

Stabhyli

August 15, 2013

1 Introduction

Stabhyli is a tool that automatically proves a hybrid systems stable. To do so, it uses a decompositional approach. The decompositional approach is based on the (discrete) graph structure large hybrid systems usually exhibit. In order to deal with the graph structure, a specialized modeling language called Hybrid Automatic Language (HAL) has been designed. This documentation is structured as follows. The Section 2 introduces HAL and Section 3 describes how to use the tool Stabhyli.

HAL is a language that is intended to be used in different contexts. Therefore at the current development state of Stabhyli not all language features are understood (and usually ignored) by Stabhyli. For example using HAL one can model probabilistic hybrid automata which will be a future focus of the development of Stabhyli. But currently the knowledge about probabilities is not exploited and thus over-approximated by nondeterminism. Features that are not yet understood or ignored are listed in Section 3.2.

To prove a hybrid system stable Stabhyli applies Lyapunov arguments to the system. The Lyapunov arguments imply that the system's energy (can be seen as distance to the equilibrium) is decreasing while system evolves. To obtain such arguments, a global Lyapunov function needs to be found. If such a function can be found, then the existence indicates the stability of the system no matter what the actual function is. To find such functions one has to generate constraint systems that representation the system's requirements on a Lyapunov function. If such a constraint system has a solution, then the existence of a suitable Lyapunov function is guaranteed. This process needs a detailed insight in the hybrid system as well as the generation of constraint systems. Thus proving a hybrid system involves multiple steps. First the analysis of the hybrid system, second the generation of a suitable constraint system, third attempting to solve the constraint system, and last interpreting the (non)-solution of the constraint system. These steps are combined and automated by Stabhyli. The result of Stabhyli is a simple answer. This answer can be either "The hybrid system is stable" or "The hybrid system can not be proven stable." Stabhyli can use three proof schemes (1) providing a single common global Lyapunov function for all modes, (2) providing a piecewise global Lyapunov function, and (3) providing a piecewise global Lyapunov function by decomposing the hybrid system. Using (3) Stabhyli gives feedback of which parts of the automaton caused the proof to fail.

A model of a hybrid system might not only include continuous variables that represent the plant (which usually need to be stabilized) but also variables that represent internal variables of the controller, such as a timer or a register that may contain a snapshot of other variables. Therefore, variables that need to converge can be marked as convergent in HAL. Note that for some modes the convergence of marked variables may depend even on the convergence of some unmarked variable. Thus for some modes variables that are not marked as convergent have to be treated as if they were marked convergent. Stabhyli automatically deduces for every mode if a variable needs to be marked as convergent, based on the variables the are marked by the user.

The templates that Stabhyli uses for the Lyapunov functions are polynomial. Therefore polynomial dynamics and polynomial predicates (for mode invariants and jump guards) are understood. Note that HAL can be used to model transcendental functions, but Stabhyli is not capable of dealing with such functions at the moment. Additionally not all polynomials can be transformed into a sums-of-squares which means that Stabhyli will fail to compile some polynomials.

2 Hybrid Automata Language (HAL)

HAL is an input language specialized to model hybrid systems as hybrid automata. In the following sections we will give the grammar of HAL. Basic rules are given in Backus-Naur Form and more complex structures are given as syntax diagrams in order to allow a visualization that is easier to read. We will start with the basic rules, followed by the rules of the objects a hybrid automaton consists of (namely modes and jumps), followed by the specification of interfaces and submodules (used to model hierarchical hybrid systems), and we will end with the overall structure of the file written in HAL.

2.1 Basic Nonterminals

The following grammar defines the basic nonterminals of the input language.

$\langle real \rangle ::= '-'? [0-9]^+ \mid '-'? [0-9]^* '.' [0-9]^+$

$\langle id \rangle ::= [a-zA-Z]^+ [a-zA-Z0-9]^*$

$\langle mathop \rangle ::= '-' \mid '+' \mid '*' \mid '/' \mid '^'$

$\langle comparator \rangle ::= '<' \mid '>' \mid '<=' \mid '>=' \mid '!=' \mid '=='$

$\langle logicop \rangle ::= 'and' \mid '&\&' \mid 'or' \mid '||' \mid 'nand' \mid 'xor' \mid 'equiv' \mid 'implies'$

<real> The nonterminal **<real>** matches a simple (real or integer valued) number. *Numbers* occur in mathematical expression used for jump guards, mode invariants, differential equations, updates, and probabilistic distributions.

