ForFET^{SMT}
Formal Feature Corner Case Evaluation Tool

Abstract

This document acts as a guide to using the Formal Feature Evaluation Tool (ForFET^{SMT}) for the evaluation of feature corners for analog mixed-signal systems modeled as hybrid automata. We prescribe a model description language HASLAC for expressing the formal model, and use the Feature Indented Assertion language for specifying quantitative attributes called *features*. The proposed tool interacts with two major third-party tools, namely SpaceEx for reachset analysis of hybrid automata, and dReach/dReal for the problem of finding feature corner stimuli using Satisfiability Modulo Theory (SMT) queries. The document describes specification languages, the tool architecture, installation and usage instructions and example models and features.

Contents

1 Philosophy and Architecture

ForFET^{SMT} is a tool for the formal evaluation of quantitative measurements, called *features*, over hybrid system models specified as hybrid automata (HA). The tool is architected to enable users to create models stored in its Model Library, and specify a list of per model features, and evaluate the *goodness* and *robustness* of models with respect to features. In order to encourage the use of model-based-designe in the Analog Mixed-Signal (AMS) Semiconductor Industry, we provide a model language prescription called the *Hybrid Automaton Specification Language for AMS Circuits* (HASLAC). To enable an easier adoption of the MBD philosophy, the model specification language is designed with syntactic elements familiar to the AMS community, specifically used for the behavioural modeling of AMS circuits.

The tool is maintatined at: `https://github.com/antoniobruto/ForFET2`

2 Specifying Hybrid Automata Models

The Hybrid Automata Specification Language for analog mixed-signal (AMS) circuits (HASLAC) is the modeling language for analog-mixed signal hybrid systems as HA. HASLAC was designed and developed keeping in mind AMS circuit engineers who are familiar with Verilog and SVA-like syntax to encourage adoption of model-based development into the AMS Semiconductor Industry.

We use the following notation to express the syntax of HASLAC.

- Keywords are written in lower case. For example: **module**, **input**, **output** are keywords.
- The symbol '|' represents a grammatical OR.
- Brackets '[' and ']' represent that the contained expression may or may not be present.
- Brackets '{' and '}' represent zero or more repetitions of the contained expression.

- A series '...' may be replaced by the most logically consistent pattern. For instance when used in as part of "a | b | ... | z", the most logically consistent pattern would be the series of all lower case letters in the English alphabet.
- A syntactic symbol may be followed by a comment that restrains the use of the symbol to the context mentioned in the comment. A comment in the syntax is surrounded by the symbol %.
- Symbols that are part of the language used to describe the syntax may also be part of the grammar, and if so are specified in single quotes ''.

2.1 Constants and Operators

```
NUMBER          ::=  [sign] REAL_NUMBER
REAL_NUMBER     ::=  UNSIGNED_NUMBER.UNSIGNED_NUMBER |
                     UNSIGNED_NUMBER[.UNSIGNED_NUMBER]
                EXP[SIGN]UNSIGNED_NUMBER
UNSIGNED_NUMBER ::=  DIGIT { _ | DIGIT }
DIGIT           ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
EXP             ::=  E | E
SIGN            ::=  + | -
OPERATOR        ::=  + | - | * | \
COMP_OPERATOR   ::=  <= | >= | ==
IDENTIFIER      ::=  LETTER { _ | IDENTIFIER }
LETTER          ::=  A | B | ...  | Z | A | B | ...  | Z
IMPLIES         ::=  '|'=>
```

2.2 HASLAC Syntax

```
MODULE_DECLARATION ::=  module ( MODULE_PORT_LIST )
                        PORT_LIST
                        PARAMETER_LIST
                        MODE_DECLARATION
                        [MODE_PROPERTIES]
                        INITIAL_CONDITIONS
                        endmodule

MODULE_PORT_LIST    ::=  SIGNAL_LIST

PORT_LIST           ::=  [INPUT_LIST ;] OUTPUT_LIST ;

INPUT_LIST          ::=  input SIGNAL_LIST ;

OUTPUT_LIST         ::=  output SIGNAL_LIST ;

SIGNAL_LIST         ::=  IDENTIFIER {[, IDENTIFIER]} ;

PARAMETER_LIST      ::=  parameter ASSIGNMENT_LIST ;

ASSIGNMENT_LIST     ::=  CONSTANT_ASSIGNMENT {[, ASSIGNMENT_LIST]} ;

CONSTANT_ASSIGNMENT ::=  IDENTIFIER = NUMBER ;

MODE_DECLARATION    ::=  mode IDENTIFIER begin DYNAMICS_LIST end

DYNAMICS_LIST       ::=  ACTIVITY ; {[, DYNAMICS_LIST]}

ACTIVITY            ::=  ddt IDENTIFIER = LINEAR_ARITH_EXPR ;
```

```
LINEAR_ARITH_EXPR   ::=  LINEAR_ARITH_EXPR + LINEAR_ARITH_EXPR |
                         LINEAR_ARITH_EXPR - LINEAR_ARITH_EXPR |
                         PRODUCT_EXPR | IDENTIFIER

PRODUCT_EXPR        ::=  CONSTANT / LINEAR_ARITH_EXPR |
                    ::=  LINEAR_ARITH_EXPR / CONSTANT |
                    ::=  CONSTANT * LINEAR_ARITH_EXPR |
                    ::=  LINEAR_ARITH_EXPR * CONSTANT

CONSTANT            ::=  NUMBER |
                         IDENTIFIER % DECLARED AS A PARAMETER %

MODE_PROPERTIES     ::=  PROPERTY {MODE_PROPERTIES};

PROPERTY            ::=  property inv INVARIANT endproperty
                         property trans TRANSITION endproperty

INVARIANT           ::=  mode==IDENTIFIER IMPLIES CNF_CONSTRAINTS ;

TRANSITION          ::=  mode==IDENTIFIER && CNF_CONSTRAINTS
                         && mode'==IDENTIFIER IMPLIES RESET_RELATION ;

CNF_CONSTRAINTS     ::=  CONSTRAINT {&& CNF_CONSTRAINTS}

CONSTRAINT          ::=  IDENTIFIER COMP_OPERATOR CONSTANT |
                         CONSTANT COMP_OPERATOR IDENTIFIER

RESET_RELATION      ::=  RESET {&& RESET_RELATION}

RESET               ::=  IDENTIFIER' == LINEAR_ARITH_EXPR

INITIAL_CONDITIONS  ::=  initial begin INITIAL_STATE_LIST end

INITIAL_STATE_LIST  ::=  INITIAL_STATE {INITIAL_STATE_LIST}

INITIAL_STATE       ::=  set begin INITIAL_CONSTRAINTS end

INITIAL_CONSTRAINTS ::=  INITIAL_CONSTRAINT ; {INITIAL_CONSTRAINTS}

INITIAL_CONSTRAINT  ::=  ASSIGNMENT |
                         ASSIGNMENT_RELATION |
                         LINEAR_CONSTRAINT

ASSIGNMENT          ::=  IDENTIFIER = IDENTIFIER |
                         IDENTIFIER = NUMBER

ASSIGNMENT_RELATION ::=  IDENTIFIER = '['CONSTANT:CONSTANT']'

LINEAR_CONSTRAINT   ::=  LINEAR_ARITH_EXPR COMP_OPERATOR
                         LINEAR_ARITH_EXPR
```

Uses of this syntax to express hybrid automaata may be found in the examples in Section 8.

3 Specifying Features

The language of features, *Feature Indented Assertions* (FIA), is used to express quantitative measurements to assess the design. The syntax and semantics for FIA is destribed below.

