Language Overview for PHAVer version 0.35

Goran Frehse

June 22, 2006

We have tried to construct a textual input language that is as user friendly as possible, while keeping the parser simple. In the syntax, we have borrowed extensively from the creators of HyTech [HHWT97], since their language is intuitively understandable. The following sections describe the syntax of PHAVer's representation of automata, states, relations etc., and brief descriptions of the analysis commands, followed by a section on the user-definable parameters. For formal definitions of hybrid automata, states, polyhedra, and PHAVer's algorithms please refer to [Fre05b, Fre05a].

1 General

Comments are preceded by either `//`, `--`, or enclosed in `/*` ... `*/`. An *identifier* is a letter plus any combination of letters, digits and the characters `_` (underscore) and `~` (tilde), where `~` is designated for joining the identifiers of locations of composed automata. A number can be given in floating point format, e.g., `3.14` or `6.626e-34`, or as a fraction, e.g., `9/5`. Note that numbers are internally represented as exact rationals, and no conversion to binary floating point format takes place (which would lead to rounding errors). E.g., the input `0.1` is parsed and represented in PHAVer as 1/10, while a 64-bit floating point representation of 0.1 would actually be the number

0.1000000000000000055511151231257827021181583404541015625.

Commands, constants, parameters and automata can occur in an arbitrary sequence. A command is terminated by `;` (semi-colon).

2 Constants

Constants are defined in the form *identifier* `:=` *expression*`;` where *expression* is any combination of expressions, identifiers and numbers with `+`,`-`,`/`,`*`,`(`,`)`.

3 Data Structures

There are four types of data structures that can be assigned to identifiers: linear formulas, sets of symbolic states, symbolic relations and automata. A *linear expression* is specified over an arbitrary set of variables, numbers and constants that can be combined using `+,-,/,*,(,)` as long as it yields a linear expression over the variables. I.e., it is not allowed to multiply two variables, or divide by a variable, and the attempt to do so will result in an error message. A linear constraint is a combination of two linear expressions with one of the signs `<,>,<=,>=,==`. A *convex linear formula* is given as a conjunction of linear constraints that are joined by `&` (ampersand). A *linear formula* is a disjunction of convex linear formulas joined by `|`. Brackets `(,)` can be used to avoid ambiguities. A linear formula can be assigned to a variable in the form *identifier* `=` *linear formula*`;`.

A *symbolic state* is a combination of a location name and a linear formula, joined by `&`, e.g., `start & x>0 & y==0`. A *set of symbolic states* is a list of symbolic states, joined by `,` (commata). A set of symbolic states of an automaton *aut* is assigned to a variable in the form *identifier* `=` *aut*`.{` *set of symbolic states* `};` Location names of automata in composition are concatenated with $\sim$ as a separator, e.g. as in `off`$\sim$`stop`. The use of the wildcards $ and ? is permitted to specify sets of symbolic states. Use the separator $\sim$ to determine which automaton in the composition the location refers to. E.g., for four identical automata $A_1||A_2||A_3||A_4$ with locations `on,off`, the identifier `$~off~$~$` refers to all locations in which A_2 is in `off`, `$~$~off~$` to those in which A_3 is in `off`, and `$~off~$` refers to both cases.

Remark Wildcards are purely syntactically understood by PHAVer and are therefore to be used with extreme caution! E.g., for sets of symbolic states `reg1=aut.{$ & true}`, `reg2=aut.{loc1 & true}`, `reg1.difference_assign(reg2)` results in `reg1=aut.{$ & true}`, while `reg2.difference_assign(reg1)` yields the empty set, `reg2=aut.{}`.

Symbolic relations are returned by the simulation relation algorithms. There are currently no provisions for specifying relations. Identifiers can be assigned to other identifiers simply using `=`.

4 Automata

In the following, let *var_ident* be an identifier defining a variable, *loc_ident* a name for a location, *label_ident* an identifier defining a synchronization label. An automaton with identifier *aut* is specified in the following form:

```
automaton aut
   contr_var:  var_ident, var_ident, . . . ;
   input_var:  var_ident, var_ident, . . . ;
   parameter:  var_ident, var_ident, . . . ;
   synclabs:   lab_ident, lab_ident, . . . ;
   loc loc_ident:while invariant wait { derivative };
       when guard sync label_ident do {trans_rel} goto loc_ident;
       when guard sync label_ident goto loc_ident;
       when . . .
   loc loc_ident:   while . . .
   initially:  initial_states;
end
```

The *invariant* and the *guard* are linear formulas over the state and input variables and the parameters. The *derivative* definition depends on the dynamics:

- For "linear" (LHA) dynamics, it is a convex linear formula over the state variables. E.g., `0 <= x' & x' < 1` for $\dot{x} \in [0, 1)$.
- For affine dynamics, it is a convex linear formula over the variables and their derivatives. The differentiated variables are indicated by `'` (single quote), e.g., `x' == -2 * x` for $\dot{x} = -2x$.

