

# COMP\*

## User Guide

Răzvan Diaconescu and Ionuț Țuțu

Simion Stoilow Institute of Mathematics  
of the Romanian Academy, Romania

`razvan.diaconescu@imar.ro`, `ionut.tutu@imar.ro`

## 1 Overview

COMP is a specification language and analysis tool that supports the formal development of component-based systems in a modular, hierarchical fashion: complex systems are built by putting together subsystems/components, which may have their own components, and so on. This allows for a powerful and efficient verification method, guided by the hierarchical structure of the system under consideration.

From a foundational perspective, COMP embodies the behavioural-abstraction paradigm, which gives prominence to the observable behaviour of systems over their structural representations or the functions that they may perform. The mathematical roots of the language belong to the area of hidden algebra. In particular, the main programming units of COMP, called object modules, consist of hidden-algebra declarations of data or state sorts, data operations, actions, observations, and axioms that bring everything together and define the semantics of programs. Structured specifications are obtained through parallel, synchronized, or indexed compositions of object modules – a defining feature of COMP – which enable the hierarchical construction of larger and more complex objects from simpler components.

In this document, we go over the steps needed in order to install and use the tool, illustrate the way it supports the specification of component-based systems – including various commands for inspecting specifications and for working with local files – and demonstrate the formal-verification capabilities of COMP.

---

\*This work was supported by a grant of the Romanian Ministry of Education and Research, CCCDI – UEFISCDI, project number PN-III-P2-2.1-PED-2019-0955, within PNCDI III.

## 2 Obtaining and running COMP

COMP is part of SpeX, a multi-language formal-specification environment written in Maude that supports various specification languages by means of a generic notion of information *processor* – one for each language integrated into the environment. Among the language processors that are currently part of SpeX, COMP is one of the most advanced, taking full advantage of the environment’s computational capabilities.

Therefore, in order to use COMP, both Maude and SpeX need to be installed. GNU/Linux and macOS binaries of Maude are available at its GitHub site. To be able to run the latest distributions of SpeX and COMP, we recommend installing a recent version of Maude: 3.2 or newer. To install Maude, it suffices to:

1. Extract the files in the downloaded ZIP archive to a convenient directory; this can be done, for instance, on a GNU/Linux machine, from a terminal, using a shell like Bash, through the following command:

```
sudo unzip Maude-3.2.1-linux.zip -d /usr/local/maude-3.2.1/
```

2. Make a discoverable link to the Maude executable; for example, provided that `/usr/local/bin/` is in the PATH of directories where executable files are located:

```
sudo ln -s /usr/local/maude-3.2.1/Linux64/maude.linux64 \
/usr/local/bin/maude
```

3. Set the MAUDE\_LIB environment variable appropriately (to the directory where `prelude.maude` and other Maude files from the ZIP archive were extracted):

```
export MAUDE_LIB=/usr/local/maude-3.2.1/Linux64
```

To make this setting persistent, you can add the above line to your `.bashrc` file (or to a similar startup file like, say, `.zshrc` in case you are using Z shell).

With Maude installed, you can proceed to downloading the latest distribution of SpeX from its GitLab site. You can install SpeX using the following commands:

4. Extract the files in the downloaded archive:

```
tar -xzf spex-22.09.tar.gz
```

5. Configure, make, and install SpeX (from its source-tree directory):

```
cd spex-22.09/ && ./configure && make && sudo make install && cd -
```

You may now safely remove the downloaded archive and the directory `spex-22.09`. The SpeX libraries should be installed to `/usr/local/share/spex/` and a shell script at `/usr/local/bin/spex` should be available in order to launch the environment.