<id> The nonterminal **<id>** matches simple identifiers. *Identifiers* occur as names for modes as well as names for variables in mathematical expression used for jump guards, mode invariants, differential equations, and updates.

<mathop> The nonterminal **<mathop>** matches the most commonly used mathematical operators. The *mathematical operators* $-$, $+$, $*$, and $/$ have their usual meanings and $^$ denotes the power operator.

<comparator> The nonterminal **<comparator>** matches all common comparators. *Comparators* will occur in boolean expression. The comparators have their usual meaning on real numbers e.g. $a \leq b$ evaluates to true if and only if a is less or equal to b . Note that the equality has to be written as $==$ in order to visually distinguish a comparison from the explicit setting of a variable to a value.

<logicop> The nonterminal **<logicop>** matches the most commonly used logical operators. *Logical operators* have their usual meanings. The operators **&&** and **||** are C++-style aliases for the operators **and** and **or**, respectively.

2.2 Expressions

The following grammar formally introduces *mathematical expressions* as well as *boolean expressions*.

```

<mathexp> ::= <real> | <id>
           | '-' <mathexp>
           | '(' <mathexp> ')'
           | <mathexp> <mathop> <mathexp>

<boolexp> ::= 'true' | 'false' | <mathexp> <comparator> <mathexp>
           | 'not' <boolexp>
           | '!' <boolexp>
           | <boolexp> <logicop> <boolexp>

```

<mathexp> The nonterminal **<mathexp>** recursively matches *mathematical expressions*. Mathematical expressions can be either real numbers, variable (denoted by their names), or combined using negation, parentheses, or mathematical operators. These kind of expressions occur in differential equations, boolean expressions, updates, and Lyapunov functions.

<boolexp> The nonterminal **<boolexp>** recursively matches *boolean expressions*. Boolean expressions can be either **true**, **false**, comparisons of mathematical expressions, or combined using negation (where **!** is a C++-style alias for **not**) and logical operators. Boolean expressions occur in jump guards and mode invariants.

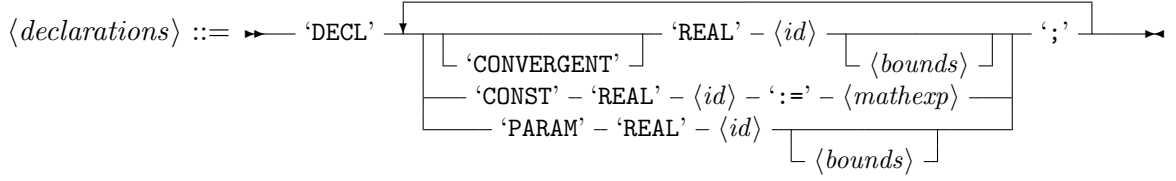
2.3 Declarations of Variables

The following syntax diagram formally defines *declarations of variables*.

```

<bounds> ::= ➡ 'IN' ┌─── '[' - <real> ────┐ , ┌─── <real> - ']' ───────────────────➡
                  │ ┌─── '(' - <real> ───┘ │ ┌─── <real> - '(' ─────────────────┘
                  │ │ ┌─── '(' - '[' - 'INF' ─┘ │ ┌─── 'INF' - '(' ───────────┘
                  │ │ │ ┌─── '(' - '[' - 'INF' ─┘ │ ┌─── 'INF' - '(' ───────────┘

