OFFIS

OLDENBURGER FORSCHUNGS- UND ENTWICKLUNGSINSTITUT
FÜR INFORMATIK-WERKZEUGE UND -SYSTEME

The Rhapsody UML Verification Environment

Installation-Guide and Tutorial

$Revision: 2.74 $, $Date: 2004/05/06 15:52:37 $

Contents

1 System Requirements

1.1 Software

- Windows NT 4.0, 2000, or XP.
- Rhapsody 4.0, Buildnumber 235809, or Rhapsody 4.0.1, Buildnumber 240912, or Rhapsody 4.1, Buildnumber 360709, or Rhapsody 4.2, Buildnumber 378424.
  Note that these are *exact* requirements. Other and in particular newer versions of Rhapsody may not cooperate properly with the **ruve**.
- Cygwin 1.3.12-1 or later (available at `http://www.cygwin.com`) with at least the following packages:
  - from the category "Base":
    - the package "bash: The GNU Bourne Again SHell"
    - the package "grep: GNU grep, egrep and fgrep"
    - the package "gzip: The GNU compression utility"
    - the package "libreadline5: GNU readline and history libraries"
    - the package "sed: The GNU sed stream editor"
    - the package "sh-utils: A set of GNU utilities"
    - the package "tar: A GNU file archiving program"
    - the package "textutils: The GNU text processing utilities"
    - the package "which: displays where a program is located"
  - from the category "Devel":
    - the package "make: The GNU version of the 'make' utility"
    - the package "gcc: C, C++, Fortran compilers"
  - from the category "Libs":
    - the package "regex"
    - the package "tcltk: TCL/TK libraries"

  For remote verification (cf. sec. 3.13) the following additional cygwin packages are required:
  - from the category "Net":
    - the package "openssh: The OpenSSH server and client programs"

  Note that the cygwin installer ("setup.exe") only installs the base system by default, so you might have to select these packages explicitly in the "Select Packages" dialog (cf. fig. 1). If you didn't do so during the initial installation, simply run the installer again and add the packages this time.

1.2 Hardware

- For best performance results contemporary hardware is recommended, i.e. at least 1 GByte of memory and at least 500 MHz processor-clock.

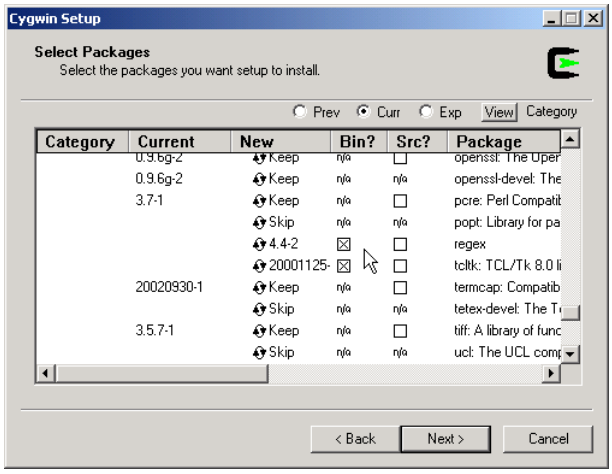


Figure 1: Cygwin installation: Select Packages.

2 Installation Guide

2.1 Unpacking

The Rhapsody UML Verification Environment (**ruve**) is provided as a `tgz` archive which should be unpacked from the cygwin-`bash` as follows[1]:

1. In the cygwin-`bash`, change to the directory where the **ruve** is to be installed[2], e.g.

   ```
   $ cd C:/Programs
   ```

2. Unpack the archive by

   ```
   $ tar -xvzf ⟨DATE⟩-uve-release-⟨RELEASENR⟩.tgz
   ```

   This unpacks all files into the directory

   ⟨*DATE*⟩`-uve-release-`⟨*RELEASENR*⟩

   to which we refer to as "*UMLVERIFROOT*" from now on.
   In the above example, *UMLVERIFROOT* would denote

   `C:\Programs\`⟨*DATE*⟩`-uve-release-`⟨*RELEASENR*⟩.

2.2 The Connection to Rhapsody

Rhapsody is made aware of the **ruve** by copying the property file `site.prp` from the package to the right location within the Rhapsody installation (and in some cases by additionally modifying Rhapsody's file `siteC++.prp`):

1. [Recommended] Don't have any Rhapsody running for the next steps.

[1] We encountered problems when using windows tools like WinZip® for unpacking the archive. The usage of the `tar`(1) program delivered with cygwin is highly recommended.
[2] Please choose a directory which does not contain spaces.

2. Find out the complete path of the Rhapsody installation, e.g.

   ```
   C:\Programs\Rhapsody
   ```

   which we refer to as "*RHAPSODYROOT*" from now on.

3. [Optional] Find out the complete path of the XMI Toolkit for Rhapsody installation, e.g.

   ```
   C:\Programs\XMIToolkit4Rhapsody
   ```

   which we refer to as "*XMI4RHAPROOT*" from now on.

4. Run the installation-script *UMLVERIFROOT*`\install.sh` in the cygwin-`bash`[3], e.g.

   ```
   $ UMLVERIFROOT/install.sh
   ```

   The script is supposed to find out the path of the cygwin installation (*CYGWINROOT*) and *UMLVERIFROOT*, but it has to ask the user to provide *RHAPSODYROOT* and *XMI4RHAPROOT*. If the installer finds suitable installations paths by itself, a suggestion is presented in square brackets which can be accepted by simply pressing `Return`.

   If the installer is not able to determine the exact build number of your Rhapsody version, it asks you to provide this number (which can be found in the 'About Rhapsody ...' dialog under Rhapsody's 'Help' menu).

   The installation-script enters these paths into a `site.prp` file which is then copied into the Rhapsody installation. The original `site.prp` is copied to `site.prp.bak` if there is no previous backup.

   If a 'XMI Toolkit for Rhapsody' installation was found, the two config files `toolkit.ini` and `properties.ini` are patched such that the exporter will produce XMI descriptions of the models which are suitable for **xuve** (XMI UML Verification Environment) [OFF03b]. The original files are copied to `toolkit.ini.bak` and `properties.ini.bak` if there are no previous backups.

[3] Alternatively you can double-click the install.bat in an explorer window.

In the example, the complete installation dialog reads:

```
RUVE installer -- version ‘‘<RELEASENR>’’.

Please enter the Rhapsody root directory:
  [C:\Programs\Rhapsody]

Please enter the 'XMI Toolkit for Rhapsody' root directory,
or enter 'none' if the toolkit is not installed:
  [none]

Using:

  OSTYPE       = cygwin
  CYGWINROOT   = C:\cygwin
  UMLVERIFROOT = C:\Programs\⟨DATE⟩-uve-release-⟨RELEASENR⟩
  RHAPSODYROOT = C:\Programs\Rhapsody
  RHAP_BUILDNO = 235809
  XMI4RHAPROOT = none
  GNUTOOLSPATH =

site.prp backed up to C:\Programs\Rhapsody/Share/Properties/site.prp.bak.

Installation done.
```

5. The installation-script checks if all necessary cygwin tools are installed. If this is not the case, the missing packages are listed and the script stops with the message `Installation failed!`. See section 1.1 on how to install new packages in cygwin. Then, re-run the installation script.

6. In some Rhapsody installations, the file

   RHAPSODYROOT`\Share\Properties\siteC++.prp`

   contains a line which begins with

   ```
   Property Environment Enum
   ```

   but does not contains the enumerator “Verification”. This causes “Verification” not to appear in the “Environment” choice-list in the “Settings”-tab of a configuration (see sec. 3.4–3.6).
   The installation script checks for this case and fixes the `siteC++.prp` if needed. The installation dialog then contains the two additional lines:

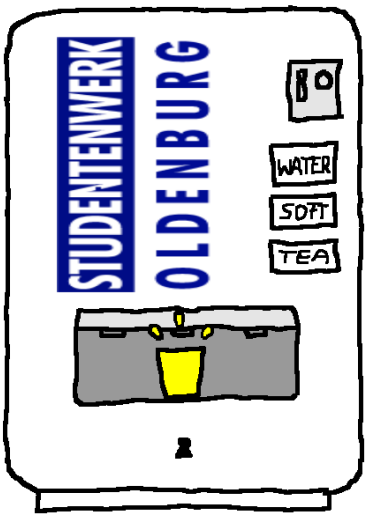
   ```
   siteC++.prp backed up to C:\Programs\Rhapsody/Share/Properties/siteC++.prp.bak.

   siteC++.prp fixed ('Verification' added to 'Environment').
   ```

3 Tutorial

3.1 The Running Example: Vending Machine

The VendingMachine sells drinks. Water at the price of 50 cent, a softdrink at the price of 1 euro, and tea at the price of 1.5 euro. As coins are inserted, lamps on a choice panel signal the possible choice, i.e. after inserting 50 cent, water will be enabled unless the water stock is empty.



Concerning money, the VendingMachine is not very sophisticated. It is for example not possible to buy water if only a 1 euro coin is inserted, since the machine does not keep track whether it already has a 50 cent coin for change. Furthermore there is a gambling component: the machine only signals if a particular drink is in stock if the corresponding amount of money is inserted. I.e. if the machine is out of water, one can only insert more money and try for another type of drink or leave the money in the machine. Supernumerary coins are given back immediately, e.g. if 1.5 euro are already inserted, no more coins are accepted.

The VendingMachine is modeled as a composite class with four parts (cf. fig. 2).

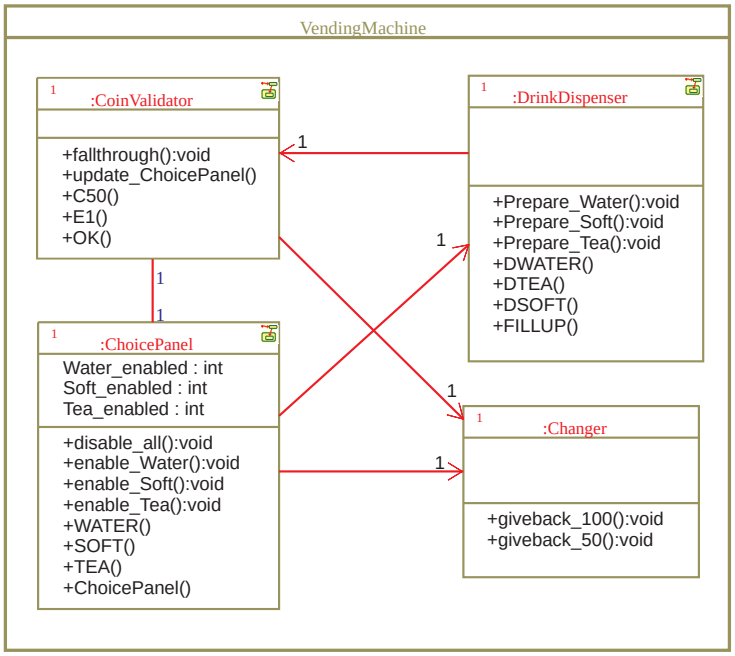


Figure 2: The Vending Machine: Class Diagram.

There is a CoinValidator which accepts coins 'C50' or 'E1', keeps track of the already inserted money by the states within state 'waitOK', and enables the lamps at the ChoicePanel by entry-actions of the corresponding states. For example if only 1 euro coins are inserted, then only the choice of softdrink is enabled since the machine cannot give back the 50 cent change if water would be selected.[4]

If supernumerary coins are inserted, the Changer's methods 'giveback_50()' or 'giveback_100()' are called to give the money back immediately. Figure 3(a) shows the Statechart of the CoinValidator.

Within the ChoicePanel, the private attributes 'Water_enabled', 'Soft_enabled', 'Tea_enabled' model the choice-lamps which can be controlled by the public methods 'enable_Water()', 'enable_Soft()', and 'enable_Tea()', respectively. Within these methods, the ChoicePanel checks whether the DrinkDispenser has the corresponding drink in stock by looking at the state of the DrinkDispenser.

The ChoicePanel accepts choices 'WATER', 'SOFT', and 'TEA'. If the chosen type of drink is actually enabled, the ChoicePanel starts the DrinkDispenser by sending 'DWATER', 'DSOFT', or 'DTEA', resp., and then disables all choices. Figure 3(b) shows the Statechart of the ChoicePanel.

The DrinkDispenser is modeled as an AND-state with three parts, one for each type of drink (assume there are three different outlets for drinks) but they are operated sequentially since only one drink can be chosen at a time.

In every compartment of the AND-state there are four states which model the amount of drinks in stock[5] If a request for an available drink arrives, the drink is prepared by calling a (dummy) method, e.g. 'Prepare_Water()', and the CoinValidator is sent an 'OK' event to have it accept another set of coins.

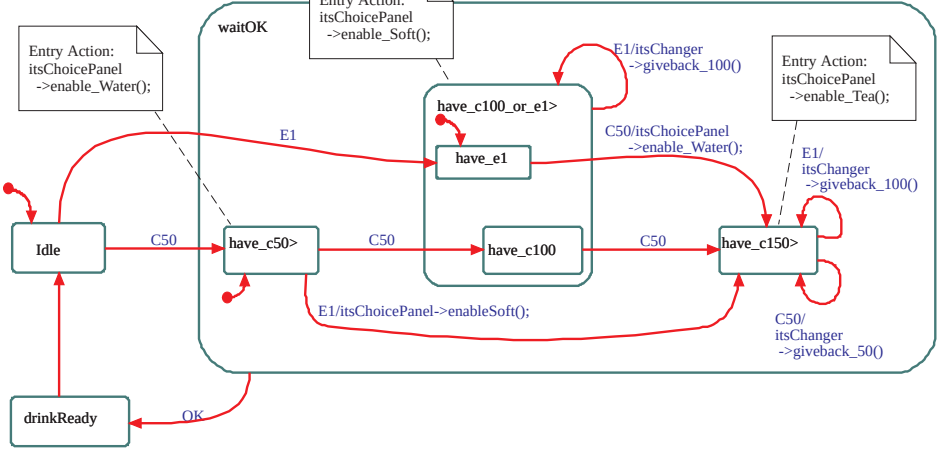
All drink stocks can simultaneously be reset by sending a 'FILLUP' event, which also enables the choice-lamps at the ChoicePanel according to the already inserted amount of money, such that a customer who inserts 1.5 euro into the machine with no drinks in stock can wait for a fill-up and then make his choice.

Note that the DrinkDispenser assumes that it is not started if there is no drink in stock, which is in fact guaranteed by the ChoicePanel. Otherwise it would not send the 'OK' event to the CoinValidator, thus reaching a deadlock. Figure 3(c) shows the Statechart of the DrinkDispenser.

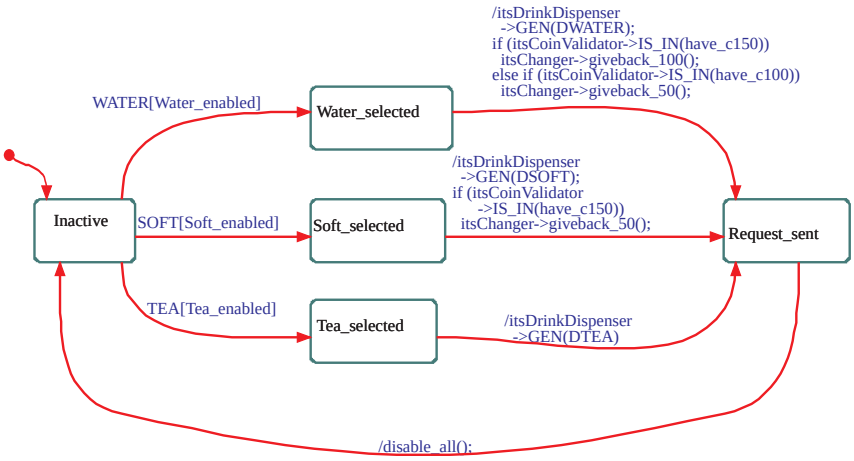
Note that the vending machine uses a special option of **ruve** which operates the statecharts in a way which corresponds to "run idle" in the simulation in Rhapsody, i.e. external events are only accepted if the event queue is empty.

[4] The design of the system contains the following flaw: It was assumed that there is no history connector needed in 'have_c100_or_e1' as destination of the self-loop, as long as the outgoing transition with trigger 'C50' starting in the default-state ensures that water is enabled in 'have_c150'.
One observable consequence is, that the sequence 'E50', 'E50', 'E1', 'FILLUP', 'WATER' does not yield water if the vending machine had been out of water at the beginning of the sequence (since the method 'update_ChoicePanel()', which is a reaction on 'FILLUP', only enables water if the CoinValidator is in 'have_c100').
Another consequence is, that the sequence 'E50', 'E50', 'E1', 'WATER' yields water and only 1 euro change. This is not observable by the verification environment since the Changer provides only dummy methods and no real implementation.

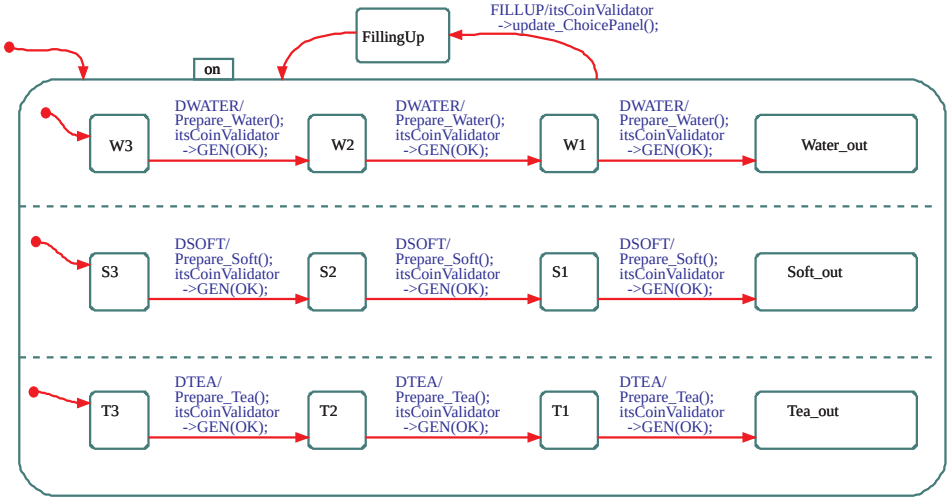
[5] this could alternatively be modeled as counters of type integer, but the current encoding is more efficient for verification.



(a) CoinValidator's Statechart



(b) ChoicePanel's Statechart



(c) DrinkDispenser's Statechart

Figure 3: The Vending Machine: Statecharts.

This ensures for example that no more money is accepted once a choice is made, since then there are always internal events in the queue until the final 'OK' event is dispatched (cf. paragraph 3.2.2 for details about external events).

For the verification examples in sections 3.4 – 3.6, a system is used which contains exactly one instance of class VendingMachine including one instance of each of its part.

3.2 Terminology

A *system configuration* is a valuation of all attributes of all alive objects in the system, which comprises in particular the active states of the reactive objects' statecharts.

The **ruve** considers only system configurations at *step boundaries*, i.e. after taking a transition. Taking a transition comprises executing the exit action of the current state, executing the actions of the transition, and the entry actions of the destination state. For AND-states, taking multiple transitions in different compartments of the AND-state due to the same cause (dispatched event or run-to-completion step) counts as *one* transition.

By a *run* of the system we denote a sequence of system configurations, where two subsequent system configurations c_1 and c_2 in the run are related in that c_2 is reachable from c_1 by taking a transition.

A *run-to-completion step* consists of taking one or more transitions, until a stable configuration without enabled outgoing transitions is reached.

For a Rhapsody design built using the means provided in appendix A, the **ruve** is able to

- solve "drive-to-state" tasks,

  e.g. "is it possible for the DrinkDispenser to reach state 'Water_out' if only 'C50' coins and 'WATER' requests are used?", and to

- solve "drive-to-property" tasks,

  e.g. "is it possible for the ChoicePanel that attribute 'Soft_enabled' has a value of 1 while 'Water_enabled' is still 0 if any type of coins and choice requests are used?", and to

- accomplish "invariance-check" tasks,

  e.g. "is it true that attribute 'Tea_enabled' having a value of 1 implies 'Water_enabled' is 1 if one enters any sort of coin and choice requests?", and to

- accomplish "pattern-check" tasks,

  e.g. "is it true that whenever a 'C50' coin is received by the CoinValidator, then the attribute 'Water_enabled' of the ChoicePanel has a value of 1 one step later?",

by checking all possible runs of a Rhapsody design starting at the step boundary just after the initialization phase, i.e. after all initial objects have been created and all reactive objects' statecharts have taken the transition originating at the root state's default state [i-L02].

For the "drive-to-state" and "drive-to-property" tasks, the **ruve** generates a prefix of a run which leads to a system configuration with the wanted properties *if* such a run exists, otherwise it indicates that the state or property is not reachable. Such a run is called *witness trace*, since it witnesses the reachability of the state or property, or simply *trace*.

For the "invariance-check" and "pattern-check" tasks, the **ruve** indicates that the stated property holds *if* it actually holds, and otherwise generates a prefix of a run which *contradicts* the property. Such a run is called *counterexample*, or *error path*, since invariants usually state wanted properties, the system presumably contains an error if the property does not hold, or also simply *trace*.

When the **ruve** solves a task, three major phases can be distinguished and observed in Rhapsody's "Output Window":

1. *Model-generation*, i.e. the transformation of the UML-model into a transition-system representation suitable for the underlying model-checker.

2. *Model-checking*, i.e. actually solving the task on the level of transition-systems.

3. *Trace-generation*: depending on the result of the model-checking phase, a trace is generated, translated back from the transition-system level into the vocabulary of the UML-model, and prepared for visualization.[6]

Right after the model-checking phase, the **ruve** prints a summary of the outcome of the current task, i.e. whether a state or configuration is reachable or not, or whether a "invariance-check" or "pattern-check" property holds or not.

The summary includes the type of task ("invariance-check", "pattern-check", "drive-to-state", "drive-to-property"), the C++ expressions given by the user to describe e.g. the configuration to drive to, the used assumptions (if any), and the mode and set of external events, if given by the"ExternalEventTrace" property.[7]

If a witness-trace or counterexample exists, the summary closes by announcing the following trace-postprocessing phase.

Prefixes of runs are presented in form of a *timing diagram* which shows the attribute valuations and statechart configurations at every system configuration of the run and a live sequence chart (LSC) which shows only the event communication in the run (cf. sec. 3.4).

3.2.1 Caveats

Between Step Boundaries Note that, according to the notion of a step induced by step boundaries, it is not possible to drive to properties which hold only "between step boundaries".

For example when the VendingMachine is filled up by sending an event 'FILLUP', the action at the corresponding transition withing the DrinkDispenser calls the method 'update_ChoicePanel()' of the ChoicePanel to update the choice lamps. The implementor of method 'update_ChoicePanel()' chose to set the lamps for water, softdrink, and tea sequentially in reverse order:

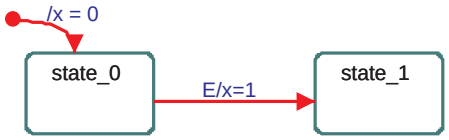
[6] To be precise, it is the model-checking phase itself which already generates the trace, but it is restricted to only those parts of the model which are *necessary* for the task, other parts are not considered. To present a complete trace, this trace is *simulated*, i.e. intuitively executed stepwise, *including* all parts of the model to provide sensible values for the unnecessary parts, and thus may require a significant amount of time.

[7] The summary of the last run is also written to the file `ruve-result.txt`.

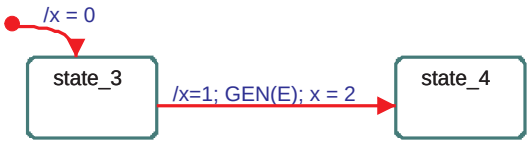
```
if (IS_IN(have_c150))
  itsChoicePanel->enable_Tea();
if (IS_IN(have_c100_or_e1) || IS_IN(have_c150))
  itsChoicePanel->enable_Soft();
if (IS_IN(have_c50) || IS_IN(have_c100) || IS_IN(have_c150))
  itsChoicePanel->enable_Water();
```

Thus there exists a system configuration where the lamp for tea is on but the one for water is not.

This configuration *cannot* be found by a "drive-to-property" task since the **ruve** considers only the result of the whole action, and the overall result is never a combination where tea is enabled at the ChoicePanel and water is not.



(a) Event reception



(b) Event sending

Figure 4: Observing events and conditions.

Event- and Non-Event Property Expressions Note that event queries in expressions (cf. sec. A.7.1) are evaluated in the *starting* state of the transition which is taken on receiving an event resp. which sends the event in its action part when taken. A combination of event and non-event queries in a property expression holds in a system configurations c iff it holds with

1. the non-event sub-expressions evaluated wrt. to c and
2. the event sub-expressions evaluated wrt. to the event-receiving and -sending on a transition from c to some system configuration c' following c in the same run.

For example let the statechart in fig. 4(a) be the statechart of a class C with attribute x. Then it is possible to solve the "drive-to-property" task for the property[8]

```
root->p_C->x == 0 && ER_E( ENV, root->p_C )
```

[8]Here, `ER_E` denotes the observation of receiving an event of type 'E'. Analogously, `ES_E` denotes the observation of *sending* an event of type 'E'. See section A.7.1 for a detailed description.

but not for

```
root->p_C->x == 1 && ER_E( ENV, root->p_C )
```

since when in state 'state_0', `x == 1` does not hold, and when in 'state_1' the event receive query does not hold according to the above introduced interpretation.

Analogously, let the statechart in fig. 4(b) denote the statechart of a class D with attribute x. Then it is possible to solve the "drive-to-state" task for the property

```
root->p_D->x == 0 && ES_E( root->p_D, root->p_D )
```

but not for

```
root->p_D->x == 1 && ES_E( root->p_D, root->p_D )
```

since the value of 'x' is only considered at step boundaries, and also not for

```
root->p_D->x == 2 && ES_E( root->p_D, root->p_D )
```

since when in state 'state_3', `x == 2` does not hold, and when in 'state_4' the event send query does not hold.

Thus to avoid mis-interpretations, it is recommended not to change attributes on transitions whose event sending is referenced in a query expression or to send all referenced events at the beginning of the action s.t. the actual values between step boundaries match the values observed at the step boundaries. In the latter example, if the action is changed to `GEN(E); x = 1; x = 2;`, then the value of 'x' is the same value it has in state 'state_3', thus it is more intuitive that only the first of the above "drive-to-state" tasks is solvable.

3.2.2 Events and EventQueues

In the following, we call an event instance generated *within* the model an *internal event* and an event generated by the environment according to the "ExternalEventTrace" property an *external event*. Note that the distinction is not class-based, but characterizes the origin of an event instance.

EventQueue Length In the current version of **ruve**, the user has to provide the maximum length of the event queue – that is, how many events can be stored at most in the queue.

The most important indicator of how many event places you need, is the maximum number of `GEN` calls in one run-to-completion step. If the model needs to emit 'OMStartBehaviorEvents' (see below) to start objects with unstable initial states, the event queue length has to be set at least to the number of 'OMStartBehaviorEvents' which may simultaneously be present in the queue.

Note that "triggered operations" are also mapped to event communication. So you have to increase the length of the queue by one whenever there is communication via "triggered operations" in the model.

External events are *in fact* not propagated through the queue, and thus do not need to be considered when determining the queue length. The verification environment may repeatly *decide* not to take the top-level event of queue but

rather *guess* an event from the environment. With this strategy, a very fine-grained interleaving of internal actions and external events is simulated. You can enforce a more coarse interleaving if you set the property 'ExternalEvent-OnlyWhenIdle' to 'True'. Now, external events are only accepted if the internal event queue is empty. In the visualization of the error path, the external events are indeed visible *in the queue* since some possible sending point in time is calculated by the visualization tool.