3.1 Constants and Operators

```
RATIONAL          ::= -?(([0-9]+|([0-9]*\.[0-9]+))(([eE][i+]?[0-9]+)?)
ARITHOP           ::= \+|\-|\* |\/
LEFT_SQR_BRKT     ::= [
RIGHT_SQR_BRKT    ::= ]
RELOP             ::= < | > | <= | >= | ==
EVENTTYPE         ::= @\+ | @\- | @
BINLOGICOP        ::= (&&) | (\|\|)
ATOM              ::= (([_a-zA-Z]+)([_a-zA-Z0-9\.]*))
```

3.2 Feature Indented Assertion Syntax

```
FEATURE_SPEC        ::= feature ATOM ( PARAMETER_LIST ); FEATURE_DEFINITION

PARAMETER_LIST      ::= PARAMETERS
                        | ϵ

FEATURE_DEFINITION  ::= begin FEATURE_BODY end

FEATURE_BODY        ::= VARDECL FEATUREDECL

VARDECL             ::= var PARAMETERS ;
                        |ϵ

PARAMETERS          ::= PARAMETERS, ATOM
                        | ATOM

FEATUREDECL         ::= SEQUENCE_EXPR |-> FEATURE_ASSIGNMENT ;

SEQUENCE_EXPR       ::= ( SEQUENCE_EXPR ) DELAY ( SEQUENCE_EXPR )
                        | EXPR, Assignment
                        | EXPR

EXPR                ::= PORVExpr
                        | PORVExpr && EVENT
                        | EVENT && PORVExpr
                        | EVENT

PORVExpr            ::= ( CONJUNCT \|\| CONJUNCT )
                        | CONJUNCT

CONJUNCT            ::= CJNCTEXPR
                        | first_match ( CJNCTEXPR )

CJNCTEXPR           ::= PORV && CJNCTEXPR
                        | PORV

PORV                ::= ArithExpr RELOP ArithExpr

EVENT               ::= EVENTTYPE ( PORV )
                        | first_match ( EVENTTYPE ( PORV ) )

ASSIGNMENT          ::= ASSIGNMENT, ATOM = ArithExpr
                        ATOM = ArithExpr

DELAY               ::= ## LEFT_SQR_BRKT RATIONAL:$ RIGHT_SQR_BRKT
                        | ## RATIONAL

ArithExpr           ::= ArithExpr ARITHOP ArithExpr )
```

```
                        | ATOM
                        | RATIONAL
                        | $time
                        | ( ArithExpr )

FEATURE_ASSIGNMENT   ::= ATOM = ArithExpr
```

- The non-terminal `ATOM` in the `FEATURE_ASSIGNMENT`, must match the feature name produced by the non-terminal `ATOM` in the partial production rule `FEATURE_SPEC -> feature ATOM ( PARAMETER_LIST );`

Please note that the main difference between the feature specification languages of the two tools are:

1. Each subexpression is a disjunction of conjunct terms, that together may be conjuncted with an event.
2. An event is specified as $@^*(PORV)$, where $* \in \{+, -, \ \}$.

Uses of this syntax to express features may be found in the examples in Section 8.

4 Tool Directory Structure

This section described the implementation items that makeup the ForFET^{SMT} tool, the directory structure of the code-based and lists the pre-requisites and build instructions for compiling the tool into a binary.

The tool directory structure is depicted in Figure 1.

At the root of the tool you may find the following content:

- **[docs]:** *Directory* : Contains documentation pertaining to the tool and the language reference manual for modelling language for HASLAC for Hybrid Automata specification.
- **[forFET]:** *Directory* : Contains the code for the tool ForFET^{SMT} and example models and features. The contents of this directory are exposed later.
- **run:** This script is to be used to setup the work-directory and build the binary. The run-script internally calls a build-script and supresses minor compiler warnings.
- **ReadME:** A readMe document briefly detailing installation and execution instructions for ForFET^{SMT}.

The code for the tool is contained in the **[forFET]** directory. The contents of this directory are as follows:

- **[lib]:** Contains the library of hybrid automaton examples that can be used out of the box and can be run when the tool is setup. When a new model is to be used, the following steps must be followed:
  1. Add the name of the model, ⟨ModelName⟩ in `lib/modelList.txt`.
  2. Add the model ⟨`ModelName`⟩`.ha` in `lib/models`.

  To add feature specifications for the model the following steps must be followed:

```
ROOT
├── ForFET
│   ├── [parsers]
│   │   ├── featureParser.l           ⎫
│   │   ├── featureParser.y           ⎪
│   │   ├── haslacParser.l            ⎪
│   │   ├── haslacParser.y            ⎬ Parsers
│   │   ├── drhParser.l               ⎪ (Lex files and Yaac files)
│   │   ├── drhParser.y               ⎪
│   │   ├── spaceExOutputParser.l     ⎪
│   │   ├── spaceExOutputParser.y     ⎭
│   │   │
│   │   ├── preSpaceEx                ⎫
│   │   ├── preHASLAC                 ⎬ Temlate Files
│   │   ├── preFeature                ⎪ Used by the Parsers
│   │   └── preDRH                    ⎭
│   ├── [lib]
│   │   ├── modelList.txt             (Line seperated list of model names)
│   │   ├── [models]                  (Model files labeled as ⟨ModelName⟩.ha)
│   │   └── [features]                (Model directories labeled ⟨ModelName⟩)
│   │       └── [⟨ModelName⟩]
│   │           └── featureList.txt   (Line seperated list of features)
│   ├── [work]                        (Generated SpaceEx and dReach Model Files)
│   ├── [spaceex]                     (SpaceEx Binaries)
│   ├── build   /* The build script */
│   ├── structs.c/h
│   └── fbfvTool.c
│
├── run     /* The run script : Used to build and setup the tool */
├── ReadME
└── [docs]
    └── tool_manual.pdf
```

Figure 1: ForFET^{SMT} Tool File and Directory Structure

1. Create the directory: `lib/features/⟨ModelName⟩`.
2. Create the feature list as line seperated feature names in the file: `lib/features/⟨ModelName⟩/featureList.txt`
3. Add the feature: `⟨FeatureName⟩.dat` in: `lib/features/⟨ModelName⟩/`.

- **[parsers]:** Contains Lex and Yaac files for the parsers used by the tool to parse the feature description, hybrid automaton specification, and third party tool outputs. Addition support files for building the parsers are also present in this folder.

- **[spaceex]:** When the tool is executed the spaceex binaries are expected to be in this directory, but depending on your setup, may also be placed elsewhere and specified in the configuration. SpaceEx executable binaries are required for running the SpaceEx Hybrid Automaton Reachability Analysis Tool.
- **[work]:** Contains intermediate files such as models and trace files generated by the tool for feature analysis.

Other files present in the tool's [**forFET**] directory and their function is detailed below:

- **build.sh:** is the build-script used to compile the parsers and the supporting code for the tool ForFET^{SMT}.
- **struct.c, structs.h:** All structures and supporting functions used for creating and maintaining the feature specification and the hybrid automaton can be found in the structs files. `structs.h` contains the structures and function definitions of all supporting functions, whereas `structs.c` contains the implementations of all functions specified in the header file.
- **fbfvTool.c:** All supervising methods that control the overall process of accepting the hybrid automaton and feature specification inputs, parsing them, computing the product of the feature automaton and the hybrid automaton, executing third party tools and finally displaying the results of analysis are present in `fbfvTool.c`.
- **forFET**: is the binary for ForFET^{SMT}, generated using the build-script, and once the supporting tools are successfully running on the machine.
- **degault.cfg:** is the example configuration file.

Additional 3rd party tools such as *Hyst* may also be present in the directory [forFET].

5 Installation Instruction

In this section we list the dependencies of ForFET^{SMT} and instructions for building and running the tool.

