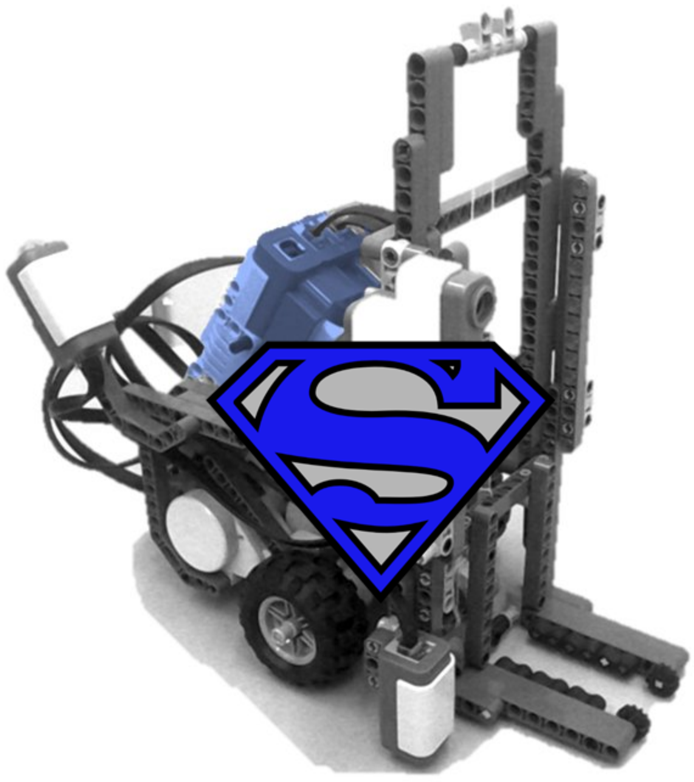


Spectra Language & Spectra Tools User Guide

Software Modeling Lab, Tel Aviv University

May 2023

Table of Contents

Introduction

Spectra is a specification language for reactive systems. The Spectra Tools are a collection of tools for performing various analyses of Spectra documents.

Outline of this Document

This document gives an overview of the Spectra language and the Spectra Tools. First, we describe the Spectra language, a specification language for reactive systems, in Chapter Spectra Language. Then we describe the Spectra Tools and the analyses they support in Chapter Spectra Tools Analyses. Parts of example specifications appear throughout the document. The complete example specifications can be found in Chapter Example Specifications.

Comments and Suggestions

We are currently improving the Spectra language and enhancing the capabilities of the Spectra Tools. We are interested in feedback on this document, the language, and the tools. Please do not hesitate to get in touch via the contact information found on our web pages:

https://www.cs.tau.ac.il/~maozs/
https://ringert.blogspot.com/

The SYNTECH Project

The Spectra language and Spectra Tools are part of the larger SYNTECH[1] project.

Reactive synthesis is an automated procedure to obtain a correct-by-construction reactive system from a given specification. Examples of these systems include the software controllers of robotic systems. Despite recent advancements on the theory and algorithms of reactive synthesis, e.g., efficient synthesis for the GR(1) fragment of linear temporal logic, many challenges remain in bringing reactive synthesis technologies to the hands of software engineers.

[1] Website of the SYNTECH project: https://smlab.cs.tau.ac.il/syntech/

The SYNTECH project is about bridging this gap. It addresses challenges that relate to the change from writing code to writing specifications, and the development of tools to support a specification-centric rather than a code-centric development process.

The Spectra Language

The Spectra language is a specification language for reactive systems. The language supports temporal constructs, including selected temporal operators. Spectra also provides high-level language constructs that allow engineers to make concepts like monitoring or counting explicit in specifications.

We give an overview of the different elements of the language. We introduce elements by a brief motivation, examples of the syntax of elements, and optional remarks.

Importing other Spectra Files

Spectra supports a mechanism for importing other Spectra documents. An import makes available the following elements from imported document:

- Pattern Definitions
- Predicate Definitions unless the predicate body references variables of the imported specification

The keyword for importing other files is `import`. The keyword has to be followed by a relative path enclosed by quotation marks of the file to import. Multiple files can be imported using separate import statements. Imports precede the Specification Declaration.