To set the properties, right-click on a configuration and select "Features".In the "Properties"-tab, select "Subject" "CPP_CG" and "Meta Class" "Verification". In this list, double-click "MaxEventQueueLength" resp. "ExternalEventsOnlyWhenIdle" and change the values appropriately.

Event Quantities For each event class, the **ruve** reserves a certain amount of memory in its internal representation to store internal events of this class and for each event class used in the "ExternalEventTrace" property, there is a single additional memory place reserved for external events.

By default, the number of memory places for events is set to the length of the event queue,, i.e. if the event queue length would be set to 3 for the VendingMachine, then there would be space reserved for 3 'WATER' events, 3 'DWATER' event, 3 'E1' events, etc., although, for example, the 'E1' event is never generated from within the model, but intended to be exclusively used for external events.

The default is reasonable since events are constructed when they are inserted into the queue and destructed after being dispatched to the destination object, thus the event queue length provides an upper bound on the number of required event places.

There are cases where the default is not "right" and needs to be corrected manually. When applied to external events, the upper bound is typically too large, which has negative impact on execution time of the model-checking task. If these event class is *only* used from the environment, the number of internal memory places should be set to zero. Additionally, when the length of the event queue *len* is greater than one, there might be events classes in the model for which it is not necessary to have *len* event objects alive simultaneously. Reducing the number to a sufficient level will reduce the model-checking complexity. In the special case when the event queue is completely filled with events of type *Ev and* the dispatching of the first *Ev* leads to a sending of a new event again of type *Ev* in the same run-to-completion step, then the upper bound is too small by 1 since the dispatched event is destroyed only after taking the last transition of the run-to-completion step. This may lead to erroneous results since allocation of the *Ev* for sending fails, which causes the internal memory management for the model to produce unpredictable values.

For these cases, one can explicitly set the number of memory places to reserve for each user-defined event class by setting the Property "MaxEventQuantities" to a semicolon-separated (possibly empty) list of pairs of event class name and non-negative quantity of the following form:

$$\langle \text{EventClass} \rangle \, , \langle \text{Quantity} \rangle \; [\; ; \langle \text{EventClass} \rangle \, , \langle \text{Quantity} \rangle \;]^*$$

Note that the list must not contain *any* spaces and that event class name has to provided in its qualified form, i.e. prefixed by the package name. Event

quantities for event classes not present in the model are silently ignored.

For example, to reserve no memory for the external 'WATER' events and four places of 'DWATER' event in the VendingMachine (although there is no need to do so), one would write:

```
Default::WATER,0;Default::TEA,4
```

The property is located directly below the 'MaxEventQueueLength" property. To set it, double-click "MaxEventQuantities" and change the value appropriately.

OMStartBehaviorEvents The verification environment uses the same mechanism as Rhapsody to ensure that reactive objects which have transient transition originating at their initial states[9]are "started properly", namely 'OMStartBehaviorEvents'. These events are processed regularly by the queue and ensure that the flow of control is eventually passed to a started object which stayed in his initial state waiting to take untriggered (but potentially guarded) transitions. Since the default to reserve one 'OMStartBehaviorEvent' per reactive object would to be expensive for verification purposes, the user has to manually decide how many of these events his model needs. This number n has can be *added* to the "MaxEventQuantity" list described in the previous paragraph, in the following form:

⟨...;⟩`OMStartBehaviorEvent,n`

For models where every reactive object is stable after the initial transition, the default of zero 'OMStartBehaviorEvents' is of course sufficient.

To ensure that all of the settings above do not lead to 'queue overflows' or 'out of memory' problems during the model-checking phase, a predefined verification task called "CheckMemoryBounds" can be performed. Chapter 3.11 describes this procedure in more detail.

3.2.3 SMI Modus

For internal use only.

[9]the initial state is the target state of the default transition

3.3 Global Settings

For the **ruve** to work, a few global properties have to be set. They are best set once in the "Features" menu of the project s.t. they are the default value in all configurations which are created during the tutorial.

1. Start Rhapsody.
2. Open *UMLVERIFROOT*`\examples\TheVendingMachine\TheVendingMachine.rpy`
3. Right-click on the project TheVendingMachine in a browser window and select "Features" (cf. fig. 5[10]).
4. Select the "Subject" "CPP_CG" and the "Meta Class" "Package" and set "DefineNameSpace" to "True" (cf. fig. 6).
5. Select the "Subject" "CPP_CG" and the "Meta Class" "Relation" and set "DataMemberVisibility" to "Public" and "ImplementWithStaticArray" to "FixedAndBounded".
6. Select the "Subject" "CPP_CG" and the "Meta Class" "Verification" and set "ExternalEventOnlyWhenIdle" to "True" and reduce all external events to an internal quantity of zero by setting "MaxEventQuantities" to "Default::C50,0; Default::E1,0; Default::WATER,0; Default::SOFT,0; Default::TEA,0; Default::FILLUP,0;" (without spaces, cf. paragraph 3.2.2).

   Additionally, you should reduce the integer range to "0 to 7", by setting "IntegerLowerBound" to "0" and "IntegerUpperBound" to "7".

[10] All screenshots in this tutorial are taken from a Rhapsody 4.0 version. Never versions look slightly different, but the corresponding elements should be easily recognizable.

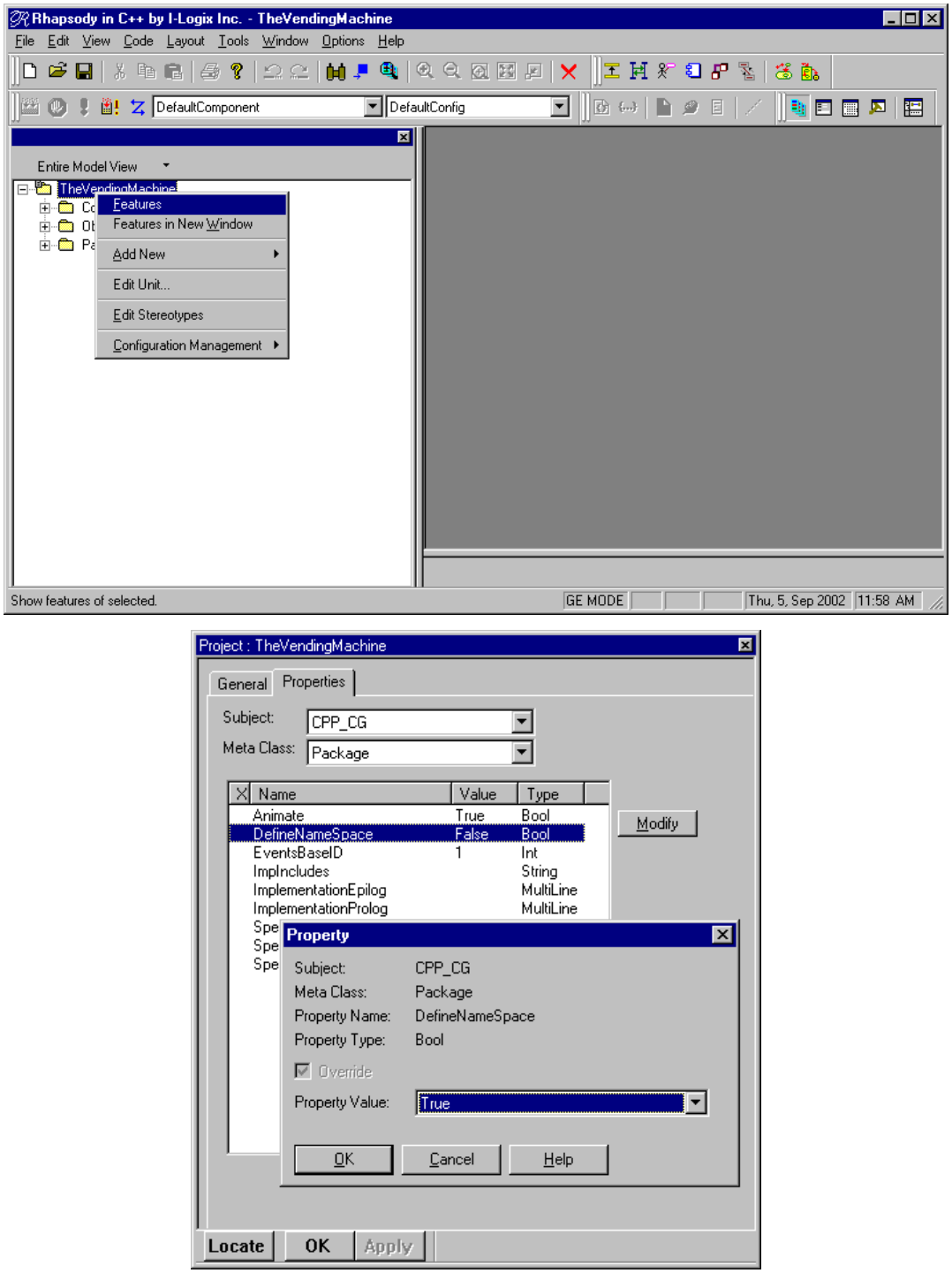


Figure 6: Setting global properties, Category Package.

3.4 "Drive to State"

1. Task:

   Check, whether it is possible for the CoinValidator object to reach state 'drinkReady' if only 'C50' coins and 'WATER' requests are sent from the environment.

2. Create a new configuration "VerifyDTS" within component "DefaultComponent" (cf. fig. 7).

3. Right-click on "VerifyDTS" in the browser and select "Features" to open the features dialog.

4. In the "Initialization"-tab, choose "Derived", check the box "VendingMachine", and uncheck the box "Generate Code for Actors" (cf. fig. 8).

5. In the "Settings"-tab, set

   - "Instrumentation" to "None",
   - "Time-Model" to "Simulated",
   - "Statechart Implementation" to "Flat", and
   - "Environment" to "Verification" (cf. fig. 9).

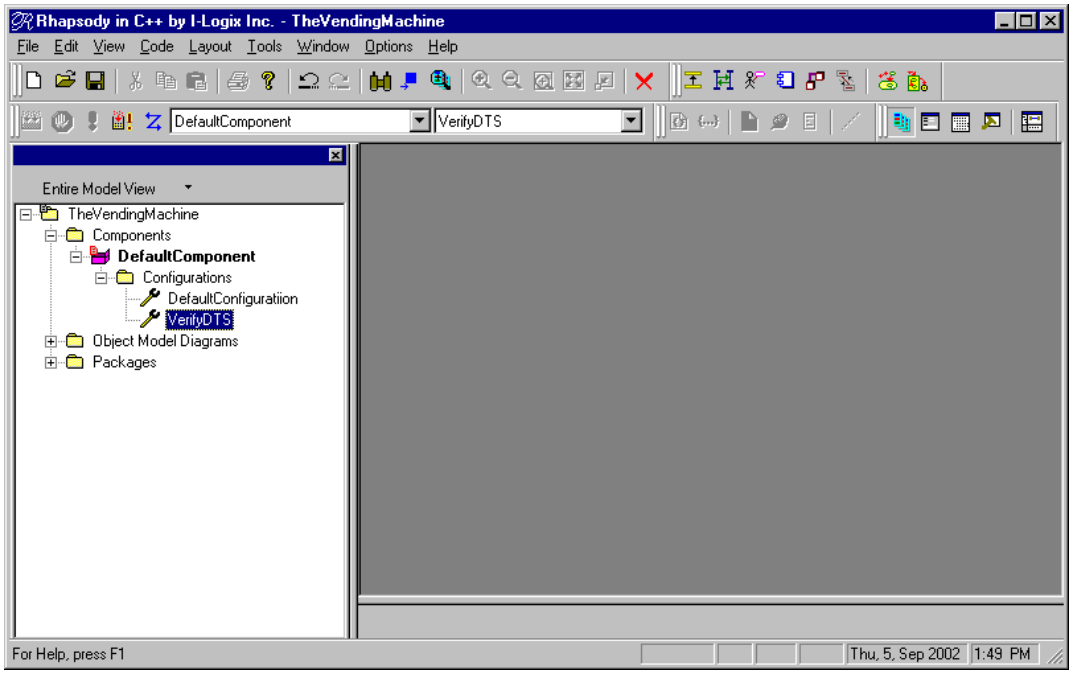


Figure 7: Creating the new configuration "VerifyDTS" within the component.

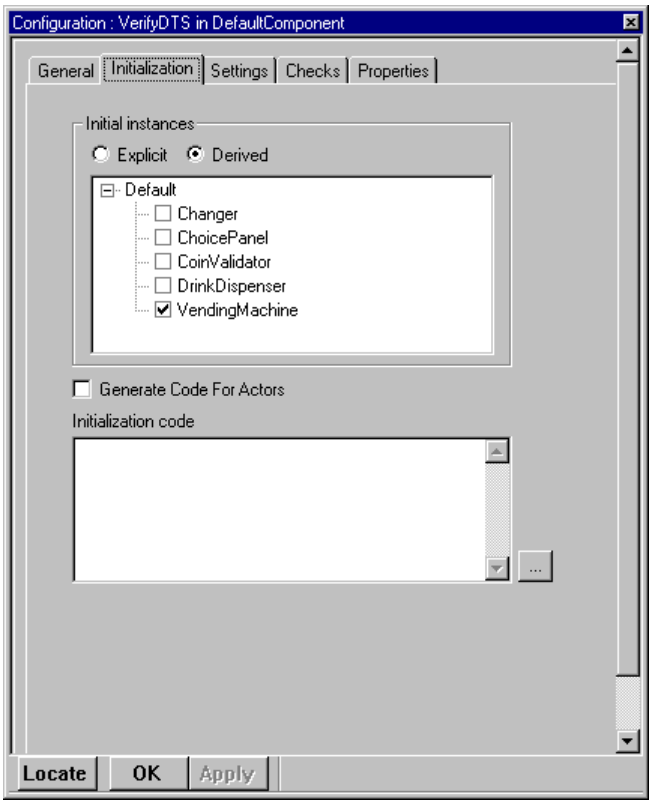


Figure 8: Initialization.

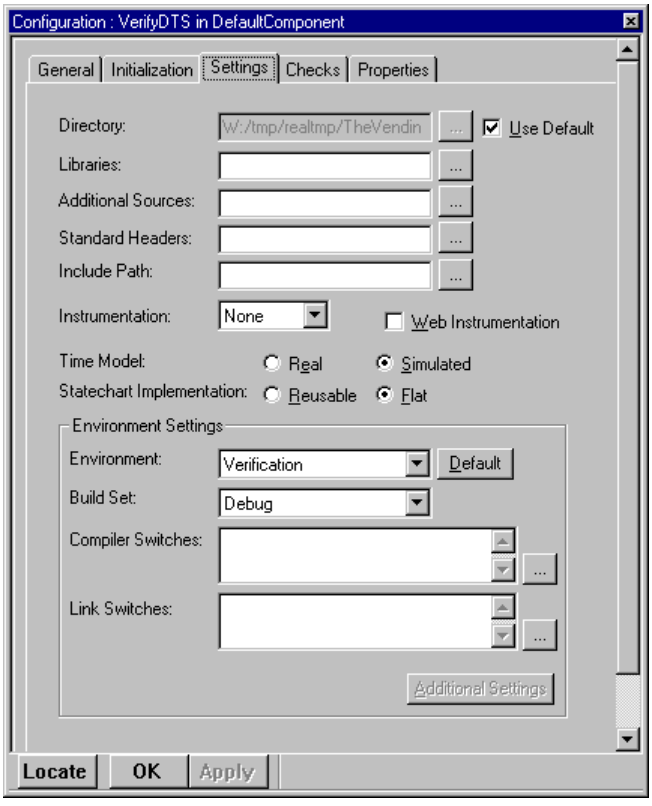


Figure 9: Settings.

5. In the "Properties"-tab, select "Subject" "CPP_CG" and "Meta Class" "Verification" and

   - for "Spec" choose "DriveToProperty"[11]
   - set "Spec::DriveToProperty" to the C++ expression

     ```
     root->p_VendingMachine->itsCoinValidator->IS_IN(drinkReady)
     ```

     which must not contain a semicolon (";") (cf. fig. 10).

     Here, 'root' is the name of a "root object" which is only visible inside the verification model and serves as the owner of all objects created in 'main()'.

     In the example, the object of class VendingMachine is created in the function 'main()' in the generated C++ code and assigned to the variable 'p_VendingMachine', i.e. an object created in 'main()' is accessible with a navigation expression starting at 'root' followed the objects' name in 'main()'.

   - set "ExternalEventTrace" to

     ```
     root->p_VendingMachine->itsCoinValidator, C50; \
     root->p_VendingMachine->itsChoicePanel, WATER; \
     ```

     to determine the set of events which may be sent to the system from the environment (cf. fig. 11). If this property is empty, a "closed system" is examined, i.e. all consumed events have to be generated within the system. There is then effectively no "environment" which could send events of types which are not sent in the system itself like the coin or choice events in the running example.

     The syntax of this property is lines (!) of the form

     ⟨Destination⟩ `,` ⟨Event⟩ `; \`

     where ⟨Destination⟩ is a navigation expression starting at `root`.
     Note that leaving out the backslash ("\"), spreading a single entry over multiple lines, or entering whitespace *behind* the backslash makes Rhapsody produce an invalid Makefile, i.e. the Generate/Make/Run (see below) will stop immediately with an error message from the make program!

   - set "ExternalEventModus" to "ndet", that is, whenever the system becomes idle, one of the events from "ExternalEventTrace" can be chosen non-deterministically and sent to the system.

     Setting the "ExternalEventModus" to "det" causes exactly as many events as in "ExternalEventTrace" to be sent to the system in exactly the order they are listed in "ExternalEventTrace". In the current example, first a single 'C50' event, then an event 'WATER', and then no further events would be sent.

[11] "drive-to-state" tasks are in fact a subset of "drive-to-property" tasks with a particular form of properties

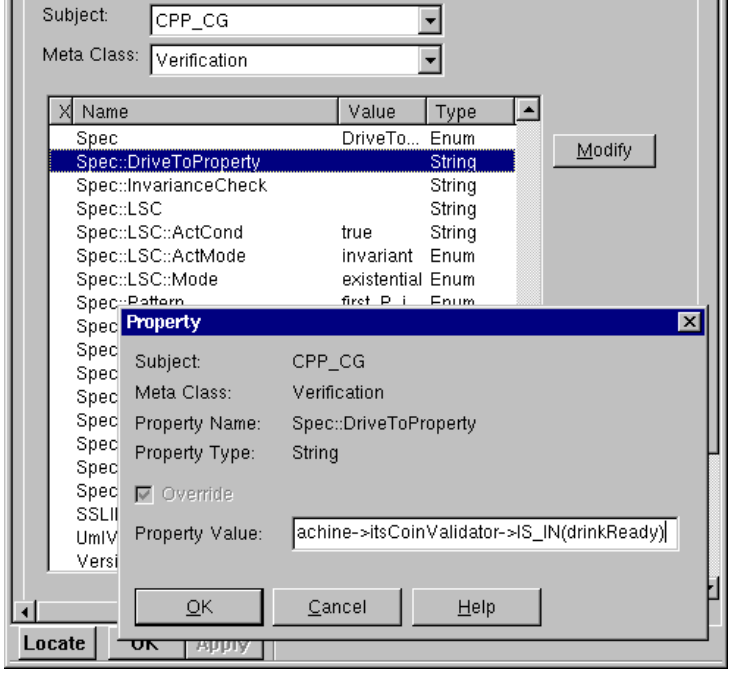


Figure 10: Denoting the state to drive to by a C++ expression.

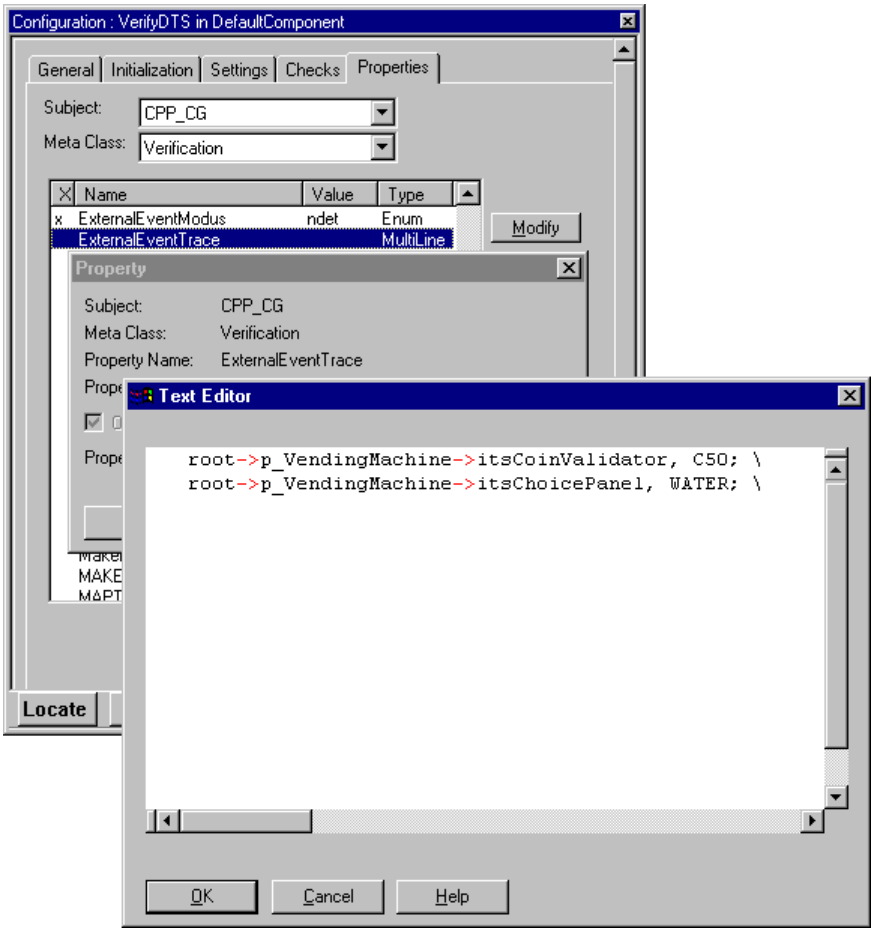


Figure 11: External events.

6. Make “VerifyDTS” the active configuration (right-click on the configuration and select “Set as Active Configuration” (cf. fig. 12)).

7. From the “Code” menu, select “Generate/Make/Run” (cf. fig. 13).

8. Confirm to create a new directory for the new configuration where all temporary files and the result will be created (the “Directory” should not be changed in the “Settings”-tab of the configuration).

9. In the executor-window one can observe the Rhapsody code-generation and lots of output from the **ruve** (cf. fig. 14).

Finally the timing diagram and LSC viewer pop up and shows the prefix of a run which leads to the requested state as a timing diagram (cf. fig. 17) and as an LSC (cf. fig. 18). Figures 15 and 16 describe the usage of the timing diagram and LSC viewer, respectively.

The timing diagram contains one *waveform* per object and attribute and an additional waveform per reactive object which shows the configuration of the object in terms of basic states (e.g. “`Default::CoinValidator[1]'s statechart`”). A waveform “`EventQueue`” shows the content of the event-queue. Depending on the kind of specification, several waveforms prefixed by “`Spec::`” denotes the current evaluations of the specified properties. The numbers at the bottom of the waveform window denote step boundaries. Note that the values to the right of the step boundary 0 are the values just after the initial step (cf. sec. 3.2), since the way to this step boundary where all objects (except for events) have been created and all reactive objects have left their default state is “pre-executed” such that the system configurations *before* this step boundary are also not considered for the verification task.

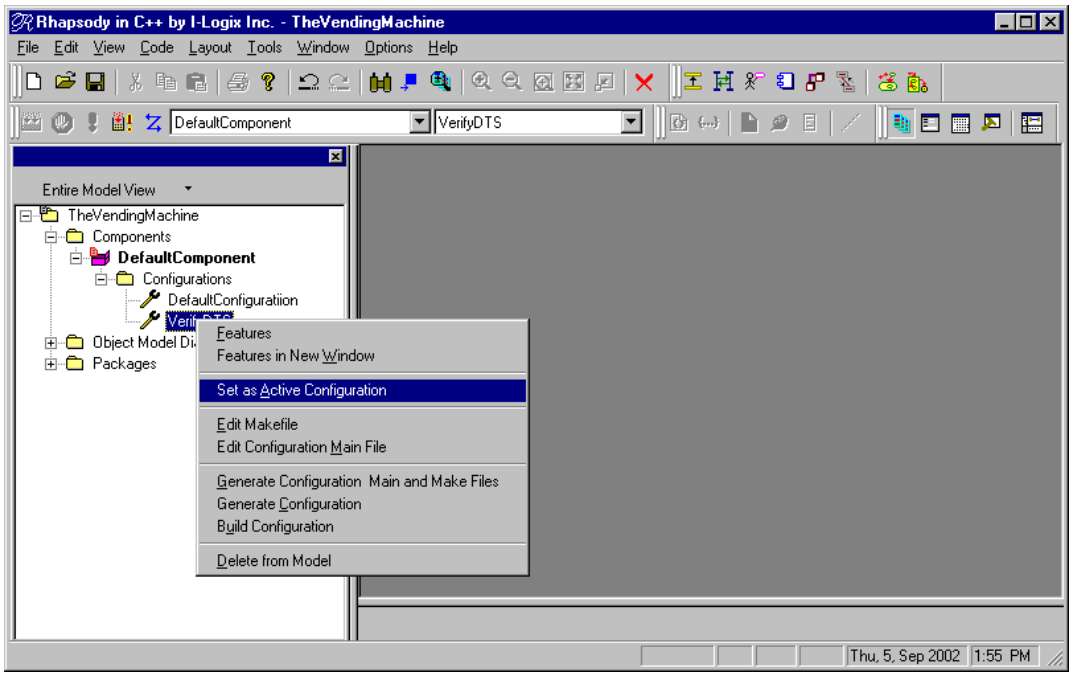


Figure 12: Setting the active configuration.

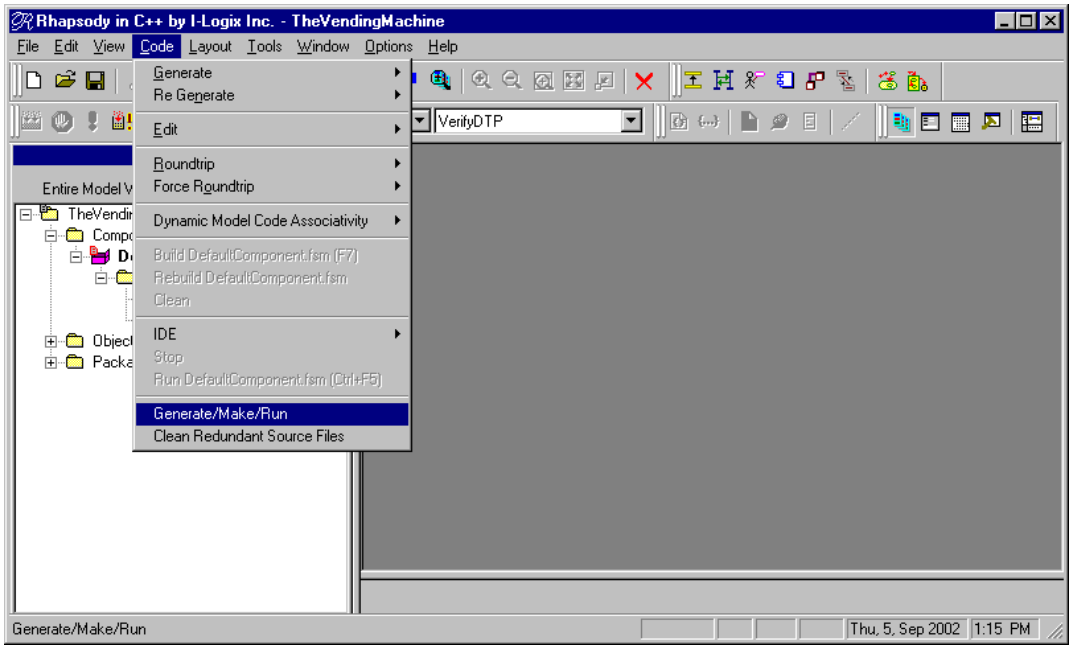


Figure 13: Starting the **ruve**.