5.1 Libraries and 3rd Party Tools

The tool requires the following libraries and packages to run:

- C/C++ compilers: g++ was used to build the tool
- Lexical Analyzer lex
- Bison C Parser
- Java Runtime Enviroment (Required for "Hyst" - a language converter for converting between a variety of HA modeling languages).
- Libraries: glib-2.0, json-glib-1.0, C/C++ standard libraries for 32 bit binaries (ia32-libs on Ubuntu 10.04 and later). Ofcourse, one would need to determine what works depending on the linux distribution being used.

- Download SpaceEx binaries from:

  `http://spaceex.imag.fr/sites/default/files/downloads/private/spaceex_exe-0.9.8f.tar.gz`

  Extract spaceex from the tarball. Ensure that the SpaceEx binary executes on your system. Running the command: `chmod +555 spaceex` in the directory `spaceex_exe` for the spaceex script should make the binary executable.

- Download and extract dReach and dReal3 to accessible paths. Links for dReach and dReal3:

  `https://github.com/dreal/dreal3/releases`.

5.2 Building Tool Binaries

To build the tool, execute `./buildForFET.sh` from the tool's root folder.

The tool generates parser and code files during the build process within the forFET directory and creates the tool binary "forFET".

Once the tool is setup, you can proceed to use the tool by first creating hybrid automaton models and writing features, which the tool then automatically accepts.

6 Tool Usage

To use the tool, create a configuration file (instructions in Section 6.1. The configuration file specifies paths for the working directory (where ForFET^{SMT} stores intermediary model files), the library for storing models and features, the paths for third party tools dReach and spaceEx. Before running `forFET` ensure that the paths specified are usable and that the working directory can be written into. Also ensure that the tool `Hyst.jar` is in the same directory as the binary for `forFET`.

Using the tool involves executing the binary:

```
./forFET <path-config-file>
```

The binary `forFET` is present in the ForFET tool directory when first built, but depending on your internal IT policy may be placed elsewhere. For instance, it may be in your user `bin` path.

6.1 Necessary Configurations

The tool uses a configuration which specifies paths for 3^{rd} party tools, the model/feature library, and a write-enabled workspace directory

An example configuration is as follows:

```
work_PATH=work/
libs_PATH=lib
dReach_PATH=../../../dReal-3.16.06.02-linux/bin/dReach
spaceEx_PATH=./spaceex/spaceex_exe/spaceex
```

The following paths are specified:

- `work_PATH`: Writable directory where model files and intermediate files are written.

- `libs_PATH`: Library of models and features
- `dReach_PATH`: The path containing the `dReal` and `dReach` binaries.
- `spaceEx_PATH`: The path containing the `spaceex` script.

Incorrect path specifications are indicated by the tool. Ensure that binaries are all "executable".

6.2 Interacting with ForFET^{SMT}

The steps followed by the tool are as follows:

- **Indicate Path Statuses:** Indicates if all 3^{rd} party tools, library and work paths are accessible.
- **Choice of Model:** The tool presents an enumerated list of models. Here you may enter a number to choose which model you wish to analyze.
- **Edit Model Parameters:** The tool presents an enumerated list of model parameters that may be edited by choosing a index for the parameter (as displayed in the list) and entering the parameter's new value. The value is a rational number.

  When satisfied with the parameter values, enter the number zero, `0`, when asked for the next choice of parameter.
- **Specifying Tool Parameters:** The tool has three core parameters it uses, a *timescale* which indicates a time-precision to be used, and a *trace time horizon* which limits the lengths of runs explored to a specified value. The units of all numbers is "seconds". The last parameter is a *maximum expected rational value*, this is the largest number you expect the tool to see. By specifying this number, you would help bound the search space, the domains of variables in the model. The largest rational value is also one that must be as large or larger than the chosen time horizon.
- **Choice of Feature:** A list of enumerated features is presented from which you may choose the feature for which you wish to evaluate the model.
- **SpaceEx Scenario**: We use the tool SpaceEx to obtain conservative estimates of the feature. SpaceEx uses two core methods for achieving this, *LGG*, which is more mature and accurate but compuationally more expensive; and *STC*, which is less mature but produces quicker results. Please note that SpaceEx is also an academic tool and is not guaranteed to always produce an output. If for one choice of algorithm, LGG or STC, analysis fails, please try a different choice in a new attempt.
- **SpaceEx Analysis Results:** Results of the SpaceEx analysis are displayed. The results show an interval for the feature.
- **Extremal Analysis - Path Length:** For extremal analysis we use an SMT solver, which requires a bound to be specified on the number of automaton location/mode changes. This must be specified here as the path length. The tool explores all paths of length upto the specified bound.

- **Extremal Analysis - Precision Limit:** The extremal analysis uses a search over SMT problem instances. The search terminates when it finds conclusive feature extremals. It decides on termination when a given precision limit on the feature value is reached. This value-precision-limit must be specified here.

- **Feature Extremal Corners:** Once the extremal analysis converges, it indicates convergence by providing the feature corner values and identifier numbers for JSON trace files describing concrete traces that achieves these corners in the model.

6.3 Usability Extensions

ForFET^{SMT} provides the following usability characteristics:

- **Environment Validation:** ForFET^{SMT} validates the environment, including write-access permissions for the work-directory, and indicates any access permission problems encountered to the user when ForFET is executed.

- **Scripting:** ForFET^{SMT} allows the user to use it in a script. It allows for standard UNIX piping for inputs and outputs. The usage is

```
./forFET [config] [<input] [>output]
```

  Note that artifacts contained within square brackets are optional.

- **Logging:** ForFET^{SMT} logs major analysis steps and analysis options in a log, "analysisLog.txt", created within the "work" directory.

- **Auto-save inputs:** All inputs to ForFET^{SMT} within a run are saved to a log file, "`run_timestamp`", created within the "work" directory. The `timestamp` is the timestamp of the run. The file can be reused by piping it as an input, to re-run any previous executions of ForFET, in a non-interactive mode of execution.

- **Extremal Trace File Recording:** Extremal analysis produces a trace corresponding to each extremal feature corner and all intermediate search steps. Due to the existence of special "*urgent*" modes used by ForFET^{SMT}, standard tools are unable to process the traces generated by the tool. We post-process the trace to enable the dReal/dReach ODE-Visualization tool to be used directly on the trace files generated by ForFET^{SMT}.

7 Standard Features in FIA

In this section, we provide a library of formal feature specifications based on the notion of standard patterns for real-time systems [2].

In Ref [1], the authors propose constructing monitor automata based on standard patterns for the verification of control and embedded systems. While the specifications are non-quantitative in nature, and provide a Boolean outcome (success/failure), they also demonstrate that state-machine representations of specification can be easier to interpret, although are challenging to correctly prepare, since they need to be constructed by hand for every new pattern that is observed. Since these automata are parameterized, once prepared for a pattern, for a new design under test it is straightforward to ge nerate an instance of the monitor. With features, we have a specification language from which monitors may be automatically generated. Given a pattern, constructing the monitors involves specifying the pattern using an SVA-like language, for which an automatic translation is possible into monitoring automata.

We provide a library of features for fundamental real-time patterns inspired from [2, 1]. When dealing with AMS systems, it is important to understand margins, both for attribute requirements that the design successfully adheres to (as a measure of robustness), and margins of failure (as a measure of how badly the design fails to meet the specification). A feature, by construction, expresses a quantity. Real-time patterns that we study are defined strictly for situations involving timing.

Since features measure quantities and are not used to enforce requirements, we use the quantities representing the feature evaluations as indications of the behaviour of the system. It is important to note that, in the construction of features, the behaviour is encoded as a sequence. Hence any sub-expression of the sequence that fails to match in a sub-trace of an execution causes there to be no match of the feature's sequence expression in that sub-trace. Hence when encoding requirements as features, we intelligently use this condition of *"no-match"* to our advantage, typically to indicate when requirements are satisfied. On the other hand, we use the upper or lower bound on the feature range to indicate failures of a requirement.