```



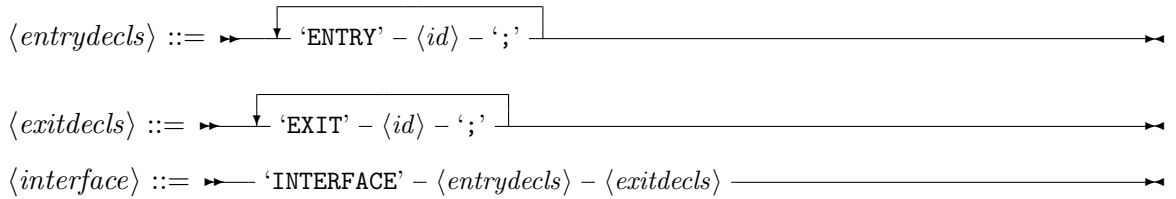
<bounds> The nonterminal **<bounds>** matches unbounded, half-bounded, and finite intervals which are either closed, half-open, or open. *Intervals* occur only in declarations and are used as global invariants on variables. Global invariants have to be conjugated with mode invariants and jump guards.

<declarations> The nonterminal **<declarations>** matches a *list of declarations of variables*. It allows the user to define three types of variables: (1) continuous variables, (2) constants, and (3) parameters. All three types are real-valued. Type (1) can optionally be declared as convergent which means that the variable has to converge in the stability analysis. Variables that are not marked as convergent will only be included if the convergence of the variable is crucial for the convergence of another variable which is in turn marked as convergent. Furthermore, type (1) allows the user to specify an interval which acts as a global invariant for all modes invariants and jump guards. Type (2) allows to define constants that will be replaced by their value in all occurrences. Type (3) allows the user to define parameters of the hybrid system. Parameters can have optional bounds specified by an interval.

2.4 Declaration of an Interface

HAL allows to create interfaces for modules. Using this feature, the entire automaton acts as a module which can be used as a submodule in other automata. This allows the user to hierarchically create automata comparable to calling functions in high-level programming languages. Such submodules (functions) can be entered (called) on entry ports and exited (left).

The following syntax diagram formalizes the *definition of an interface*.



<entrydecl> The nonterminal **<entrydecl>** matches a *list of declarations of entry ports*. An entry port consists only of a name specified by $\langle id \rangle$. An entry port can be used within the definition of the automaton as the source of a jump. The meaning of an entry port is as follows: Each time the entry is activated by an automaton which uses this automaton as a submodule, one of the jumps connected to the entry port will be (nondeterministically) taken.

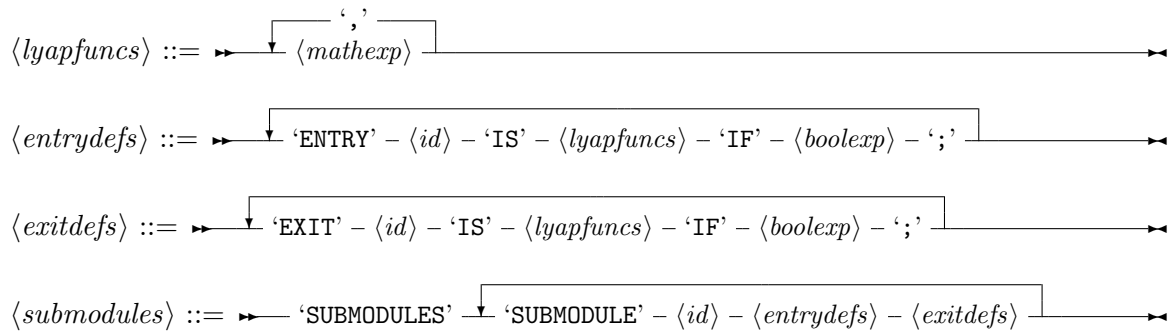
<exitdecl> The nonterminal **<exitdecl>** matches a *list of declarations of exit ports*. An exit port is similar to an entry port except that it can only be used as the target of a jump. The meaning of an exit port is, that as soon as the exit port gets activated the control is handed back to the automaton on the next higher hierarchical level, where the exit port will be used as the source of a jump.

<interface> The nonterminal **<interface>** matches an *interface for a submodule*. An interface specifies how the control can be handed over to a submodule. It can be entered on entry ports and will only be left on exit ports.

2.5 Definition of Submodules

HAL allows the user to use submodules in the automaton specified via interfaces. A submodule consists of multiple entry ports and exit ports. Both types of ports have associated Lyapunov functions used to perform the composition. Entry (resp. exit) ports additionally have an entry (resp. exit) condition. The definition of a submodule should be auto-generated from an interface. Thus there is no need to have a detailed insight in the Lyapunov function's meaning.