Note that parameter uncertainties can be incorporated by using inequalities.

A transition is specified by `when` ... `goto` statements. There always must be a synchronization label associated with the transition. A linear formula *trans_rel* specifies the jump relation μ after a `do` statement, where the post-transition value of the variable is indicated by `'` (single quote). State variables that are not changed by the transition must be specified explicitly in the jump relation, e.g., `x'==x & y'==y`. [1] The reset of a variable to 0 would be defined, e.g., by `z'==0`. If no jump relation μ is specified, the controlled variables remain constant and the inputs may change arbitrarily. The restriction that parameters remain constant is always added automatically to the jump relation. Instead of a single label per transition one may specify a comma-separated list, which adds one copy of the transition per label.

Automata are composed using `&` (ampersand), e.g., `comp_aut = aut1 & aut2;`

The initial states are specified as a set of symbolic states after `initially:`, which may include wildcards.

5 Commands

PHAVer provides commands for computing reachable sets of states and simulation relations, plus a number of commands for the manipulation and output of data structures. In the following list, square brackets [] are used to

[1]Although this might seem tedious and is different from the HyTech language, we find it avoids errors and confusion when composing automata.

indicate optional arguments. *identifier* is used to denote the identifier for an arbitrary object, *formula_ident* for a linear formula, *state_ident* for a set of symbolic states, *rel_ident* for a symbolic relation and *aut_ident* for an automaton. Let *state_or_rel_ident* stand for either a set of symbolic states or a relation. Let *state_list* be an explicit comma separated list of symbolic states, e.g., `start & t==0, stop & t==1`. Recall that objects can be copied with an assignment *new_identifier* `=` *old_identifier*`;`

5.1 General

- `echo "`*text*`";`
  Displays *text* and starts a new line.

- `who;`
  Displays a list of identifiers currently in memory, as well as the sizes of automata.

- *identifier*`.print(`[`"`*file_name*`"`][`,`*method*]`);`
  If *file_name* is specified, writes a representation of *identifier* to the file *file_name*, otherwise to the standard output. An optional integer *method* determines the format:

  – `0`: (default) Location names and linear formulas are produced in textual form that can be read by PHAVer. E.g., `aut.print("test",0)` saves an automaton description in the file `test`. It can be parsed directly by PHAVer, e.g., using `phaver test`.

  – `1`: Output the linear formulas as a sequence of linear constraints in floating point form. Equalities $\phi = 0$, where ϕ is some linear formula, are converted to $\phi \geq 0 \wedge -\phi \geq 0$. The coefficients of a constraint $\sum a_i x_i + b \bowtie 0$ are output separated by spaces, one constraint per line. Convex formulas are separated by a blank line. No location information is given. This form can be used for output with polyhedral visualization packages. The order of variables is the same as in the list provided by the automaton output.

  – `2`: Output the linear formulas as a sequence of vertices in floating point form. The vertices belonging to a convex formula are separated by a blank line. No location information is given. This form can be used for output with plotting tools like `graph`. If 2-dimensional, the points are in counter-clockwise order and represent a closed line for each convex formula, i.e., the last point is equal to the first. The order of variables is the same as in the list provided by the automaton output. There is also a script `plot_2d_vertices.m` to plot this output with Matlab.

5.2 Reachability Analysis

- *state_ident*`=`*aut_ident*`.reachable;`
  Returns the set of states reachable in the automaton *aut_ident* from the initial states.

- *state_ident1*`=`*aut_ident*`.reachable(`*state_ident2*`);`
  Returns the set of states reachable in the automaton *aut_ident* from the states in *state_ident2*.

- *state_ident1*`=`*aut_ident*`.is_reachable(`*state_ident2*`);`
  Computes the set of reachable states, but stops as soon as a state in *state_ident2* is found. Returns the states that were found to be reachable at the time of termination.

- *state_ident1*`=`*aut_ident*`.is_reachable_fb(`*state_ident2*`);`
  Verifies whether *state_ident2* is reachable from the initial states using forward/backward refinement.