We have the following real-time specifications involving predicates/events p, q and s over real-valued analog signals:

Manifest: The time elapse, between a time-point when q is true and a time-point when p is true. This property is used to identify manifestations of p, and is modeled when we wish to ensure that p remains absent. The feature is specified as follows:

We describe features in this section, that can be used as templates for developing new features, and the construction of these features will also aid the interested reader to use the semantics of features develop custom feature templates.

```
feature manifest
begin
var t1, t2;
q, t1:=$time ##[0:$] p, t2:=$time |-> manifest:=t2-t1;
end
```

Manifest Timed: The time elapse (bounded by a real-value T), between a time-point when q is true and a time-point when p is true. This property is used to identify manifestations of p, however bounded to occurrences beyond an insensitivity period T, and is modeled when we wish to ensure that p remains absent T time after q is true. The feature is specified as follows:

```
feature manifestTimed
begin
var t1, t2;
q, t1:=$time ##[T:$] p, t2:=$time |-> manifestTimed:=t2-t1-T;
end
```

The evaluation of the feature *manifest* is described in Figure 2.

Separation: The separation in time (bounded by a real-value T), between a time-point when q is true and the time-points when p is first true. Unlike the feature *manifest*, this feature represents the need for p to be true within a real-valued time bound T after q is true. A feature range where the value of the upper bound of the range is T indicates a failure, that is at least one instance exists where p remains false for a separation of T time. The feature is specified as follows:

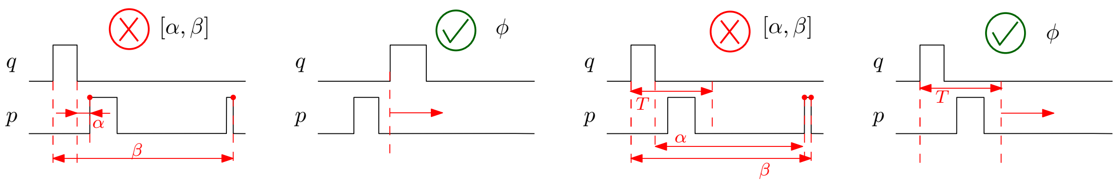


Figure 2: Manifest - The first pair represents the untimed case, while the second has a lower bound on the region of sensitivity. The first of each pair describes a situation of a failure.

```
feature separation
begin
var t1, t2;
q, t1:=$time ##[0:T] first_match(!p), t2:=$time
|-> separation:=t2-t1;
end
```

The evaluation of the feature is described in Figure 3.

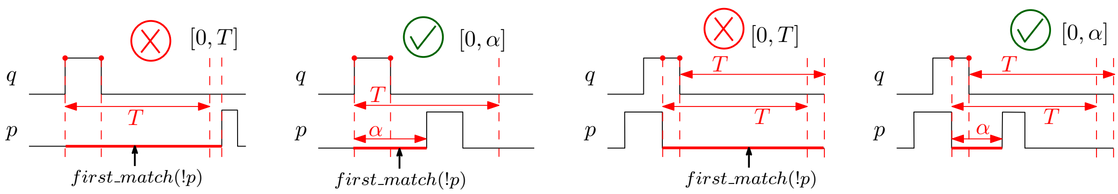


Figure 3: Separation - The leftmost of each pair represents a failure of p to be separated from time-points where q is true by at most T time, while the rightmost represents success.

Duration: The time elapse, after q is true, from the time-point when p becomes true, to the first time-point when p becomes false. For each occurrence of p after q is true, p must remain true for at least T time before becoming false again. A lower bound of any value under T for the feature indicates a failure. The feature is specified as follows:

```
feature duration
begin
var t1, t2;
q ##[0:$] @+(p), t1:=$time ##[0:$] first_match(@-(p)), t2:=$time
|-> duration:=t2-t1;
end
```

The evaluation of the feature *duration* is described in Figure 4.

Reaction: The time elapse, after q is true, from a time-point when p becomes false, to the first time-points when s is false. For each occurrence of p after q, when p becomes false, s must react within an amount of time T, that is s must remain false for an amount of time less than T. Hence, an upper bound of T when evaluating the feature *reaction* indicates a failure of s to react to p falling. The feature is specified as follows:

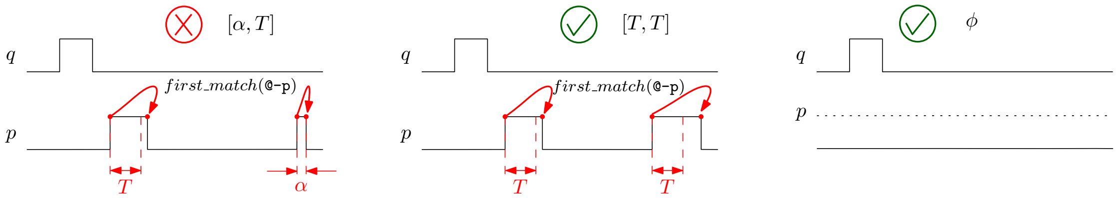


Figure 4: Duration - The leftmost execution has an instance where the duration of p is below T. The last two executions (from the left) indicate feature values where the lower bound on the values would be greater than T, where p remains true for at least T time.

```
feature react
begin
var t1, t2;
q ##[0:$] @-(p), t1:=$time ##[0:T] first_match(!s), t2:=$time
|-> react:=t2-t1;
end
```

The evaluation of the feature is described in Figure 5.

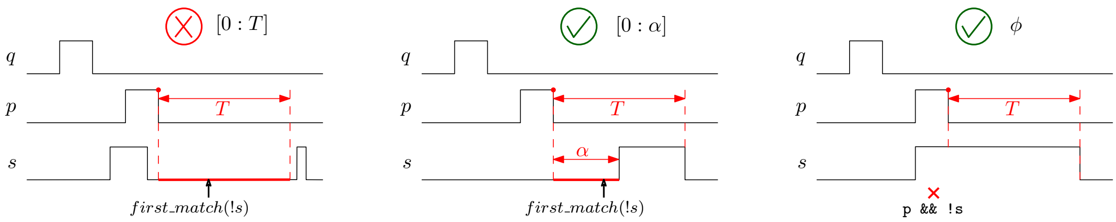


Figure 5: Reaction - The leftmost is an execution where s fails to react to the negative edge of p in T time. The last two cases (from the left) indicate a feature range where the upper bound on the range would be less than T, indicative of successes, where s reacts to the negative edge of p within T time.

Response: The time elapse, after a time-point where q is true, from a time-point where p is true and s is false, to the first time-points when s is false. Similar to reaction, the feature *response* requires s to be true within T time of p being true. Evaluations of the feature on various executions is described in Figure 6.

Invariance: The time elapse (bounded by a real value T), after q is true, from a time-point when p is true, to the first time-points when s is false. The invariance feature may then be used to check if s is invariant for at least an amount of time T whenever p is true. The way the feature is described, the time-elapse measures the duration when $p$is true and s is false. The feature is specified as follows:

```
feature invariance
begin
var t1, t2;
q ##[0:$] p, t1:=$time ##[0:T] first_match(!s), t2:=$time
|-> invariance:=t2-t1;
end
```

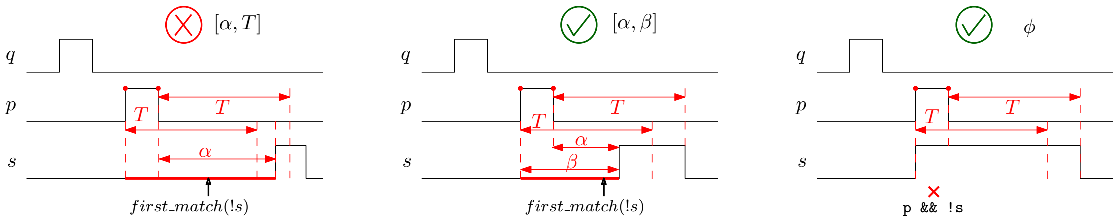


Figure 6: Response - The leftmost is an execution where s fails to respond to p in T time. The last two cases (from the left) indicate a feature range where the upper bound on the range would be less than T, indicative of successes, where s responds to p within T time.