The COMP interpreter can be installed in a similar manner to SpeX. Once you download the latest distribution of the tool from its GitLab site, COMP can be installed using the following commands:

6. Extract the files in the downloaded archive:

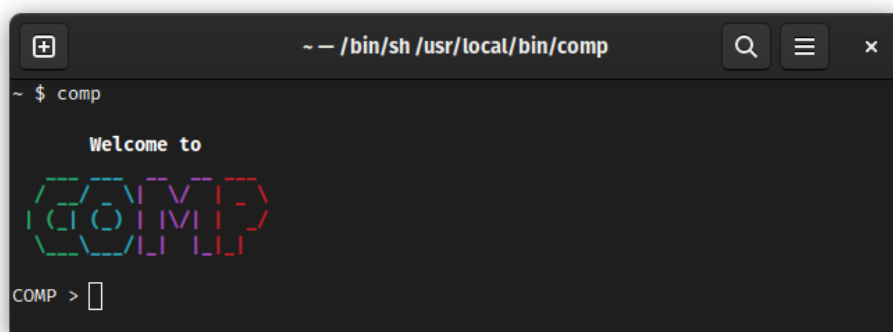
```
tar -xzf comp-22.09.tar.gz
```

7. Configure, make, and install COMP (from its source-tree directory):

```
cd comp-22.09/ && ./configure && make && sudo make install && cd -
```

Once more, similarly to the installation of SpeX, you may remove the downloaded archive and the directory `comp-22.09`. The COMP libraries should be installed to `/usr/local/share/comp/` and a shell script at `/usr/local/bin/comp` should be available in order to launch the tool – independently from SpeX.

You can now run COMP from the command line to be greeted with the following message. Any subsequent user input (declarations of system specifications or commands) under the `COMP >` prompt is meant to be handled by the COMP interpreter.



### 3 Writing COMP specifications

A COMP specification consists of a series of modules that introduce either data types or behavioural objects and are interrelated through various data-importation and object-composition operators. The full specification capabilities of the tool are presented in the language-definition document of COMP, which is available at the COMP homepage. Here, we focus instead on some of the main specification features of the language. We introduce them by means of a simple example of a wrist watch.

The object we have in mind has states that may change under three possible actions: (a) `tick`, which governs the inner workings of the watch; (b) `inc-min`, which indicates that one of the push-buttons of the watch has been pressed in order to adjust its minute hand; and (c) `inc-hour`, which indicates that one of the push-buttons of the watch has been pressed in order to adjust its hour hand. Furthermore, for each state, there are three observations we can make, namely the positions of the second, minute, and hour hands of the watch. The aim is to design/specify a wrist watch where none of the two push-buttons interfere with timekeeping.

#### Data-type declarations

First, we specify what kind of values can be displayed by the wrist watch. These are natural numbers that the watch increments – while it runs – up to specific bounds: 23 in the case of the hour indicator, 59 for the minute and second indicators. We capture displayed values using a binary operation on natural numbers, `inc-or-reset`, that returns either the successor of its first argument when, by doing so, it yields a value less than its second argument (as in modular arithmetic), or 0 otherwise.

To code such an operation, we begin by loading the file `Nat.comp`, which provides basic support for working with natural numbers and is available in the COMP library of examples. We load the file by typing the following command at the COMP prompt.

```
load Nat.comp
```

For this to succeed, a copy of the file needs to be saved in the working directory where COMP has been launched; alternatively, you can provide an explicit path (relative to the working directory, or absolute) to `Nat.comp`.

We specify the values displayed by the wrist watch by declaring a new data-type module, called `DISPLAY/VALUES`, as follows (the code can be typed directly at the COMP prompt or to an external file that is then loaded into COMP):

```
data DISPLAY/VALUES is
```

As it stands, the module declaration is incomplete. We proceed by importing the

module NAT/ORD, which provides knowledge about natural numbers and their typical order relations (needed for reasoning about upper bounds).

```
protecting NAT/ORD .
```

The import is `protecting` (as opposed to `extending` or `including`), meaning that none of the declarations that follow can alter the data types in NAT/ORD in any way. Most data-type imports are of this kind.