Example

```
import "../patterns/DwyerPatterns.spectra"
import "../patterns/MyPatterns.spectra"
```

The above example imports patterns defined in the files DwyerPatterns.spectra and MyPatterns.spectra.

Specification Declaration

Each Spectra document is a specification. Specifications have names. Each specification is defined in a separate file.
The keyword for specifications is `spec`.

Example

```
spec Rover
```

Remarks

- As a convention, the name of a specification should match the name of the file it is defined in.
- As a convention, the name of a specification should be capitalized.
- Currently, there is no technical significance to the name of a specification.

Variable Declarations

All variables in a specification are either environment controlled (input) or system controlled (output). Variable declarations state the type and the name of the variable.

Example

The following example declares an environment controlled variable `gravityDetected` of type boolean. In addition, it declares two system controlled variables.

The variable `speed` can have integer values from 0 to 7 (including 0 and 7). The variable `mode` can have one of the three enumeration values *SEEK*, *MEASURE*, or *IDLE*.

```
env boolean gravityDetected; // whether sensors detect gravity
sys Int(0..7) speed; // the speed set by the system
sys {SEEK, MEASURE, IDLE} mode; // current system mode
```

Variable Types

Spectra supports the following elements to be provided as variable types:

- `boolean` for Boolean variables with values true and false
- `Int`(l..u) for $l, u \in \mathbb{N}$ and $l < u$ to declare bounded integer variables
- Enumerations of concrete values, e.g., {*SEEK*, *MEASURE*, *IDLE*}
- Type Definitions

Arrays

All variables can be declared as (multi-dimensional) arrays by appending the size of each dimension inside square brackets. The following example declares a one-dimensional array (of length 2) of bounded integers from 0 to 7:

```
sys Int(0..7)[2] pos;
```

Arrays are accessed as follows:

```
pos[0]=5 // valid access of cell 0
pos[2]=5 // ERROR index out of bounds
```

Remarks

- Variable names must be unique.
- As a convention, variable names should start with lowercase letters.

Assumptions and Guarantees

Intuitively, in every state the environment provides a next input via environment variables (without seeing what the system will output in response). Then, the system outputs the next output via system variables based on the current state and the next input the environment has provided.

We describe the behavior of environments by assumptions on the environment behavior observable from the variables it controls. There are three basic types of assumptions:

- Initial assumptions: constraints over initial states of the environment (may only include environment variables)
- Safety assumptions starting with the temporal operator `always/alw`: constraints over current and next state (may only include `next`() around environment variables)
- Liveness assumptions starting with the temporal operator `alwaysEventyally/alwEv`: may include the `next`() operator around any variable

We describe the required behavior of the system by guarantees. There are three basic types of guarantees in Spectra which differ from assumptions in the variables that may appear in them:

- Initial guarantees: constraints over initial states of the environment (may include any variable)
- Safety guarantees starting with the temporal operator `always/alw`: constraints over current and next state (may include `next`() around any variables)
- Liveness guarantees starting with the temporal operator `alwaysEventyally/alwEv`: may include the `next`() operator around any variable

A special kind of assumptions and guarantees supported by Spectra in addition to the ones above are instances of Pattern Definitions.

Example

The following example shows an initial assumption with the name "`inBeginningNoGravity`", a safety assumption without a name, and a liveness assumption with the name "`eventuallyDetectGravity`".

```
asm inBeginningNoGravity:
  gravityDetected = false;

// once gravity is detected it does not go away
asm
  alw gravityDetected -> next(gravityDetected);

asm eventuallyDetectGravity:
  alwEv gravityDetected;
```

The next example shows an initial guarantee without a name, a safety guarantee without a name spanning multiple lines, and a liveness guarantee with the name "`doSomethingUseful`".

```
gar mode = IDLE; // start in IDLE mode

gar // if no gravity then no speed and be IDLE
```

```
  alw gravityDetected = false ->
          next(speed = 0) & next(mode) = IDLE;

gar doSomethingUseful:
  alwEv mode != IDLE;
```

Remarks

- Safety assumptions and guarantees may not include nested `next()` operators, i.e., safety constraints may not specify further than one step into the future.
- The `next()` operator applies to all variables in its scope, e.g., `next(x=x)` is a tautology while `next(x)=x` means that x will keep its current value in the next step. Constants are not affected, e.g., `next(x=1)` is the same as `next(x)=1.`
- Guarantees and assumptions can appear in any order in the Spectra document (their appearance can also be mixed).
- Some analyses use the names of assumptions and guarantees for providing reports otherwise the names have no significance.
- Names of assumptions and guarantees have to be unique.