5.3 Simulation Checking

- *rel_ident*`=get_sim(`*aut_ident1*,*aut_ident2*`);`
  Returns a simulation relation for *aut_ident1*$\preceq$*aut_ident2*.

- `is_sim(`*aut_ident1*,*aut_ident2*`);`
  Computes a simulation relation and displays whether *aut_ident1*$\preceq$*aut_ident2*.

- `is_bisim(`*aut_ident1*,*aut_ident2*`);`
  Computes a simulation relation and displays whether *aut_ident1*$\preceq$*aut_ident2*.

- `ag_sim(`*aut_ident1*,*aut_ident2*,*aut_ident3*,*aut_ident4*`);`
  Computes a simulation relation using assume/guarantee reasoning and displays whether *aut_ident1* $||$ *aut_ident2* $\preceq$ *aut_ident3* $||$ *aut_ident4*.

5.4 Partitioning

- *aut_ident*`.set_partition_constraints((`*lin_expr1*,δ_{1min},δ_{1max}`),`
  `(`*lin_expr2*,δ_{2min},δ_{2max}`),...);`
  Defines the partitioning constraints used in subsequent analyses. A location will be split by a constraint of the form *lin_expr1*$\leq c$, where c is the center of the location with respect to the linear expression. δ_{1min} and δ_{1max} define the minimum and maximum extent of every location in the partitioning process. The constraints are prioritized according to the partitioning parameters. The maximum values, δ_{1max},..., may be omitted.

5.5 Queries

- *identifier1*`.contains(`*identifier2*`);`
  Writes whether the object *identifier2* is contained in the object *identifier1* of the same type to the standard output.

- *identifier*`.is_empty;`
  Writes whether the object *identifier* is empty to the standard output.

- *state_ident*`=`*aut_ident*`.inital_states;`
  Copies the initial states of *aut_ident* to *state_ident*.

- *state_ident*`=`*aut_ident*`.get_invariants;`
  Copies the invariants of *aut_ident* to *state_ident*.

5.6 Manipulation Commands

5.6.1 States and Relations

- *state_ident*`.rename(`*var_ident1*,*var_ident2*`);`
  Replaces the name of variable *var_ident1* with *var_ident2*.

- *identifier*`.remove(`*var_ident*,*var_ident*,...`);`
  Existential quantification over the specified variables.

- *identifier*`.project_to(`*var_ident*,*var_ident*,...`);`
  Existential quantification over all except the specified variables.

- *identifier*`.get_parameters(`*bool*`);`
  Performs existential quantification over state and input variables, i.e., non-parameters. A boolean parameter *bool* specifies the quantification over locations:

  – `false`: Disjunction, the parameters are common to all locations. E.g., compute the set of reachable states, intersect it with a set of desired states, and get the parameters for which all desired states are reachable with option `false`.

  – `true`: Conjunction, the parameters occur in any of the locations. E.g., compute the set of reachable states, intersect it with a set of forbidden states, and get the parameters for which any of the forbidden states are reachable with option `true`.

- *state_ident*`=`*state_or_rel_ident*`.loc_union;`
  Returns the states unified over the locations, attributed to the location wildcard `$`.

- *state_ident*`=`*state_or_rel_ident*`.loc_intersection;`
  Returns the states intersected over the locations, attributed to the location wildcard `$`.

- *identifier1*`.intersection_assign(`*identifier2*`);`
  Intersects *identifier2* with *identifier1* and puts the result into *identifier1*.

- *identifier1*`.difference_assign(`*identifier2*`);`
  Subtracts *identifier2* from *identifier1* and puts the result into *identifier1*.

- *rel_ident1*`=`*rel_ident2*`.inverse;`
  If *rel_ident2* is a relation R, then *rel_ident1* is assigned R^{-1}.

- *rel_ident1*`=`*rel_ident2*`.project_to_first;`
  If *rel_ident2* is a relation $R(p, q)$, then *rel_ident1* is assigned $R' = \{p | \exists q : R(p, q)\}$.

5.6.2 Automata

- *aut_ident*`.add_label(`*lab_ident*`);`
  Adds the label *lab_ident* to the alphabet of the automaton *aut_ident*. Can be used to add an extra label that is dedicated for partitioning to an existing model.

- *aut_ident*`.reverse;`
  Reverses causality (time and transitions) in the automaton. This may be used to implement backwards reachability by reversing the automaton, and then using standard forward reachability. Don't forget whether you want to change the initial states in the process.