The following syntax diagram formalizes *definitions of submodules*.



<lyapfuncs> The nonterminal **<lyapfuncs>** matches a *list of Lyapunov functions* given as mathematical expressions. It allows the user to supply Lyapunov function arguments for submodules used for the stability verification. Lyapunov functions occur in definitions of input ports and output ports of submodules.

<entrydefs> The nonterminal **<entrydefs>** matches a *list of definitions of entry ports* of a submodule. An entry port is identified by its name within the automaton's definition. An entry port is active if the entry condition specified by **<boolexp>** evaluates to true. Entry ports can be used as the targets of jumps. The definition of an entry port occurs in the definition of a submodule.

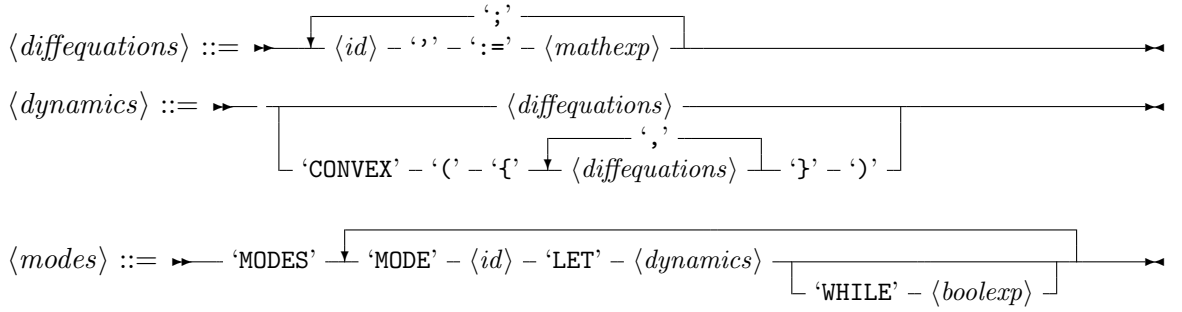
<exitdefs> The nonterminal **<exitdefs>** matches a *list of definitions of exit ports* of a submodule. An exit port is similar to an entry port except that an exit port can be used as the source of a jump. As definitions of entry ports, the definition of an exit port occurs in the definition of a submodule.

<submodules> The nonterminal **<submodules>** matches a *list of definitions of submodules* separated by semicolons. Each *submodule* consists of a name specified by **<id>**, a list of entry ports, and a list of exit ports. Note that the number of Lyapunov functions of each port (entry and exit) has to be equal and the order is important because the i -th Lyapunov function of each port needs to be a valid combination of Lyapunov functions for the submodule.

2.6 Definition of Modes

A *mode* describes the evolution of continuous variables. Modes usually model discrete states, where differential equations or convex differential inclusions inside a mode describe the continuous evolution of the variables. A convex differential inclusion is a convex combination of differential equations per variable. Therefore a set of differential equations per variable can also be viewed as a convex differential inclusion with a single corner point. A trajectory of the hybrid system evolves along a convex differential inclusion as long as the mode's invariant evaluates to true. A trajectory might jump from a mode to another mode as long as the jump guard evaluates to true (see Section 2.7).

The following syntax diagram formalizes *definitions of modes*.



<diffequations> The nonterminal **<diffequations>** matches a *list of differential equations* one for each variable separated by semicolons. Differential equations occur in dynamics. Note that multiple differential equations for a single variable are not allowed.

<dynamics> The nonterminal **<dynamics>** matches continuous *dynamics*. Dynamics are either a single list of *differential equations* or a *convex differential inclusion*. A *convex differential inclusion* consists of lists of differential equations where each list is separated by commas. Dynamics occur in a definition of a mode.

