored) specification acts as the environment of the system under verification, thus exercising the inputs of the system and reacting to the resulting outputs. The framework is typical of the verification of untimed systems such as speed-independent

asynchronous circuits. In such case, the verification consists in building the (untimed) reachability space of the resulting closed system and then checking for violations of the given failure conditions. TRANSYT implements this functionality through the `uverif` command. The command automatically mirrors the specification to build the environment of the system under verification, and builds the final closed system. Automatic failure conditions for input-output conformance, persistency and dead-lock can be also computed if appropriate options are specified (see below).

This command is particularly interesting for our purposes, due to the possibility of just building the closed system and the failure conditions for verification, and leaving the resulting system available for later manipulation. In particular for the verification of the timed behavior with the `tverif` command.

What follows is the on-screen output of running the `help uverif` command. A brief description is provided for the different options.

```
ts > help uverif
ts:: uverif [options] <ts name spec.> <ts name impl.>

     Untimed verification of a system <ts name impl.> versus its specification
     <ts name spec.> .

     Options:
      -dbxN               Temporal setting of the verbose level to N.
      -VnotConformance    Do not check conformance of the implementation with respect
                          to the given specification.
      -VnotPersistency    Do not check persistency in the implementation.
      -Vdeadlock          Check also the presence of dead-locks in the closed system.
      -Vclose             Build the closed system and create fails only.
      -Vnotdestroy        Do not destroy intermediate TSs after verification.
```

C.4 The `tverif` command

The `tverif` command of the TRANSYT tool implements the relative timing-based verification approach for timed systems presented in Chapter 4.

This command has two possible forms. The first one is similar to that of the `uverif` command. That is, two systems (the system under verification and its specification system) are given at the command line, together with flags to set the automatic construction of failure conditions (input-output conformance, persistency and dead-lock). This form of the `tverif` command can be superseded by the combination of the `uverif` command to build the closed system for verification, and then the second form of the `tverif` command to verify the closed system. Actually, the second form of the command for the verification of properties on a system is the most commonly used. See Chapter 5 for examples.

A number of options to control the execution and the output produced by this command are available. We describe them in the following sections.

C.4.1 Output

In Chapter 5 the on-screen textual output of the `tverif` command was shown in several examples. This output is very concise since it intends just to provide a sort of progress indicator of the verification process.

Despite of the brief on-screen output the execution provides much more detailed information in a sort of *log* file. Namely, a file with the name of the system being verified and the `.out` extension is generated. The amount of information in the file depends on the verbosity level specified with the `-dbxN` option, where `N` is a decimal digit. Bigger values of `N` imply more verbosity. A detailed description of the contents of the *log* file would require a lot of space, hence it is not given here. Simply say that, among other debugging information, details about the state where the failure trace stops, which is the event causing the failure, which strategies are used to build the event structure, etc. are provided for each iteration of the verification process.

Regarding the *log* file, finally say that of the `-HTML` option is specified, the *log* information is generated in the form of a browsable `HTML` file. The file has the name of the system under verification plus the extension `.html`. The file contains hyperlinks to a number of other files for the different objects (traces, CES, etc.) generated at each iteration of the verification process (see below).

The execution of the `tverif` command also creates a directory named with the name of the system under verification plus the extension `.files`. The directory contains, upon demand, a number of files with specific information for each iteration. Namely, the following different types of information can be provided at each iteration:

- The failure trace leading from the initial state of the system up to the state where the failure being verified was detected.

- A CES capturing the causality relations of all the events in the failure trace. Since the failure trace can be very long in some cases, the construction of this CES can also take a significant amount of time. This object is not necessary for the verification process, but can be useful for debugging purposes, for example.

- The suffix of the failure trace necessary to build a CES that provides enough timing information as to prove the non-existence of the failure in the timed domain.

- The resulting LzCES that captures the relative timing constraints that disprove the failure being verified.