We can now declare the `inc-or-reset` operation and define it using two axioms: one matching the case where the displayed value (first argument) can be safely incremented without exceeding the bound (second argument); and one matching the case where the displayed value needs to be reset to 0.

```
op inc-or-reset : Nat Nat -> Nat .
vars N, B : Nat .
ax inc-or-reset(N, B) = s N if s N < B = true .
ax inc-or-reset(N, B) = 0 if s N < B = false .
```

Besides `inc-or-reset`, we also introduce constants used in the later part of the specification of the wrist watch to refer to certain natural numbers, whose normal-form representations are otherwise too cumbersome to write – obtained from 0 through repeated applications of the successor operation. To that end, we need the addition and multiplication of natural numbers, which we import from NAT/MULTIPLICATION.

```
protecting NAT/MULTIPLICATION .
ops 2, 5, 12, 24, 58, 59, 60 : () -> Nat .
ax 2 = s s 0 .
ax 5 = s s s s 0 .
ax 12 = (2 * 5) + 2 .
ax 24 = 2 * 12 .
ax 58 = (24 + 5) * 2 .
ax 59 = s 58 .
ax 60 = 5 * 12 .
```

We signal the completion of the data-type module by typing `enddata` at the COMP prompt. In response, the interpreter prints an appropriate message (`Loading module`) indicating that the module `DISPLAY/VALUES` has been loaded successfully and can now be used in subsequent declarations and/or tests, proofs, etc.

```
enddata
```

We can check the loaded definition of `DISPLAY/VALUES` (or of any other data-type or behavioural-object module) by asking the interpreter to print it on the screen.

```
show module DISPLAY/VALUES
```

Moreover, we can inspect the sorts, operations, and axioms declared in `DISPLAY/VALUES`, and also test the definition of `inc-or-reset` using term rewriting. For example:

```
open DISPLAY/VALUES
  list operations
  reduce inc-or-reset(2, 60) .
  reduce inc-or-reset(59, 60) .
close
```

Here, the first command opens the module, and in this way instructs the interpreter to expect a different kind of input, where commands, e.g., are specific to data-type modules, and where the non-logical symbols used are those declared in `DISPLAY/VALUES`. The two `reduce` commands should return 3 – i.e., `s s s 0` – and 0, respectively. Finally, we close the opened module to allow for further module declarations.

## Behavioural-object declarations

The wrist-watch system we intend to specify consists of two kinds of counters as subobjects. More precisely, it consists of one counter up to 24 (for hours) and two counters up to 60 (used for minutes and seconds). We introduce these as objects with explicit states, referred to by the sort `Display`, only one action (for incrementing the counter) and one observation (retrieving the value of the counter in a given state).

```
bobj UP-T0-24-COUNTER with states Display is
  protecting DISPLAY/VALUES .
  obs value : Display -> Nat .
  act inc_ : Display -> Display .
  ax value(inc X:Display) = inc-or-reset(value(X:Display), 24) .
endbo

bobj UP-T0-60-COUNTER with states Display is
  protecting DISPLAY/VALUES .
  obs value : Display -> Nat .
  act inc_ : Display -> Display .
  ax value(inc X:Display) = inc-or-reset(value(X:Display), 60) .
endbo
```

These are base objects – i.e., without components – hence, according to the specification methodology of COMP (see the language-definition document) we need to axiomatize how the observable value of a counter changes under the effect of the `inc` action. That is the role of the two equations declared in the modules `UP-T0-24-COUNTER` and `UP-T0-60-COUNTER`. The only difference between them is that the upper bound used for `inc-or-reset` is 24 in one case and 60 in the other.

We are now ready to specify the watch as a compound object with three counters:

```

bobj WATCH is
  syncing (UP-T0-24-COUNTER as HOUR)
    and (UP-T0-60-COUNTER as MINUTE)
    and (UP-T0-60-COUNTER as SECOND) .

```

We use the `as` construct to give names to the three component objects of `WATCH` – i.e., within `WATCH`, they are referred to using the names `HOUR`, `MINUTE`, and `SECOND`; and we synchronize the three components, instead of using a simple parallel composition, because incrementing the second or the minute counters may trigger, when their values are reset to 0, an automatic incrementation of the minute or hour counters.