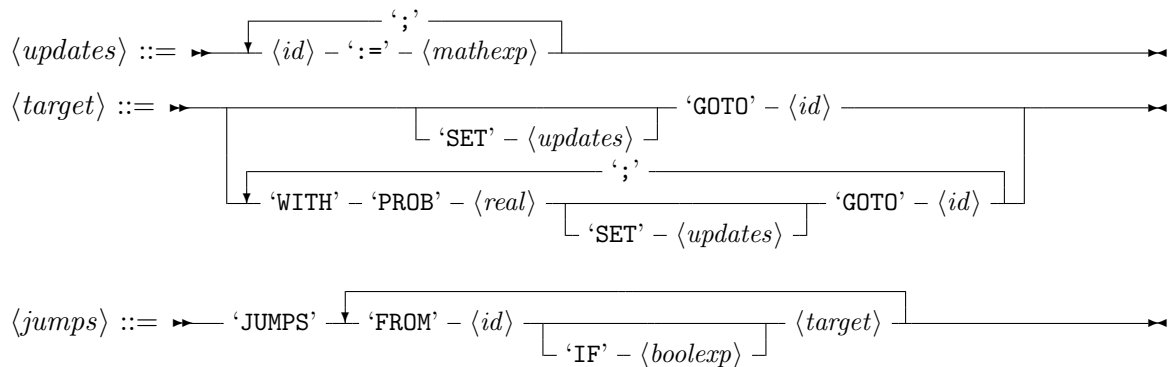
<modes> The nonterminal **<modes>** matches a *list of definitions of modes*. A single mode consists of a name **<id>**, continuous dynamics **<dynamics>**, and an optional mode invariant **<boolexp>**. Definitions of modes occur in the definition of the automaton.

2.7 Definition of Jumps

A *jump* is what is called a transition in the domain of automata theory or a switch in the domain of dynamical systems. Jumps can be used in different fashions. They allow a user to model discrete evolutions of the system (i.e., entering a new mode) and to model discrete modifications of the continuous variables (e.g., resetting a timer). A jump might be guarded via a boolean expression such that any trajectory is allowed to follow the jump if and only if the boolean expression evaluates to true. Note that a trajectory of a hybrid system is not forced to take a jump once the jump's guard evaluates to true as long as the invariant of the current mode holds. Therefore the user introduce nondeterminism as soon as a mode's invariant and a jump's guard have a nonempty intersection or different jumps originating in the same mode have a nonempty intersection of their guards. The user may also replace these kind of nondeterminism by probabilism. To do so the user has to specify a jump with multiple targets. A target is then chosen based on a distribution that has to be specified within the jump's definition. Each target of a jump might have an update to modify the valuation of the continuous variables.

An *update* sets a subset of the continuous variables to new values where the new values can also depend on the current values of other continuous variables. Note that exchanging the valuation of two variables will look like this $x := y ; y := x$ because updates will be performed in parallel such that variables that occur on the right side always refer to the valuation of the variables when the jump was initiated.

The following syntax diagram formalizes *definitions of jumps*.



<updates> The nonterminal <updates> matches a list of *updates* per variable separated by semicolons. Updates occur in jumps. Note that multiple updates for a single variable are not allowed.

<target> The nonterminal <target> matches a *jump targets* which could be either a single jump target or a list of probabilistic jump targets. Jump targets occur in jumps. Each jump target can have an optional update and must specify the name of the target mode. When specifying probabilistic targets each target also needs a probability $p > 0.0$ and the sum of the individual probabilities has to be equal to 1.0. Note that each name of a mode have to refer to a defined mode.

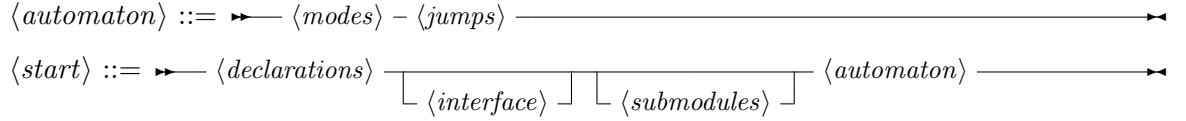
<jumps> The nonterminal <jumps> matches a list of *jumps* each from one mode to one or multiple modes (probabilistic jump). Note that self loop (i.e., jumps from a mode to

itself) as well as multiple (probabilistic) jumps between modes are allowed. It is not allowed that either the name of the source mode or target mode refers to an undefined mode.