- The complete lazy state space of the system. Producing this output may take a considerably amount of time unless the system is really small, say less than a few hundred states. In any case, the later manual analysis of more than a hundred states

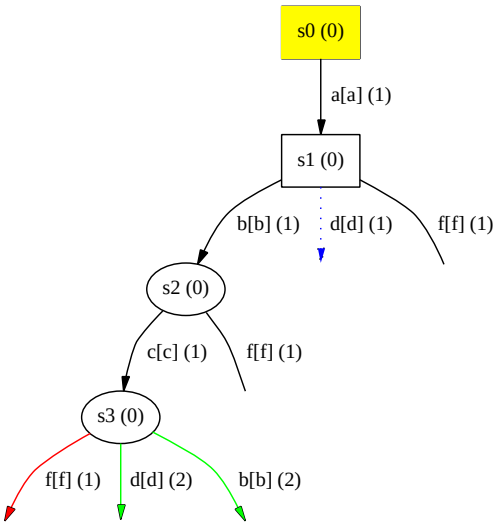

Figure C.1 DOT file corresponding to the failure trace in Figure 5.6 (a3).

might result in a very tedious task. Therefore, the usefulness of this information is often reduced to debugging purposes.

The generation of the files for the different objects can be controlled respectively with the following options: `-VwriteTraceN`, `-VwriteESN`, `-VwriteSuffixN`, `-VwriteTESN` and `-VwriteSTD`. Where `N` is a decimal digit which specifies the format of the output for each object. More precisely, `N` may take the following values: `0` (default) to produce no output for the object; `1` to produce a DOT (`.dot`) file for graphical visualization or printing of the object; `2` to produce a text (`.txt`) file for manual manipulation or interchange of the object with other users; or `3` to produce an XML (`.xml`) file for inter-operability between applications. The same option can be specified with different values for `N`, thus producing output files with different formats for the same object. Notice that in case of the `-VwriteSTD` option, no value for `N` can be given. If the option is specified, a DOT file is generated for the lazy state space of the system at the given iteration. Thus, the evolution of the incremental refinements can be graphically analyzed, for example.

Failure trace and suffix

A failure trace starts from the initial state of the system under verification. The trace reflects the sequence states and firings of events that lead to the failure state, as well as information about the enabled and firable events at each visited state. The trace ends in a state where the firing of an event either violates a transition failure condition, or lead to a state where an state failure condition is violated. The following explanations correspond

to the visual aspect of the DOT file for a failure trace (`-VwriteTrace1` option) or suffix (`-VwriteSuffix1` option).

States are depicted by means of either ovals or boxes. States inside a boxes correspond to *nodal states*. If the initial state of the system belongs to the trace or the suffix being depicted, the corresponding box or oval is filled in yellow. States are named following an increasing sequence of numbers. Since the same state can be visited more than once along a trace, between parentheses the occurrence number of the state inside the trace is shown.

Transitions between states are labeled with the name of the event producing the transition, followed by the name of the label the event belongs to, and the occurrence number of the event along the trace in parentheses. Other events enabled at each state are depicted as hanging arcs. If an enabled event is not firable in the state, a hanging line is just drawn. Different colors and types of lines are used to depict special events:

- Dotted arcs correspond to events that are disabled by the event firing in the given state.
- Blue arcs correspond to events which are in conflict with the event that fires in the given state. That is, is the event with the arc in blue fired, it will disable the event that fires currently in the trace.
- Red arcs correspond to events that cause a failure situation in the given state.
- Finally, green arcs correspond to events firable concurrently with some failure event, and could *scape* from the failure situation.

Figure C.1 depicts the visualization of the DOT file generated for the failure trace in the third iteration of the verification of the example in Section 5.2. In the figure: states s_0 and s_1 are nodal; event `f` is not firable in states s_1 and s_2; event `d` is disabled by the firing of event `b` in state s_1, and the firing of event `d` in state s_1 would disable `b`; finally, the firing of `f` in state s_3 causes a failure situation, which could be *escaped* if a second occurrence of `d` or `b` fired first.

Similar text-based or XML-based descriptions are produced by TRANSYT for a given failure trace (`-VwriteTrace2` and `-VwriteTrace3` options, respectively) or a suffix of the failure trace (`-VwriteSuffix2` and `-VwriteSuffix3` options, respectively). Details on the formats of the resulting files can be found in [PPb].