The states of the object `WATCH` are in this case implicit because we haven't included any `with states` declaration. We can refer to them using the sort `State` (introduced automatically by the interpreter), as in the following declaration of variables:

```

var X : State . vars H, M, S : Nat .

```

The main action of the object `WATCH`, which captures the operation of the inner ticking mechanism of the wrist watch, is defined as follows:

```

act tick_ : State -> State .
ax HOUR/Display(tick X) = inc HOUR/Display(X)
  if MINUTE/value(X) = 59 and SECOND/value(X) = 59 .
ax HOUR/Display(tick X) = HOUR/Display(X)
  if not MINUTE/value(X) = 59 or not SECOND/value(X) = 59 .
ax MINUTE/Display(tick X) = inc MINUTE/Display(X)
  if SECOND/value(X) = 59 .
ax MINUTE/Display(tick X) = MINUTE/Display(X)
  if not SECOND/value(X) = 59 .
ax SECOND/Display(tick X) = inc SECOND/Display(X) .

```

In line with the specification methodology of COMP, we need to axiomatize how the action `tick` affects the three counter components of the watch. This is achieved through a series of equations (some of which are conditional) where the components of `WATCH` are accessed using the projections `HOUR/Display`, `MINUTE/Display`, and `SECOND/Display` – which are generated automatically by the interpreter.

The remaining two actions, used for setting the watch by separately adjusting (incrementing) the minute and the hour indicators, are defined as follows:

```

act inc-min_ : State -> State .
ax HOUR/Display(inc-min X) = HOUR/Display(X)
  if not MINUTE/value(X) = 59 .
ax MINUTE/Display(inc-min X) = inc MINUTE/Display(X) .
ax HOUR/Display(inc-min X) = inc HOUR/Display(X)
  if MINUTE/value(X) = 59 .
ax SECOND/Display(inc-min X) = SECOND/Display(X) .

act inc-hour_ : State -> State .
ax inc-hour X = inc X :: HOUR .

```

Note the use of the `::` syntax in the axiomatization of `inc-hour`. This indicates that only the hour value of the state `X` is updated according to the action `inc` from the component `HOUR`, which is of kind `UP-TO-24-COUNTER`. No other component is affected.

We conclude the definition of `WATCH` with the declaration of a macro operation (which we write `H : M : S`) that makes it easier to refer to watch states with specific hour (`H`), minute (`M`), and second (`S`) values when performing tests or doing proofs.

```

op _:_:_ : Nat Nat Nat -> State .
ax value(HOUR/Display(H : M : S)) = H .
ax value(MINUTE/Display(H : M : S)) = M .
ax value(SECOND/Display(H : M : S)) = S .
endbo

```

Similarly to data-type modules, we can open object modules in order to inspect their sorts, operations, axioms, and to perform tests by term rewriting. In addition, for object modules, we also gain access to the states, projections, actions, and observations that are automatically generated by the interpreter when declaring compound objects. We can execute, for instance, the following commands to get all the actions and observations of the object `WATCH`.

```

open WATCH
  list State actions
  list State observations
close

```

This yields, besides the actions `tick`, `inc-min`, and `inc-hour` that we have explicitly declared in `WATCH`, three other implicit actions and observations, which are automatically generated by the interpreter; each of them matches an action (`inc`) or an observation (`value`) from one of the three components of the watch.



## 4 Verifying properties

COMP supports the verification of behavioural properties of objects through a **check** command, which becomes available once an object module is opened for verification. The most general form of the command is:

$$\mathbf{check} \langle \mathit{property} \rangle [\mathbf{forall} \langle \mathit{pre-constr} \rangle] [\mathbf{given} \langle \mathit{post-constr} \rangle] .$$

where  $\langle \mathit{property} \rangle$  is the property (behavioural equality) to be verified, and  $\langle \mathit{pre-constr} \rangle$  and  $\langle \mathit{post-constr} \rangle$  are optional constraints on the variables used in  $\langle \mathit{property} \rangle$  and on the variables generated during the automatic verification (for indexed compositions), respectively. Those constraints are arbitrary sentences; they may contain strict or behavioural equalities, negations, conjunctions, and so on.