The evaluation of the feature on executions is shown in Figure 7.

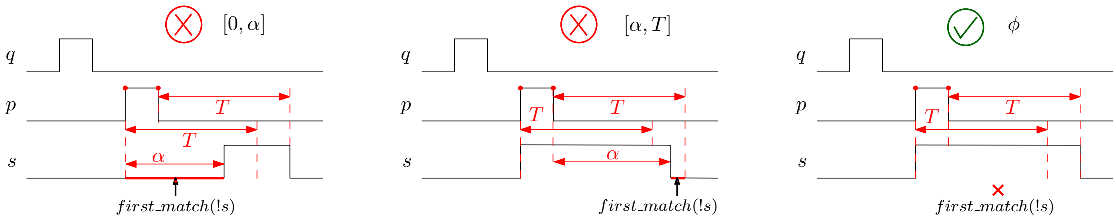


Figure 7: Invariance - The first two cases (from the left) indicate a non-empty feature range, indicative of a failure to maintain the invariance condition for s. The rightmost is an execution where invariance is maintained.

7.1 Feature Monitors

Seven feature patterns were described. Each pattern represents a desired or undesired behaviour, while the feature evaluation function measures a quantity robustness. Writing formal specification can be a daunting task. Not only is is a challenge to translate English description into the formalism called for, it is exacerbated by the problem of having complete specifications written for every desired and (or) undesired scenario. In this section we deal with the former, with the aim of making it easier to describe features.

It has been found, in the control and embedded domains, translating specification textually, into formal specification can be difficult. English, which is the most widely used language for written specification, can be ambiguous in its meaning. Understanding the intent embedded in such specification is non-trivial. We therefore propose that specifications be expressed using monitors. This is not the first time that this has been proposed. In Ref [1] a number of standard patterns were translated into monitoring state machines (automata) for checking assertions on formal models of hybrid systems.

Since features allow quantitative expression, and tools exist for monitoring features over simulation runs (DyFET) and formally for hybrid automata models of the circuit, we find it would be useful to prepare monitoring automata for features.

Table 1 presents feature monitors for the feature patterns described in Section 7.

In the table, each feature monitor has a number of locations labeled with L_i, $i \in \mathbb{N}$. L_1 is the initial location for all monitors, and described the situation where a match of the activation signal q is awaited. Since the monitor is non-deterministic, the transition from L_1 to L_2 can be taken when q is observed, or control may stay in L_1 to await future time-points where q is true. This power allows the analysis to analyze an infinite number of matches of the feature on executions of the circuit. To enforce movement during a specific region or on the first occurrence of an event we use `first match`, indicated as the function `fm(.)` in the monitors. The semantics of first match are pictorially described in Section 7.

Each location L_i, $i > 1$, represents that part of the behaviour being described has match. The location labeled $\langle final \rangle$ indicates that at the location, a complete match of the behaviour described has been observed and that the feature has been computed.

In each location, two timer variables are used. The variable lt measures the time spent in the location, while variable t measures the cumulative time and is analogous to `$absoluteTime` in most circuit simulators. In the monitors, q, p and s are predicates over signal variables of the circuit.

Above every transition, a Boolean condition is specified. This condition guards movement between the source and destination location, in the direction of the edge. The notation `@-(`p`)` represents the negative edge of the predicate p.

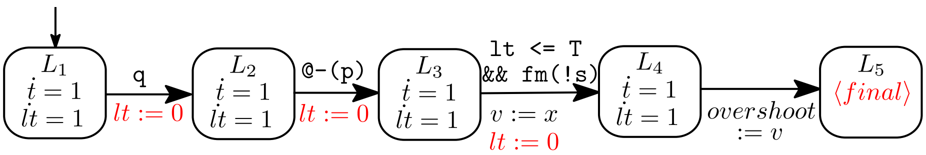
Below each transition, assignment operations are described, allowing assignments to be made to variables of the monitor. These variables are uninterpreted and similar in semantics to local variables in SVA, the difference being that variables used with features are real-valued. One may define as many variables as necessary. In our use, in developing these monitors, two local variables (in addition to the timers lt and t) were sufficient.

7.2 Using Feature Templates

The standard feature templates described in Section 7 have implications in real practice. In this section, we describe how standard feature templates can be used to describe frequently evaluated real-valued timing attributes for an AMS circuit.

Rise Time: Rise time is defined generically as the time for a signal x to rise from k_1 to k_2. With a known upper bound T_{rise} on the value of *rise time*, one can define $q \equiv (x == k_1)$, with $p \equiv (x == k_2)$, and measure the *Separation* between q and p. One can also alternatively define $p \equiv (x \leq k_1)$ and $s \equiv (x \geq k_2)$ and measure the *reaction* of s after p, where $q \equiv p$, or alternately $q \equiv en$, where en is a Boolean enable of the circuit.

Overshoot: Overshoot is defined with respect to the largest value of a signal x with respect to a rated value c_x for x. Although, overshoot is not a timing attribute of the circuit, it represents a *reaction* of the circuit when x goes above c_x. The attribute we wish to compute is the maximum deviation of x from c_x as a reaction to x increasing beyond c_x. While defining $p \equiv (x \leq c_x)$ and $s \equiv p$, we may use a single local variable to *overshoot* that is assigned the value of x when p is false. The feature are its associated monitor are given below.




```
feature overshoot
begin
var v;
q ##[0:$] @-(p) ##[0:T] first_match(!s), v:=x
|-> overshoot:=v;
end
```

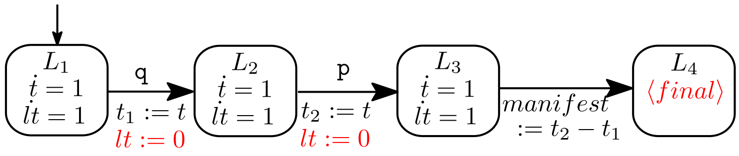
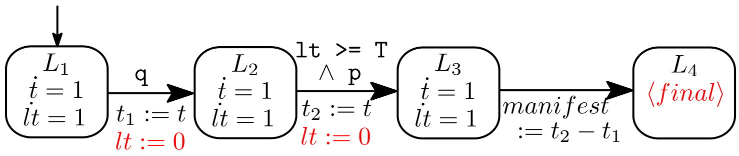
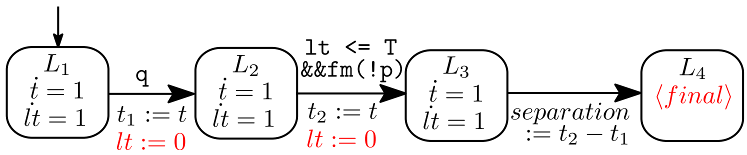
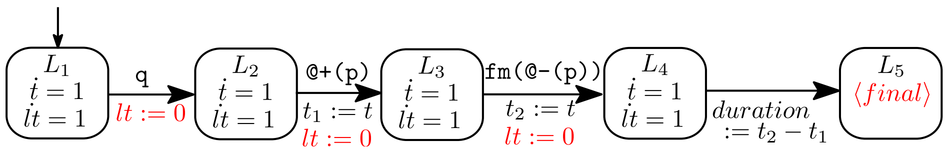
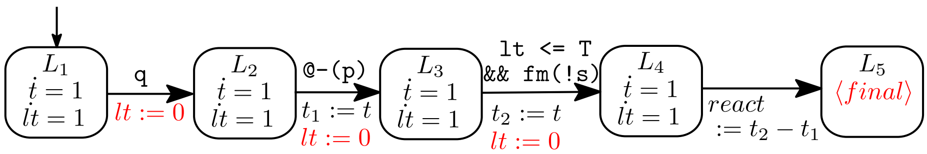
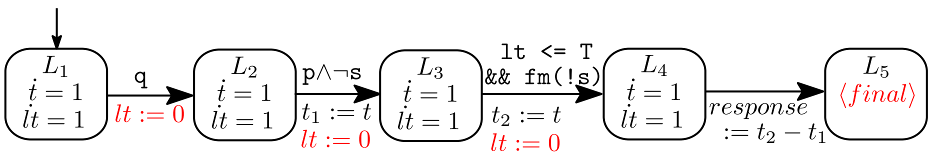
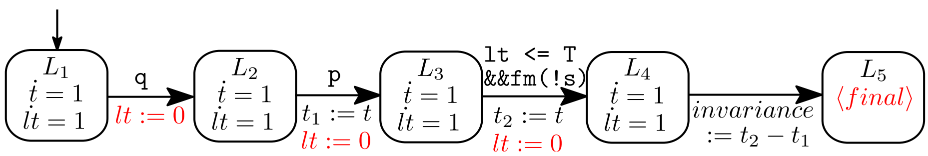
Feature	Feature Monitor
Manifest (Untimed)	
Manifest (Timed)	
Bounded Separation	
Duration	
Bounded Reaction	
Bounded Response	
Bounded Invariance	