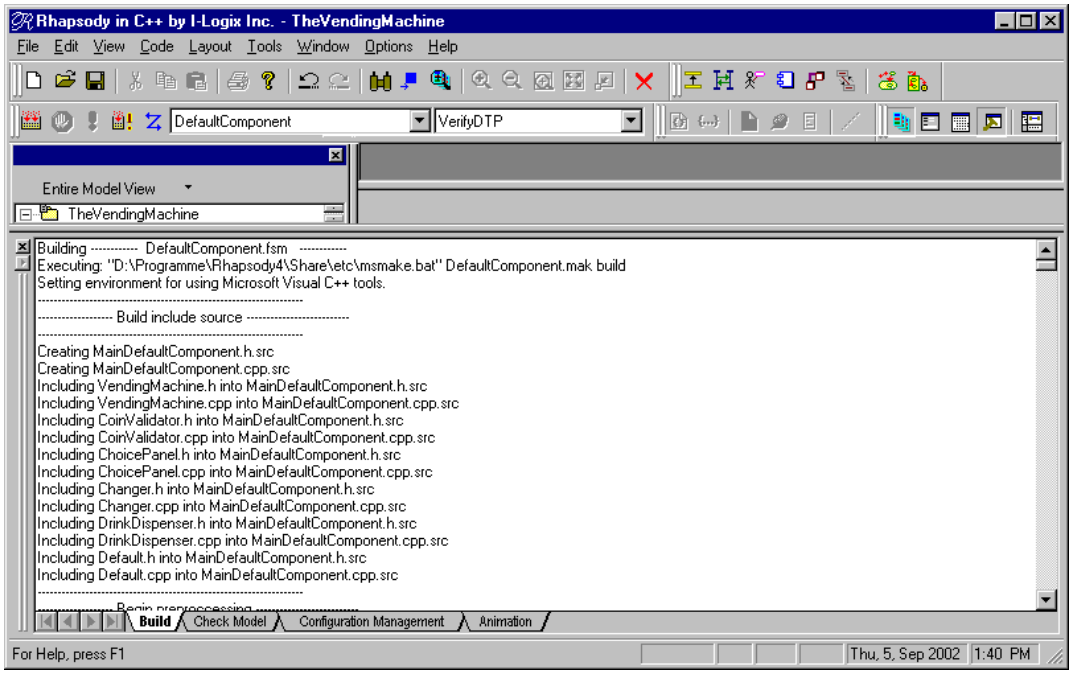


Figure 14: **ruve** output in Rhapsody's "Output Window".

- when the pointer is on the name or waveform window, the cursor keys scroll the names or waveforms respectively.
- holding down the shift key and left-clicking in the waveform window shows a ruler which can be used to identify the step.
- the "search" text field can be used to search attribute waveforms by regular expressions. All matching names are highlighted in red and the viewer scrolls to the first matching waveform.
- the "filter" text field can be used to "filter out" waveforms from the view by a regular expression. If the radiobutton "pos" is on, then those waveforms which match the pattern remain in the viewer, when "neg" is on, only those which *do not* match the pattern remain.
- the "reset" button clears both text-fields and causes the initial set of waveforms to be shown.
- the waveform can be zoomed by "Zoom In" and "Zoom Out" in the "View" menu.
- many properties like colors, fonts, etc. are customizable from the menus below "Miscellaneous/Configure".

Figure 15: Usage of the Timing-Diagram Viewer.

- currently to view LSCs, the editor LSCEdit is "abused" – only the obvious viewing functionality of the LSCEdit like using the scrollbars and the "Quit" menu-item are meant to be used.

Figure 16: Usage of the LSC Viewer.

The LSC contains one *instance line* per object participating in event communication during the prefix of a run and possibly an instance line for the "environment" representing the source of external events.

The example in figs. 17 and 18 together read as follows:

- The six links called `its<ClassName>` corresponds to the associations between the four parts of the vending machine as shown in fig. 2. Note that objects have identities starting with 1 for every class.
- The waveform `p_VendingMachine` denotes the identity of the composite object which holds the four parts.
- The first step shows the initial configuration of the statecharts in terms of basic states, i.e. since the DrinkDispenser has entered its AND state the three basic states 'W3', 'S3' and 'T3' became active.
  Furthermore, the `enabled` flags of the ChoicePanel have been initialized with zero by calling 'disable_all()' in the constructor.
- After the first step, an external event 'C50' was generated and inserted into the queue.

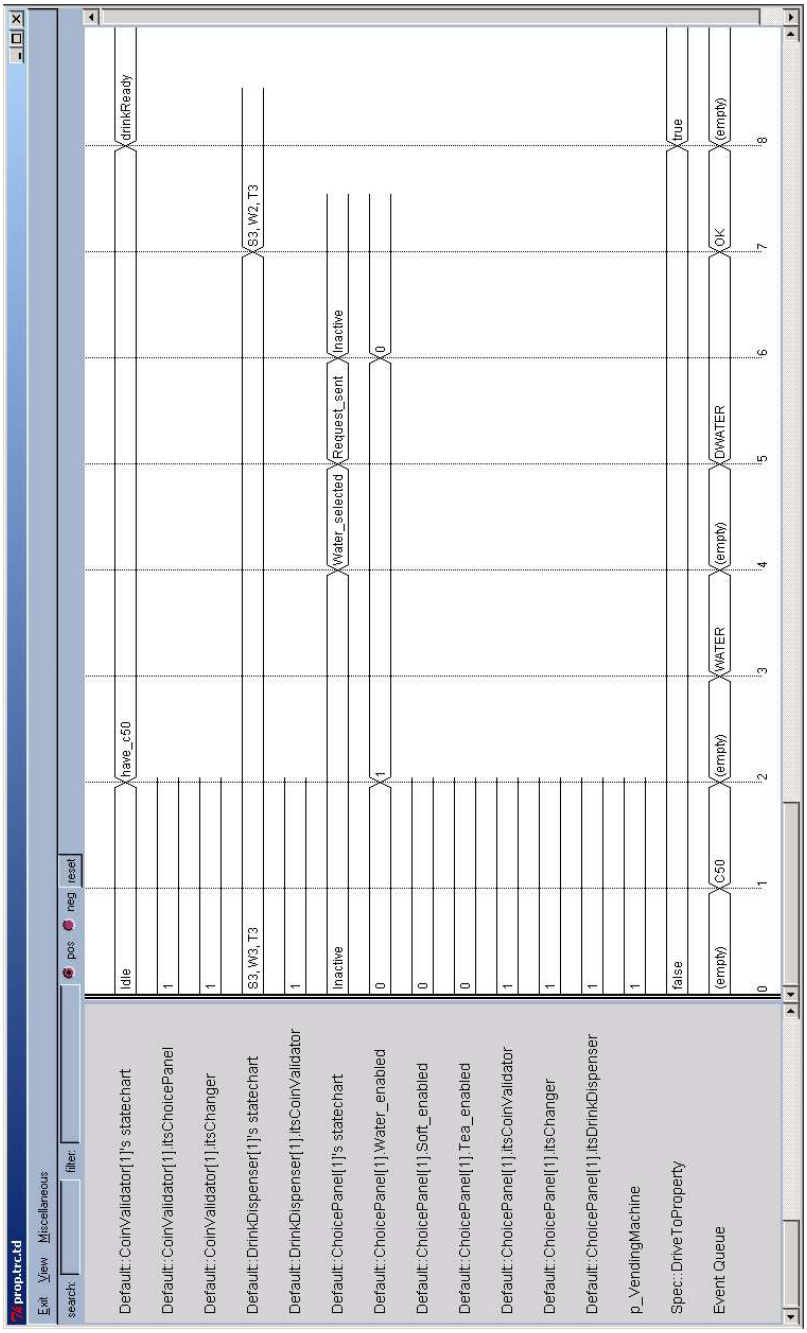


Figure 17: A trace leading to state 'Water_dispend'.

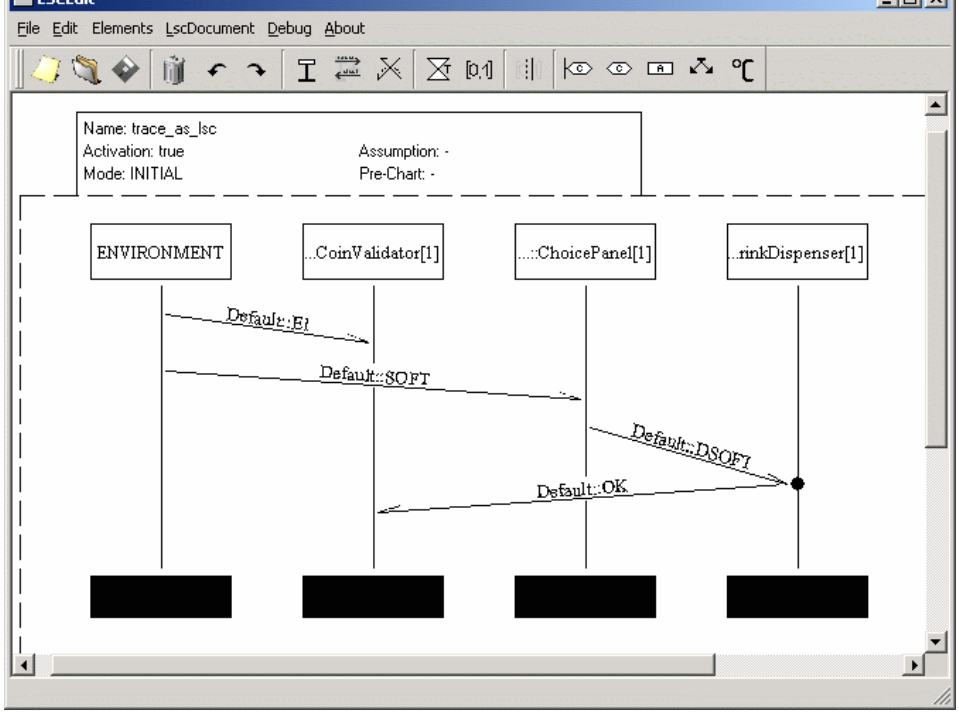


Figure 18: LSC.

- This 'C50' event is dispatched in the second step and the CoinValidator therefore moves to the state 'have_c50'. Here, the entry action is performed which sets `Water_enabled` to 1.
- After reaching the state 'have_c50' in the third step, an external event 'WATER' is generated. The event is dispatched in the fourth step and causes the ChoicePanel to enter the 'Water_selected' state.
- Since 'Water_selected' has an outgoing transition with an empty guard, the object directly proceeds to the state 'Request_sent'. With this transition, a 'DWATER' event was generated and sent to the DrinkDispenser.
- Again, we have an enabled outgoing transition and the ChoicePanel goes back to the 'Inactive' state in the next step.
- Now the 'DWATER' event is processed by the DrinkDispenser which leads to the statechart configuration 'S3', 'W2' and 'T3'. An 'OK' event is sent to the CoinValidator.
- The 'OK' is dispatched to the CoinValidator in the next step, which therefore enters the wanted state 'drinkReady'.

Before starting the next verification task, the viewers have to be closed (only then, the log in Rhapsody's execution window is closed by "Done.").

3.4.1 Drive to Configuration

"Spec::DriveToProperty" is an arbitrary C++ expression (according to appendix A.7) thus for example the following expression could be used for a "drive-to-configuration" task which asks for a system configuration where the CoinValidator is in state 'have_c100_or_e1' and the ChoicePanel is in 'Water_selected':

```
 root->p_VendingMachine->itsCoinValidator->IS_IN(have_c100_or_e1)
&&  root->p_VendingMachine->itsChoicePanel->IS_IN(Water_selected)
```

(only reachable, if two 'C50' coins are inserted).

3.4.2 On "Stutter" Steps

A *transient state* is a state with an outgoing transition without a trigger, e.g. with a guard only. If a transient state is reached, the outgoing guards are only evaluated at the beginning of the *next* step and if no guard holds, the system stays in the current state and becomes stable, i.e. ends the run-to-completion step. These *stutter-steps* are also visible in the generated traces.

In the VendingMachine example, there is no stutter step since all transitions end at states where all outgoing transitions either have no guard and no trigger (such that the run-to-completion step never stops there) or are all annotated with triggers such that the state is not transient.

3.4.3 On "Unnecessary" Event Sendings in the LSC

For some paths generated by "drive-to" tasks, the LSC shows a *cold* event sending, i.e. only the sending but not the reception happens during the considered prefix of a run.

Cold events may even show up if they are not required, if the configuration to drive to is already satisfied in the state which is the source of the transition at which the observed event is sent. This is correct behavior of the **ruve** since the configuration of attributes is considered together with event sending (cf. sec. 3.2.1). If the system configuration to drive to is visited only for a single step, and if leaving the system configuration causes an event to be sent, then this event sending is always observed together with the attribute configuration.

3.5 "Drive to Property"

1. Task:

   > Check, whether it is possible for the ChoicePanel that attribute 'Soft_enabled' has a value of 1 while 'Water_enabled' is still 0 if any sort of coins and choice requests are sent from the environment.

2. Create a new configuration "VerifyDTP" within component "DefaultComponent".[12]

3. Right-click on "VerifyDTP" in the browser and select "Features" to open the features dialog.

4. Set up the "Initialization"- and "Settings"-tab as in sec. 3.4.

5. In the "Properties"-tab, select "Subject" "CPP_CG" and "Meta Class" "Verification" and

   - for "Spec" choose "DriveToProperty"
   - set "Spec::DriveToProperty" to the C++ expression

     ```
     root->p_VendingMachine->itsChoicePanel->Soft_enabled == 1
     &&
     root->p_VendingMachine->itsChoicePanel->Water_enabled == 0
     ```

     (cf. fig. 19).
   - set "ExternalEventTrace" to

     ```
     root->p_VendingMachine->itsCoinValidator, C50; \
     root->p_VendingMachine->itsCoinValidator, E1; \
     root->p_VendingMachine->itsChoicePanel, WATER; \
     root->p_VendingMachine->itsChoicePanel, SOFT; \
     root->p_VendingMachine->itsChoicePanel, TEA; \
     ```

     to allow any sort of coins and choice requests (cf. fig. 20).
   - set "ExternalEventModus" to "ndet".

6. Make "VerifyDTP" the active configuration.

7. From the "Code" menu, select "Generate/Make/Run".

8. Confirm to create a new directory for the new Configuration.

Finally a timing diagram and an LSC viewer pop up and show as a timing diagram and an LSC a prefix of a run which leads to a system configuration in which the wanted property holds (cf. fig. 21).

Remember to close the viewers before the next verification task.

[12] Configurations can be copied by holding down the Ctrl-Key and dragging the configuration onto the Component node where it should be copied to.

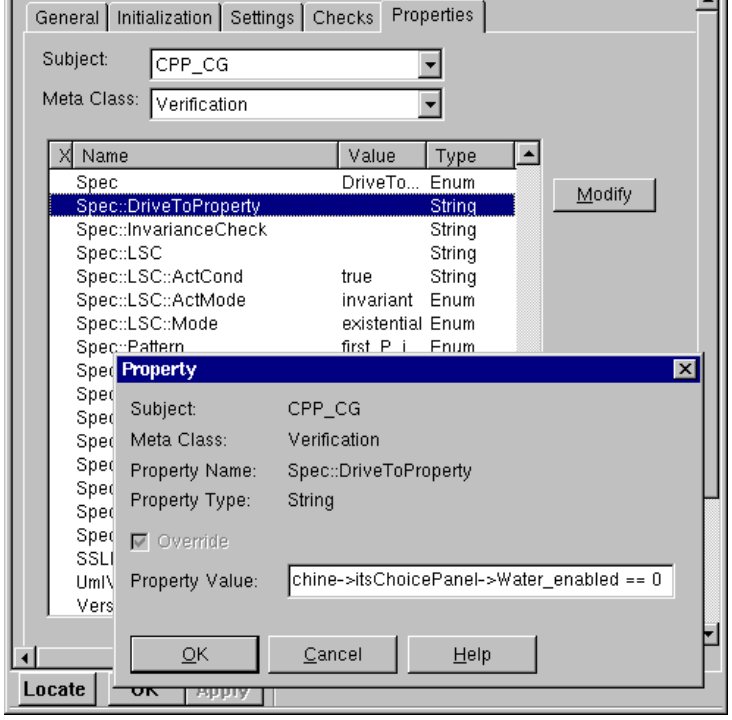


Figure 19: Denoting the property to drive to by a C++ expression.

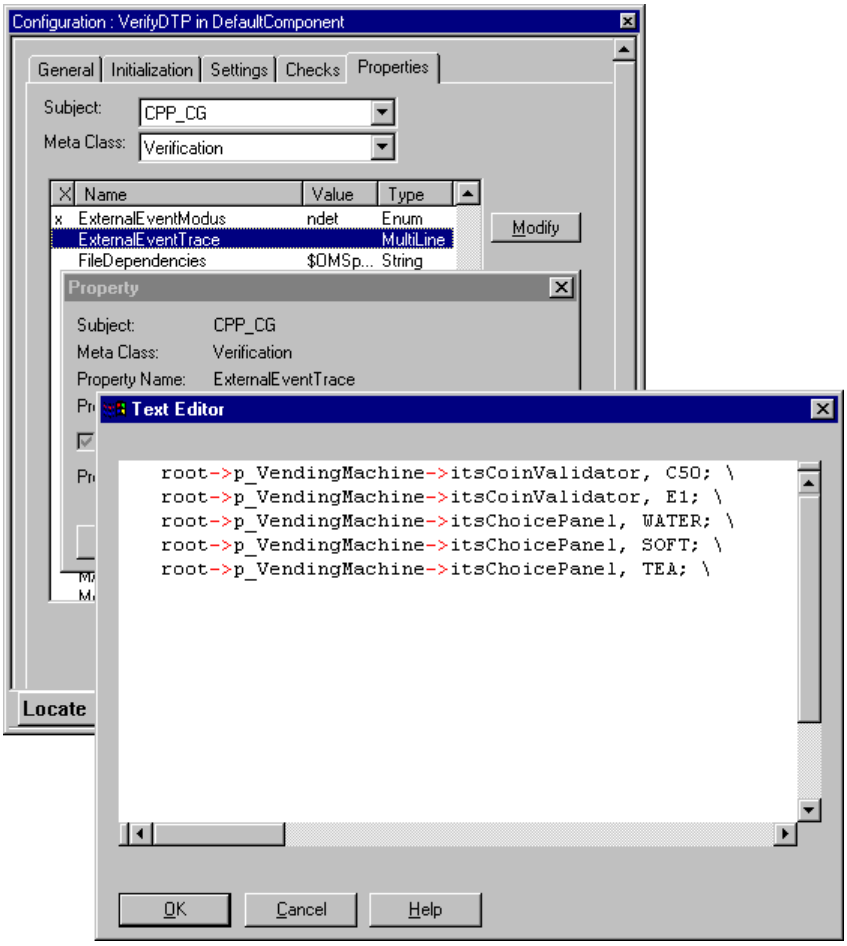


Figure 20: External events.

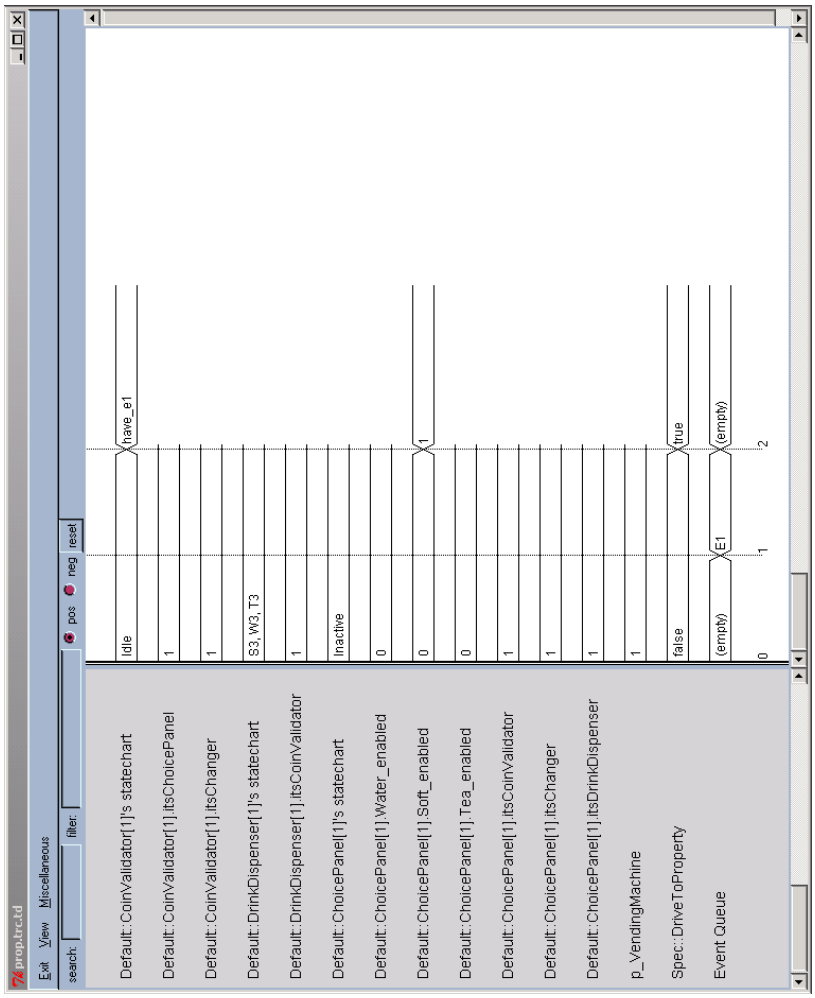


Figure 21: A trace leading to the wanted property configuration.

3.6 "Invariance Check"

1. Task:

   Check, whether the DrinkDispenser never runs out of drinks.
   (This is (unfortunately) not an invariant of the model).

2. Create a new configuration "VerifyCI" of component "DefaultComponent".

3. Right-click on "VerifyCI" in the browser and select "Features" to open the features dialog.

4. Set up the "Initialization"- and "Settings"-tab as in sec. 3.4.

5. In the "Properties"-tab, select "Subject" "CPP_CG" and "Meta Class" "Verification" and

   - for "Spec" choose "InvarianceCheck"
   - set "Spec::InvarianceCheck" to the C++ expression

     ```
     !(root->p_VendingMachine->itsDrinkDispenser->IS_IN(Water_out)
     &&
     root->p_VendingMachine->itsDrinkDispenser->IS_IN(Soft_out)
     &&
     root->p_VendingMachine->itsDrinkDispenser->IS_IN(Tea_out))
     ```

     (cf. fig. 22).
   - set "ExternalEventTrace" to

     ```
     root->p_VendingMachine->itsCoinValidator, C50; \
     root->p_VendingMachine->itsCoinValidator, E1; \
     root->p_VendingMachine->itsChoicePanel, WATER; \
     root->p_VendingMachine->itsChoicePanel, SOFT; \
     root->p_VendingMachine->itsChoicePanel, TEA; \
     ```

     to allow any sort of coins and choice requests (cf. fig. 23).
   - set "ExternalEventModus" to "ndet".

6. Make "VerifyCI" the active configuration.

7. From the "Code" menu, select "Generate/Make/Run".

8. Confirm to create a new directory for the new Configuration.

Finally a timing diagram and an LSC viewer pop up and show as a timing diagram and as an LSC a counterexample, i.e. a prefix of a run which leads to a system configuration where no more drinks are available. Since the resulting trace contains about 80 steps, fig. 24 only displays the start and the end sequence of it.

Remember to close the viewers before the next verification task.

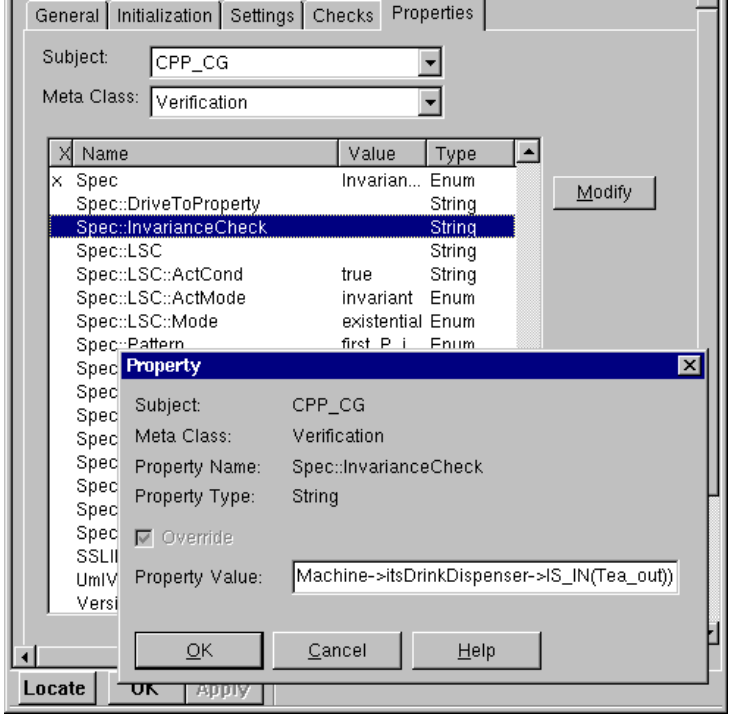


Figure 22: Denoting the property to check by a C++ expression.

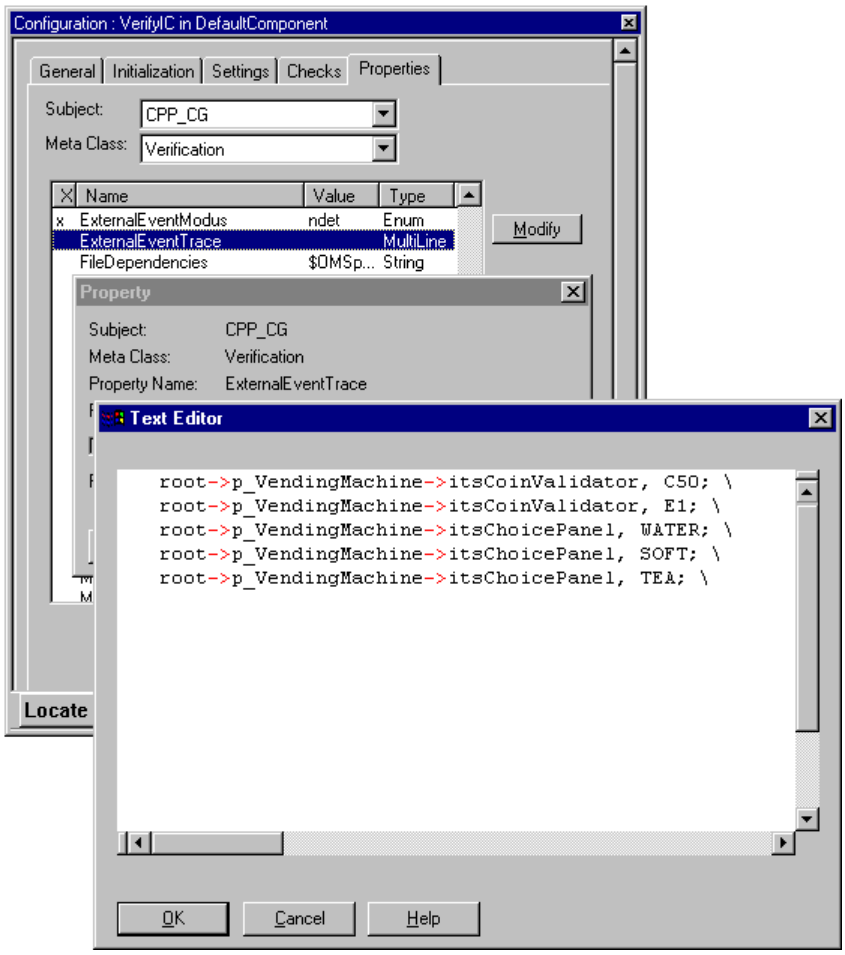


Figure 23: External events.