- *aut_ident*`.initial_states(`*state_ident*`);`
  Replaces the initial states of *aut_ident* with *state_ident*.

- *aut_ident*`.invariant_assign(`*state_ident*`);`
  Replaces the invariants of *aut_ident* with *state_ident*.

6 Parameters

The following is a summary of the parameters used in PHAVer. A parameter is defined in the form *identifier* `=` *value*`;`. The default setting is given in brackets, and the type is boolean unless specified otherwise.

6.1 General

- `ELAPSE_TIME` (true): Can be used to switch off the time-elapse operator. Useful for the analysis of purely discrete systems, but the speed-up is modest.

6.2 Reachability Analysis

- `REACH_MAX_ITER` (0): Integer specifying the maximum number of iterations used, i.e., the number of discrete transitions explored. Only active if > 0. A value of -1 returns the initial states after partitioning and time elapse.

- `CHEAP_CONTAIN_RETURN_OTHERS` (true): Determines the type of containment test used:

  - false: exact, i.e., a convex polyhedron p is considered contained in a non-convex polyhedron q if the difference $p \setminus q$ is empty.
  - true: a convex polyhedron p is considered contained in a non-convex polyhedron $q = q_1 \cup \ldots \cup q_n$ (a union of convex polyhedra) if there is a convex polyhedron q_i in the union that contains it, i.e., $\exists i \in \{1, \ldots, n\} : q \subseteq q_i$. This method is generally faster than exact testing, although it results in more polyhedra.

- `USE_CONVEX_HULL` (false): Use convex-hull over-approximations. Highly recommended when using on-the-fly partitioning, and usually a good idea.

- `REACH_STOP_USE_CONVEX_HULL_ITER` (1000000000): Integer specifying the maximum number of iterations for which the convex-hull over-approximation is used. Can be set to a lower value to improve termination.

- `REACH_USE_BBOX` (false): Causes the over-approximation of the post-transition states with a bounding box. Can be used to force termination, but usually leads to excessive over-approximation.

- `REACH_USE_BBOX_ITER` (1000000000): Integer specifying the frequency n (a number of iterations) with which the bounding-box over-approximation is used. It is only applied at one iteration, then followed by n iterations with the normal setting. Can be set to a lower value to improve termination. Note that it is independent of `REACH_USE_BBOX`.

- `REACH_ONLY_EXPLORE` (false): Toggles a special *exploration* mode: There is no testing if newly reached states are contained in previous ones. Terminates only if the number of iterations is set with `REACH_MAX_ITER`.

- `SEARCH_METHOD` (0): Experimental feature, directing the search order according to 0: breadth-first incl. doubles (HyTech-like), 1: post horizon-based, 2: predecessor ratio, 3: depth-first, 4: breadth-first, 6: topological sort of reachable states, 7: topological sort of all states. So far, only 0, 6 and 7 are useful. Use 6 and 7 mainly with `PARTITION_CHECK_TIME_RELEVANCE` (`_DURING` and/or `_FINAL`) to true.

- `SNAPSHOT_INTERVAL` (0): Experimental feature. Writes every n iterations the currently reachable set into files. Projects to the first two variables.

6.3 Overapproximations

- `CONSTRAINT_BITSIZE` (0): Integer specifying the number of bits used in constraints, i.e., in the polyhedral computations. Equalities are not affected. If a constraint can not be specified with that amount of bits, a error is thrown. Only active if > 0.
- `REACH_BITSIZE_TRIGGER` (0): Integer threshold for limiting the number of bits used. The bits are reduced as specified in `CONSTRAINT_BITSIZE` only if they exceed this threshold. Can significantly improve termination and reduce the over-approximation, usually set to $10 - -30\times$ the limit.
- `REACH_STOP_USE_BITSIZE` (1000000000): Integer specifying the maximum number of iterations for which the number of bits are constrained. Can be set to a lower value to improve termination.
- `LIMIT_CONSTRAINTS_METHOD` (1): Integer specifying the method with which the number of constraints is reduced:
  - 0: constraints are evaluated according to slack (slow).
  - 1: constraints are evaluated according to angle (very fast).
- `REACH_CONSTRAINT_LIMIT` (0): Integer specifying the maximum number of constraints allowed in a convex polyhedron. Exceeding polyhedra will be over-approximated as specified by `LIMIT_CONSTRAINTS_METHOD`. The limiting is performed before the time-elapse operator, so that the resulting number of constraints can be higher. Usually set to at least 2^n if n is the number of variables. Only active if > 0.
- `REACH_CONSTRAINT_TRIGGER` (0): Integer threshold for limiting the number of constraints. A convex polyhedron is reduced to `REACH_CONSTRAINT_LIMIT` constraint once it exceeds this threshold. Can significantly improve termination, usually set to 2–3$\times$ the constraint limit. Boundedness of the polyhedron is preserved, with more constraints if necessary. Only active if > 0.
- `TP_CONSTRAINT_LIMIT` (0): Integer specifying the maximum number of constraints used for describing the derivative. Exceeding derivative formulas will be over-approximated as specified by `LIMIT_CONSTRAINTS_METHOD`. Boundedness of the polyhedron is preserved, with more constraints if necessary. Only active if > 0.