To illustrate the verification process, we consider the following property of **WATCH**:

*Pressing the minute-incrementation pusher does not interfere with the internal ticking of the watch.*

Formally, what we need to verify is that the behavioural equality

$$\mathbf{tick} \ \mathbf{inc-min} \ (H : M : S) \sim \mathbf{inc-min} \ \mathbf{tick} \ (H : M : S)$$

holds (i.e., it is semantically entailed by the specification) for all possible values  $H$ ,  $M$ , and  $S$  of the hour, minute, and second counter, respectively, of the watch.

We begin by opening the object module **WATCH** and by adding (using the command **let**) two lemmas on natural numbers that we use during the verification.

```
open WATCH
let ax not M:Nat = N:Nat if M:Nat < N:Nat = true [label: Lemma-1.1] .
let ax M:Nat < s N:Nat = true if M:Nat < N:Nat = true [label: Lemma-1.2] .
```

The first one states that the ‘less than’ relation on natural numbers is irreflexive; and the second that increasing an upper bound of a number yields another upper bound.

Next, we let  $<58$  and  $<59$  be any two natural numbers that are less than 58 and less than 59, respectively. We need these during checking in order to account for situations where the second or the minute indicator can be safely updated without triggering an update of the minute or hour indicator. Similarly to the two lemmas above, we introduce the two constants and their defining properties using **let**.

```
let ops <58, <59 : () -> Nat .
let ax <58 < N:Nat = true if N:Nat = 58 .
let ax <59 < N:Nat = true if N:Nat = 59 .
```

With these preparations in place, we can verify that there is no interference between `inc-min` and `tick` simply by invoking `check`; but we must still distinguish several cases, depending on whether `M` is at most 57, or 58, or 59, and also on whether `S` is at most 58, or 59. The final form of the `check` command is as follows:

```
check tick inc-min (H:Nat : M:Nat : S:Nat)
  ~ inc-min tick (H:Nat : M:Nat : S:Nat)
forall (M:Nat = <58 or M:Nat = 58 or M:Nat = 59)
  and (S:Nat = <59 or S:Nat = 59) .
```

If all goes well (as it is the case in this example), we get a successful message of the form: `Proved! The property holds.` Otherwise, the interpreter provides a proof trace that can be used for debugging purposes, which we discuss in the next section.

To get a more in-depth view of the verification process, we can ask the interpreter to print how many equalities are examined during an execution of `check` – that is because, according to the verification methodology of `COMP`, a behavioural equality between states of a compound object is often component-wise decomposed into behavioural equalities between the projections of those states; moreover, a behavioural equality between states of a base object is also decomposed into a sequence a strict equalities.

```
show check stats
```

Executing the above `check` command once more, we notice that 6 behavioural equalities are examined at the top level of the object hierarchy (because we consider three possible instances of the variable `M` and two possible instances for `S`) and 18 behavioural equalities are examined at the base level (because each of the 6 top-level equalities is decomposed along the projections `HOURL/Display`, `MINUTE/Display`, and `SECOND/Display` into three other equalities, one for each component of `WATCH`).

The scope of `show check stats` is local to the last opened module and it ends once the module is closed. Other opened modules are unaffected by it. You can also restore the original abridged `check` messages using the command `hide check stats`.

```
close
```

## 5 Debugging facilities

Consider a second property of the wrist watch, namely that pressing the hour-incrementation pusher does not interfere either with the internal ticking of the watch. Before anything else, we attempt to check the property directly:

```
open WATCH
  check tick inc-hour (H:Nat : M:Nat : S:Nat)
    ~ inc-hour tick (H:Nat : M:Nat : S:Nat) .
```