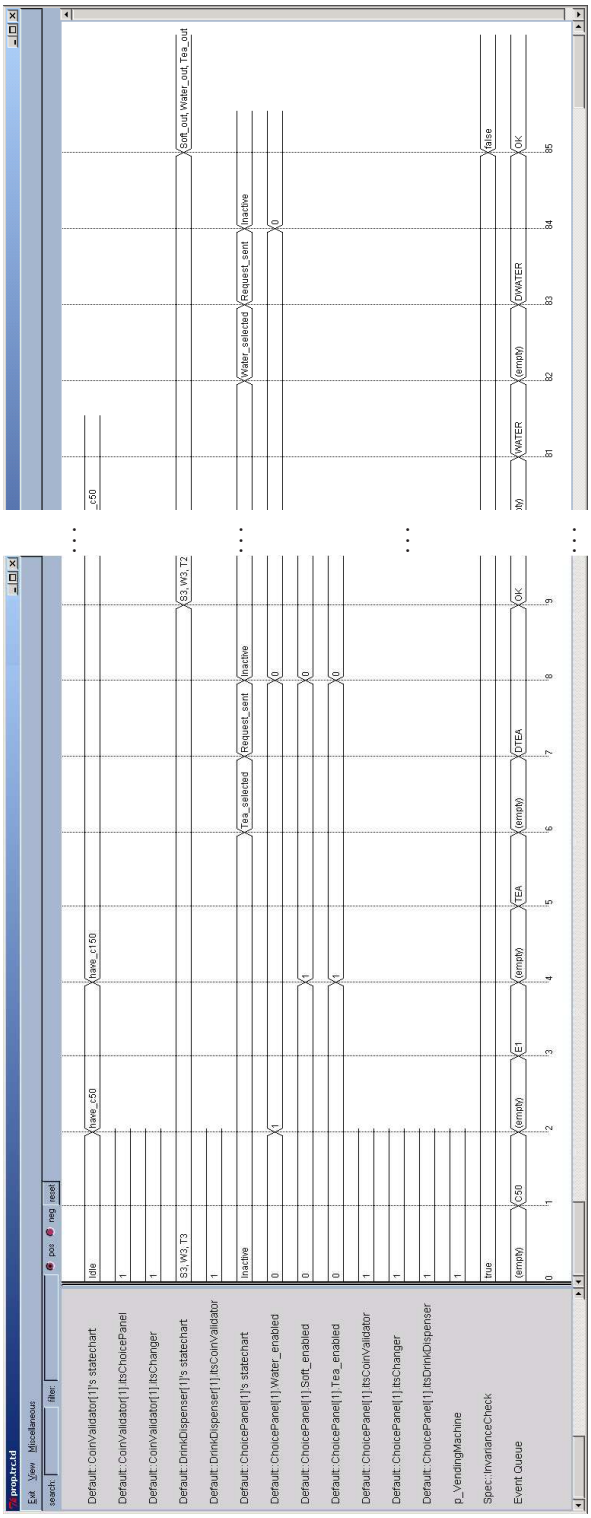


Figure 24: Parts of a trace leading to a configuration with no drinks left.

3.6.1 A true Invariant

An example of a true invariant is that having run out of water implies that water is not enabled on the ChoicePanel:

This invariant can be checked using the following expression for "Spec::InvarianceCheck":

```
!root->p_VendingMachine->itsDrinkDispenser->IS_IN(Water_out)
||
!root->p_VendingMachine->itsChoicePanel->Water_enabled
```

Recall that for a true invariant, no timing diagram and no LSC viewer pop up, but the log in Rhapsody's "Output Window" summarizes that the property holds.

3.7 The Pattern Library

The "drive-to-state", "drive-to-property" and "invariance-check" tasks are in fact instances of the specification pattern "inv_P__immediate" from the OFFIS Pattern Library [OSC01].[13]

Actually all patterns from the pattern library are accessible from the **ruve** by setting the corresponding Rhapsody properties. Property "Spec::Pattern" selects a pattern by name. For the verification task, the necessary pattern variables have to be bound. For every variable described in the pattern manual [OSC01], there is a Rhapsody property like "Spec::Pattern::P_Expr".

The pattern manual [OSC01] documents which pattern requires which variable and how this is already encoded in the pattern's names.

1. Task:

   > Check, whether it is true that whenever a 'C50' coin is received by the CoinValidator, then the attribute 'Water_enabled' of the ChoicePanel has a value of 1 one step later.
   > (This is not the case, since 'Water_enabled' is not set if there is no water in stock).

2. Create a new configuration "VerifyPAT" of component "DefaultComponent".

3. Right-click on "VerifyPAT" in the browser and select "Features" to open the features dialog.

4. Set up the "Initialization"- and "Settings"-tab as in sec. 3.4.

5. In the "Properties"-tab, select "Subject" "CPP_CG" and "Meta Class" "Verification" and

   - for "Spec" choose "PatternCheck"
   - for "Spec::Pattern" choose "inv_P_implies_finally_Q_B__immediate" (cf. [OSC01] for an in depth explanation of the corresponding temporal logic formula)
   - set "Spec::Pattern::max_X_Val" to 1 (for "one step later"), and set "Spec::Pattern::P_Expr" to the C++ expression

     ```
     ER_C50( ENV, root->p_VendingMachine->itsCoinValidator )
     ```

     (for "the CoinValidator receives a 'C50' event from the environment") and set "Spec::Pattern::Q_Expr" to the C++ expression

     ```
     root->p_VendingMachine->itsChoicePanel->Water_enabled
     ```

     (cf. fig. 25).
   - set "ExternalEventTrace" to

[13] For "drive-to-state" and "drive-to-property" tasks, the negation of the property "Spec::DriveToProperty" is used as P and for "invariance-check" tasks "Spec::InvarianceCheck" itself is used as P, since if $\neg P$ does not always hold, the model-checker generates as a counterexample a prefix of a run which leads to a system configuration in which $\neg P$ does not hold. But "not $\neg P$" is just P, i.e. the system configuration one wanted to drive to.
Consequently, if P never holds, then $\neg P$ is true and thus the model-checker does not generate an error path.

```
root->p_VendingMachine->itsCoinValidator, C50; \
root->p_VendingMachine->itsCoinValidator, E1; \
root->p_VendingMachine->itsChoicePanel, WATER; \
root->p_VendingMachine->itsChoicePanel, SOFT; \
root->p_VendingMachine->itsChoicePanel, TEA; \
root->p_VendingMachine->itsDrinkDispenser, FILLUP; \
```

to allow any sort of coins and choice requests and filling up the machine.

- set "ExternalEventModus" to "ndet".

6. Make "VerifyPAT" the active configuration.
7. From the "Code" menu, select "Generate/Make/Run".
8. Confirm to create a new directory for the new Configuration.

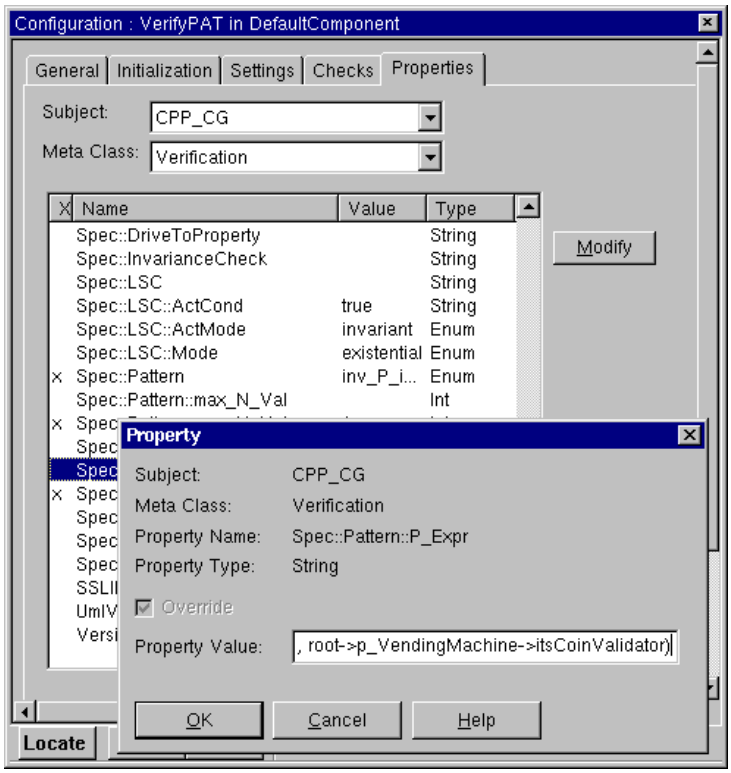


Figure 25: Denoting the properties of the pattern to check by C++ expressions.

Finally a timing diagram and an LSC viewer pop up and show as a timing diagram and as an LSC a counterexample, i.e. a prefix of a run where the claimed property does not hold. Since the resulting trace contains about 30 steps, fig. 26 only displays the start and the end sequence of it.

Since the **ruve** produces a counterexample, it claims that the property does not hold and in order to understand what "goes wrong" in the counterexample

it is usually not sufficient only to look at the last state but one needs to identify the points in time where parts of the specification hold or do not hold.

This is due to the fact that a "pattern-check" task usually refers to multiple points in time in contrast to "drive-to" tasks where the timing diagram and the LSC simply end at a configuration the system was supposed to drive to or "invariance-check" tasks where typically the last state in a counterexample is a state where the invariant does not hold. In the example, the specification refers to a point in time where P (= the CoinValidator receives a 'C50' event) holds and where Q (= the lamp for water is switched on) holds and states that whenever P holds, then at most one step later the property Q holds.

With patterns of the "implies" form, one typically reads the counterexample in both the timing diagram viewer and the LSC viewer backwards and expects that in the last state the conclusion does not hold while there exists an earlier state where the premise indeed holds, thus overall the implication is violated.

In the concrete example, considering the last state of the counterexample in the timing diagram – since Q is rather a "configuration" property which talks about variable values than an event communication property – shows that the water lamp is not enabled (but disabled since step 24, cf. ① in fig. 26), thus Q does not hold. Now reading the LSC backwards – since the P refers to event communication – shows that the CoinValidator has indeed received a 'C50' event before (cf. ② in fig. 27). In particular considering the LSC shows, that the event reception takes place in the last-but-one step, thus the last-but-one step is a point in time where P holds but one step later Q does not hold, thus the system does not satisfy the specification.

Reading the whole timing diagram (cf. ③ in fig. 26) one finds that the reason for this violation is simply that the vending machine runs out of water and thus of course does not enable the lamp for water when a 'C50' is inserted.

Remember to close the viewers before the next verification task.

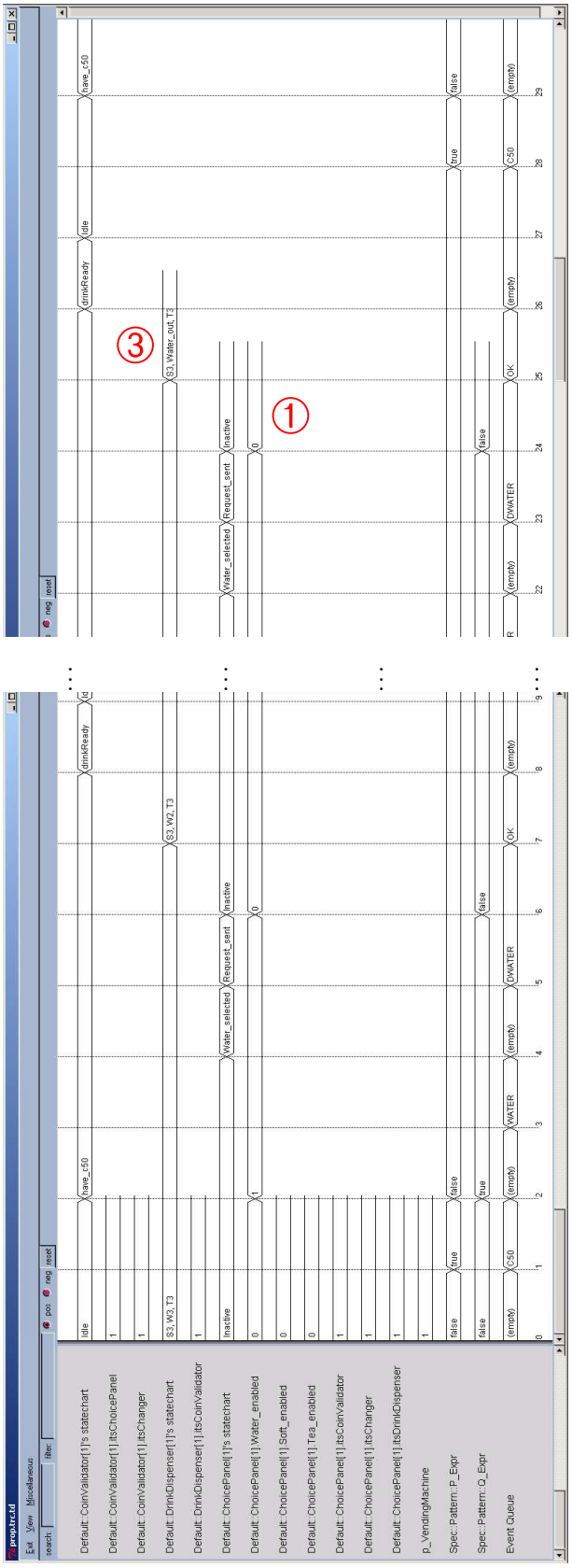


Figure 26: Parts of a trace showing that sending 'C50' does not cause 'Water_selected' to be set if the machine has no water in stock.

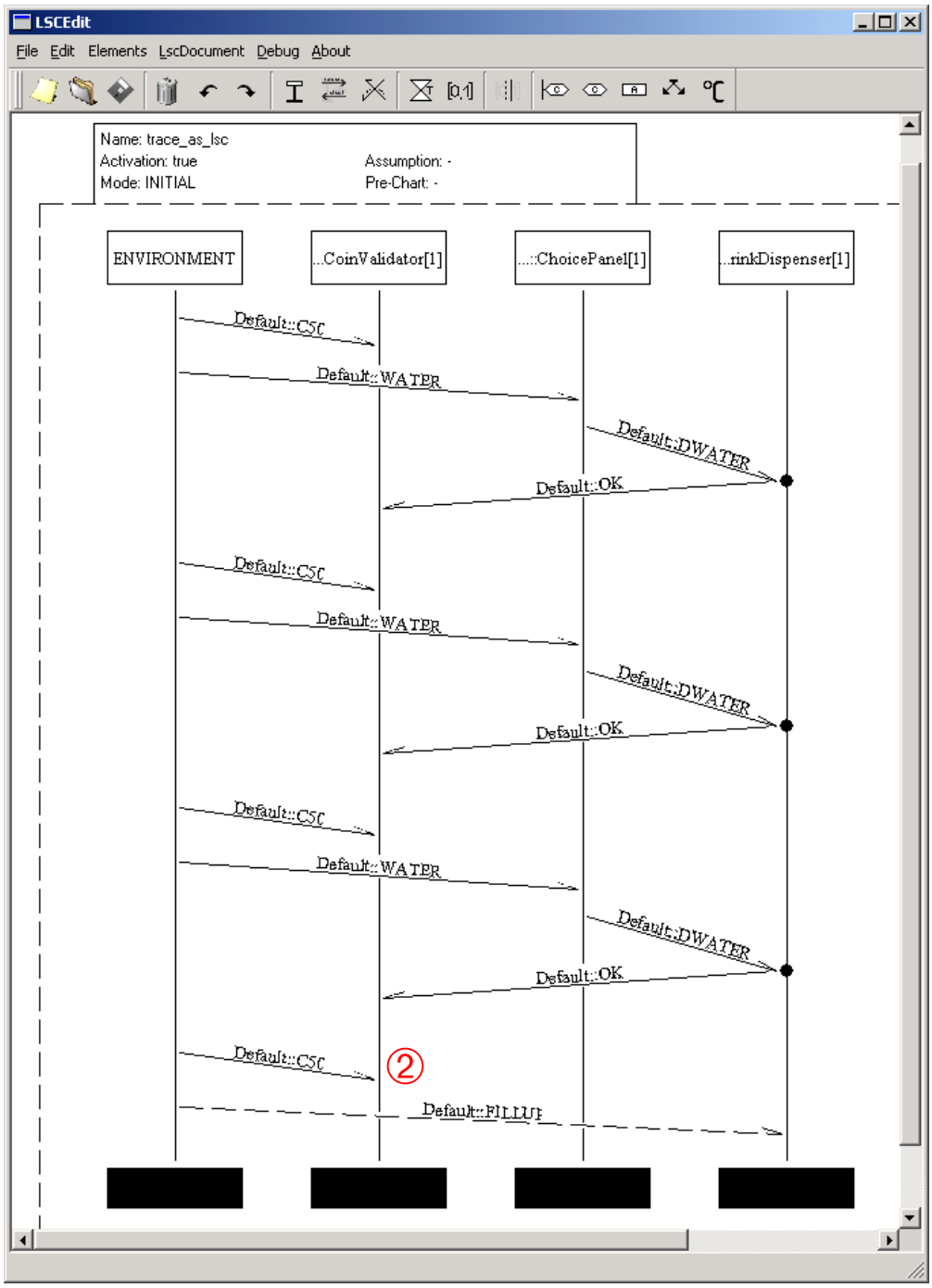


Figure 27: The sequence of events disproving the property shown as LSC.

3.8 Assumptions

Assumptions are used to restrict the environment of model. In that sense, the "ExternalEventTrace" property is already an assumption since it restricts the type of events sent to the model. But with the "ExternalEventTrace" alone is not possible to assume e.g. that the users of the VendingMachine insert 'E1' events only after inserting 'C50' before. Therefore the **ruve** also allows to describe assumptions by the patterns introduced in section 3.7.

1. Task:

   > Check, whether it is true that whenever a 'C50' coin is received by the CoinValidator, then the attribute 'Water_enabled' of the ChoicePanel has a value of 1 *ten* step later, *assuming* that a 'FILLUP' event is sent at most *ten* steps after one of the drinks runs out of stock.
   > (On first sight, one expects this to hold, since events are only sent from the environment if the machine is idle s.t. the 'FILLUP' always arrives just in time. But the outcome presented below shows that the property happens to *not* hold for the special case of the 'Water_enabled' flag and the granted times.)

2. Create a new configuration "VerifyPATas" of component "DefaultComponent".

3. Right-click on "VerifyPATas" in the browser and select "Features" to open the features dialog.

4. Set up the "Initialization"- and "Settings"-tab as in sec. 3.4.

5. In the "Properties"-tab, select "Subject" "CPP_CG" and "Meta Class" "Verification" and

   (a) set up the property to be verified just as in section 3.7, except that "max_X_Val" should now be 10 instead of 1, i.e.

   - for "Spec" choose "PatternCheck" (as in section 3.7)
   - for "Spec::Pattern" choose "inv_P_implies_finally_Q_B_immediate" (as in section 3.7)
   - set "Spec::Pattern::max_X_Val" to 10 (for "ten steps later", other than in section 3.7), and set "Spec::Pattern::P_Expr" to the C++ expression

     ```
     ER_C50( ENV, root->p_VendingMachine->itsCoinValidator )
     ```

     (for "the CoinValidator receives a 'C50' event from the environment", as in section 3.7), and set "Spec::Pattern::Q_Expr" to the C++ expression

     ```
     root->p_VendingMachine->itsChoicePanel->Water_enabled
     ```

     (as in section 3.7; cf. fig. 25).
   - set "ExternalEventTrace" as in section 3.7 to allow any sort of coins and choice requests and in particular filling up the machine.
   - set "ExternalEventModus" to "ndet".

(b) state the assumption, i.e.

- for "Assumption1::Pattern" choose "inv_P_implies_finally_Q_B_immediate"
- set "Assumption1::Pattern::max_X_Val" to 10 (for "ten steps later"), and set "Assumption1::Pattern::P_Expr" to the C++ expression

```
root->p_VendingMachine->itsDrinkDispenser->IS_IN(Water_out)
||
root->p_VendingMachine->itsDrinkDispenser->IS_IN(Soft_out)
||
root->p_VendingMachine->itsDrinkDispenser->IS_IN(Tea_out)
```

(for "some drink is out of stock"),
and set "Assumption1::Pattern::Q_Expr" to the C++ expression

```
ER_FILLUP(ENV, root->p_VendingMachine->itsDrinkDispenser)
```

(for "the DrinkDispenser receives a 'FILLUP' event from the environment").

6. Make "VerifyPATas" the active configuration.
7. From the "Code" menu, select "Generate/Make/Run".
8. Confirm to create a new directory for the new Configuration.

Finally a timing diagram and an LSC viewer pop up and show as a timing diagram and as an LSC a counterexample, i.e. a prefix of a run in which the vending machine is filled up within 10 steps, but where the last 'C50' does not cause the lamp for water to be switched on within 10 steps due to a bug in the design: after fillup, the water lamp is not enabled if the CoinValidator is in state 'have_e1', since then there would be no money to give back change. But this state is not only reachable by inserting an 'E1', but also by the sequence shown in trace, since 'have_e1' is the default state within 'have_c100_or_e1'.

Note that this counterexample is also a counterexample for the case *without* the assumption, but is not necessarily produced, since the model-checking algorithm usually finds the shortest and "simplest" path. In this case, the result is usually a trace where the machine simply runs out of water without filling it up.

Figure 28 displays only the end sequence of the trace starting from the state where the machine runs out of water, since the trace contains nearly 40 steps,

Remember to close the viewers before the next verification task.

The other assumptions, "Assumption2" and "Assumption3", are used analogously to the example. All assumptions with a "::Pattern" different from "none" are considered in *conjunction* and it is checked whether the conjunction of the assumptions *implies* "Spec".

Assumptions can also be used together with Drive-to tasks but not with temporal-logic.

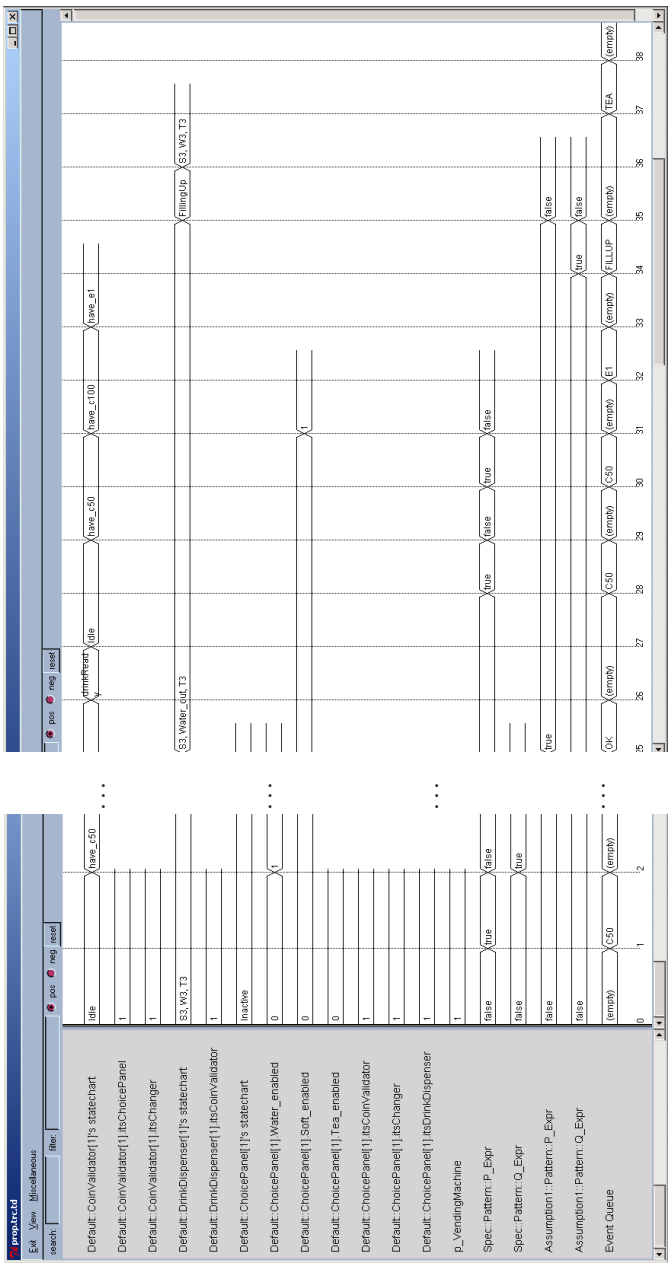


Figure 28: Parts of a trace showing that sending ‘C50’ does not cause ‘Water_selected’ to be set if the CoinValidator is in state ‘have_e1’ when the machine is filled up.

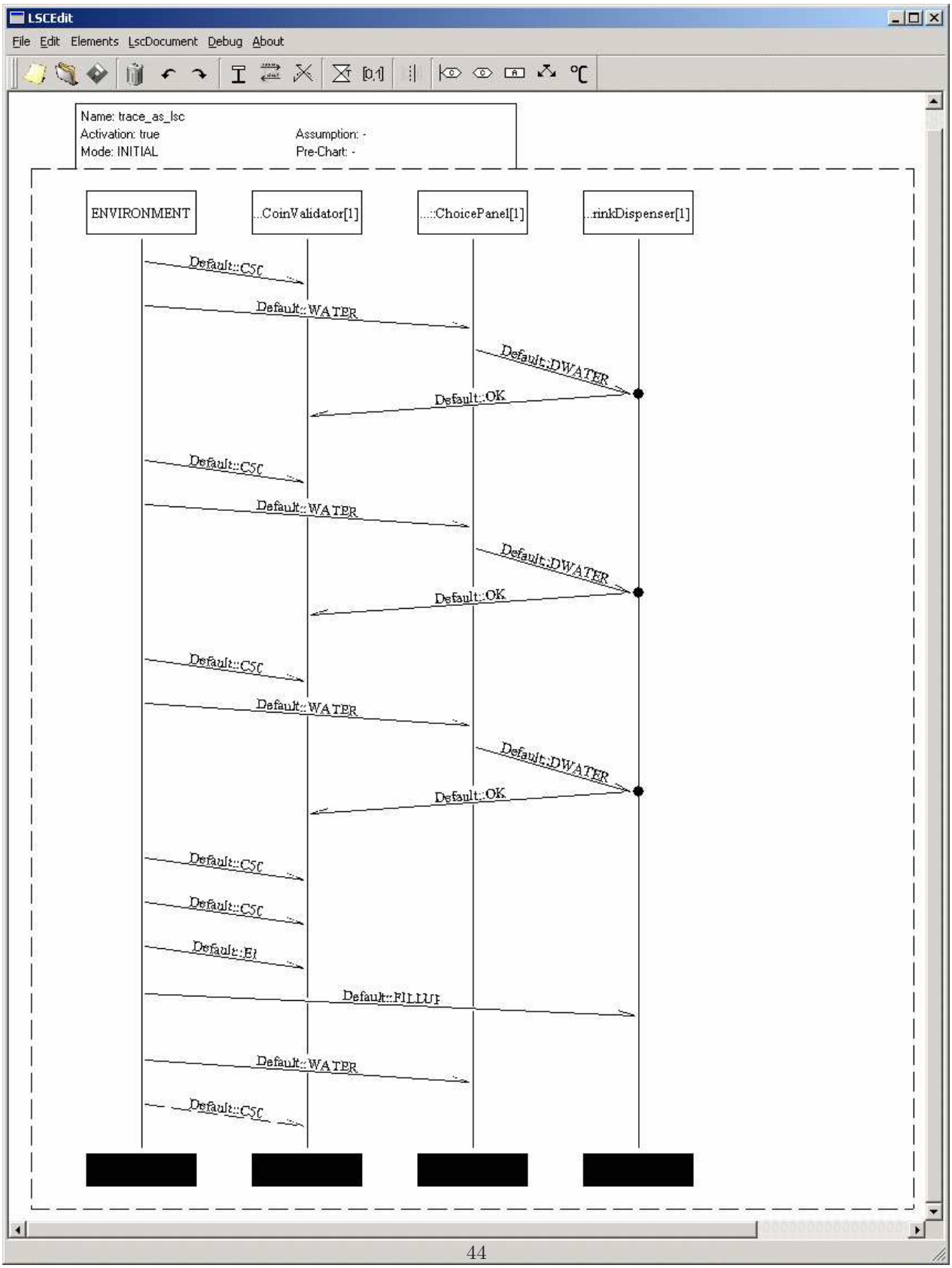


Figure 29: The sequence of events disproving the property shown as LSC.