Safety Assumptions/Guarantees and State Invariants

Safety guarantees and assumptions are always constraining transitions (even when the operator **next**() does not appear in them). However, some safety constraints do not contain the operator **next**(). Spectra detects these cases and treats the safety constraints as state invariants (they hold in the initial state and all next states) instead of transitions.

Without the automatic detection as state invariants, the meaning of the following safety guarantee would have been: states where `mode=MEASURE` and `speed!=0` have no valid successor, i.e., are a deadlock and these states violate the safety guarantee.

```
gar measureSpeedDeadlock: alw mode=MEASURE -> speed=0;
```

In some analyses, e.g., Strategy Synthesis, the difference will likely not be observed because the system will avoid deadlocks. However, in other analyses the non-translated constraints could lead to confusing results, e.g., for the Well-Separation Check.

PastLTL Operators

Spectra supports the use of PastLTL operators that evaluate formulas over the past interaction of the system and environment. These operators can be used in almost all places including assumptions and guarantees. PastLTL operators can also be nested.

PastLTL operators receive Boolean formulas as arguments and all PastLTL operators yield a Boolean evaluation. For Boolean formulas ψ and ϕ the PastLTL operators supported by Spectra are:

- `PREV` ϕ (also `Y`ϕ): ϕ was true in the previous state
- `ONCE` ϕ (also `O`ϕ): ϕis true now or was true in any past state
- `HISTORICALLY` ϕ (also `H`ϕ): ϕis true now and was true in all past states
- ψ `SINCE` ϕ (also ψ `S` ϕ): ψ is true now, ϕis true now or ϕ was true in some past state and when ϕ became false immediately ψbecame true and stayed true up until now
- ψ `TRIGGERED` ϕ (also ψ `T` ϕ):

Example

The following example uses PastLTL operators in an assumption and in a guarantee.

The assumption states that dock requests will only be sent by the environment once the system signaled that it is ready.

The guarantee expresses that the system will stay ready if a dockRequest has been true since some time where ready was true.

```
env boolean dockRequest;
sys boolean ready;
sys boolean docking;

asm alw next(dockRequest) -> ONCE(ready);

// stay ready if seen dockRequest since ready
gar alw (dockRequest SINCE ready) -> next(ready);
```

Remarks

- The PastLTL operator `PREV()` is very different from the general operator `next`(). The argument and the evaluation of `PREV()` are both Boolean while the evaluation of `next`() is of the type of its argument. As an example, `PREV(speed)=2` has two type inconsistencies (argument of `PREV()` is not Boolean and Boolean evaluation of `PREV()` cannot be compared to the number 2) while `next(speed)=2` is well-typed.
- PastLTL operators in assumptions cannot be used inside a `next`() operator (in guarantees they can).
- Note that the following two guarantees are quite different (not only syntactically different). A violation of the first is only detected one step later than a violation of the second:
  - `gar alw (PREV(speed=0) -> speed=1);`
  - `gar alw (speed=0 -> next(speed=1));`

Type Aliases

Spectra allows the definition of types (technically type aliases). Defined types can be used within variable or parameter declarations.

Type aliases start with the keyword `type`, a name, and the assignment of a type to the name.

Example

The following example defines three types:

- `Floors`: with integer values from 0 to 10
- `Position`: an array of integers from 0 to 7 of length 3
- `RoverMode`: an enumeration of items

```
type Floors = Int(0..10);
/** 3-dimensional position coordinates */
type Position = Int(0..7)[3];
type RoverMode = {SEEK, MEASURE, IDLE};
```

The next example shows how to use the last type alias in the declaration of a variable:

```
sys RoverMode mode; // current system mode
```

Remarks

- Type alias names must be unique.
- As a convention, type names should start with an uppercase letter.
- Comments preceding a type alias starting with `/**` will appear in tooltips when using the alias.