CES and LzCES

For each failure trace, a CES can be generated which captures the causal information between all the events appearing in the trace. Also the portion of the CES corresponding to the chosen suffix of the failure trace, can be generated in the form of a LzCES including the relative timing relations. The following explanations correspond to the visual aspect of the DOT file for a CES (`-VwriteES1` option) or a LzCES (`-VwriteTES1` option).

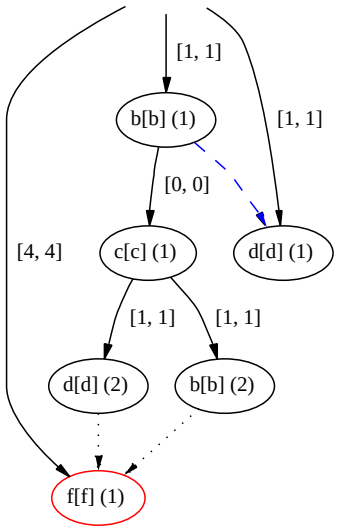

Figure C.2 DOT file corresponding to the LzCES in Figure 5.6 (b3).

Events are depicted as ovals and are labeled with the name of the event, followed by the name of the label the event belongs to, and the occurrence number of the event in the failure trace in parentheses. Events corresponding to failure situations in the trace used to derive the CES are colored red. The *root events* (*i.e.* with no predecessor) of the CES are those events enabled at the initial state of the failure trace or suffix used to derive it.

Arcs representing causality relations between events are drawn as black arrows. Such arcs are annotated with the delay ranges of the destination event. The delays are used for timing analysis in the corresponding iteration of the verification process. Disabling relations between events are depicted as dashed blue arcs, from the disabler to the disabled event, according to the information extracted from the trace. Finally, the relative timing relations are drawn as dotted black arcs, indicating that the source event fires before the destination event.

If the initial state of the failure trace or suffix used to derive the CES or the LzCES was a nodal state, then the causal arcs leading to the root events start in an imaginary common event which enabled all the root events simultaneously. And the arcs are annotated with the corresponding $[min, max]$ delay intervals. Conversely, the arcs leading to the root events are parallel and are annotated with $[0, max]$ delay intervals, if the initial state of the failure trace or suffix was not nodal.

Figure C.2 depicts the visualization of the DOT file generated for the LzCES in the third iteration of the verification of the example in Section 5.2. The LzCES was built using the suffix starting from state s_1 in the failure trace of Figure C.1. In the figure: the arcs leading to the root events start from a common point since the initial state of the failure

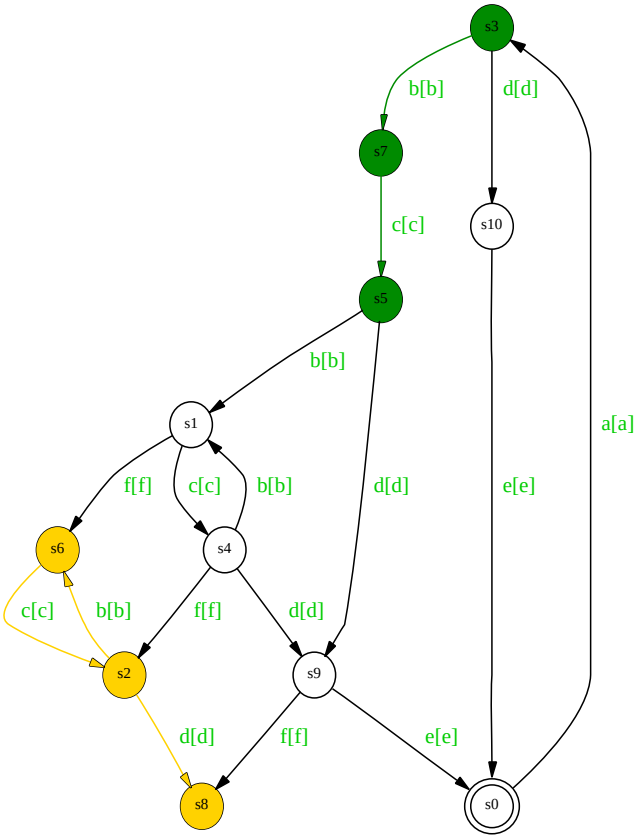

Figure C.3 DOT file corresponding to the LzTS in Figure 5.6 (d3).

trace was nodal; event b(1) disables event d(1); and two relative timing relations from events b(2) and d(2) to event f are depicted.