3.9 Verify a LSC specification

The **ruve** is able to verify UML models against a specification in form of a "Life Sequence Chart" (LSC, [DH01]). Currently, the LSC has to be of the "static binding" kind, that is, all instance lines are bound to a concrete object of the model, which must not change during the run of the system. Future implementations will comprise dynamic bindings as described in [DW03, KW02]. For the semantics of LSCs we refer to [Klo03, DW03].

1. Task:

   > Check, whether it is true that whenever a customer wants to buy a water drink (thus, inserts at least one 'C50' coin, then possibly other coins, followed by requesting a 'WATER') and the vending machine is not out of water drinks, then a water is prepared and dispensed to the customer.

2. Create a new configuration "VerifyLSC" of component "DefaultComponent".

3. Right-click on "VerifyLSC" in the browser and select "Features" to open the features dialog.

4. Set up the "Initialization"- and "Settings"-tab as in sec. 3.4.

5. In the "Properties"-tab, select "Subject" "CPP_CG" and "Meta Class" "Verification" and

   - for "Spec" choose "LifeSequenceChart"
   - set "ExternalEventTrace" to

     ```
     root->p_VendingMachine->itsCoinValidator, C50; \
     root->p_VendingMachine->itsCoinValidator, E1; \
     root->p_VendingMachine->itsChoicePanel, WATER; \
     root->p_VendingMachine->itsChoicePanel, SOFT; \
     root->p_VendingMachine->itsChoicePanel, TEA; \
     root->p_VendingMachine->itsDrinkDispenser, FILLUP; \
     ```

     to allow any sort of coins and choice requests and filling up the machine.
   - set "ExternalEventModus" to "ndet".

6. Make "VerifyLSC" the active configuration.

7. From the "Code" menu, select "Generate/Make/Run".

8. Confirm to create a new directory for the new Configuration.

Right after starting the verification, the LSC editor "LSCEdit" pops up in order for the specification to be drawn. Thus in contrast to the other specification kinds, e.g. patterns, the specification is not entered and stored as a set of "Properties", but stored in two separate files which are edited using the LSCEdit.

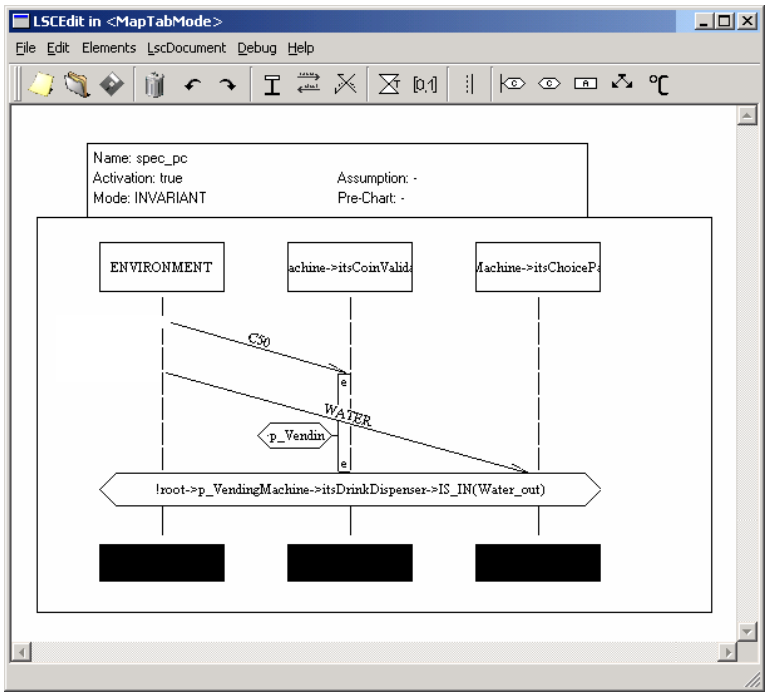


Figure 30: The pre-chart of the LSC. The local invariant reads: "`!(ER_WATER(ENV, root->p_VendingMachine->itsChoicePanel) || ER_SOFT(ENV, root->p_VendingMachine->itsChoicePanel) || ER_TEA(ENV, root->p_VendingMachine->itsChoicePanel))`"

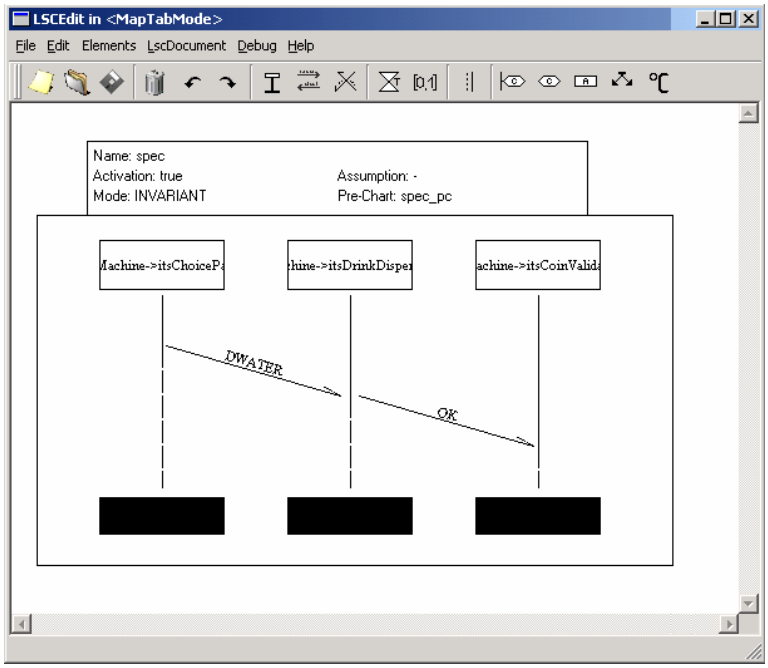


Figure 31: The main chart of the LSC specification.

In the following we briefly recall the intuition of an LSC corresponding to this section's verification task and then provide a step-by-step guide how the LSC is drawn using the LSCEdit.

The LSC for this task consists of a pre-chart (depicted in fig. 30) and a main chart (depicted in fig. 31). The specification reads as follows:

> *[pre-chart]* Whenever the machine receives a 'C50' followed by a 'WATER', and between these two events no other drink request are received (thus, the first C50 coin is indeed used to buy the water), and when receiving the 'WATER' the vending machine is not out of water drinks, then *[main chart]* finally a 'DWATER' event is produced internally, directly followed by an 'OK' event.

To create this specification in the LSCEdit the following steps are needed:

1. Set "Name" to "`spec_pc`" and "Activation" to "`true`", activate the "PRE-CHART" radio button, switch "Mode" to "Iterative", and click "Okay".
2. Choose "Elements → Add Instance", set "Name" to "`ENVIRONMENT`", and click "Okay".
3. Add two more instances with names
   "`root->p_VendingMachine->itsCoinValidator : CoinValidator`"
   and
   "`root->p_VendingMachine->itsChoicePanel : ChoicePanel`"[14].
4. Choose "Elements → Add AsyncMessage". You will see a couple of dots appearing on the instance lines to which we refer to as "locations".
5. Click on the second location of the "Environment" instance, then on the fourth location of the "CoinValidator" instance.
6. In the following dialog, set "Name" to "`C50`".
7. Add another async message, from the fourth location of "Environment" to the 8th location of the "ChoicePanel" instance, with "Name" set to "`WATER`".
8. Choose "Elements → Add condition" and click on the 8th location of "Environment", then on the 8th location of "CoinValidator", and then *right*-click on the 8th location of "ChoicePanel".
9. Set "Expression" to
   "`!root->p_VendingMachine->itsDrinkDispenser->IS_IN(Water_out)`"
   and "Condition" to "Hot", then click "Okay".
10. Choose "Elements → Add local invariant", then first click on the fourth location, and afterwards on the 8th location of the "CoinValidator".
11. Set "Expression" to
    "`!(ER_WATER(ENV, root->p_VendingMachine->itsChoicePanel)`
    `|| ER_SOFT(ENV, root->p_VendingMachine->itsChoicePanel)`
    `|| ER_TEA(ENV, root->p_VendingMachine->itsChoicePanel))`".

[14] an instance name must either be "`ENVIRONMENT`" or of the form "`<object-navigation-expression> : <class>`"

12. Set “Temperature” to “Hot” and both “Top” and “Bottom” to “Exclusive”, then click “Okay”.[15]

13. The pre-chart is now finished, so choose “File → New LSC” to start the main chart.

14. In the dialog, set “Name” to “`spec`”, “Activation” to “`true`” and “State” to “universal”.

15. Click on “Edit” next to the (currently empty) list of “pre-charts” and activate “spec_pc”

16. Close both dialogs by clicking “Okay”.

17. In the main chart, add three instances with names
“`root->p_VendingMachine->itsChoicePanel : ChoicePanel`”
“`root->p_VendingMachine->itsDrinkDispenser : DrinkDispenser`”
“`root->p_VendingMachine->itsCoinValidator : CoinValidator`”,
all of them with “Start Temperature” set to “Hot”.

18. Draw an async message “`DWATER`” from the second location of “ChoicePanel” to the fourth location of “DrinkDispenser”.

19. Draw an async message “`OK`” from the fourth location of “DrinkDispenser” to the sixth location of “CoinValidator”.

20. That’s it. Choose “File → Save LSCDocument” to save the LSC.

All elements of the LSC can be deleted, modified or moved within the LSC editor. Please see the online help (“Help → Mini Help”) for some explanation.

If you now close the editor, the verification will continue automatically. When you start the verification of the “VerifyLSC” configuration the next time, the saved LSC will be loaded and you can change the specification or simply close the editor without modifications.

Finally a timing diagram and an LSC viewer pop up and show as a timing diagram and as an LSC a counterexample. Since the resulting trace contains about 40 steps, fig. 32 only displays the start and the end sequence of it.

The **ruve** has detected that the specification does not hold, thus it shows a run of the system where the prechart is traversable, but the main chart is violated. In the resulting LSC (fig. 33) we see three times the same sequence of ‘C50’, ‘WATER’, ‘DWATER’, ‘OK’ events, thus the machine is out of water drinks now (i.e., in the corresponding timing diagram, the statechart configuration of the DrinkDispenser in step 25 reads “`S3, Water_out, T3`”). The following ‘C50’ event is the entry point of the pre-chart traversation. After a sequence of ‘C50’, ‘E1’, ‘FILLUP’ (which do not violate the local invariant within the pre-chart), the ‘WATER’ is the next event which has to be observed

[15] This local invariant needs some explanation: What we actually want to specify is that between the ‘C50’ and ‘WATER’ no other drink selections are sent. Since the sending point of external events is not visible in the expression language (cf. A.7.1), we move the local invariant to the “CoinValidator” and talk about the reception of external events. Alternatively (and more generic) we could add a shared condition “true” on locations number four of both the “CoinValidator” and “ChoicePanel”, and draw the local invariant between the dummy condition and the reception of ‘WATER’ (locations four to eight) of the “ChoicePanel”.

for fulfilling the pre-chart. On reception time of the 'WATER' event, the machine is not out of water (due to the previous 'FILLUP'), thus the pre-chart is completely traversed.

Now that the pre-chart has been observed, the trace LSC shows a violation of the liveness property of the main chart, i.e. an infinite sequence of 'WATER' requests[16], which *does not finally* produce the 'DWATER' and 'OK' events.

In order to see why the 'WATER' event of the pre-chart does not dispend a water, we take a look at the statechart of the CoinValidator (fig. 3(a)). The sequence of two 'C50' events leads to the state 'have_c100'. An 'E1' causes first the self loop of 'have_c100_or_e1' and then the default transition to 'have_e1' to be taken. Hence the machine still has calculated the right amount of money, but it has lost the information that it actually has received two C50 coins rather than one E1 coin.

The following 'FILLUP' invokes the 'update_choicePanel()' method which is responsible for activating the drink-enable-flags. But since the machine "thinks" it has only a single E1, the 'Water_enabled' is not set, because the machine would not be able to give back the right amount of change. The following 'WATER' events now do not trigger any drink dispensing since the corresponding guard is not evaluated to `true` in fig. 3(b).

Remember to close the viewers before the next verification task.

[16] which by accident emulate an "angry customer", impatiently hammering on the water button

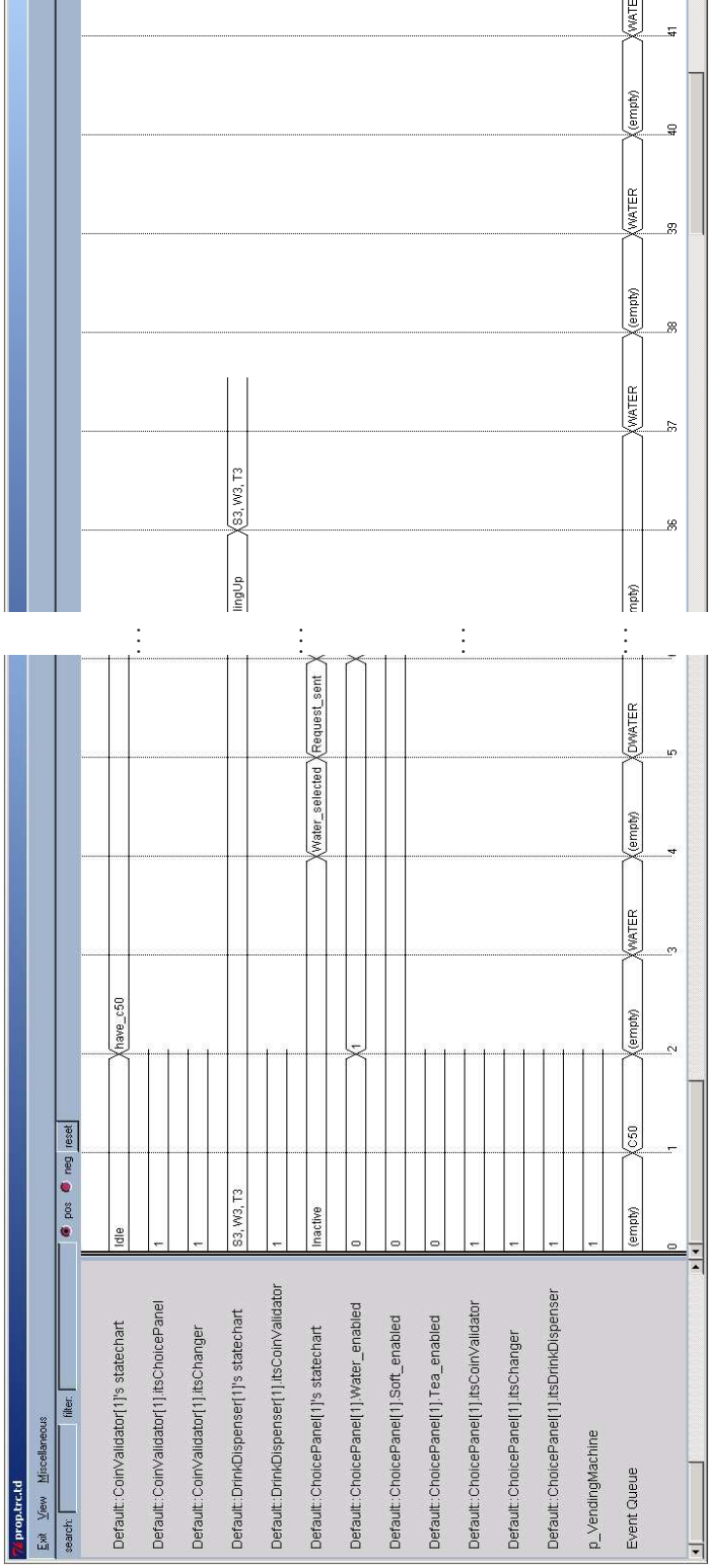


Figure 32: Parts of a trace disproving the specified property.

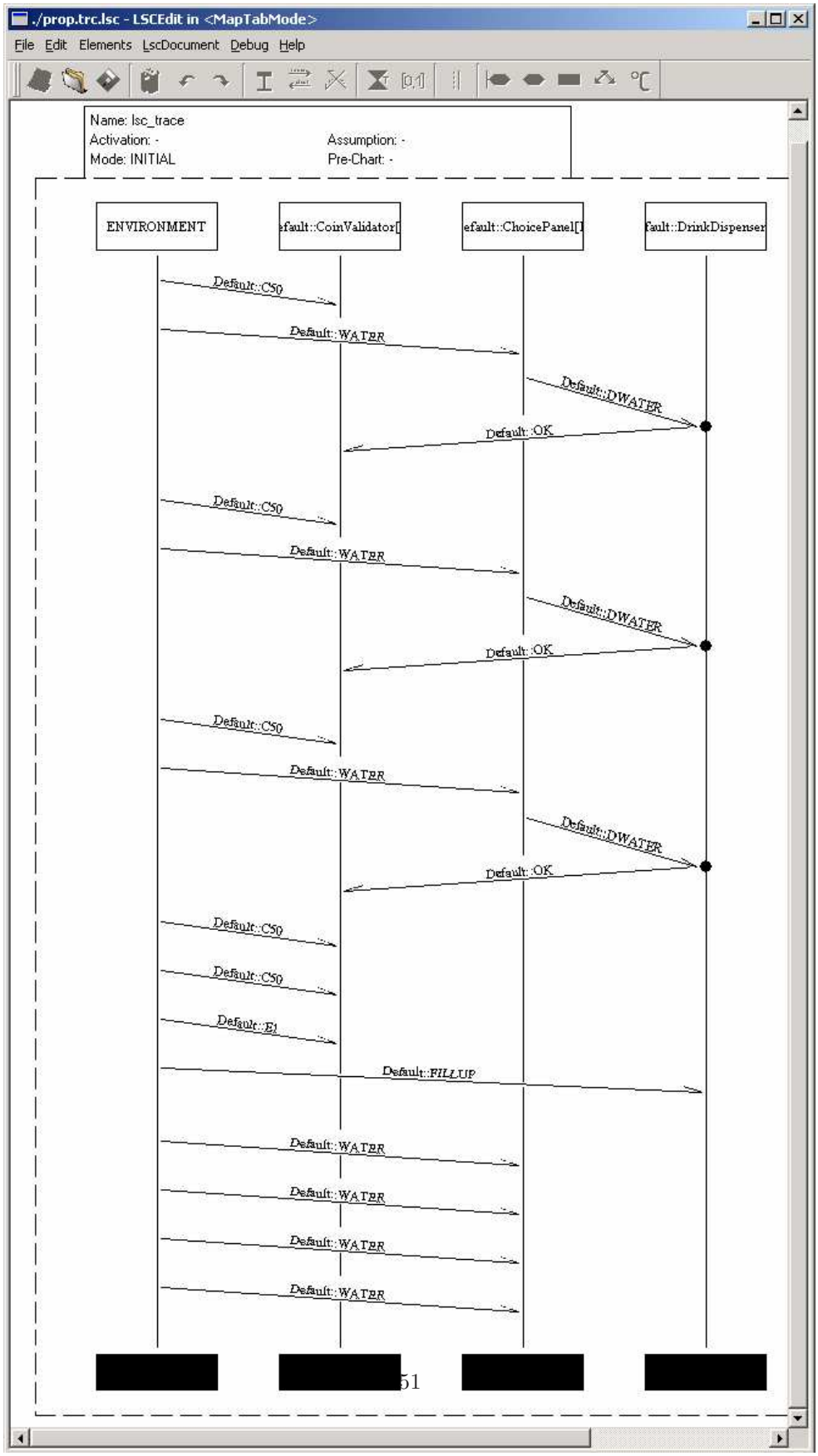


Figure 33: The sequence of events disproving the property shown as LSC.

3.9.1 LSCs in Omega LSC XML format

A Live Sequence Chart specification can not only be entered using the LSCEdit, but also imported if it is provided in the Omega LSC XML format [OFF04]. It is then first translated into the format of the LSCEdit and can be edited using the LSCEdit before the verification run starts.Once the XML file has been translated once and edited using the LSCEdit, it is only re-translated if it is changed later and thus newer than the files in LSCEdit format. The user is then prompted by the **ruve** whether the LSCEdit files should be kept or re-generated.

Using an XML file, for example `lsc.xml`, for LSC verification requires the following actions:

1. Create a new configuration and set it up for LSC verification just as outlined in sec. 3.9.

2. Additionally, in the "Properties"-tab, select "Subject" "CPP_CG" and "Meta Class" "Verification" and

   - change "Spec::LSC::Source" to "Omega_LSC_XML" and
   - in "Spec::LSC::Omega_LSC_XML_File" enter the name of the XML file, in the example `lsc.xml`.
     The filename may be given as an absolute path or relative to the configuration directory, i.e. if the value of

     "Spec::LSC::Omega_LSC_XML_File"

     is just `lsc.xml`, then this file is searched for at the same location where Rhapsody writes the generated code for the configuration to.

3. When starting the verification run by selecting from the "Code" menu the entry "Generate/Make/Run", first the XML is translated into the format of the LSCEdit and then an LSCEdit is started as with regular LSC verification described in sec. 3.9.

Appendix C provides a complete description of the relation between "Spec::LSC::Source" of "Omega_LSC_XML" and "LSCEdit" in terms of the affected files.

Note that, independent from the kind of verification task, there is also an Omega LSC XML representation of the counterexample generated whenever a counterexample exists.

3.10 Temporal Logic

For internal use only.

3.11 Checking Memory Bounds

The **ruve** has a builtin verification task called "CheckMemoryBounds" which can be found in the "Spec" list of the verification properties. After having performed the global settings as described in section 3.3 and defined an appropriate list of external events, this task can be started just like a "normal" verification task by invoking Generate/Make/Run.

The verification checks whether any "overflows of event queues" or any "out of memory errors" may occur in the model's behavior. If this is the case, an example trace shows the exact step and the class where such an error happened. Often these errors results from too optimistic settings for the event queue length or event quantities (cf. sec. 3.2.2).

The statechart depicted in fig. 34 will cause such a memory bound violation when running with an event queue of length 1. The timing diagram in figure 35 shows that generating a second event of type 'Ev' in the third step (when the first event is still in the queue) leads to both a "queue overflow" and an "out of memory" error.

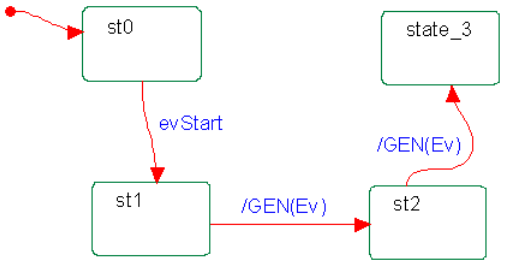


Figure 34: Erroneous statechart.

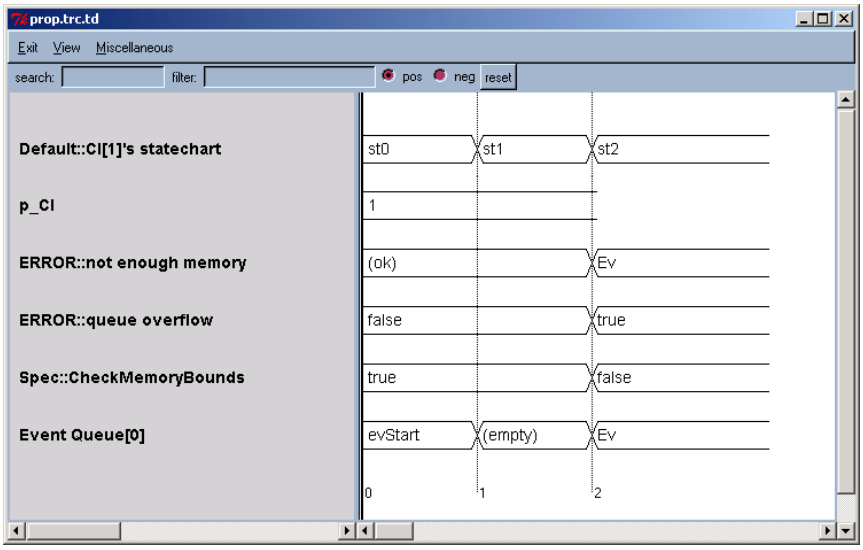


Figure 35: A trace showing memory bounds violations in step 3.

3.12 Performance Tuning

The examination of the complete state space and behavior relation of a dynamic model is a complex task, leading to time- and memory-consuming verification runs. The following guidelines may give some hints to speed up the verification:

- The event settings described in section 3.2.2 should be as aggressive as possible. Whenever it turns out to be necessary to increase the event queue length, one should always re-consider the quantity of event classes since the default quantity is the length of the event queue (verification property "MaxEventQuantities', cf. paragraph 3.2.2).
- To avoid the need for 'OMStartBehaviorEvents' you should try to design your statecharts such that they are stable after taking the default transition (cf. paragraph 3.2.2).
- **ruve** works with a bounded range for integer values. This range should be chosen as small as possibly (verification properties "IntegerLowerBound" and "IntegerUpperBound").
- Currently, the usage of triggered operations is expensive. Thus one implement the intended behavior using asynchronous event communication and primitive operations whenever possible.
- Setting the concurrency mode of classes to "active" obviously increases the number of possibles runs due to the different possibilities of scheduling. Keep in mind that there is *always* one active thread called "OMDefaultThread" started by Rhapsody – hence explicitly defining one active object per model is redundant.
- Declare class attributes local to methods if they are exclusively used there and need not be referenced in a specification.
- Arithmetic operations on attributes increase the behavioral space – this is especially true for multiplication and division.
- StrongAggregation (composition) should be preferred over WeakAggregation since the composition relation has a "constant character" in the world of a model-checker.
- Of course the overall "size" of the model (that is (among other things) the number of objects including events, number of attributes, size of the statechart) should be as small as possible.
- A critical point in a verification run is the translation of the model into a finite state machine and its representation as a BDD (binary decision diagram, [Bry86]). Crucial for efficient model-checking is a good ordering of the variables (which results in a "small" BDD). For finding a good ordering, several strategies were developed and implemented (see [Som98] for an overview). The default strategy of **ruve** is "Sift", which performs best in most of the cases. If the reordering takes an extraordinary amount of time, it is worthwhile to try out different strategies by setting the verification property "ReorderingStrategy" to the desired value. Experiments show that "Random", "SymmSift" and "Window4" perform very well in special cases.

- As described in section 1.2, contemporary hardware is recommended, i.e. at least 1 GByte of memory and at least 500 MHz processor-clock.

3.13 Remote Verification

If there is a *UMLVERIFROOT* installation on a remote host with `ssh`(1) service, then the verification task can be divided into

1. C++ code-generation and accessing the Rhapsody API locally,
2. copying the relevant files to the remote host,
3. execute the verification task on the remote host,
4. if a trace has been generated, copying the trace to the local host and starting the viewer locally.