Table 1: Feature monitors for Standard Feature Patterns (described in Section 7)

Settling Time: Settling time is defined for a signal x with respect to its rated value c_x, as the time taken for x to settle within a very small value $\delta \in \mathbb{R}$ of c_x for time α). By defining the initial event to be $q \equiv en$ for an enable signal en of the circuit, with $p \equiv (x == c_x)$, and the invariant $s \equiv (c_x - \delta \leq x \leq c_x + \delta)[^*\alpha]$, we measure the *invariance* of s after any observation of p as the settling time.

In addition to these, other analog properties can be broadly translated into standard features as follows:

- Properties dealing with safety (nothing bad should happen), of the form *if* `q` *is true then* `p` *should never happen* can be encoded as the *manifestation* of `p` after `q` (timed or untimed).
- Properties dealing with the temporal distance between an event manifestation after an enabling condition can be measured as a *separation*.
- Properties dealing with the lengths of time when a signal x is above or below a threshold c_x can be measured as *durations* for the predicate (being true or false).
- Properties dealing with reactions and responses of signals to each other (transactions), can be modeled using the features *reaction* and *response*.

8 Examples

Examples of models and features, by default, can be found in the folder "`./forFET/lib/`". If not present download them from the author's website[1], or contact your IT-Team if you are using ForFET^{SMT} in a secure environment.

8.1 Battery Charger

The hybrid automaton model of the battery charger is in Figure 8.

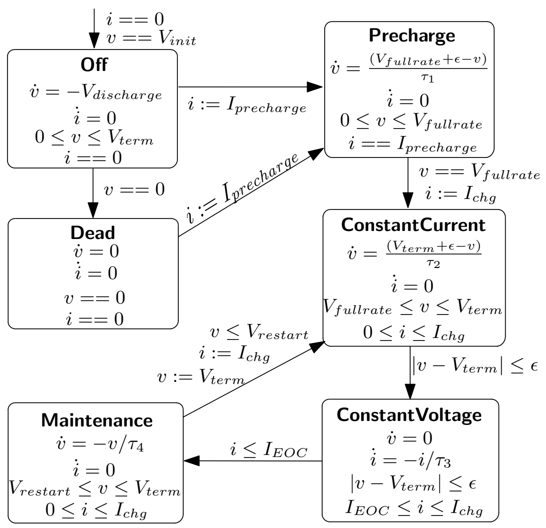


Figure 8: Battery Charger Hybrid Automaton Model

8.1.1 HASLAC Description

```
module batteryCharger(v,i)
        output v,i;

        parameter
        Vinit = 0,
```

[1]The tool is available for download at: `http://cse.iitkgp.ac.in/~bdcaa/ForFET/`

```
Vdischarge = -10e-5,
Vterm = 4.2,
Vfullrate = 3,
Vrestart = 4,
Iprecharge = 0.05,
Ichg = 1,
IEOC = 0.025,
E = 0.005,
T1 = 1200,
T2 = 150,
T3 = 150,
T4 = 1200;

mode off
begin
        ddt v = -Vdischarge;
        ddt i = 0;
end

mode precharge
begin
        ddt v = ( Vfullrate + E - v ) / T1;
        ddt i = 0;
end

mode constantCurrent
begin
        ddt v = (Vterm + E - v)/T2;
        ddt i = 0;
end

mode constantVoltage
begin
        ddt v = 0;
        ddt i = -i/T3;
end

mode maintenance
begin
        ddt v = -v/T4;
        ddt i = 0;
end

property inv off
mode == off |=> 0<=v && v<=Vterm && i==0;
endproperty

property inv precharge
mode == precharge |=> 0<=v && v<=Vfullrate && i==Iprecharge;
endproperty

property inv constantCurrent
mode == constantCurrent |=> (Vfullrate-E)<=v && v<=Vterm
&& 0<=i && i<=Ichg;
endproperty

property inv constantVoltage
mode == constantVoltage |=> (Vterm-E)<=v && v<=(Vterm+E)
&& (IEOC-E)<=i && i<=Ichg;
endproperty

property inv maintenance
```

```
        mode == maintenance        |=> Vrestart<=v && v<=Vterm
        && 0<=i && i<=Ichg;
        endproperty

        property trans off_pc
        mode == off && mode' == precharge && true |=> i'==Iprecharge && v'==v;
        endproperty

        property trans pc_cc
        mode == precharge && mode' == constantCurrent && v>=(Vfullrate - E) && v<=Vfullrate
                |=> i'==Ichg && v'==v;
        endproperty

        property trans cc_cv
        mode == constantCurrent && mode'==constantVoltage && (Vterm-E)<=v && v<=(Vterm+E)
                |=> i'==i && v'==v;
        endproperty

        property trans cv_maint
        mode == constantVoltage && mode'==maintenance && i<=IEOC
                |=> i'==i && v'==v;
        endproperty

        property trans main_cc
        mode == maintenance && mode'==constantCurrent && v<=(Vrestart+E)
                |=> i'==Ichg && v'==v;
        endproperty

        initial begin
                set begin
                mode == off;
                i == 0;
                v == 0;
                end
        end

endmodule
```

8.1.2 Features

We analyze the following features for the battery charger model.