Similar text-based and XML-based descriptions are produced by TRANSYT for a given CES (`-VwriteES2` and `-VwriteES3` options, respectively) or a LzCES (`-VwriteTES2` and `-VwriteTES3` options, respectively). Details on the formats of the resulting files can be found in [PPb].

Lazy state space

With option `-VwriteSTD` a DOT file for the lazy state space of the system under verification can be generated at each iteration of the process. A circle is drawn for each state, which are numbered. The initial state is indicated by a double circle. Also, those states covered by the last refinement are colored green, whereas the remaining failure states are colored yellow. Arcs between states indicate the transitions of the system. The lazy transitions are not depicted.

Figure C.2 shows the visualization of the DOT file generated for the lazy state space after the third refinement of the verification of the example in Section 5.2.

C.4.2 Construction of the failure trace

At the time of writing, only two options allow to control the way the failure traces are built at each iteration of the verification process.

`-VfailTraceN` This option indicates the way a failure situation from which the trace will be built is searched. If the value of `N` is set to `1` or `2`, it specifies respectively that a BFS or a chained partial traversal must be executed until a failure state is reached. If `N` is set to `3`, a fast symbolic simulation search for failure states is used. Although the BFS-based method (default) is the slowest, it guarantees the construction of the shortest possible failure traces. Conversely, the simulation-based approach is the fastest, but at the cost of often building much longer failure traces. The approach based on a partial traversal with chaining represents an intermediate alternative.

`-VextendTraceN` This option indicates whether once the failure trace is built, it must be extended to beyond the failure state found, in order to capture other states that also present failure situations. If `N` is set to `0` (default), no extension is performed. Whereas if `N` is set to `1` or `2`, the trace is extended visiting states (if any) where the same failure situation is given, or visiting any subsequent failure state, respectively. Notice that covering several failure situation may often yield to more complex CESs and LzCESs. Although this may reduce the number of iterations of the verification process, it also complicates the timing analysis, the enabling-compatible product, and the readability of the timing constraints for back-annotation.

C.4.3 Construction of the LzCES

In order to build the LzCES at each iteration of the verification process, by default, the smallest possible suffix of the failure trace is taken such that it contains at least one failure situation. Then, the suffix is extended backwards until the timing analysis on the CES resulting from the suffix proves the non-existence of the failure, or contradicts the firing order of the events in the trace. That is, the shortest suffix is taken that yields a CES from which it can be proved that the trace is not timing-consistent.

Several options allow to control the way the LzCES is derived from an appropriate suffix of the failure trace.

`-Acapf` This option indicates that the shortest suffix of the failure trace that can be used to build the LzCES must contain all the failure situations present in the trace. Since more failure situations are covered by a single CES, less iterations of

the verification process are potentially required, at the expense of increasing the size of the LzCES at each iteration, and thus compromising its readability for later back-annotation.

`-Apreconc` Similarly to the previous option, if the option `-Apreconc` is specified, all the events which are concurrent with the root events of the LzCES according to the failure trace, must be included also in the LzCES. In such a way, the later enabling-compatible product will cover an area of the state space with less *entry-points*, hence producing less state splitting.

`-AtaComplete` This option forces to take the whole failure trace as the suffix used to build the LzCES. This results interesting in some cases, since a complete view of the causality and timing relations between the events in the trace is captured under a single, although often big, structure. Moreover, the enabling-compatible product with the resulting LzCES often results in a very localized refinement of the state space, since many conditions on the enabledness of events must be satisfied.

`-AfailGuided` This option forces the use of heuristics which try to reduce the size of the resulting LzCES for refinement. The heuristics focus the analysis on the particular failure situation being analyzed in a given iteration of the verification process, and discard timing relations unrelated to the failure.

`-AfilterTedges` In a similar vein, is this option is specified the size of the LzCES is tried to be reduced. In this case, the timing relations that do not help to prove the non-existence of the failure situation in all the failure states of the trace, are ignored. The result is that all the events in the final LzCES are related to the failure situation being analyzed. This and the previous option are recommended in order to improve the readability of the resulting LzCESs for later back-annotation.