This might be useful if the local machine is slow and there is a fast verification host at hand or the local machine should not be blocked by the verification task. Furthermore, the remote copies can be used to re-execute all existing verification tasks on the remote host in batch mode by copying the all new relevant files to the remote host, execute all remote `RUN.sh` from a script on the remote host, and copy back all generated traces.

1. Task:

   Execute the same task as "VerifyCI" remotely on host `vhost` as user `vuser` in directory `/tmp` (cf. sec. 3.6).

2. Create a new configuration "VerifyCIremote" of component "DefaultComponent".
3. Right-click on "VerifyCIremote" in the browser and select "Features" to open the features dialog.
4. Set up the "Initialization"- and "Settings"-tab as in sec. 3.4.
5. Right-click on "VerifyCIremote" in the browser and select "Features" to open the features dialog.
6. Set up everything as for "VerifyCI" in sec. 3.6.
7. In the "Properties"-tab, select "Subject" "CPP_CG" and "Meta Class" "Verification" and
   - set "RemoteHost" to "vhost", the name of the remote host,
   - set "RemoteUser" to "vuser", the name of the account on the remote host,
   - set "RemoteTmp" to "/tmp", the name of the directory below which the directory structure for relevant and temporary files should be established on the remote host (see below for an example),
   - set "RemoteUmlVerifRoot" to the path of the *UMLVERIFROOT* on `vhost`, e.g.

     `/opt/local/`⟨*DATE*⟩`-uve-release-`⟨*RELEASENR*⟩
   - set "RemoteGCC" to the name of the GNU C-compiler command, usually simply "gcc"

- set "RemoteGNUToolsPath" to the path to the GNU tools on the remote host followed by a colon (":"). If the GNU tools are in the `PATH` anyway on the remote host, then this property may be left empty.
- set "MAKETARGET" to "verify_remote".

Note that all these properties except for MAKETARGET are best set up globally (cf. sec. 3.3) if the setup for remote verification is not supposed to be the same for all verification tasks.

8. Make "VerifyCIremote" the active configuration.
9. From the "Code" menu, select "Generate/Make/Run".
10. Confirm to create a new directory for the new Configuration.
11. Confirm to run the newly created executable (which is in fact an `.bat` file).

Then an MS-DOS command window pops up and `ssh`(1) prompts for the password of `vuser` on `vhost` and

- sets up the directory

  `vhost:/tmp/vuser/TheVendingMachine/DefaultComponent/VerifyCIremote/`

- copies the relevant files to the newly created remote directory,
- executes the verification command `RUN.sh` on the remote host, and
- if the run results in a trace, copies it back to the local host.

If a trace has been generated, a timing diagram and an LSC viewer pop up locally as usual.

3.14 "Reviewing" and Saving Traces

All temporary files of the verification run, and in particular the generated traces, are written into the target directory of the configuration, i.e. the same place where the generated C++ code goes to.

Once the timing diagram and LSC viewers have been closed, the trace can be viewed by the following steps:

1. In the cygwin-`bash`, change to the directory of the configuration, e.g.

   `cd` *UMLVERIFROOT*`/examples/TheVendingMachine/DefaultComponent/VerifyDTS`

2. There are two possibilities to make the cygwin-`bash` aware of the location of the helper tools needed by the timing diagram viewer. Either set the environment variable

   `export` *UMLVERIFBIN*`=`*UMLVERIFROOT*`/bin/cygwin`

   or extend your PATH variable

   `export` *PATH*`=`*UMLVERIFROOT*`/bin/cygwin:$PATH`

   (These settings can be made permanent by including them into the personal `.bashrc`.)

3. Then simply call

   UMLVERIFROOT`/bin/cygwin/reviewtrc.sh`

   which displays the latest automatically generated trace in the current directory as timing diagram and LSC.

To save traces, change to the directory of the configuration and copy the files

```
prop.trc.td
prop.trc.map
prop.trc.lsc
prop.trc.xml
```

to another directory.[17] The trace and LSC can be viewed by calling

UMLVERIFROOT`/bin/cygwin/reviewtrc.sh` *THE_COPY*

from the cygwin-`bash` if *UMLVERIFBIN* or *PATH* have been set as in step 2 above. Here *THE_COPY* refers to the full path to the newly created copy of the `.td` file.

[17] `prop.trc.xml` is the Omega LSC XML [OFF04] representation of the trace.

A Supported Rhapsody

A.1 Components and Configurations

The "unit of verification" is a "Configuration" of a "Component" of a "Project", thus "Components" for verification

- should not contain anything other than a "Configuration" (thus no "Folders" or "Files" etc.), and in particular
- must not contain nested "Components".

A.2 Packages and Namespaces

- The system to be verified must be contained in a single package.
- A class must not have the same name as the project.
- Namespaces must not be used explicitly in the user code.

A.3 Object Model Diagram

An Object Model Diagram may contain:

- Classes,
- Class hierarchies with single inheritance,
- Directed or undirected associations between classes of a fixed multiplicity N,
- Compositions and Aggregations with a fixed (part-)multiplicity N.

Note that although the aggregation relation has no special semantics, its multiplicity serves as an indication on how many objects are created *at most*. This is similar to the composition relation – with the difference that these parts are created when the compound object is created and destructed when the compound object is destructed.

Thus, both relations together determine a maximum amount of internal memory to represent runtime objects. Simple association relations do not influence this computation, they only provide the possibility to store "existing" objects at their link ends.

A.4 Statecharts

A Rhapsody Statechart may contain:

- Basic states, OR-states, AND-states.
- Default states.
- Transitions (according to sec. A.6).
- Condition connectors.
- History connectors.

- Termination connectors.
- Simple fork and join connectors, i.e. transitions which split to reach multiple destinations in different compartments of an AND-state or transitions which originate at multiple sources in different compartments of an AND-state (*not* junction connector).
- Diagram connectors and sub-machines.

States may have entry-actions and exit actions.

A.5 Event-Queue

At the moment, an event queue with a fixed length N is supported (see section 3.2.2). The behavior in case of a queue overflow is undefined, i.e. the outcome may be anything from "the property holds" to arbitrary traces. If a trace was produced, a special waveform called `ERROR::queue overflow` will indicate the exact step where the overflow occurred. Unfourtunatly, if the overflow happens in the initialization step of the system (i.e. when the default transition of the starting objects are taken), the overflow error is not visible in the trace (since the init step is not visible).

A.6 Transitions Annotation

The annotations of transitions have Rhapsody's usual form of:

Trigger `[` *Guard* `]` `/` *Statementlist*

Trigger is the name of an event class, but not a time event ("`tm()`"). *Guard* is an expression according to A.7 of type `int`.[18] Both *Trigger* and *Guard* (together with the square brackets) may be missing.

A.7 Expressions

An *Expression* can be one of:

- a decimal literal,
- a variable or constant name,
- an attribute access of the form

  Attributename or *Variable*`->`*Attributename*

  where *Variable* is of pointer-to-object type,
  To access attributes of other objects, the get/set methods should be used (see below).
- a call of a primitive or triggered operation (but see section A.11 below for restrictions) of the form

  Methodname or *Variable*`->`*Methodname*

[18] the expression `1 == 2` is considered to be of type `int`

where *Variable* is of pointer-to-object type,
The methods generated by Rhapsody to get/set attributes can be used.

- a state query of the form

  `IS_IN(`*State*`)` or *Variable*`->IS_IN(`*State*`)`,

  where *State* is the name of a state as shown in the Statechart in Rhapsody and where *Variable* is of pointer-to-object type,

- an application of a unary operator of the form

  op Expression,

  if *Expression* is of type `int` and *op* is one of:

  `!`, `-`

- an application of a binary operator of the form

  Expression op Expression,

  if both *Expressions* are of type `int` and *op* is one of:

  `||`, `&&`,
  `+`, `-`, `*`, `/`,
  `<`, `<=`, `==`, `!=`, `>=`, `>`

- a conditional expression of the form

  Expression `?` $Expression_1$ `:` $Expression_2$

  where *Expression* is of type `int` and $Expression_i$ are both of the same pointer-to-object type or of type `int`.

Note that (unlike for C++) we consider as expressions only those terms which do not have side-effects and that expressions cannot be used on the left hand side of an assignment.

Overflows in expressions should be avoided. Since for model-checking the range of integers is significantly reduced compared to 32- or 64bit-values on real targets, the behavior of the model in case of overflows may not match the behavior of execution on a real target (cf. sec. A.9).

A.7.1 Property Expressions

A *property expression* is an expression used as the value of a Rhapsody property which take C++ expressions as specifications, for example "Spec::DriveToProperty".

Property expressions must not contain a C++ conditional expression (`(?:)`) or a unary minus[19] since they are not supported for property expressions.

In addition to plain C++ expressions (excluding conditional expression and unary minus), property expressions may contain

[19] workaround: write `(0-a)` instead of `-a`

- *event send/receive queries* of the form

  (ES_|ER_)⟨Event⟩((⟨Sender⟩, ⟨Receiver⟩[, ⟨Parameterlist⟩])

  where ⟨Event⟩ is the name of an event class, and ⟨Sender⟩ and ⟨Destination⟩ are navigation expression starting at 'root' (cf. sec. 3.4), or ENV denoting the sender and receiver to be observed. ⟨Parameterlist⟩ is a (possibly empty) comma-separated list of actual event parameter values to be observed. This list must exactly match the number and the order of the formal parameters of event ⟨Event⟩.

  Event send queries start with ES_, *event receive queries* start with ER_. It is not possible to observe the *sending* point of an external event, only the point in time of *reception* – thus, ER_⟨Event⟩(ENV, ⟨Receiver⟩) is ok while ES_⟨Event⟩(ENV, ⟨Receiver⟩) is forbidden.

- *state changed queries* of the form

  ⟨Object⟩->STATE_CHANGED_(TO|FROM)(⟨statename⟩)

  where ⟨Object⟩ is a navigation expression to a reactive object starting at 'root' and ⟨statename⟩ is a name of a state of ⟨Object⟩'s statechart. For instance

  ```
  root->p_C->STATE_CHANGED_TO(st)
  ```

  becomes true if the object 'p_C' has entered state 'st' and was at some state *different from* 'st' in the previous step, and

  ```
  root->p_C->STATE_CHANGED_FROM(st)
  ```

  becomes true if the object 'p_C' was at state 'st' in the previous step and now has entered some state *different from* 'st'.

  Note that these queries do not talk about entering or exiting states in full consequence, since they do not become true when taking self-loops (that is why we call them state *change* queries).

A.8 Statements

A *Statement* is one of:

- an expression according to sec. A.7
- an assignment of the form *Variable* = *Expression*, an assignment of the form

  Variable op Expression,

  where *op* is one of:

  =, +=, -=, *=, /=

- a pre- or postincrement of the form

  op Variable or *Variable op*

  where *op* is one of:

  `++`, `--`

- a sequential composition *Statement* `;` *Statement*,
- a block `{` *Statement* `}`,
- a conditional statement `if` *Expression Statement* `else` *Statement*,
- an event sending

  `GEN(`*Event Parameterlist*`)` or *Variable*`->GEN(`*Event Parameterlist*`)`,

  where *Event* is the name of an event class, *Parameterlist* is a possibly empty list of parameters matching the signature of a constructor of *Event* and where *Variable* is of pointer-to-object type.

Switch statements and any kinds of loops are not supported yet.

A.9 Data Types

Constants, attributes, local variables, method parameters and return values can have a type of:

- `int`,
- pointer-to-object, e.g. `VendingMachine*`.

Methods (but not triggered operations) can have return type `void`.

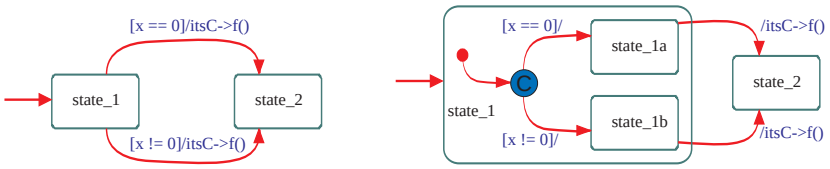
Type `bool` is not supported because Rhapsody does not offer it in the choice list of predefined types on which the **ruve** depends.

Only functions can have local variables and the local variables must be declared in the outermost block of the function, thus for example the `if`-block of a conditional statement cannot have its own local variables.

Local variables must not have the same name as a parameter (no "shadowing").

Structured types (other than classes) are not supported yet.

The value range of the `int` type can be controlled by the properties "CPP_CG"/"Verification"/"IntegerLowerBound" and "IntegerUpperBound". Default is the range $-64 \ldots 63$ which could cause the model-checking task to run out of memory (or time) if there are many attributes of type `int` in the model. The range can then be reduced to a smaller range which is still sensible wrt. to the model. The right bound must not be smaller or equal than the left bound.



(a) Unsupported Statechart

(b) Transformed Statechart

Figure 36: For each state, the triggered operation calls within all action parts on outgoing transitions of this state shall be to triggered operations with different names.

A.10 Methods

Methods can have parameters, return values, and local variables of types according to sec. A.9 and as body a statementlist according to sec. A.8. Overloading[20] is not yet supported.

A.11 Triggered Operations

A triggered operation may have parameters and return values of types according to sec. A.9, but the return value may not be of type `void`.

Calls of triggered operations may occur anywhere in the action part of a transition within a statechart, but not within a primitive operation.

Any number of outgoing transitions or static reactions may contain triggered operation calls in their action part and there may be any number of triggered operation calls within a single action part (including entry- and exit-action) as long as they all have different names.

For example, the statechart depicted in figure 36(a) is not supported since triggered operation 'f()' is called on both outgoing transitions. Cases of this type can usually be worked around by inserting additional states like, for example, the statecharts in figure 36(b) is supported and even keeps the property that `IS_IN(state_1)` evaluates to `true` until 'state_2' is reached.

Triggered operations are internally mapped to event communication. For each triggered operation of each class, an event class called

⟨triggered operation name⟩_⟨class name⟩_Event

is introduced to the model. This event is able to carry actual event parameters of the triggered operation as well as the reply value.

A call of the operation is then realized by sending an event of this class, whereby the event is put *in front* of the callee's event queue. After the callee has consumed this event, the execution of the 'reply()' statement will set the reply value, and will sent the event back to the caller.

This mechanism allows to specify triggered operations as part of the requirement in terms of asynchronous event communication. Currently, it is not possible to specify the reply value, only the actual parameters are visible.

[20] multiple methods with the same name but different signatures

Figure 37: statechart of classes C and D

As an example, consider a simple model with two classes `C` and `D`, and a triggered operation 'TO()' with two formal parameters $p1$ and $p2$. The statecharts of `C` and `D` are shown in figure 37.

A LSC specification which requires that the triggered operation 'TO()' finally occurs is displayed in figure 38. Note that

- the event reception of the "call event" and the sending of the "reply event" has to be specified *simultaneously*, since the 'reply()' statement is executed in the action part of the trigger reception in fig 37(b),
- both the "call event" and the "reply event" have to carry the actual parameters of the call,
- even if the reply value is not directly visible, it can of course be stored in a class variable which then can be used in the specification.

The witness-trace and -lsc are also talking about the newly introduced event class. For your convenience, the resulting LSC is patched such that it shows the actual parameters in the "call event", but indeed the reply value in the "reply event" (cf. fig. 39).

A.12 The "gray zone"

Note that appendix A presents the features of Rhapsody and C++ which are "known to work" whereas the document [OFF03a] lists the features of Rhapsody and C++ which are "known not to work".

Both documents are not complementary, i.e. there is a "gray zone" in between. Entering the "gray zone" is on own risk, since the **ruve** *as a research prototype is not always able to recognize unsupported features and may produce completely unpredictable output.*

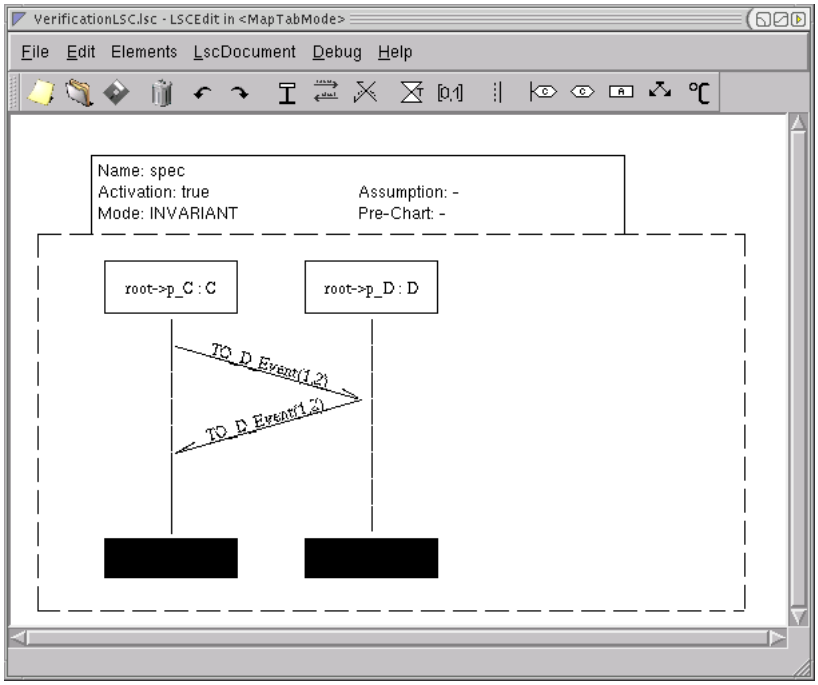


Figure 38: An LSC specifying the occurrence of 'TO()'

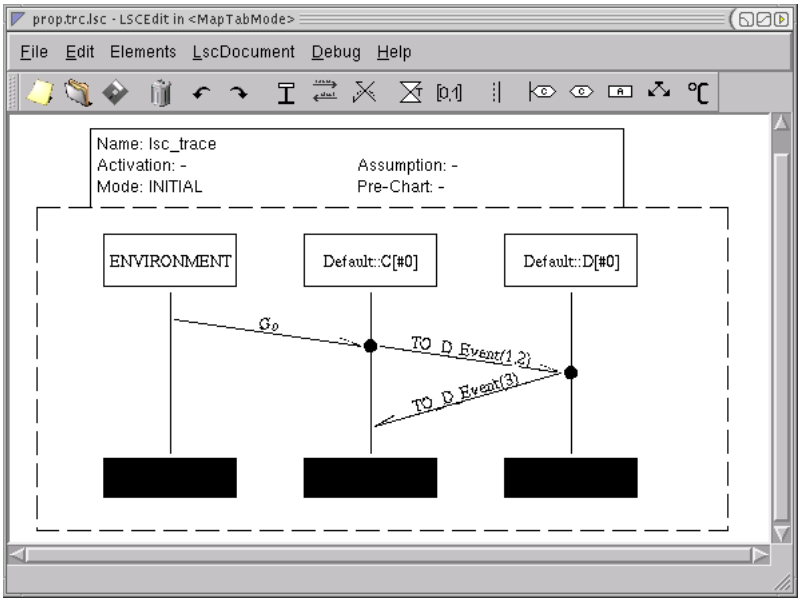


Figure 39: A witness LSC showing the occurrence of 'TO()'

B Troubleshooting

***Symptom*:** "Generate/Make/Run" immediately fails, saying something like "missing separator".

***Diagnosis*:** A trailing backslash in one of the external event lines in property "ExternalEventTrace" is missing (see section 3.4).

***Symptom*:** "Generate/Make/Run" finishes immediately, saying something about "Code Generation done. 0 Error(s), 0 Warning(s), 0 Message(s)".

***Diagnosis*:** Possibly you forgot the close the trace viewers which popped up after the previous run of **ruve**.

***Symptom*:** Verification stops immediately and complains about missing files.

***Diagnosis*:** Make sure that CPP_CG::Verification::MAKETARGET is set to 'verify' in order to invoke the complete verification run (ie. model-generation, model-checking, and trace-generation). 'verify_only' is only possible if you have already performed a verification on this configuration with flag CPP_CG::Verification::KeepIntermediateFiles set to 'true'.

***Symptom*:** Model-generation fails, the last output talks about "rhap2ssl".

***Diagnosis*:** Possibly the used version of Rhapsody is not an officially supported one (see section 1.1).

***Symptom*:** Model-generation fails, the last output talks about a "C++ frontend-error" which is concretized earlier in the output as "OMNotifier::notifyToOutput("Internal error – no last state in history"')".

***Diagnosis*:** One of your history connectors seems to have no outgoing transition (which serves as a default transition when no history information is available). It is also necessary that all sub-composite states within a composite state containing a history connector contain also a history connector, that determine a default transition when no history information is available.

***Symptom*:** Model-generation fails, the end of the output talks about "Begin Compile merged" and at the end of the output a warning and an error are reported, the warning with message "Cant resolve the Designator \rootState_subState\ in Context ENTITY WORK.*PACKAGE*::*CLASS*\" and the error with message "Argument >SSLNamePtr name< is a NULL Pointer" and *PACKAGE* is equal to *CLASS* in the first message.

***Diagnosis*:** Possibly the design does not adhere to the recommendations of section A.2, but uses the same name for both a package and a class within the package. Renaming either the package or the class should then solve the issue.

***Symptom*:** The outcome of **ruve** is completely unexpected, e.g. the trace seems not to be related to the property at all or **ruve** says, the property holds, although it is expected to be false.

***Diagnosis*:** Possibly **ruve** erronously re-uses files from a previous run. Try to rename (or remove) the configuration's directory by hand and re-start the

ruve.

If the task uses assumptions, check whether the assumption does not evaluate to *false*, since then every property holds. This can be checked by setting the Specification to *false* (i.e. set "Spec" to "InvarianceCheck" and "Spec::InvarianceCheck" to "false") – if **ruve** still reports that the property holds, then the assumption is suspicious.

To be continued...

C Technicalities

LSC source LSCEdit vs. Omega LSC XML

This section describes the mechanism behind the values "LSCEdit" and "Omega_LSC_XML" of property "Spec::LSC::Source" (cf. sec. 3.9).

An LSC specification consists of two files,

- a so called *LSC file*, which contains the structural information, i.e. instance lines, messages, conditions, etc., and
- a so called *Maptab file*, which contains a mapping table, that provides the mapping, like the C++ expression for a condition or the event annotation of messages.

For the configuration "VerifyLSC" of section 3.9, these files are expected in the same directory where the directory `VerifyLSC`, into which Rhapsody generates the C++ code of the configuration, is located under the names:

- `VerifyLSC.lsc` and
- `VerifyLSC.map`.

In the example of section 3.9 this is

- `.../TheVendingMachine/DefaultComponent/VerifyLSC.lsc` resp.
- `.../TheVendingMachine/DefaultComponent/VerifyLSC.map`.

When the "Spec::LSC::Source" (cf. sec. 3.9.1) is set to "LSCEdit", then the LSCEdit operates on these files, thus it will in particular store the drawn LSC directly at that location.

When the property "Spec::LSC::Source" is set to "Omega_LSC_XML", then the translation of the file given in "Spec::LSC::Omega_LSC_XML_File" is also stored into these two files and then the LSCEdit is run on them for final adjustments of the specification.

This "adjusted" specification is kept as long as the XML source does not change; when the XML file is newer then the two files, then the user is prompted whether to keep the files or to re-translate the XML. If the XML should not be re-translated, the prompt can be suppressed by setting "Spec::LSC::Source" to "LSCEdit", and if the result of the translation should not be opened with LSCEdit on each verification run, running the LSCEdit can be suppressed by *additionally* setting property "Spec::LSC::runLSCEdit" to "False".[21]

[21] property "Spec::LSC::runLSCEdit" has no effect when "Spec::LSC::Source" is set to "Omega_LSC_XML"!

References

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[DH01] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, July 2001.

[DW03] Werner Damm and Bernd Westphal. Live and Let Die: LSC-based Verification of UML-Models. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures*, number 2852 in LNCS. Springer-Verlag, 2003.

[i-L02] i-Logix. *Rhapsody User Guide*, 2002. Release 4.0.

[Klo03] Jochen Klose. *Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior*. PhD thesis, Universit"at Oldenburg, 2003.

[KW02] J. Klose and B. Westphal. Relating LSC Specifications to UML Models. In Hartmut Ehrig and Martin Grosse-Rhode, editors, *Proceedings INT2002- International Workshop on Integration of Specification Techniques for Applications in Engineering*, April 2002.

[OFF03a] OFFIS. The Rhapsody UML Verification Environment - Restrictions, June 2003.

[OFF03b] OFFIS. The XMI UML Verification Environment - User Documentation, June 2003.

[OFF04] OFFIS. (deliverable(s) on xml for lsc exchange), January 2004.

[OSC01] OSC, i-Logix. Statemate Magnum Certifier Pattern Library User Guide, 1.3, 2001. http://www.ilogix.com.

[Som98] F. Somenzi. *CUDD: CU Decision Diagram Package Release*, 1998. http://www.async.ece.utah.edu/∼myers/ee5740_98/cudd/cudd.ps.

Index

OLDENBURGER FORSCHUNGS- UND ENTWICKLUNGSINSTITUT
FÜR INFORMATIK-WERKZEUGE UND -SYSTEME

The Rhapsody UML Verification Environment

Restrictions

Revision : 1.40

To apply the Rhapsody UML Verification Environment on Rhapsody models, the user has to take care of restrictions described in this paper. The document is structured as follows: The first section contains general assumptions about UML-models to be supported and of which some are already implied by the more specialized restrictions. The sections 2 to 7 describe restrictions on the following levels: *Design*, *Specification*, *Errorpath*, *UML*, *Rhapsody* and *C++*. Section 8 contains a list of potentially problematic constructs and finally section 9 is a list of possible C++ translation errors (if an error is the result of an unmet restriction, then there is a reference to the proper location in the document).

The restrictions mentioned in this paper are also valid for the XMI Verification Environment. You will find additional, XMI-specific restrictions in [3].

1. Assumptions

(1) UML-models are of the "task type", i.e.
   (a) a task comprises a single active object and a set of passive objects; every object belongs to exactly one task and during lifetime it does not change the task it belongs to
   (b) the active object of a task has an own thread-of-control, a queue and does event handling for its reactive objects (objects which have a statechart); it can call methods of itself and its passive objects and may have a statechart of its own
   (c) inter-task communication is restricted to event communication

(2) UML-models are bounded over the lifetime, in particular the lifetime of the model can be separated into three phases:
   (a) Finitely many non-event objects are created in the initialisation phase (statecharts of this objects must take the initial transition). In Rhapsody, this corresponds to the entries made in the "Initialization"-tab of a "configuration". For all of this objects, its components (strong aggregates) and recursivly all sub-components are also created at this time (the multiplicity of strong-aggregation relations must be bounded). No other objects may be created at initialisation time.

   (b) During runtime, (weak) aggregates of objects may be created or destructed. Together with the creation or destruction of an aggregate, all its components and recursivly all sub-components are created resp. destructed. The multiplicity of weak-aggregation relations must be bounded.
   (c) The destruction phase is possibly never reached, if the model does not terminate (in this case the initially created objects don't terminate).

2. Design

(1) *Annotation Language* Only (a subset of) C++ is allowed as annotation language (In the XMI Verification Environment it is also possible to use the action language defined for the Omega kernel language).

(2) *Expression Overflow* Overflow must not occur within expressions (due to limited, machine-dependent integer-representation), e.g.

```
return ((0xffffffff + 1) == 0) ?  1 :  0;
```

yields 1 if compiled for a 32bit machine but the corresponding expression yields 0 in (the intermediate language) SMI

(3) *Initial Transition* Initial transitions within statecharts have neither guards nor triggers

(4) *Libraries* External libraries (i.e. libraries not defined by the user) must not be used, including the standard C library and the standard template library, but excluding the Rhapsody Framework (see (6.3)). Import from other Rhapsody projects is not supported.

(5) *Queue* Designs which need a queue length greater than a predefined n are not supported.