Time to Charge: *The time for the battery to charge from ϵ volts (completely drained of charge) to the battery's rated voltage.*

```
feature ChargeTime(Vterm,epsilon);
begin
        var t1, t2;
        (state == PRECHARGE) && @+(V >= epsilon), t1 = $time
        ##[0,$] (state == CV) && @+(V == Vterm), t2 = $time
                |-> ChargeTime = t2-t1;
end
```

Restoration Time: *Time taken for the voltage to rise from Vrestart, in the maintenance mode, to Vterm, in the constant voltage mode.*

```
feature restorationTime;
begin
        var t1,t2;
        (state == maintenance) && @+(v <= Vrestart), t1=$time ##[0:$]
```

```
            (state == cv && i == IEOC) && @+(v >= Vterm), t2=$time
                    |-> restorationTime = t2 -t1;
    end
```

8.2 Nuclear Charger

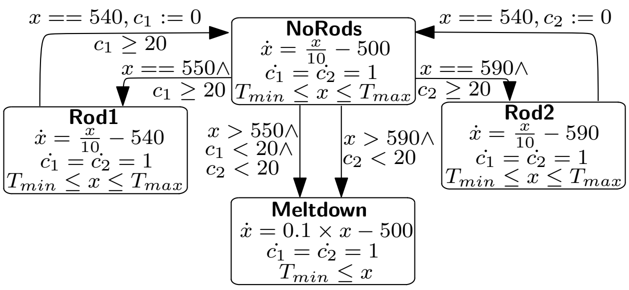


Figure 9: Nuclear Reactor Cooling Hybrid Automaton Model

8.2.1 HASLAC Description

```
module nuclearReactor(x,c1,c2)
        output x,c1,c2;

        parameter
        minTemp = 500,
        maxTemp = 600,
        R0 = 50,
        R1 = 56,
        R2 = 60;

        mode noRods
        begin
                ddt x = 0.1*x - R0;
                ddt c1 = 1;
                ddt c2 = 1;
        end

        mode rod1
        begin
                ddt x = 0.1*x - R1;
                ddt c1 = 1;
                ddt c2 = 1;
        end

        mode rod2
        begin
                ddt x = 0.1*x - R2;
                ddt c1 = 1;
                ddt c2 = 1;
        end

        mode meltdown
        begin
                ddt x = 0;
                ddt c1 = 0;
                ddt c2 = 0;
```

```
        end

        property inv noRods
        mode == noRods |=> x<=maxTemp && x>=minTemp;
        endproperty

        property inv rod1
        mode == rod1 |=> x<=maxTemp && x>=540;
        endproperty

        property inv rod2
        mode == rod2 |=> x<=maxTemp && x>=540;
        endproperty

        property inv meltdown
        mode == meltdown |=> x<=maxTemp && x>=minTemp && c1<=20 && c2<=20;
        endproperty

        property trans noRods_rod1
        mode == noRods && mode' == rod1 && x == 550 && c1>=20
                |=> x' == x && c1' == c1 && c2' == c2;
        endproperty

        property trans noRods_rod2
        mode == noRods && mode' == rod2 && x == 590 && c2>=20
                |=> x' == x && c1' == c1 && c2' == c2;
        endproperty

        property trans rod1_noRods
        mode == rod1 && mode' == noRods && x == 540
                |=> x' == x && c1' == 0 && c2' == c2;
        endproperty

        property trans rod2_noRods
        mode == rod2 && mode' == noRods && x == 540
                |=> x' == x && c1' == c1 && c2' == 0;
        endproperty

        property trans noRods_meltdown
        mode == noRods && mode' == meltdown && c1<(R1 - 1)*10 && c2<20 && x>(R1 - 1)*10
                |=> x' == x && c1' == c1 && c2' == c2;
        endproperty

        property trans noRods_meltdown
        mode == noRods && mode' == meltdown && c2<20  && x>(R2 - 1)*10
                |=> x' == x && c1' == c1 && c2' == c2;
        endproperty

        initial begin
                set begin
                mode == noRods;
                x == 530;
                c1 == 20;
                c2 == 20;
                end
        end

endmodule
```

8.2.2 Features

Unsafe Operating Temperature: *A temperature of the reactor that if reached can lead to an unsafe meltdown of the reactor.*

```
feature unsafe();
begin
        var temp;
        (state == meltdown), temp = x
                |-> unsafe = temp;
end
```

Safe Operating Temperature: *Temperatures of the reactor reached during its safe operation.*

```
feature unsafe();
begin
        var temp;
        (state == Rod1) || (state==Rod2), temp = x
                |-> unsafe = temp;
end
```

8.3 Cruise Control

The cruise control model HA is shown in Figure 10.

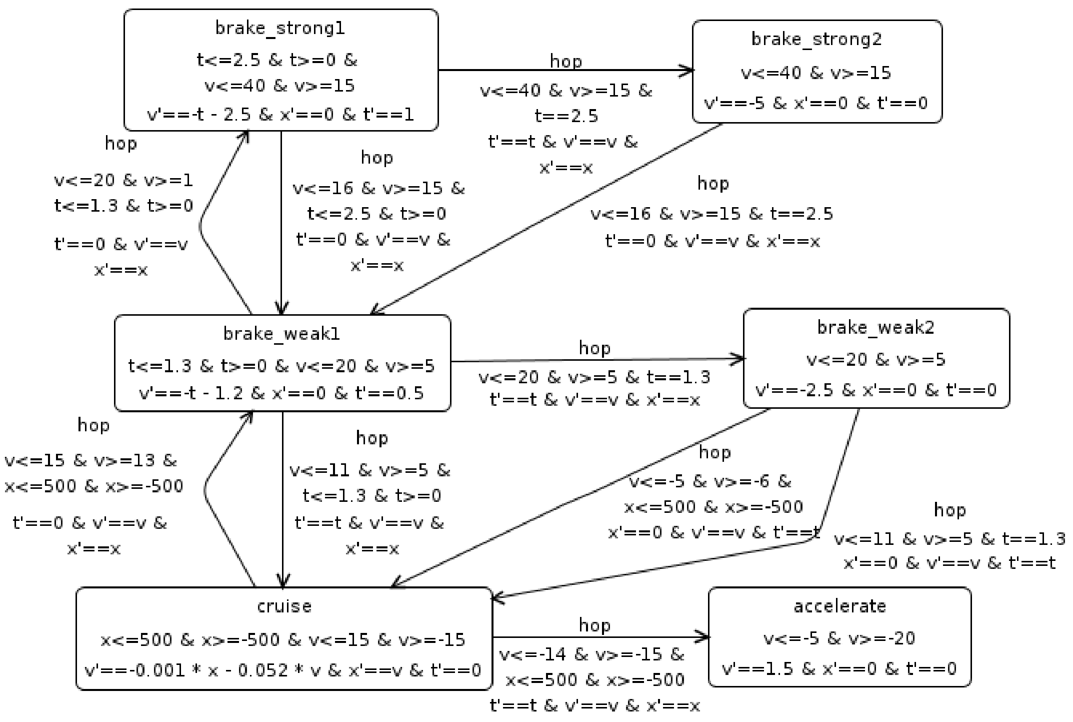


Figure 10: Cruise Control Hybrid Automaton Model *(A snippet from the SpaceEx Model Editor)*