2.8 Input File

The input file structure is very simple. It has three sections. The first section declares variables, constants, and parameters. The second section defines all the modes that occur in the automaton. The third section defines the jumps that a trajectory may follow.

The following syntax diagram formally defines *the input file*.



<automaton> The nonterminal **<automaton>** matches a *definition of an automaton*. A automaton consists of a list of modes specified by **<modes>** and a list of jumps specified by **<jumps>**. The nonterminal **<automaton>** occurs only in the start nonterminal **<start>**.

<start> The start nonterminal **<start>** matches the *input file*. The input file starts with a list of declarations of variables followed by an optional declaration of the automaton's interface, an optional list of definitions of submodules used within the automaton and ends with the definition of the automaton.

3 The Tool

Stabhyli is a simple command line tool currently available for Linux systems. As the tool (in its current distribution version) is build as a static binary for x86-32 system it should run out of the box. This section lists the feature that Stabhyli currently has as well as features that HAL which are not yet understood or ignored by Stabhyli. This section ends with the description of the command line options that Stabhyli has and a short example of an automaton that can be used as an input for Stabhyli.

3.1 Features

The following is a list of features that Stabhyli currently has.

- graph-based analysis and decomposition of the graph structure in order to verify stability by solving smaller problems.
- automatic generation and solving of constraint systems that prove a hybrid system stable
- identification of components of a hybrid system that cause the stability proof to fail

3.2 Missing Features

The following features of HAL are not yet implemented in Stabhyli.

- support for constants (will result in an exception)
- support for parameters (currently threaded as normal variables)
- support for global invariants on variables (currently ignored)
- support for submodules (submodules and interfaces will get parsed but ignored)

The following implementation specific features will be included in future versions of Stabhyli.

- changing default settings via a config file
- optimized generation of the Z-polyhedron (Newton polytope) for the generation of the sums-of-squares representation of polynomials (currently a rough heuristic is used)
- automatic translation from hybrid systems written in HLang to hybrid automata

3.3 Invocation

Stabhyli has parameters, all are optional except the parameter `--hal` which specifies the input file. A detailed description of the parameters can be found in Table 1.

For example, if a user has written a model of a hybrid system using HAL and wants to verify this model using Stabhyli then an simple invocation could be

```
stabhyli --hal example.hal.
```

If a common global polynomial Lyapunov function with exponents from 1 to 4 should be found, the invocation could be

```
stabhyli --proof common --min-lf-exp 1 --max-lf-exp 4 --hal example.hal.
```

3.4 Good to know

- Stabhyli currently tries to detect and handle some (implicit) equality constraints. This is experimental but so far allows to handle systems where the jump constraints require some Lyapunov functions to have identical values for some states.
- Stabhyli does not yet use multiple precision arithmetic. Like the background solver (CSDP), Stabhyli only uses double precision arithmetic.
- Having the famous Lyapunov theorem in mind, Stabhyli knows three types of constraints:

Parameter	Type	Default	Description
<code>--min-lf-exp</code>	optional	0	Sets the minimal exponent used for variables within the Lyapunov function's polynomial template.
<code>--max-lf-exp</code>	optional	2	Sets the maximal exponent used for variables within the Lyapunov function's polynomial template.
<code>--proof</code>	optional	decompose	Sets the scheme used for the verification of stability. Usable schemes are decompose , piecewise , and common . The common scheme tries to find a single common global Lyapunov function for all modes that proves hybrid system stable. The piecewise scheme tries to find a piecewise global Lyapunov function (one per mode) that proves the hybrid system stable. The decompose scheme tries to find piecewise Lyapunov function that proves the hybrid system stable by decomposing the hybrid system into small subcomponents.
<code>--split</code>	optional	zipper	Sets the heuristic used for the decision which nodes will be split. Usable heuristic are, zipper , product and pairwise . The product sorts the nodes according to the product of incoming and outgoing edges and selects the first node for splitting. The pairwise heuristic sorts the nodes according to the pairwise comparison of incoming and outgoing edges. The zipper works like the product heuristic but prefers nodes with either incoming or outgoing degree of one.
<code>--hal</code>	required	-	Sets the file which contains the automaton that should be verified stable.