Despite of the above options, two additional options control the use of nodal points information in order to build the LzCES. If a nodal point is reached during the extension of the suffix of the trace, the delays of the root events in the LzCES can be set to the corresponding $[min, max]$ ranges. Otherwise, the conservative $[0, max]$ delay range must be used. Although it is generally desirable to consider nodal points for that purpose, sometimes the conservative timing analysis yields LzCESs that produce a better refinement of the state space during the enabling-compatible product. By default, only global nodal points are taken into account. In order to consider also the local nodal points, the `-Elnp` option must be specified. The `-Enotgnp` option disables the use of global nodal points information.

C.4.4 Timing analysis

Two algorithms for timing analysis on the CES are implemented in TRANSYT: an exact algorithm due to [MD92] and a faster approximate algorithm due to [CDY99]. Since the latter algorithm is conservative, it often provides timing relations which produce *less aggressive* state space refinements during the enabling-compatible product. Due to this fact, some failure situations which do not exist in the timed domain of the system under verification, cannot be properly refined from the state space. That is, the verification process can result in conservative *false negatives*. The exact timing analysis algorithm is used by default. In order to select the approximate timing analysis algorithm, the `-EapproxTA` option must be specified to the `tverif` command.

Finally, if option `-EcheckConc` is specified, only timing relations between concurrently firable events in the CES are taken into account. In such a way, the resulting LzCES is simplified so that less redundant back-annotation information is provided. However, the simplification of the LzCES can result in more iterations of the verification process, to include the discarded timing relations in the analysis of other failure situations.

C.4.5 Miscellaneous

Several options allow to control other miscellaneous aspects of the verification process. Namely:

`-VtraverseFreqN` Although a complete traversal of the state space of the system is not required along the iterations of the verification process, it can be enforced every `N` iteration with the `-VtraverseFreqN` option, where `N` is a decimal digit. The complete traversal allows, among other things, a precise tracking of the remaining failure situations in the refined state space. Also, the `-VwriteSTD` option only applies to iterations where the complete state space has been computed. By default, no complete traversal is performed until the last iteration of a succeeding verification process, *i.e.* `N` equals `0`.

`-VreorderFreqN` The boolean variables used to represent the system symbolically can be reordered every certain number of iterations in order to improve the size of the BDDs. Despite of the immediate memory savings, the subsequent iterations of the verification process often complete faster thanks to the improvement in the boolean operations. Nevertheless, it must be taken into account that variable reordering is a very costly operation and should not be invoked too often. The value of `N` in the option `-VreorderFreqN` fixes the frequency of the variable reordering process, where `N` is a decimal digit. By default no variable reordering is performed, *i.e.* `N` equals `0`.

`-VminimizeFreqN` As it has been shown in Appendix B the implementation of the enabling-compatible product using BDDs requires the incorporation of several boolean variables to encode the configurations of the GRC and to distinguish the states where the timing analysis applies and those states where it does not apply. As a result, the size of the BDDs that encode the transition relations of the events of the system, sometimes grow exaggeratedly. However, as the number of iterations increases, some of those variables may become redundant, since later refinements may have pruned the part of the state space where those variables made sense. For this reason, TRANSYT can try to remove some of the redundant variables and recompute the simplified equations periodically. The option `-VminimizeFreqN` indicates the frequency of such variable removal, where `N` is a decimal digit (`N` equals `0` by default).

Remark that at the time of writing, this option is still in experimental use. However, the preliminary experiments show promising reductions in the sizes of the BDDs and also in the CPU times.

`-VnitersN` This option allows to specify the number of iterations of the verification approach that must be performed. As usual, `N` is a decimal digit. This allows an incremental verification process controlled by the user. If this option is not give, the verification process proceeds with as many iterations as required in order to prove the correctness of the system according to the properties under verification, or a counterexample failure trace that proves its incorrectness is found.

`-Vpwp` This option is equivalent to pruning from the state space, the delays relation between simultaneously enabled pairs of events, before the actual verification process starts. That is, this option is equivalent to the `prune_dr -pwp` command.