(6) *Recursion* No recursion is allowed, in the sense that for every object o and every method f (where different implementations in a class hierarchy are considered different methods) in every call chain f is called for o at most once. This excludes direct recursion, where a method calls itself, and indirect recursion where another method g calls f for o where it has been originally called from.

3. Specification

(1) *Liveness* Liveness properties are not supported.

(2) *Methods* Methods from the model cannot be used in the C++ expression used as specification for drive-to-property or check-invariance, except for `state_IN()` methods of states, which are the basis of `IS_IN` macro. In addition, the `abs()`-function (which is part of the Verification Framework) may be used to calculate the absolute value of an integer.

(3) *Multiplicity Constraints* Multiplicities on associations could be interpreted as constraints, if one writes the right temporal logic formula and runs invariance check (no support to generate this formula yet).
Automatic "on the fly" checking that *all* associations keep their multiplicity constraint is not supported.

(4) *Object Naming* Object naming is restricted to fully qualified names starting with `root`. Navigation expression over associations or weak aggregations which link ends are not initialized in the first step are not supported. Free variables are not supported.

(5) *Specification Language* The specification is an instance of a synchronous pattern (including drive-to-state, drive-to-property, check-invariance) or a CTL formula.

(6) *Time* Neither time annotations in an LSC nor worst-case-timing analysis based on annotations in the model are supported.

4. Errorpath

(1) *Simulations* There's no translation from the model-checker's counter-examples to a Rhapsody simulation of the error.

5. UML

(1) *Actors* Actors are not supported.

(2) *Deferred Events* The concept of deferred events is not supported.

(3) *Delays* OXFDelay is not supported.

(4) *Inheritance* Multiple Inheritance is not supported yet.

(5) *Multiplicity* Associations, aggregations and compositions with unfixed multiplicities are not supported.

(6) *Multiplicity* If a class D is inherited from a class C, then D must not be a strong aggregate or a component of C.

(7) *Statecharts* `doActions` within a state are not supported.

(8) *State names* Statenames must be unique for a class, that is states in different orthogonal regions or within different submachine states or across composite states must be named unique within the whole 'root' statemachine of a class.

(9) *Time Events* Time events are not supported, i.e. a model must not contain a `tm()` transition.

(10) *Scopes* Hierarchical scopes are not supported, in particular:
  (a) nested packages
  (b) nested classes
  (c) class-declarations in operations

(11) *Stereotypes* Stereotyped relations are ignored.

6. Rhapsody

(1) *Components* Exactly one component declaration and exactly one configuration for the whole model is taken into account for the verification(cf. [2]).

(2) *Framework Implementation* Exploiting knowledge about the framework implementation, e.g. accessing attributes of OMReactive directly, is not supported (cf. (6.3))

(3) *Framework Interface* The supported interface of the Rhapsody framework OXF comprises: `GEN`, `REPLY` (but cf. 6.7), `IS_IN`

(4) *Instances* No usage of explicit instantiations of classes is allowed

(5) *Projectname* The name of the project must not be the name of a class.

(6) *Run-To-Completion Steps* The behavior of Rhapsody for infinite Run-To-Completion Steps is not supported.

(7) *Triggered Operations* Calling a triggered operation is supported in transition actions, not in primitive operations.

(8) *Version* Only Rhapsody Version 4.2 is supported

7. C++

(1) *asm-Statements* Asm statements are not supported.

(2) *Call by reference* Call by reference is not supported.

(3) *Casts* Explicit casts in the user code are not supported. Note that special forms of casts are possible when using the workaround for enumeration-types mentioned in [1].

(4) *Comma Operators* The comma expression operation is not supported.

(5) *Constants* Constants of the simple types mentioned in 7.25 are supported, but no constant attributes, and const-declaration of methods.

(6) *Conversions* Conversion operator functions for classes are not supported.

(7) *Ellipsis* Functions with ellipsis are not supported.

(8) *Exceptions* Exceptions are not supported, i.e. no `throw`, no `try`, no `catch`.

(9) *Expression Statements* Expressions as statements are only allowed, if the topmost expression node is an assignment or a functioncall.

(10) *Goto* Goto and label statements are not supported (Nevertheless, forward jumps should work, but are not well tested).

(11) *Loops* Loops in general are not supported (no `while`, `do-while` and `for`). Nevertheless, a special form of a `for`-loop may be used, see [1].

(12) *Modulo Operators* The modulo expression operation is supported, but a modulo assign operator is not supported.

(13) *Namespaces* Different name spaces for structs and other types not supported, e.g. in C++.

```
int A = 0; struct A  ...  ;
```

is possible; the A's are different and to separate them, a use of the latter A has to be prefixed by `struct`.

(14) *Namespaces* The `using`-statement is not supported.

(15) *Overloading* Flat operators like `operator+` for structs are not supported.

(16) *Plain Operators* Plain (i.e. not member function) operators are restricted to integral types. Bitwise operators (`|`, `&`, `^`, `~`, `|=`, `^=`, `&=`) and shift operators (`<<`, `>>`, `<<=`, `>>=`) are not supported.

(17) *Pointers* The operator `*` as (pointer-)type constructor (except for pointer types to classes) or as dereferencing operator is not supported. The operator `&` is not supported as address-of operator. In particular function-pointers, call-by-reference (except for objects) and pointer-to-member are not supported.

(18) *Pointer Arithmetics* Pointer arithmetics (like adding 1 to a pointer) are not supported. Pointers can only be assigned and compared for (in)equality.

(19) *References* The operator `&` to construct reference types is not supported.

(20) *sizeof() Operator* The `sizeof()` operator is not supported.

(21) *C Static* Static local and global variables are not supported.

(22) *C++ Static* Static attributes or methods are not supported.

(23) *Switch* The *fall through* C++-semantic of switch-statements is not supported (That is: We only support switch statements, if the last statement of a switch-case is a break)

(24) *Templates* The declaration and/or usage of template classes is not supported.

(25) *Types* In general: Only the simple integer kind is allowed (i.e. no explicit short/long or signed/unsigned integers, no char, no bool), no floats, no typedefs, no enums, no structs, no unions, no arrays (cf. [2]). The range of the integer type is determined by the values of the configuration properties

`CPP_CG.Verification.IntegerLowerBound`

and

`CPP_CG.Verification.IntegerUpperBound`

The previous restriction is only excepted by the two workarounds for special array- and enumeration-types described in [1].

(26) *Type Definitions* Type definitions within function bodies are not supported.

(27) *Variables* Attributes or local variables of class type are not supported, i.e. only pointers ("identities") of them are supported.

(28) *Virtual Base Classes* Virtual base classes are not supported.

8. Modeling Guidelines

(1) avoid side-effects in guards, i.e. don't call functions which modify attributes and don't use assignment-expressions or increment-operators
(this is actually required by the UML 1.4: "Guards should not include expressions causing side effects. Models that violate this are considered ill formed." (p. 2-165))

(2) event parameters can only be used at the action of the first transition in the run-to-completion step triggered by the event (other access is only possible by a "dirty hack" and this is not supported (see above))

(3) don't modify attributes that occur in conditions inside the statechart by primitive operations (since otherwise you might observe the situation that the object is in stable state although there is a transition which seems to be enabled (since the condition now holds due to modifications by the primitive operation) and that it remains in stable state until the next event is dispatched (this is UML semantics: the enabled-ness of a transition is checked only if there is a current event, i.e. during event dispatching))

9. C++ Translation Errors

Many restrictions mentioned in the previous sections are checked when the C++ code is translated by the tool `edg2ssl`. This section contains a list of all possible errors and its errorcode in the backend-part of `edg2ssl` (on the other hand the front-end part checks the syntactical correctness of the C++ code like a conventional compiler). Some of the errors below are not the result of a restriction (and therefore off topic in the context of this document).

(1) Dynamic initializations are not supported yet (*discretionary error*)

(2) Using declarations are not supported yet (*discretionary error*) [page 4 (7.14)]

(3) Local static variable initializations are not supported yet (*discretionary error*)

(6) Not enough memory (*catastrophe*)

(8) An elaborated type specifier is needed (not supported yet) (*error*) [page 4 (7.13)]

(9) There exists an unknown kind of a hidden name (*error*)

(11) A variable or field has an unsupported type (e.g. a class type) (*error*) [page 5 (7.27)]

(12) A Switch-clause does not end with a break (*discretionary error*) [page 5 (7.23)]

(13) Type declarations within functions are not supported yet (*error*) [page 5 (7.26)]

(14) Something unexpected happened (*internal error*)

(15) Qualification needed (*error*)

(16) Unsupported address base kind (*discretionary error*)

(17) Unsupported constant representation kind (*discretionary error*)

(18) Unsupported dynamic init kind (*discretionary error*)

(19) Unsupported constructor init kind (*discretionary error*)

(21) Unsupported special function kind (*discretionary error*)

(23) Unsupported initialization kind (*discretionary error*)

(24) Unsupported type kind (*discretionary error*)

(25) Unsupported statement kind (*discretionary error*)

(27) Unsupported expression operation kind (*discretionary error*)

(28) Unsupported expression node kind (*discretionary error*)

(29) A constant pointer to a routine (*discretionary error*) [page 4 (7.17)]

(30) A constant pointer to a variable (*discretionary error*) [page 4 (7.17)]

(31) A constant pointer to a _GUID structure for Microsoft __uuidof operation (*discretionary error*) [page 4 (7.17)]

(32) A constant pointer to a label (used for the GNU address-of-label extension) (*discretionary error*) [page 4 (7.17)]

(33) A constant of kind pointer-to-member (data or function) (*discretionary error*) [page 4 (7.17)]

(34) A constant of kind dynamic_init (*discretionary error*)

(35) A List of constants in initialization (*discretionary error*)

(36) A repeated initialization constant in an array (*discretionary error*)

(37) A nontype parameter in a class template declaration (*discretionary error*) [page 5 (7.24)]

(38) The change of the current object in an aggregate initializer (C99) (*discretionary error*)

(39) Initialization to zero (defined to be the same as default initialization of a static object) (*discretionary error*)

(40) Initial value of a simple object is established by a call of a routine that returns a class object via a copy constructor (*discretionary error*)

(41) Initial value of a nonconstant aggregate object (array or class) is represented by a list of constant entries (some of which will refer to nonconstants) (*discretionary error*)

(42) Initial value is established by a bitwise copy (*discretionary error*)

(43) Object to be initialized is a virtual base class (*discretionary error*) [page 5 (7.28)]

(44) A conversion operator function (*discretionary error*) [page 4 (7.6)]

(45) Initialization to zero (static or dynamic) (*discretionary error*)

(46) Either dynamic or aggregate-constant initialization of a local static variable. The variable itself does not point at the initializer; rather the initialization is represented by a local static variable init entry (*discretionary error*)

(47) Routine type (*discretionary error*) [page 5 (7.25)]

(49) Struct type (*discretionary error*) [page 5 (7.25)]

(50) Union type (*discretionary error*) [page 5 (7.25)]

(51) Pointer to member types (*discretionary error*) [page 5 (7.25)]

(52) Type parameter in a (class or function) template declaration (*discretionary error*) [page 5 (7.24)]

(53) Goto-statement (*discretionary error*) [page 4 (7.10)]

(54) Label-statement (*discretionary error*) [page 4 (7.10)]

(55) Asm-statement (*discretionary error*) [page 4 (7.1)]

(56) Try-block-statement (*discretionary error*) [page 4 (7.8)]

(57) Statement to set the size of a variable length array type (*discretionary error*)

(58) Declaration of a variable or typedef with variably modified type (*discretionary error*)

(59) A statement, which marks where a variable with a variable length array type should be deallocated (*discretionary error*)

(60) Cast of a pointer to a class to a pointer to a direct derived class (*discretionary error*) [page 4 (7.3)], [page 4 (7.3)]

(61) Cast of a pointer to a member of a class to a pointer to a member of a direct base class (*discretionary error*) [page 4 (7.3)]

(62) Cast of a pointer to a member of a class to a pointer to a member of a direct derived class (*discretionary error*) [page 4 (7.3)]

(63) Cast of an lvalue in pcc mode (*discretionary error*) [page 4 (7.3)]

(64) Dynamic Cast (*discretionary error*) [page 4 (7.3)]

(65) Static Cast (*discretionary error*) [page 4 (7.3)]

(66) Generic const cast (*discretionary error*) [page 4 (7.3)]

(67) Generic reinterpret cast (*discretionary error*) [page 4 (7.3)]

(69) Floating negation (*discretionary error*) [page 5 (7.25)]

(71) Integer bitwise complement ("~" operator) (*discretionary error*) [page 4 (7.16)]

(74) Floating pre increment (*discretionary error*) [page 5 (7.25)]

(75) Floating pre decrement (*discretionary error*) [page 5 (7.25)]

(76) Pointer pre increment (*discretionary error*) [page 4 (7.18)]

(77) Pointer pre decrement (*discretionary error*) [page 4 (7.18)]

(78) An expression placed above an expression that is a struct rvalue, produces the address of the struct. This is used in implementing subscripting of rvalue arrays in C mode (an extension to ANSI/ISO C) (*discretionary error*) [page 5 (7.25)]

(79) "%" operator (*no error*)

(81) Pointer difference (*discretionary error*) [page 4 (7.18)]

(82) Pointer to member equality (*discretionary error*) [page 4 (7.17)]

(83) Pointer to member inequality (*discretionary error*) [page 4 (7.17)]

(84) Structure assignment (*discretionary error*) [page 5 (7.25)]

(86) Pointer to member assignment (*discretionary error*) [page 4 (7.17)]

(91) Remainder assign operator (*discretionary error*) [page 4 (7.12)]

(92) Floating add assign operator (*discretionary error*) [page 5 (7.25)]

(93) Floating subtract assign operator (*discretionary error*) [page 5 (7.25)]

(94) Floating multiply assign operator (*discretionary error*) [page 5 (7.25)]

(95) Floating divide assign operator (*discretionary error*) [page 5 (7.25)]

(96) Pointer add assign operator (*discretionary error*) [page 4 (7.18)]

(97) Pointer subtract assign operator (*discretionary error*) [page 4 (7.18)]

(98) Left shift assign operator (*discretionary error*) [page 4 (7.16)]

(99) Right shift assign operator (*discretionary error*) [page 4 (7.16)]

(100) Bitwise and assign operator (*discretionary error*) [page 4 (7.16)]

(101) Bitwise or assign operator (*discretionary error*) [page 4 (7.16)]

(102) Exclusive or assign operator (*discretionary error*) [page 4 (7.16)]

(104) Member of a struct or union, where the first operand is a struct/union value, the second (given by an enk_field node) is the member (field) (*discretionary error*) [page 5 (7.25)]

(105) First operand is an address of a struct/union, second a bit field (*discretionary error*) [page 5 (7.25)]

(106) First operand is a struct/union value, second a bit field (*discretionary error*) [page 5 (7.25)]

(107) Extraction of a value of a bit field (*discretionary error*) [page 5 (7.25)]

(108) Selection of a field identified by a pointer to a (data) member (*discretionary error*) [page 4 (7.17)]

(109) Static member selection of kind p→m (*discretionary error*) [page 4 (7.22)]

(110) Static member selection of kind lvalue.m (*discretionary error*) [page 4 (7.22)]

(111) Static member selection of kind rvalue.m (*discretionary error*) [page 4 (7.22)]

(112) Left shift ("<<" operator) (*discretionary error*) [page 4 (7.16)]

(113) Right shift (">>" operator) (*discretionary error*) [page 4 (7.16)]

(114) Bitwise and ("&" operator) (*discretionary error*) [page 4 (7.16)]

(115) Bitwise or ("|" operator) (*discretionary error*) [page 4 (7.16)]

(116) Exclusive or ("^" operator) (*discretionary error*) [page 4 (7.16)]

(117) Comma operator (*discretionary error*) [page 4 (7.4)]

(118) A normal function pointer for a C++ virtual member function (*discretionary error*) [page 4 (7.17)]

(119) Call of a "destructor" for a class or simple type that does not have one (operand is an lvalue) (*discretionary error*)

(120) Call of a "destructor" for a class or simple type that does not have one (operand is an rvalue) (*discretionary error*)

(123) A call of a function identified by a pointer to a member (*discretionary error*) [page 4 (7.17)]

(124) va_start macro reference (*discretionary error*) [page 4 (7.7)]

(125) va_arg macro reference (*discretionary error*) [page 4 (7.7)]

(126) va_end macro reference (*discretionary error*) [page 4 (7.7)]

(127) <stdarg.h> variant of va_copy macro reference (*discretionary error*) [page 4 (7.7)]

(128) <varargs.h> variant of va_start macro reference (*discretionary error*) [page 4 (7.7)]

(129) Generic negation (*discretionary error*) [page 5 (7.24)]

(130) Generic post increment (*discretionary error*) [page 5 (7.24)]

(131) Generic post decrement (*discretionary error*) [page 5 (7.24)]

(132) Generic pre increment (*discretionary error*) [page 5 (7.24)]

(133) Generic pre decrement (*discretionary error*) [page 5 (7.24)]

(134) Generic addition (*discretionary error*) [page 5 (7.24)]

(135) Generic subtraction (*discretionary error*) [page 5 (7.24)]

(136) Generic multiplication (*discretionary error*) [page 5 (7.24)]

(137) Generic division (*discretionary error*) [page 5 (7.24)]

(138) Generic equality (*discretionary error*) [page 5 (7.24)]

(139) Generic inequality (*discretionary error*) [page 5 (7.24)]

(140) Generic greater than (*discretionary error*) [page 5 (7.24)]

(141) Generic less than (*discretionary error*) [page 5 (7.24)]

(142) Generic greater than or equal (*discretionary error*) [page 5 (7.24)]

(143) Generic less than or equal (*discretionary error*) [page 5 (7.24)]

(144) Generic assignment (*discretionary error*) [page 5 (7.24)]

(145) Generic add assign operator (*discretionary error*) [page 5 (7.24)]

(146) Generic subtract assign operator (*discretionary error*) [page 5 (7.24)]

(147) Generic multiply assign operator (*discretionary error*) [page 5 (7.24)]

(148) Generic divide assign operator (*discretionary error*) [page 5 (7.24)]

(149) Generic unary "&" (*discretionary error*) [page 5 (7.24)]

(150) Generic ".*" field selection (*discretionary error*) [page 5 (7.24)]

(151) Generic "→ ⋆" field selection (*discretionary error*) [page 5 (7.24)]

(152) lvalue (*discretionary error*) [page 5 (7.24)]

(153) rvalue (*discretionary error*) [page 5 (7.24)]

(154) Called function details are not known (*discretionary error*) [page 5 (7.24)]

(155) Initialization of a temporary within an expression (*discretionary error*)

(156) throw expression (*discretionary error*) [page 4 (7.8)]

(157) a variable declaration with an initializer that appears as the condition of an if-, switch-, while- or for-statement (*discretionary error*)

(158) Introduction of an object lifetime (*discretionary error*)

(159) typeid expression (*discretionary error*)

(160) A sizeof expression that cannot be evaluated until runtime (*discretionary error*) [page 4 (7.20)]

(161) Nonstandard construct "&..." (address of ellipsis) (*discretionary error*) [page 4 (7.7)]

(162) The address of a routine (*discretionary error*) [page 4 (7.17)]

(163) Templates are not supported yet (*discretionary error*) [page 5 (7.24)]

(164) Ellipsis are not supported yet (*discretionary error*) [page 4 (7.7)]

(165) Float constant (*discretionary error*) [page 5 (7.25)]

(166) Float type (*discretionary error*) [page 5 (7.25)]

(167) Float post increment expression operation (*discretionary error*) [page 5 (7.25)]

(168) Float post decrement expression operation (*discretionary error*) [page 5 (7.25)]

(169) Float addition expression operation (*discretionary error*) [page 5 (7.25)]

(170) Float subtraction expression operation (*discretionary error*) [page 5 (7.25)]

(171) Float multiply expression operation (*discretionary error*) [page 5 (7.25)]

(172) Float divide expression operation (*discretionary error*) [page 5 (7.25)]

(173) Float "==" expression operation (*discretionary error*) [page 5 (7.25)]

(174) Float "!=" expression operation (*discretionary error*) [page 5 (7.25)]

(175) Float ">" expression operation (*discretionary error*) [page 5 (7.25)]

(176) Float "<" expression operation (*discretionary error*) [page 5 (7.25)]

(177) Float ">=" expression operation (*discretionary error*) [page 5 (7.25)]

(178) Float "<=" expression operation (*discretionary error*) [page 5 (7.25)]

(179) Float assignment expression operation (*discretionary error*) [page 5 (7.25)]

(180) Pointer post increment expression operation (*discretionary error*) [page 4 (7.18)]

(181) Pointer post decrement expression operation (*discretionary error*) [page 4 (7.18)]

(182) Pointer addition expression operation (*discretionary error*) [page 4 (7.18)]

(183) Pointer subtraction expression operation (*discretionary error*) [page 4 (7.18)]

(184) Pointer ">" expression operation (*discretionary error*) [page 4 (7.18)]

(185) Pointer "<" expression operation (*discretionary error*) [page 4 (7.18)]

(186) Pointer ">=" expression operation (*discretionary error*) [page 4 (7.18)]

(187) Pointer "<=" expression operation (*discretionary error*) [page 4 (7.18)]

(189) Static local variable (*discretionary error*) [page 4 (7.22)]

(190) Static member routine (*discretionary error*) [page 4 (7.22)]

(191) Static variable (C-style) (*discretionary error*) [page 4 (7.21)]

(192) Static routine (C-style) (*discretionary error*) [page 4 (7.21)]

(193) Statement unrecognized (*discretionary error*) [page 4 (7.9)]

(194) Maximal nesting of the operator $\star$ to construct pointer types is 2 (*discretionary error*) [page 4 (7.17)]

(195) This kind of an array is unsupported (*error*) [page 5 (7.25)]

References

[1] OFFIS: "The Rhapsody Verification Environment", How to go round some restrictions, May 2004

[2] OFFIS: "The Rhapsody Verification Environment", Installation-Guide and Tutorial, May 2004

[3] OFFIS: "The XMI Verification Environment", Features and Restrictions, May 2004

OLDENBURGER FORSCHUNGS- UND ENTWICKLUNGSINSTITUT
FÜR INFORMATIK-WERKZEUGE UND -SYSTEME

Rhapsody Verification Environment

How to go round some restrictions

$Revision: 1.1 $, $Date: 2004/02/12 13:46:27 $

1 Introduction

This document gives hints how to use some UML-/C++-features, which are in general still restricted in the Rhapsody Verification Environment, but are usable in special cases. By using these features you are entering the *gray zone*, that is you are forced to use special workarounds which may be not exactly what you want or what you already used in your model and which are (at the moment) only semi-supported by us. In many cases this document could neverless be helpful to the user of the Rhapsody Verification Environment.

2 Arrays

Arrays in general are not supported, but for RUVE (yet not for XUVE) it is possible to define *one dimensional* `int`-arrays in the following way:

1. Add a new type (e.g. named `Array_of_5_int`) to the package of your Rhapsody model.

2. In the *Features* → *C++ Declaration*-field of the type enter a declaration in the following form (the following declaration corresponds to the example array `Array_of_5_int` mentioned above):

   ```
   typedef int %s[5];
   ```

3. Now you can create class attributes and give them the array-type in the following way: Under *Features* → *Attribute type* you have to select *Use existing type* and choose your named array-type (e.g. `Array_of_5_int`) from the *Type*-dropdownlist.

4. You have to deselect the following properties of all attributes which are of an array type:

   ```
   CPP_CG.Attribute.AccessorGenerate
   CPP_CG.Attribute.MutatorGenerate
   ```

   Instead, you may of course explicitly define own accessor- or mutator-functions in the class of the attribute.

At the moment we only support such array types for *attributes*, but local variables of an arraytype are not supported yet. Note that we also do not support pointer to arrays, especially it is not directly possible to use arrays as parameters for primitive operations, events or triggered operations. Alternatively you may define an own class with an array attribute and use the pointer to this class.

3 Enumerations

Enumeration types are only supported in the C++ sense:

- enumeration values correspond to integer values, that is they may be used in arithmetical expressions
- two different named enumeration values (e.g. `ev_a` and `ev_b`) may have the same integer value, that is the expression `ev_a` == `ev_b` may result to `true`.
- each enumeration type defines an integer range, which can be represented by the smallest bitfield that is necessary to contain all named enumeration values (example: the type `enum example_et {ev_a`$= -1$`, ev_b`$= 2\}$ defines an integer range from $-4(= -2^2)$ to $3(= 2^2 - 1)$). Thus it is possible that an enumeration variable may store an integer value without the existance of a corresponding enumeration name. The result of an enumeration variable after a range overflow is not defined.
- integer values may be converted to enumeration values (example: `x = example_et(-3)`).

You may use enumerations in RUVE (yet not in XUVE), if the use is not in conflict with this C++ sense. Enumeration types have to be defined by a typedef and have to be used by refering to the name defined by the typedef (analogue to arrays, see previous section). We do not support overloading of functions by using same named functions, were the signatures have the same number of parameters and only differ in the parameter types (for example: we do not support the declaration of two functions `f(int)` and `f(example_et)` in the same scope).

It is not necessary that the range of each enumeration fits into the int-range defined by the properties

```
CPP_CG.Verification.IntegerLowerBound
CPP_CG.Verification.IntegerUpperBound
```

This could be very helpful, if you need for special variables a greater range than the range defined by these properties (for example, if you want to calculate the average of three int values by calculating the sum of the three values into a special `res` variable and afterwards dividing `res` by 3 - in this case you need a greater range for `res` to prevent an overflow).

On the other hand, enumerations could be used to minimize the state space (and thus the verification time) of the model by using for each variable an enumeration type with the smallest possible int-range.

We do not yet support the `bool` type, but you may define a two-valued enumeration type with corresponding int-values 0 and 1. You may, after a conversion operation to the enumeration type, assign 'simple' boolean expressions to such a variable[1] and compare them with other simple boolean expressions (by using the expression operations '`==`' or '`!=`').

4 Loops

Loops in general are not supported, but we support a special form of the `for`-loop, which could be very helpful if you want to navigate through a relation with a multiplicity greater than 1:

```
for (int <var> = <civ1>; <var> < <civ2>; <var>++) ...
```

Strictly speeking, the following conditions must be met:

- The counter variable `<var>` must be of type `int`
- The declaration of `<var>` must be within the for-loop initialization
- `<var>` must be initialised directly within the declaration by assigning a constant `int`-value `<civ1>`
- nothing else may be done in the initialisation field
- the `for`-loop condition expression must be an `int`-'lower than' expression operation; the left operand must be the counter variable and the right operand must be a constant integer value
- the `for`-loop increment expression must be the `int`-post increment operator applied on the counter variable
- the counter variable may not be changed within the `for`-loop statement

For example, the following `for`-loop sends an event `E` to all 4 parts of a composition `itsC`:

```
for (int i=0; i<4; i++) { itsC[i]->GEN(E); }
```

[1] let `true`, `false` and variables of a $\{0,1\}$-valued enumeration type be a 'simple' boolean expressions. The result of the operations '`&&`', '||' or '`!`' on 'simple' boolean expressions is also a 'simple' boolean expression

OLDENBURGER FORSCHUNGS- UND ENTWICKLUNGSINSTITUT
FÜR INFORMATIK-WERKZEUGE UND -SYSTEME

The XMI UML Verification Environment

User Documentation

$Revision: 1.10 $, $Date: 2004/05/07 14:37:30 $

This documentation describes the usage of the tools contained in the XMI UML Verification Environment. Please see *The Rhapsody UML Verification Environment (Installation-Guide and Tutorial)* [7] for a detailed description of the system requirements and the installation procedure.

It is possible to start XUVE directly from Rhapsody, see section 5. You will find a description of the nondetermism-support in section 6.