Table 1: Parameters of Stabhyli

- Constraints that are **Non-Essential** will not be checked for satisfaction. Such constraints are used to direct the solver or to bound variables.
- Constraints that are **Soft** can be violated only a bit (currently the threshold for eigenvalues is $-1e^{-8}$). Such constraints are the constraints that involve α, β, γ since if there is a slight violation one can usually choose smaller α, γ or larger β to satisfy the constraint.
- Constraints that are **Hard** should not be violated and the threshold for rejecting solutions is set to $-1e^{-21}$. The constraints obtained by the transitions between modes are **Hard**.
- Stabhyli uses gap terms to make the search for a solution more robust the downside is that this also makes it harder to find a solution. And for some input Stabhyli might not be able to certify stability just because of the gaps. Note that scaling the input system might help in that case. The gaps are:
 - $1e^{-8}$ for constraints involving α or β ,
 - $1e^{-10}$ for constraints involving γ ,
 - $1e^{-12}$ for transition constraints, and
 - $1e^{-7}$ for other constraints (e.g. positivity of the sum of some parameters – like linear/conic combinations).
- Because of the used data structures (sets and maps) the memory allocation order can effect the order in which elements are handled and thereby introduces non-determinism. Thus it might happen that a second run leads to a different result. Also rounding and machine (im-)precision can make two runs yielding different results.
- Intentionally adding specific invariants/guards to the automaton could help finding a solution. So having an invariant $5 \leq x$ might not help the solver if the Lyapunov function template does not involve x but only x^2 . Here replacing the invariant with $25 \leq x^2$ could be a good idea. In practices some invariants/guards could make the problem very difficult since the solver is required to set their parameters to exactly 0. Note that in some cases Stabhyli does recognise such useless invariants/guards and removes them.

External Libraries:

CSDP background solver for SDP problems.

ATLAS/BLAS/LAPACK used by CSDP.

SDPA 7 (experimental) alternative background solver.

BISON/FLEX generating the parser and lexer for HAL and HLang files.

PPL (experimental) solving some linear problems.

CGAL Geometry based pruning of the vector of monomials for the sums-of-squares decomposition.

GMP (experimental) Some code is already aware of multiple precision arithmetic.

3.5 Example Automaton

The Figure 1 list a short example of an automaton that models a heating machine. It consists of three modes `normal`, `heat`, and `cool` and a single variable `temp`. The goal of that automaton is to stabilize the relative temperature `temp` which can be seen as the difference to the desired temperature. The mode `heat` (resp. `cool`) models saturation such that the temperature can not be raised (resp. lowered) more than one degree per time unit.

Using Stabhyli to proof this hybrid system stable could be done using a common global Lyapunov function. Example invocations are given in Figure 2.

```

1  DECL
2      CONVERGENT REAL temp;
3
4  MODES
5      IN    normal
6      LET   temp' := -0.1 * temp
7      WHILE -10 <= temp AND temp <= 10
8
9      IN    heat
10     LET   temp' := 1
11     WHILE -200 <= temp AND temp <= -10
12
13     IN    cool
14     LET   temp' := -1
15     WHILE 10 <= temp AND temp <= 200
16
17  JUMPS
18     FROM normal IF temp <= -10 GOTO heat
19
20     FROM heat IF temp >= -10 GOTO normal
21
22     FROM normal IF 10 <= temp GOTO cool
23
24     FROM cool IF 10 >= temp GOTO normal

```

Figure 1: Example Automaton “heater”

```

1  $ stabhyli --proof common --hal heater.hat
2  Set proof method to common
3  Set automaton filename to heater.hat
4  Common Lyapunov function: +100*temp^2
5  The hybrid system is stable
6  $ stabhyli --proof common --hal heater.hat --max-lf-exp 1
7  Set proof method to common
8  Set automaton filename to heater.hat
9  Set maximal exponent for Lyapunov templates to 1
10 The hybrid system can not be proven stable

```

Figure 2: Using Stabhyli to prove “heater” stable