C.4.6 Summary of the `tverif` command

What follows is the on-screen output of running the `help tverif` command. A brief description is provided for all the aforementioned options.

```
ts > help tverif
ts:: tverif [options] <ts name spec.> <ts name impl.>

     Timed verification of a system <ts name impl.> versus its specification
     <ts name spec.> .

     Specific options:
      -VnotConformance    Do not check conformance of the implementation with respect
                          to the given specification.
      -VnotPersistency    Do not check persistency in the implementation.
      -Vdeadlock          Check the presence of dead-locks in the system.
```

```
ts:: tverif [options] <ts name>

     Timed verification of a system <ts name>.

     Specific options:
      -VnitersN         Perform only N iterations. If N=0 (default) iterate normally.

     Common options:
      -dbxN             Temporal setting of the verbose level to N.
      -Vnotdestroy      Do not destroy intermediate TSs after verification.
      -Vpwp             Perform atom's pairwise pruning before starting verification, based
                        on their delays.
      -VfailTrace       Indicates the way the failure trace is searched: (1) to perform a
                        partial traversal; (2) perform a partial traversal using chaining;
                        (3) to perform a fast simulation (bughunt).
      -VextendTraceN    Extend traces following fails. N may take the values: 0 for
                        no extension, 1 for extensions following a single fail, or 2
                        for multiple fails extensions (default N=0).
      -VtraverseFreqN   Force reachability analysis every N iterations (default N=1).
      -VreorderFreqN    Force reorder of BBD variables every N iterations (default
                        N=0, i.e. no variable reorder).
      -VminimizeFreqN   Force minimization of redundant BBD variables every N
                        iterations (default N=0). N must be multiple of that specified
                        with -VtraverseFreq option.
      -VwriteTraceN     Write the failure trace up to the initial state at each iteration.
                        N indicates the output format: 0 for no output, 1 .dot, 2 for .txt,
                        and 3 for .xml .
                        The different options are cumulative so that several outputs
                        can be produced. Default value is 0.
      -VwriteSuffixN    Write the portion of the failure trace used at each iteration.
                        N indicates the output format: 0 for no output and 1 for .dot .
                        outputs can be produced. Default value is 0.
      -VwriteESN        Write the complete ES for the full failure trace, at each iteration.
      -VwriteTESN       Write the timed ES portion at each iteration.
                        N indicates the output format: 0 for no output, 1 .dot, 2 for
                        .txt and 3 for .xml.
                        The different options are cumulative so that several outputs
                        can be produced. Default value is 0.
      -VwriteSTD        Write the resulting STD after each iteration.

     Options applicable when building timed Event Structures from a Trace:
      -Acapf           Capture all fail states in the trace when building the ES.
      -Apreconc        Capture ES preconcurrency following the trace.
      -AtaComplete     Builds an ES for the complete trace, then perform timing analysis, etc.
                       This may result interesting is some cases, however the fail removal is
                       very local and thus it is not suitable for hard verification processes.
      -AfailGuided     Uses the failure transitions as a source for heuristics which try to
                       reduce the size of the ESs and focus more locally around the
                       particular failure being analyzed.
      -AfilterTedges   Tries to filter (remove) those timing edges which do not actually
                       remove the fail from all the failure states of the trace. This is
                       intended to reduce the size of the resulting timed ESs such that
                       all the atoms that appear in it, are related to the fail being
                       analyzed.
```

```
Options applicable when building Event Structures:
 -Elnp          Use local nodal points when possible.
 -Enotgnp       Do not use global nodal points information. If this option is
                selected, -Elnp is selected automatically, otherwise the entry
                condition of the enabling compatible composition would not work
                in some cases.
 -EenablingsN   Sets the level of detail to take into account in order to put
                non-enabling information to the GRC created from an ES. N=1 means
                to use the minimum of information, i.e. a fast algorithm but
                with the need of more extra encoding variables. N=2 means to use
                information from the trace used to build the ES (if any), i.e.
                a more complex algorithm but with almost no extra encoding.
                Default value is N=2.

Timing analysis options:
 -EapproxTA     Use the approximate algorithm for timing analysis.
 -EcheckConc    Indicates whether not to consider only timing arcs between concurrently
                firable vertexes (if set), but just consider any possible timing arc
                (if not set). By default all possible timing arcs are considered.
```