8.3.1 HASLAC Description

```
module cruiseControl(v,x,t)
output v,x,t;

mode brake_strong1
begin
ddt v = -t - 2.5;
ddt x = 0;
ddt t = 1;
end

mode brake_strong2
begin
ddt v = -5;
ddt x = 0;
ddt t = 0;
end

mode brake_weak1
begin
ddt v = -t - 1.2;
ddt x = 0;
ddt t = 0.5;
end

mode brake_weak2
begin
ddt v = -2.5;
ddt x = 0;
ddt t = 0;
end

mode cruise
begin
ddt v = -0.001*x - 0.052*v;
ddt x = v;
ddt t = 0;
end

mode accelerate
begin
ddt v = 1.5;
ddt x = 0;
ddt t = 0;
end


property inv strong1
mode == brake_strong1 |=> v>=15 && v<=40 && t>=0 && t<=2.5;
endproperty

property inv strong2
mode == brake_strong2 |=> v>=15 && v<=40;
endproperty

property inv weak1
mode == brake_weak1 |=> v>=5 && v<=20 && t>=0 && t<=1.3;
endproperty

property inv weak2
mode == brake_weak2 |=> v>=5 && v<=20;
```

```
endproperty

property inv cruise
mode == cruise |=> v>=-15 && v<=15 && x>=-500 && x<=500;
endproperty

property inv accelerate
mode == accelerate |=> v>=-20 && v<=-5;
endproperty

property trans s1w1
mode == brake_strong1 && mode' == brake_weak1
&& t>=0 && t<=2.5 && v>=15 && v<=16
|=> t' == 0 && v'==v && x'==x;
endproperty

property trans s1s2
mode == brake_strong1 && mode' == brake_strong2
&& t==2.5 && v>=15 && v<=40
|=> t' == t && v'==v && x'==x;
endproperty

property trans s2w1
mode == brake_strong2 && mode' == brake_weak1 &&
t==2.5 && v>=15 && v<=16
|=> t' == 0 && v'==v && x'==x;
endproperty

property trans w1s1
mode == brake_weak1 && mode' == brake_strong1 &&
t>=0 && t<=1.3 && v>=18 && v<=20
|=> t' == 0 && v'==v && x'==x;
endproperty

property trans w1w2
mode == brake_weak1 && mode' == brake_weak2 &&
t==1.3 && v>=5 && v<=20
|=> t' == t && v'==v && x'==x;
endproperty

property trans w1c
mode == brake_weak1 && mode' == cruise &&
t>=0 && t<=1.3 && v>=5 && v<=11
|=> t' == t && v'==v && x'==x;
endproperty

property trans w2c
mode == brake_weak2 && mode' == cruise
&& t==1.3 && v>=5 && v<=11
|=> x' == 0 && v'==v && t'==t;
endproperty

property trans cw1
mode == cruise && mode' == brake_weak1 &&
x>=-500 && x<=500 && v>=13 && v<=15
|=> t' == 0 && v'==v && x'==x;
endproperty

property trans ca
mode == cruise && mode' == accelerate &&
x>=-500 && x<=500 && v>=-15 && v<=-14
|=> t' == t && v'==v && x'==x;
```

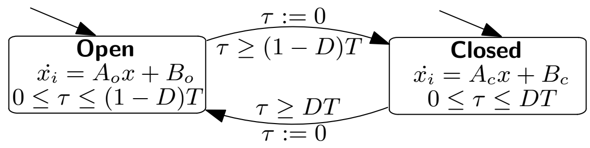


Figure 11: Buck Regulator Hybrid Automaton Model

```
endproperty

property trans ac
mode == brake_weak2 && mode' == cruise &&
x>=-500 && x<=500 && v>=-6 && v<=-5
|=> x' == 0 && v'==v && t'==t;
endproperty

initial begin
set begin
mode == brake_strong1;
x==0;
t>=0;
t<=2.5;
v>=0;
v<=40;
end
end
endmodule
```

8.3.2 Features

Time for Speed Capture from a precise speed difference for strong braking from a precise speed difference, `k`, to a zero difference between target and actual speed, i.e. when the target speed is reached.

```
feature speedCapturePrecise(k);
begin
var t1, t2;
((state == brake_strong2) && (v == k), t1=$time
##[0,$] ((state == cruise) && @+(v == 0), t2=$time
|-> speedCapturePrecise = t2-t1;
end
```

Time for Speed Capture from a range of speed differences from any speed difference within the range of [k_1,k_2] to a zero difference between target and actual speed, i.e. when the target speed is reached.

```
feature speedCaptureRange(k1,k2);
begin
var t1, t2;
(v >= k1 && v <= k2), t1 = $time ##[0,$]
((state == cruise) && @+(v == 0), t2 = $time
|-> speedCaptureRange = t2-t1;
end
```

8.4 Buck Regulator

The buck regulator model HA is shown in Figure 11.

8.4.1 HASLAC Description

```
module buck(v,i,t)
output v,i,t;

parameter
a00o = 0,
a01o = -21052.6316,
a10o = 38095.2381,
a11o = -40100.2506,
a00c = 0,
a01c = -21052.6316,
a10c = 38095.2381,
a11c = -40100.2506,
bounds = 1000,
T = 1e-05,
b0o = 0,
b1o = 0,
b0c = 21052.6316,
b1c = 0,
Vs = 12,
D = 0.51667;


mode closed
begin
        ddt t = 1;
        ddt v = (a10c * i + a11c * v + b1c * Vs);
        ddt i = (a00c * i + a01c * v + b0c * Vs);
end

mode open
begin
        ddt t = 1;
        ddt v = (a10o * i + a11o * v + b1o * Vs);
        ddt i = (a00o * i + a01o * v + b0o * Vs);
end

property inv closed
mode == closed
|=> v<=bounds && v>=-bounds && i<=bounds &&
i>=-bounds && t<=D * T && t>=0;
endproperty

property inv precharge
mode == open
|=> v<=bounds && v>=-bounds && i<=bounds &&
i>=-bounds && t<=(1-D) * T && t>=0;
endproperty

property trans closed_open
mode == closed && mode' == open && t>=D * T
|=> i'==i && t'==0 && v'==v;
endproperty

property trans open_closed
mode == open && mode' == closed && t>=(1-D) * T
|=> i'==i && t'==0 && v'==v;
endproperty

initial begin
set begin
```

```
mode == closed;
i == 0;
v == 0;
t == 0;
end
end

endmodule
```

8.4.2 Features

Settle Time: Time taken for the output voltage to settle to below $Vr + \epsilon$, where Vr is the rated voltage for the regulator, for two successive openings of the capacitor switch.

```
feature settleTime(Vr,E);
begin
        var st;
        (x1>=Vr+E) ##[0:$] @+(state==Open) && (x1<=Vr+E), st=$time
        ##[0:$] @+(state==Open) && (x1<=Vr+E)
                |-> settleTime = st;
end
```

Peak Overshoot: The maximum peak value of the voltage response curve measured from the desired response of the system.

```
feature overshoot(Vr);
begin
        var v1;
        (state == closed) && (x1 >= Vr), v1 = x1
                |-> overshoot = v1;
end
```

9 Proposed Extensions and Usability Enhancements

Through interactions with our liaisons on this project, a number of extensions and usability enhancements have been proposed. In this section we details those that have not yet been implemented. Note that this list will evolve with future versions of the tool.

9.1 Extensions to ForFET^{SMT}

During the cycle of evaluation, our liaisons proosed a number of extensions that would be useful to the use of ForFET^{SMT}. Here we list interesting ideas that may become enhancements to the tool in the future, but have not been incorporated into the present version of ForFET^{SMT}.

1. ***Global Time for Features, and Sharing of Feature Results across Analyses:*** It is proposed that there feature analyses may be synchronized using a global timer that is used across multiple feature analyses. Additionally, the results of one feature analysis may be used to guide a different feature analysis (synchronously/asynchronously).

2. ***Batch Feature Analyses:*** It is proposed that multiple features be annalyzed simulataneously. The sharing of information across feature analyses from Proposal 1 would also be relevant when analyzing multiple features, synchronously or asynchronously.

9.2 Usability Enhancements

1. ***Multiple Library Sources:*** ForFET^{SMT} allows the user to specify a library of models and features that are then available for analysis. It is proposed that users be allowed to specify multiple library paths, indicating multiple sources for models and respective features, that taken together would constitute a larger model/feature library.

2. ***Comments/Descriptions for Model Paramenters*** A formal (hybrid automaton) model may be treated as a model skeleton that is instantiated during analysis by assigning values to the paramenters of the model. Different combination of parameters yield models with possibly uniquely different behaviours. To aid with identification of parameters and decide on alterations to their values, it would be useful to provide a facility to associate comments with the parameters in the model. The comments associated with parameters are displayed during an interactive session with ForFET^{SMT}, as an aid to the user, to give insights into how a parameter change affects the model behaviour.

Acknowledgements

The authors gratefully acknowledge SRC funding for the partial financial support of this research through Tasks 2740 and 2267.

References

[1] G. Frehse et al. A toolchain for verifying safety properties of hybrid automata via pattern templates. In *ACC*, pages 2384–2391, June 2018.

[2] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *ICSE*, ICSE ’05, pages 372–381. ACM, 2005.