Defines

Defines introduce shortcuts for expressions. They are similar to macros. Expressions abbreviated by defines can evaluate to any type.

Defines start with the keyword **`define`**. The keyword is followed by a name, the operator :=, and a Spectra expression to be associated with the name. Multiple definitions can follow a single **`define`** keyword.

Example

```
type Floors = Int(0..10);
env Floors elevatorLocation;
env Floors request;

define
  requestedUp := request > elevatorLocation;
  requestedDown := request < elevatorLocation;

define
  openRequest := request != elevatorLocation;
```

A convenient way to improve readability and avoid the introduction of additional variables is the definition of constant arrays. The size of the constants array follows its name in square brackets [ ] (in regular arrays the size is declared as part of the type). The values are written in curly brackets { } and separated by commas.

Example

```
sys Int(0..15)[NUM_OF_ROBOTS] targetLocation;
define
  NUM_OF_ROBOTS := 3;
define
  basePositions[NUM_OF_ROBOTS] := { 0, 5, 14 };

gar ini targetLocation[0] = basePositions[0];
```

Remarks

- Be careful when placing the operator `next`() inside defines and using the defined name inside the scope of another `next`(). In this case the application would happen twice. This is not allowed.
- Defines can reference other elements, e.g., other defines, variables, monitors, or counters.

Predicate Definitions

Predicates allow for encapsulation and reuse of parameterized, Boolean expressions. Predicate definitions define the name of a predicate, its typed parameters, and the body of the predicate. The body of the predicate is a Boolean expression that may reference elements of the specification and the predicate parameters.

Example

The following example defines the predicate `carries` with parameter `i` of type `Item`. A reference to the predicate evaluates to true iff the value of the first or the second cell of the array `cargo[0]` equals to the value of parameter `i`.

```
/**
* item i the farmer carries (as cargo[0] or cargo[1])
* @param i
```

```
*/
predicate carries(Item i):
  (cargo[0]=i | cargo[1]=i);
```

Predicates can be instantiated by providing concrete values to parameters or by passing variables as parameters. In the following example the predicate `carries` is instantiated three times inside an assumption with different concrete values.

```
asm notCarryingAnItemKeepsItemPosition:
  alw (!carries(Wolf) -> wolfPos=next(wolfPos)) &
      (!carries(Cabbage) -> cabbagePos=next(cabbagePos)) &
      (!carries(Goat) -> goatPos=next(goatPos));
```

Remarks

- The body of a predicate can be either enclosed by ":" and ";" or by "{" and "}".
- Two major differences between defines and predicates are that
  - defines cannot have parameters and
  - predicates can only evaluate to true or false while defines can evaluate to any type.
- If predicates should be imported into other specifications the body of the predicate has to be restricted to parameter values only and not contain references to variables of the imported Spectra document.

Monitor Definitions

Monitor definitions introduce monitors with a type and a name. The body of a monitor defines how the value of the monitor is updated in each step. Monitors allow for keeping track of events over time.

The body of monitors consists of initial constraints and safety constraints. Constraints can refer to the name of the monitor as a variable. Together, the constraints have to uniquely determine the value of the monitor.

Example

The following listing defines a monitor of type **boolean** with the name `expectFinding`. The body of the monitor defines how the monitor's value is updated using an initial and a safety constraint. Initially `expectFinding` is false. Then, `expectFinding` becomes true iff the system mode is SEEK or we expected a finding and nothing is found so far.

```
/**
 * monitor whether system is expecting to find something after SEEK
 */
monitor boolean expectFinding {
  !expectFinding;
  alw next(expectFinding) = (mode=SEEK | expectFinding & !found);
}
```

The monitor can be referenced in places where any system variables can be referenced, e.g., in liveness assumptions:

```
asm alwEv !expectFinding;
```

The monitor itself is neither an assumption nor a guarantee. However, the above assumption states that the environment has the responsibility to influence the monitor's value -- here, by setting found=true some steps after seeing mode=SEEK.

Remarks

- Monitors are instantiated directly and only once.
- Monitors can be referenced multiple times, e.g., in assumptions and in guarantees at the same time.
- The reference to a monitor inside the operator **next**() cannot appear in an assumption (imagine monitors to be system variables).
- Make sure the value of the monitor is always uniquely defined! If the value is not unique then synthesized strategies might fail. If value assignments inside the monitor body are contradicting synthesis will fail.