6.4 Simulation Checking

- `PRIME_R_WITH_REACH` (true): Initialize the simulation relation with the reachable states of $P||Q$.
- `USE_CONVEX_HULL_FOR_PRIMING` (true): Use convex-hull reachability for the initialization if `PRIME_R_WITH_REACH` is true.

- `PRIME_R_WITH_DISCRETE_REACH` (true): Over-approximating initialization of the simulation relation with the locations of $P||Q$ that are reachable by discrete transitions.
- `STOP_AT_BAD_STATES` (true): Stop as soon as bad states are encountered. Only useful if `PRIME_R_WITH_REACH=true`.
- `SHOW_BAD_STATES` (false): Output bad states as they are encountered.
- `SIM_SIMPLIFY_R` (true): Simplify the simulation relation, i.e., remove redundant polyhedra and join convex unions where possible, after each difference operation. Costly, but usually indispensable.

6.5 Partitioning

- `TIME_POST_ITER` (0): Integer specifying how many iterations are performed between reachable states and restricting the derivative to those reachable states. This number equals to $k-1$ in $Post^k_{t,pr}$. Higher numbers improve the accuracy of refined dynamics, but numbers > 0 require that the derivative is re-computed every time the cell is examined, which may slow down the analysis significantly.
- `PARTITION_CHECK_TIME_RELEVANCE` (true): Eliminate partitioning transitions that are never crossed by timed transitions and are therefore irrelevant *as they are created when a location is split.*
- `PARTITION_CHECK_TIME_RELEVANCE_DURING` (true): Eliminate partitioning transitions that are never crossed by timed transitions and are therefore irrelevant *testing all connected transitions when a location is split.* This helps to additionally remove transitions, helping convergence, memory and speed.
- `PARTITION_CHECK_TIME_RELEVANCE_FINAL` (true): Eliminate partitioning transitions that are never crossed by timed transitions and are therefore irrelevant, *testing all connected transitions when the location is split to minimum critera and will not be split further.* This is may be used alternatively or in addition to `PARTITION_CHECK_TIME_RELEVANCE_DURING`, with better or worse performance.
- `REFINE_DERIVATIVE_METHOD` (2): Integer determining how the set of derivatives is determined:
  - 0: constraint-based,
  - 1: bounding box of method 0,
  - 2: projection-based,
  - 3: bounding box of method 2.
- `PARTITION_PRIORITIZE_REACH_SPLIT` (false): Prioritize constraints that have reachable states strictly on both sides.

- `PARTITION_PRIORITIZE_ANGLE` (false): Prioritize constraints according to the maximum angle spanned by the derivative in the resulting locations.
- `PARTITION_SMALLEST_FIRST` (false): Prioritize constraints according to their extent in the location. The value false corresponds to largest first, which is volume optimal.
- `PARTITION_DERIV_MAXANGLE` (1): Floating point number $\lhd_{min}$. A location is only refined if $\lhd_{min} > \lhd_{deriv}(l, S)$, where $\lhd_{deriv}(l, S)$ is the max. angle between any two derivative vectors in the location. A value smaller than 1 results in a partitioning based loosely on the "curvature" of the derivative, typically between $0.85 - 0.99$.

References

[Fre05a] Goran Frehse. *Compositional Verification of Hybrid Systems Using Simulation Relations.* PhD thesis, Radboud University Nijmegen, 2005. available at `http://webdoc.ubn.ru.nl/mono/f/frehse_g/compveofh.pdf`.

[Fre05b] Goran Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control (HSCC'05), Mar. 9–11, 2005, Zürich, CH*, volume 2289 of *LNCS*. Springer, 2005. PHAVer is available at `http://www.cs.ru.nl/~goranf/`.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, December 1997.