The status of this verification is `Open`: neither proved nor disproved. The interpreter provides a proof trace to indicate what went wrong. In this case, the trace consists of two behavioural equalities (one for the top level of the object hierarchy, one for the lower level of the hierarchy) and one final strict equality:

```
tick inc-hour(H:Nat : M:Nat : S:Nat)
  ~?~ inc-hour tick(H:Nat : M:Nat : S:Nat)

HOUR/Display(tick inc-hour(H:Nat : M:Nat : S:Nat))
  ~?~ HOUR/Display(inc-hour tick(H:Nat : M:Nat : S:Nat))

value(HOUR/Display(tick inc H:Nat : M:Nat : S:Nat :: HOUR))
  =?= inc-or-reset(value(HOUR/Display(tick(H:Nat : M:Nat : S:Nat)))), s...)
```

We notice that the `HOUR` projection of the state in the left-hand side of the original equality cannot be evaluated. That is because, based on the axiomatization of `HOUR/Display(tick ...)`, the prover cannot determine whether the minute and second indicators point to 59. Hence, we split the verification task in order to account for situations where the values of those indicators are at most 58, or 59.

```
let op <59 : () -> Nat .
let ax <59 < N:Nat = true if N:Nat = 59 .
check tick inc-hour (H:Nat : M:Nat : S:Nat)
  ~ inc-hour tick (H:Nat : M:Nat : S:Nat)
forall (M:Nat = <59 or M:Nat = 59)
  and (S:Nat = <59 or S:Nat = 59) .
```

The status of the verification is still `Open`, but for a different reason: the prover cannot infer that `<59` is distinct from 59. We address this limitation by adding the irreflexivity of the ‘less than’ relation as a lemma – just like in the previous section.

```
let ax not M:Nat = N:Nat if M:Nat < N:Nat = true [label: Lemma-2.1] .
```

Running the last **check** command once more gives us a positive result: the property holds. So, we have seen how proof traces can help us guide the verification tool using case analysis or to extend its knowledge base with lemmas. In the same manner, proof traces of desirable properties that cannot be proved or are shown not to hold can also be used to discover faults in the original design/specification.

Assume, for example, an alternate (faulty) design of adjusting the minutes of the watch, with only one equation instead of the four original ones:

```
ax inc-min X = inc X :: MINUTE .
```

Under this formalization, the property analysed in Section 4 no longer holds. Instead, running the same proof steps as before gives us the following proof trace:

```
tick inc-min(H:Nat : 58 : 59)
  ~?~ inc-min tick(H:Nat : 58 : 59)

HOUR/Display(tick inc-min(H:Nat : 58 : 59))
  ~?~ HOUR/Display(inc-min tick(H:Nat : 58 : 59))

inc-or-reset(H:Nat, 24) =?= H:Nat
```

Clearly, the last equality does not hold, which is because applying `inc-min` after `tick` in the state `(H:Nat : 58 : 59)` does not update the value of the hour indicator – although it should. This is a consequence of erroneously defining `inc-min X` as `inc X :: MINUTE`, which is local to the component `MINUTE` and has no effect on `HOUR`.

## 6 Troubleshooting

The COMP interpreter analyses user input in a modal manner, meaning that the notion of valid input varies depending on the context in which that input is evaluated: at system-level, within the declaration of a module (and in that case, it depends on the kind of module being declared), or after opening a module. This is a key feature of COMP that the interpreter inherits from SpeX, the formal-specification environment where COMP is implemented.

When the input is not adequate to the current context, or if it contains errors, the interpreter signals the issue through an appropriate warning message; moreover, it indicates where the error occurs (at the standard input / command line, or in a file, and at which line), and what is causing it. For example, trying to load an inexistent file (`WATCH.comp` as opposed to `Watch.comp`) produces the message:

```
Warning: <standard input>, line 1: No such file or directory: WATCH.comp
```

And trying to show the definition of the module `WATCH` immediately after the failed `load` command ends with:

```
Warning: <standard input>, line 2: no such module: WATCH
```

Similarly, the command `list State actions` is not recognized at system-level or after opening a data-type module. Even after opening an object module, the interpreter still checks that `State` is a valid state sort. Executing the command after opening the module `UP-T0-24-COUNTER`, for example, where states have the sort `Display`, produces:

```
Warning: <standard input>, line 7: State is not a state sort
```