1 `verify_xmi.sh`

The tool `verify_xmi.sh` allows to verify specifications for a XMI UML model. Before using it, you must be familiar with the different kinds and the syntactical form of specifications for the XMI UML Verification Environment. It supports the same specification descriptions as the Rhapsody UML Verification Environment, please read [7] for a detailed description. In addition you have to ensure that no restriction of the XMI UML Verification Environment is violated, please read the corresponding document [8]. The usage of `verify_xmi.sh` is:

```
Usage: verify_xmi.sh [OPTIONS] XMIFILE [PROPERTY...]
```

By typing[1] `verify_xmi.sh --help` you will get a list of possible options. The most common options are `--fixstate` and `--omal`. By giving the option `--fixstate`, the `verify_xmi.sh` script interprets all instances of `State` in the XMI to be in fact instances of `SimpleState`[2]. By giving the option `--omal`, the user determines that the action language within the XMI file is the Omega action language defined in [1]. Without this option, the tool assumes the Rhapsody action language, which is a subset of C++[3].

With the `PROPERTY` file you determine the specification which has to be checked by the XMI Verification Environment. If no `PROPERTY` file is given, a

[1] The CygWin-Bash will find the XMI-tools, if you add `$UMLVERIFROOT/bin` to the `PATH` Environment-variable or if you prefix each call with `$UMLVERIFROOT/bin/`

[2] Normally, there may not exists any `State` instance in the XMI, because `State` is an abstract UML meta-class. Nevertheless, instances of `States` instead of `SimpleStates` are used in XMI-files generated by the Rhapsody XMI-Toolkit.

[3] In fact, the language is only syntactically a subset of C++. Semantically the language is extended by interpreting special preprocessor-pragmas as nondeterministic choise- or interleaving-constructs, see the grammar in section 4

default file `properties.spec` is copied to the current directory and opened with your favorite editor, which has to be set in the `$EDITOR` environment variable. If `$EDITOR` is not set, WordPad (or `vi`, if WordPad cannot be found) is used as the default editor.

The syntax of the property file should be self-explanatory, if you know how to set the corresponding properties in the Rhapsody UML Verification Environment (see [7] - make sure that your editor does not introduce new linebreaks and follow the rules for multi-line properties). The value of `Configuration` in the first line is used to create a new directory with this name, where all files for the verification task are written.

As an example, we describe in the following how to verify the invariance check property for the Vending Machine[4], mentioned in section 3.6 of [7]:

- Enter the directory `$UMLVERIFROOT/examples/TheVendingMachine/`
- This directory contains an XMI file of the model, named `TheVendingMachine.xmi`. To start the verification you have to call

  ```
  $UMLVERIFROOT/bin/verify_xmi.sh TheVendingMachine.xmi
  ```

- If you do not already have a `properties.spec` file in the current directory, the following message will be given:

  ```
  No ./properties.spec, trying to find one...

  Copied default properties.spec -- please edit  [RETURN to continue].
  ```

  After pressing the `RETURN`-button, you can edit the properties.spec file.
- Set the name `VerifyCI` for the configuration by editing the line beginning with `Configuration`:

  ```
  Configuration : String = "VerifyCI"
  ```

- Make sure that the `ExternalEventModus` is set to `ndet`:

  ```
  ExternalEventModus : "det,ndet" = "ndet"
  ```

- Specify the events, which are can be send nondeterminstically by the environment:

  ```
  ExternalEventTrace : MultiLine = "root->p_VendingMachine->itsCoinValidator, C50; \
  root->p_VendingMachine->itsCoinValidator, E1; \
  root->p_VendingMachine->itsChoicePanel, WATER; \
  root->p_VendingMachine->itsChoicePanel, SOFT; \
  root->p_VendingMachine->itsChoicePanel, TEA;"
  ```

[4]The `examples`-directory contains Rhapsody models and corresponding XMI-files for the Vending Machine- and the MiniExampleExt-model, both in two versions: The first version uses a C++ subset as action language, the second the Omega action language.

This property is of type `MultiLine`, so you may use linebreaks - but you have to take care that each line (except the last line) is finished by a backslash.

- Set the kind of specification to `InvarianceCheck` (this has to be a single line without linebreaks):

  ```
  Spec : "DriveToProperty,InvarianceCheck,PatternCheck,
  TemporalLogic,LifeSequenceChart" = "InvarianceCheck"
  ```

- Set the `Spec::InvarianceCheck` property (this property also must not contain linebreaks):

  ```
  Spec::InvarianceCheck : String =
  "!(root->p_VendingMachine->itsDrinkDispenser->IS_IN(Water_out) && \
  root->p_VendingMachine->itsDrinkDispenser->IS_IN(Soft_out) && \
  root->p_VendingMachine->itsDrinkDispenser->IS_IN(Tea_out))"
  ```

- Now you are ready, save and exit the editor. The verification process will start and will (after a while) produce a resulting timing diagram and a corresponding LSC (see [7] for a screenshot).

2 xmicheck

The tool `xmicheck` allows to check an XMI UML model against a number of restrictions of the Omega Kernel Language (see section 3 of [6]). The usage of `xmicheck` is:

```
Usage: xmicheck [OPTIONS] XMIFILE...
```

`xmicheck --help` shows the possible options. The following list informs briefly (by listing the corresponding error message), which of the restrictions are taken into account by `xmicheck`. Some of them are not checked, if option `--rhap_xmi_check` is given (because some OMEGA-restrictions are always violated by Rhapsody XMI-files). If `xmicheck` should be applied on an XMI-file which was generated by Rhapsody, the option `--fixstate` is helpful to fix a bug in Rhapsody's XMI-exporter concerning the representation of `SimpleStates`. `xmicheck` itself is restricted to XMI 1.0 (see [3]) and UML 1.3 (see [4], [5]) (future versions will also support XMI 1.1 and/or UML 1.4). The checked restrictions are:

1. Abstract class (*error*)
2. Abstract operation (*error*)
3. Stereotype (*warning*) [not check with option `--rhap_xmi_check`]
4. Flow-Relation (*error*)
5. Dependency-Relation (*error*)
6. Abstraction-Relation (*error*)

7. Binding-Relation (*error*)
8. Permission-Relation (*error*)
9. Usage-Relation (*error*)
10. AssociationClass (*error*)
11. An AssociationEnd of a Composition has more than one MultiplicityRange (*error*)
12. An AssociationEnd of a Composition has an interval-range of the form [n..m] (*error*)
13. The root-AssociationEnd of a Composition has a multiplicity not equal to 1 (*error*)
14. An 'end-point'-AssociationEnd of a composition is not navigable (*error*)
15. An 'end-point'-AssociationEnd with Multiplicity n or a root-AssociationEnd of a composition is not frozen (*error*) [not check with option `--rhap_xmi_check`]
16. An 'end-point'-AssociationEnd with Multiplicity * of a composition is not add_only (*error*)
17. An AssociationEnd of an Aggregation has more than one MultiplicityRange (*error*)
18. An AssociationEnd of an Aggregation has an unbounded MultiplicityRange with lower bound not equal to zero (*error*)
19. An 'end-point'-AssociationEnd of an aggregation is not navigable (*error*)
20. The root-AssociationEnd of an Aggregation has a multiplicity not equal to 1 (*error*)
21. A root-AssociationEnd of an aggregation is not frozen (*error*) [not check with option `--rhap_xmi_check`]
22. An AssociationEnd of an Association has more than one MultiplicityRange (*error*)
23. An AssociationEnd of an Association has an unbounded MultiplicityRange with lower bound not equal to zero (*error*)
24. Time event (*warning*)
25. Change event (*error*)
26. The CallConcurrencyKind of a triggered operation is neither guarded nor sequential (*error*)
27. Choice State (*error*)
28. Fork State (*error*)
29. Join State (*error*)

30. Junction State (*error*)
31. State with Do-Activity (*error*)
32. Two aggregate classes are related to the same aggregate-parts (*error*)
33. There must be exactly one class with tag 'isRootClass' valued 'True' (*error*)
34. The root-class must be active (*error*) [not check with option `--rhap_xmi_check`]

3 omal2cpp

The tool `omal2cpp` allows to convert an XMI_{omal} file into an XMI_{C++} file. That is, the action language of method bodies, transition effects and the expression language of transition guards is translated from the Omega action language (see [1]) into the Rhapsody action language. The usage of `omal2cpp` is:

```
Usage: omal2cpp [OPTIONS] XMIINFILE XMIOUTFILE
```

At the moment `omal2cpp` is restricted to XMI 1.0 and UML 1.3 (future versions will also support XMI 1.1 and/or UML 1.4).

Note that the tool still uses the old version of the OMAL-language defined in [1]. Support for the revised grammar defined in [2] is not yet supported.

4 cpp2omal

This tool will take an XMI_{C++} file and produces an XMI_{omal} file. As actions in XMI_{C++} there are only C++ constructs allowed which correspond in a syntactical way to a language construct in the Omega action language. C++ comments of multiline-form can be used, but will be ignore by the translation.

The following grammar rules determine the syntactical subset of C++ which can be translated by cpp2omal. Most of the rule names are equal to the corresponding names in [2]. The literal `IDENTIFIER` is defined by the regular expression `[A-Za-z_][A-Za-z0-9_]*`.

4.1 The root `cpp_action`

```
<cpp_action> ::= /* empty */ | <action> | <action_list> <action>
```

This rule allows to use action lists without enclosing block brackets.

4.2 `action`

```
<action> ::= [ IDENTIFIER ':' ] (
    <elementary_action>
  | <control_action>
  | <composite_action>
  | <nondet_choice_action>
  | <interleaving_action> )
```

4.3 elementary_action

```
<elementary_action> ::= ';' /* empty action */
  | <assignment_action> ';'
  | <call_action> ';'
  | <send_action> ';'
  | <return_action> ';'
  | <reply_action> ';'
```

4.4 assignment_action

```
<assignment_action> ::= <navigation_expression> '=' <expression>
```

4.5 call_action

```
<call_action> ::= <call_expression>
```

4.6 send_action

Generating signal events in C++ must be done by using a `GEN` macro or function (like in Rhapsody's C++ framework), which will be translated into the corresponding Omega Language representation.

```
<send_action> ::= <navigation_expression> '->' 'GEN'
    '(' IDENTIFIER '('
    [ <simple_expression> ( ',' <simple_expression> )* ]
    ')' ')'
```

4.7 return_action

```
<return_action> ::= 'return' <simple_expression>
```

4.8 reply_action

The `reply`-macro is part of the Rhapsody-Framework and is used to return the value of a triggered operation.

```
<reply_action> ::= 'reply' '(' <simple_expression> ')'
```

4.9 control_action

```
<control_action> ::= 'if' <expression> <action>
  | 'if' <expression> <action> 'else' <action>
  | 'while' <expression> <action>
```

4.10 composite_action

```
<composite_action> ::= '{' <action_list> '}'
```

4.11 nondet_choice_action

There exists no language construct in C++ for nondeterministic choice actions. To enable the use of such a construct for the verification tools, we use special C++ preprocessor-pragmas[5].

```
<nondet_choice_action> ::=
  '#pragma' 'choose' <action_list> '#pragma' 'endchoose'
```

Note that this pragma-section (and the pragma-section defined in the next subsection), may not be nested within the same scope. You may instead introduce a 'pseudo'-scope by beginning a new C++-block before an inner-section and by ending the C++-block after the inner-section. Example: Suppose you want to nondeterminstically choose from four actions, where the third action consists of two (sub-)actions. The order of the execution of these subactions should be nondeterministic. Then, instead of writing

```
#pragma choose
<action1>
<action2>
#pragma cobegin
<action3.1>
<action3.2>
#pragma coend
<action4>
#pragma endchoose
```

you have to write:

```
#pragma choose
<action1>
<action2>
{
#pragma cobegin
<action3.1>
<action3.2>
#pragma coend
}
<action4>
#pragma endchoose
```

4.12 interleaving_action

Analogue to the previous nondet_choice_action-rule, we use special C++ preprocessor-pragrams to express the interleaving action:

```
<interleaving_action> ::=
  '#pragma' 'cobegin' <action_list> '#pragma' 'coend'
```

[5]Note the syntax-rules for C++ preprocessor-pragmas (in particular, there must be a linebreak before and after the action_list).

4.13 control_action

```
<control_action> ::= 'if' '(' <expression> ')' <action>
  | 'if' '(' <expression> ')' <action> 'else' <action>
  | 'while' '(' <expression> ')' <action>
```

4.14 group_action

```
<group_action> ::= '{' <action_list> '}'
```

4.15 action_list

```
<action_list> ::= <action> | <action_list> <action>
```

4.16 expression

```
<expression> ::= <call_expression>
  | <create_expression>
  | <simple_expression>
```

4.17 call_expression

```
<call_expression> ::= <navigation_expression> '->'
  [ IDENTIFIER '::' ]
  IDENTIFIER '('
  [ <simple_expression> ( ',' <simple_expression> )* ]
  ')'
```

4.18 create_expression

```
<create_expression> ::=
  'new' IDENTIFIER [ '::' IDENTIFIER ] '('
  [ <simple_expression> ( ',' <simple_expression> )* ]
  ')'
```

4.19 simple_expression

```
<simple_expression> ::=
  [ <simple_expression> '||' ] <and_expression>
```

4.20 and_expression

```
<and_expression> ::=
  [ <and_expression> '&&' ] <relational_expression>
```

4.21 relational_expression

```
<relational_expression> ::= [ <relational_expression>
    ( '<' | '<=' | '==' | '>=' | '>' | '!=' ) ] <add_expression>
```

4.22 add_expression

```
<add_expression> ::=
  [ <add_expression> ( '+' | '-' ) ] <mult_expression>
```

4.23 mult_expression

```
<mult_expression> ::=
  [ <mult_expression> ( '*' | '/' ) ] <unary_expression>
```

4.24 unary_expression

```
<unary_expression> ::= [ ( '-' | '!' ) ] <primary_expression>
```

4.25 primary_expression

```
<primary_expression> ::=
  <literal>
  | <navigation_expression>
  | '(' <simple_expression> ')'
```

4.26 literal

This rule contains the literal `INTEGER_LIT`, which is defined by the regular expression `[0-9]+` and the literal `REAL_LIT`, which is defined by the regular expression `[0-9]+\.[0-9]+`.

```
<literal> ::=
  'false' | 'true' | 'NULL' | INTEGER_LIT | REAL_LIT
```

4.27 navigation_expression

This rule contains the special literal `COLLOP_IDENTIFIER`, which is defined by the regular expression

```
("isEmpty_"|"notEmpty_"|"size_")[A-Za-z_][A-Za-z0-9_]*
```

The suffix of this regular expression determines the name of the collection. Note that this part looks different to the OMAL representation - the reason is that supported collections in Rhapsody models are C++ arrays, which does not have any member functions. Instead, there have to be for each class C and each collection c in C member functions[6] `size_c`, `isEmpty_c` and `notEmpty_c` which realize the corresponding behavior.

C++ Array-accesses are translated to the OMAL-`getAt`-collection operation.

[6]It depends on the framework, if the member functions must be implicit or explicit in the model: As an example, the XMI Verification Environment supplies these functions implicit, but for the Rhapsody Verification Environment these member functions must be defined explicit in the model, because they are not automatically generated by Rhapsody's Codegeneration.

The rule contains also a special 'params' variable, which is supplied by the Rhapsody Framework and allows to access parameters of the event-trigger or operation-trigger. This variable is translated into the name of the Reception, which is equal to the operation- resp. event-name.

```
<navigation_expression> ::=
    ( 'this' | IDENTIFIER | 'params' '->' IDENTIFIER )
    ( '->' IDENTIFIER )*
    [ '[' <simple_expression> ']' ]
  | [ ( 'this' | IDENTIFIER | 'params' '->' IDENTIFIER )
      ( '->' IDENTIFIER )* ] COLLOP_IDENTIFIER '(' ')'
```

5 Starting XUVE from Rhapsody

If your UML-model is a Rhapsody-model, you can also start XUVE from the Rhapsody-Environment. Make sure, that your model does not violate a restriction mentioned in [8], if it would be exported to XMI by the Rhapsody XMI Toolkit. To use this feature, you have to set the property

`CPP_CG.Verification.MetaInfoTool`

to the value `xmi4rhap` (the default value is `rhap2ssl`).

6 Nondeterminism-Support

The XMI UML Verification Environment (XUVE) supports nondeterministic semantic for the following `StateMachine`-constructs:

- Orthogonal regions of composite states: The execution order of orthogonal regions in composite states is chosen nondeterministically.
- Nondeterministic transition choice: The transition, which will be taken in a given state, is chosen nondeterministically from the set of triggered transitions in this state.

There exist two properties of type `Bool` to switch on these features (please read the ruve tutorial [7] for a description of how to set properties for a configuration):

`CPP_CG.Verification.NondetAndStates`
`CPP_CG.Verification.NondetTransitions`

In addition, we support two kinds of nondeterminisms within the action language: nondeterministic choice of an action from a given action-list by using the `choose`-pragma (see section 4 for the syntax), and nondeterministic interleaving of a given list of actions by using the `cobegin`- resp. `coend`-pragma.

References

[1] Ileana Ober. *Definition of the tool exchange format*, OMEGA-Milestone IST/33522/WP2.2/M2.2.1, Revision 4, March 2003

[2] Ileana Ober. *Action specification in OMEGA*, OMEGA-Milestone IST/33522/WP2.2/M2.2.1, Revision 3-a4, March 2004

[3] *OMG XML Metadata Interchange (XMI) Specification*, Object Management Group, Version 1.0, June 2000, http://www.omg.com/docs/formal/00-06-01.ps

[4] *OMG Unified Modeling Language Specification*, Object Management Group, Version 1.3, March 2000, http://www.omg.com/docs/formal/00-03-01.ps

[5] *OMG UML 1.3 Interchange Metamodel*, Object Management Group, December 2001, http://www.omg.com/docs/ad/01-12-02.xml

[6] Werner Damm, Bernhard Josko, Amir Pnueli, Angelika Votintseva *Deliverable D.1.1.2: A Formal Semantics for a UML Kernel Language*, OMEGA-Deliverable

[7] *The Rhapsody UML Verification Environment*, Installation Guide and Tutorial, OFFIS, June 2004

[8] *The XMI UML Verification Environment*, Features and Restrictions, OFFIS, June 2004

OLDENBURGER FORSCHUNGS- UND ENTWICKLUNGSINSTITUT
FÜR INFORMATIK-WERKZEUGE UND -SYSTEME

The XMI UML Verification Environment

Features and Restrictions

$Revision: 1.7 $, $Date: 2004/05/07 14:45:20 $

1 Introduction

The XMI UML Verification Environment (**xuve**) enables the formal verification of UML models given as XMI-files. Although **xuve** uses the same back-end tools and is part of the Rhapsody UML Verification Environment (**ruve**), **xuve** is independent of a Rhapsody installation and can be run at its own.

UML (and thus also its XMI representation) allows to describe the same thing in different ways. A problem for XMI as an exchange format is that UML tools often diverge on its chosen way of description. Other problems for the exchange could arise when UML tools require resp. deliver XMI-files based on different XMI- or UML-versions.

This document determines the properties of the XMI required by **xuve**:

- The XMI and UML version is determined in Section 2
- Section 3 contains a list of supported UML meta-class instances and its required representation in XMI.
- Finally section 4 determines the action language supported in the XMI

Note that all restrictions of **ruve**, documented in [4] and [5], are also valid for **xuve**.

2 XMI Version

At the moment **xuve** requires XMI-files based on XMI 1.0 (see [8]) and UML 1.3 (see [7] and [6]), which are still the versions exported for example by Rhapsody and ArgoUML. Extensions to support also XMI 1.1 and UML 1.4 will be done in future versions of **xuve**.

3 UML Meta-Classes

All non-abstract UML meta-classes can be instanciated in an XMI-file. For the verification of a given UML-model, **xuve** takes only a subset of them into account, because:

- Some meta-class instances are not relevant for the behavioral description of the model (for example, some tools use meta-classes of the package `Extension_Mechanisms` to store graphical informations of the model)
- UML allows to describe the same thing in different ways and UML-tools tend to diverge at some aspects (for example the representation of transition effects or operation-bodies are often done in different ways). In the initial version of **xuve** we support XMI-representations which are similar to Rhapsody's XMI (see the corresponding subsections below for the details)
- **xuve** is restricted to UML-models which are conform to the Omega kernel language (see [1])
- The restrictions of the Rhapsody UML Verification Environment are also valid for **xuve** (see [4]). The default behavior of **xuve** is to use the same semantical interpretation of statecharts as Rhapsody.Alternatively, xuve supports nondeterminism for two aspects of the StateMachine semantic:
  - If more than one outgoing transition of a state is triggered, then the choice which transition will be taken is nondeterministic.
  - The execution order of orthogonal regions of composite states is nondeterministic
- There are some additional restrictions for the current version of **xuve**, because the corresponding features are not yet implemented

The subsections below describe in technical detail, which instances of the UML meta-classes can be translated from XMI to the internal model-checker format SSL/SMI (instances not mentioned here may exist in the XMI-file, but they will not be translated properly):

3.1 `Model`

There must be exactly one, named instance of `Model`, as child of `XMI.content`.

3.2 `Package`

There must be exactly two instances of `Package`, as `ownedElement` of the Model. One Package named `PredefinedTypes` has to contain all basic datatypes (like `int`, `bool`, `char`), which are used in the model (They are instances of the UML meta-class `Datatype`). The other package, named by the user, contains the rest of the model. This package may optionally have a tag

`CPP_CG.Package.SpecIncludes`

set to a filename, which may contain the declaration of preprocessor-macros (in C++ syntax) to be used as integer constants in the action language.

3.3 Class

Named classes as `ownedElement` of the user-defined Package are supported (but for example no classes nested in classes). Exactly one class must have a tag `CPP_CG.Class.isRootClass` set to `True`. The meta-attribute `isActive` may be `true` or `false`.

3.4 Association

The user-defined package may contain (as `ownedElement`) Associations between classes. The `connection` of each Association has to consist of exactly two AssociationEnd-instances.

3.5 AssociationEnd

The field `isNavigable` of AssociationEnds may be `true` or `false` - the AssociationEnd must be named, if `isNavigable` is `true`. The `aggregation`-kind may be `none`, `aggregate` or `composite`.

3.6 Multiplicity

The `multiplicity` of an `AssociationEnd` must consist of exactly one MultiplicityRange, if `isNavigable` is `true`.

3.7 MultiplicityRange

The `lower`- and `upper`-bound of a `MultiplicityRange` must be set to a number n, $n > 0$.

3.8 Generalization

The user-defined package may contain (as `ownedElement`) Generalizations between classes (that is, the `child` and `parent`-field of the `Generalization` refer to a `Class` and both classes refer by the `generalization` resp. `specialization`-field back to the `Generalization`) or between signals (that is, the `child` and `parent`-field refer to a `Signal`, where again both signals refer back to the `Generalization`).

3.9 Attribute

A named `Attribute` can be a `feature` of a class. All kinds of `visibility` (`public`, `private`, `protected`) and the `ownerScope`-value `instance` are supported. The `type` has to be a basic type defined in the `PredefinedTypes` package.

Attributes may have a tag `CPP_CG.Attribute.MutatorGenerate` set to `False`. If this tag is missing, an implicit `set`-method is generated (named `set<name of the Attribute, first character upper case>`) with exactly one parameter equally typed as the attribute (the implicit generation of this method is omitted, if a method with this name already exists in the class, because of an explicit definition by the user). The `set`-method can be used in the model

to set the value of the attribute. Analogously, Attributes may have a tag `CPP_CG.Attribute.AccessorGenerate` set to value `False` to suppress the implicit generation of an accessor-function named `get<name of the Attribute, first character upper case>`.

Attributes may have a tag `CPP_CG.Attribute.SmiModus`. Possible values are `state`, `auxiliary` and `input`. See [5] for a detailed description of this property.

3.10 Operation

A named `Operation` can be a `feature` of a class. All kinds of `visibility`, all kinds of `ownerScope` and the `concurrency`-value `sequential` are supported. If the operation is a triggered operation, a tag `isTrigger` must have the value `true`. If the operation is virtual, a tag `CPP_CG.Operation.isVirtual` must have the value `True`.

3.11 Method

For each nontriggered operation there must be exactly one defining `Method`, which has to be a `feature` of the corresponding class (triggered operations may not have a defining method - instead their semantic is given by the StateMachine of the class, which has the triggered operation as a feature). The definition of the method is located in the `body`-meta attribute. Constructors and the destructor of a class may be modeled as Operations resp. Methods. They have to be named like the corresponding class (for constructors) or like the corresponding class prefixed with '~' (for destructors). If the operation of the method is virtual, the method must have a tag `CPP_CG.Operation.isVirtual` set to the value `True`.

3.12 Parameter

Operations and the corresponding methods may have parameters of kind `in` and must have at least one parameter of kind `return`. The type of a Parameter has to be a basic type, defined in the `PredefinedTypes` package.

3.13 StateMachine

Each class may have at most one `StateMachine` as `behavior`. The `top`-state of a StateMachine has to be a CompositeState. A `StateVertex` may be a PseudoState, a CompositeState (`isConcurrent` may be `true` or `false`) or a SimpleState. CompositeStates and SimpleStates must be named and may have an `UninterpretedAction` (with corresponding code in its `body`-attribute) as `entry` or `exit`.

3.14 PseudoState

A PseudoState may be of kind `initial`, `deepHistory`, `join` or `fork`. In addition, `branch`- and `final`- Pseudostates are only supported with Rhapsodys semantic: `branch`-Pseudostates are *statical conditional branches* and the effect of `final`-Pseudostates is the destruction of the instance after entering the state.

3.15 `Transition`

Transitions may be `incoming` and `outgoing` of state vertices. Each transition may have a at most one `guard` (the guard-expression has to be an instance of `BooleanExpression`), a `trigger` (of meta-class `SignalEvent` or `CallEvent`) and an `effect` (of meta-class `UninterpretedAction`), which contains the action in the `body` meta-attribute).

3.16 `SignalEvent`

The `SignalEvent` must refer (with name `signal`) to a `Signal` instance.

3.17 `Signal`

The userdefined package may have as `ownedElement` instances of the meta-class `Signal`. Each `Signal` may have instances of meta-class `Attribute` as `feature`. The `Signal` of a SignalEvent must be named.

3.18 `CallEvent`

The `operation` of a `CallEvent` must be a triggered operation.

4 Action Language

The action language in method bodies, transition effects and the corresponding expression language-subset used in transition guards may be either the Rhapsody action language, which is a subset of C++ with library functions corresponding to the actions defined informally in [1], or the Omega Action Language defined in [3] (not that the tool `omal2cpp` and the verification of XMI-files which use the Omega-action language, is still based on the nonrevised grammar defined in [2]).

References

[1] Werner Damm, Bernhard Josko, Amir Pnueli, Angelika Votintseva *Deliverable D.1.1.2: A Formal Semantics for a UML Kernel Language*, OMEGA-Deliverable

[2] Ileana Ober. *Definition of the tool exchange format*, OMEGA-Milestone IST/33522/WP2.2/M2.2.1, Revision 4, March 2003

[3] Ileana Ober. *Action specification in OMEGA*, OMEGA-Milestone IST/33522/WP2.2/M2.2.1, Revision 3-a4, March 2004

[4] OFFIS. *Restrictions For the Verification of Rhapsody UML-models*, 2002.

[5] OFFIS. *The Rhapsody UML Verification Environment*, Installation-Guide and Tutorial, 2003.

[6] *OMG UML 1.3 Interchange Metamodel*, Object Management Group, December 2001, http://www.omg.com/docs/ad/01-12-02.xml

[7] *OMG Unified Modeling Language Specification*, Object Management Group, Version 1.3, March 2000, http://www.omg.com/docs/formal/00-03-01.ps

[8] *OMG XML Metadata Interchange (XMI) Specification*, Object Management Group, Version 1.0, June 2000, http://www.omg.com/docs/formal/00-06-01.ps