However, `list Display actions` is successful in this context and returns, as expected:

```
act inc_ : Display -> Display
```

When parsing terms, axioms, properties, constraints, etc., where the syntax is user defined and depends on the algebraic signature of the current module, the COMP interpreter also points to the most likely cause of error within the user input. For instance, if we execute, after opening the module `WATCH`, the command

```
check tick inc-min (H : M:Nat : S:Nat) ~ inc-min tick (H : M:Nat : S:Nat) .
```

without specifying that `H` is a variable, and without declaring it as a constant beforehand, we get, of course, a parsing error:

```
Warning: <standard input>, line 11: unexpected token: H
```

```
|  
| check tick inc-min (H <--- here  
| _____
```

## 7 List of COMP commands

### System-level commands

**load** *<file-name>* Loads the contents of a specified file.

**eof** Prevents the interpreter from reading any further from the current input stream or file. If that stream is the standard input / command line, then the command also terminates the execution of the interpreter.

**quit** Terminates the execution of the interpreter.

### Top-level commands

**list modules** Lists the names of all modules that are currently recorded into the COMP database.

**list data modules** Lists the names of all data-type modules that are currently recorded into the COMP database.

**list bobj modules** Lists the names of all behavioural-object modules that are currently recorded into the COMP database.

**show module** *<module-name>* Displays the recorded definition of a given module.

**list opened modules** Lists the names of all opened modules.

**open** *<module-name>* Opens a given module and changes the input-evaluation context according to the kind and contents of that module. If the module has already been opened (without being closed in the meantime), then the command brings the input-evaluation context of that module back into foreground.

**close** Closes the most recently opened module and restores the previous input-evaluation context.

### Commands available after opening a data-type module

**list declarations** Lists all the declarations in the module: sorts, operations, axioms.

**list sorts** Lists all sorts declared in the module.

**list operations** Lists all operations declared in the module.

**list axioms** Lists all axioms declared in the module.

**let**  $\langle declaration \rangle$  . Extends the last opened module with a given declaration.  
**reduce**  $\langle term \rangle$  . Reduces the given term according to the axioms in the module.

## Commands available after opening an object module

**list declarations** Lists all the declarations in the module: data-type sorts, state-sorts, operations, projections, actions, observations, etc.  
**list sorts** Lists all data-type sorts declared in the module.  
**list states** Lists all state sorts declared in the module.  
**list data operations** Lists all data operations declared in the module.  
**list**  $\langle state \rangle$  **projections** Lists all projections of a given state sort declared in the module.  
**list**  $\langle state \rangle$  **actions** Lists all actions of a given state sort declared in the module.  
**list**  $\langle state \rangle$  **observations** Lists all observations of a given state sort declared in the module.  
**list axioms** Lists all axioms declared in the module.  
**let**  $\langle declaration \rangle$  . Extends the last opened module with a given declaration.  
**reduce**  $\langle term \rangle$  . Reduces the given term according to the axioms in the module.  
**solve**  $\langle query \rangle$  . Finds solutions (answer substitutions) to a given query.  
**check**  $\langle property \rangle$  [**forall**  $\langle pre-constr \rangle$ ] [**given**  $\langle post-consts \rangle$ ] . Verifies a given property under (optional) pre-constraints on the variables used in the property and post-constraints on the variables generated during the verification process.  
**show/hide check stats** Determines whether the number of equalities examined during an execution of **check** is printed together with the outcome of **check**.

## Comments

**`**  $\langle comment \rangle$  Indicates the beginning of a line comment. It can be used anywhere in the input stream and is ignored by the interpreter.  
**\***  $\langle comment \rangle$  Indicates the beginning of an echoed line comment. It can be used only at the beginning of a new line, and the comment is displayed (these are useful, e.g., when loading proof scores from file).  
**\*** ( $\langle comment \rangle$ ) Similar to **\***  $\langle comment \rangle$ , but the comment may span multiple lines.