Counter Definitions

Spectra supports the definition and use of bounded counters. Counters have a name that can be used as a reference to the value of the counter in arithmetic expressions. Counter values can be compared to other bounded integers or integer expressions.

Counters have a body with the following fields:

- Initial value assignment to counter
- `inc` (optional): constraint when to increase the counter
- `dec` (optional): constraint when to decrease the counter
- `reset` (optional): constraint when to reset the counter
- `overflow` (optional): an overflow type, either
    - `false` (default) - overflow is a violation of the counter,
    - `modulo` - an overflow restarts counting from lower bound of counter, or
    - `keep` - an overflow will result in keeping the value of the upper bound of the counter
- `underflow` (optional): an underflow type, either
    - `false` (default) - underflow is a violation of the counter,
    - `modulo` - an underflow restarts counting from upper bound of counter, or
    - `keep` - an underflow will result in keeping the value of the lower bound of the counter

Example

The listing below shows the definition of a counter named `blinks` with domain 0 to 5. The counter is initialized to 0. The counter is increased when at a station and blinking. The counter is reset when not at a station. An overflow of the counter is forbidden.

```
define blink := next(light)!=light;

counter blinks(0..5) {
      // initially no blinks
      blinks=0;
      // blinking is turning light on when it was off
      inc: atStation & blink;
      // reset counter when not at station
      reset: !atStation;
      // blink exactly 5 times
      overflow: false;
}
```

The counter `blinks` is used in the following guarantee:

```
// if we have blinked less than 5 times at station do blink
gar alw atStation & blinks < 5 -> blink;
```

Remarks

- Make sure the fields inside the counter definition do not contradict each other, e.g., ensure that `inc` does not overlap with `dec`.
- Counters are internally treated as guarantees. If a counter has contradicting `inc/dec/reset` fields, if it overflows with `overflow: false`, or if it underflows with `underflow: false` synthesis sees it as the violation of a guarantee. The specification can become unrealizable!

Pattern Definitions

Pattern definitions encapsulate reusable units of specifications. Pattern definitions have a name, a list of parameter names, and a body. The body of a pattern definition can declare variables (only visible inside the pattern) and constraints over variables and parameters. Constraints can be initial, safety, or liveness constraints. Parameters are Boolean.

Patterns can be instantiated as assumptions or guarantees. If a pattern instance is a guarantee then the system has to ensure satisfaction of all liveness constraints defined in the pattern. Otherwise, if the pattern is instantiated as an assumption the environment has to ensure satisfaction of the liveness constraints.

Parameters of pattern instances are Boolean expressions.

Example

The following pattern definition defines a pattern with the name `S_responds_to_P_globally` and two parameters s and p. The comment above the pattern will be shown in tooltips when instantiating the pattern. Comments on patterns may contain HTML tags as, e.g., supported by JavaDoc.

```
/**
 *<p>
 * <b>Kind:</b> Response: s responds to p <br>
 * <b>Scope:</b> Globally<br>
 * <b>LTL:</b> G (!p || F s) (also G(p -> Fs))
 * </p>
 */
pattern S_responds_to_P_globally(s, p) {
  var { S0, S1} state;
  // initial assignments: initial state
  ini state=S0;
  // safety this and next state
  alw ((state=S0 & ((!p) | (p & s)) & next(state=S0)) |
  (state=S0 & (p & !s) & next(state=S1)) |
  (state=S1 & (s) & next(state=S0)) |
  (state=S1 & (!s) & next(state=S1)));
  // equivalence of satisfaction
  alwEv (state=S0);
}
```

The following example instantiates the response pattern in the context of a specification. First, it is instantiated as an assumption, then it is instantiated as a guarantee. Both instances are independent of each other.

```
// instantiate response pattern for eventually reaching the
// requested floor
asm S_responds_to_P_globally(elevatorLocation = request,
                             requestedUp & command = UP);

gar eventuallyHandleOpenRequests:
  S_responds_to_P_globally(request=elevatorLocation, openRequest);
```

Remarks

- The body of patterns can only refer to variables of the pattern or its parameters. It cannot refer to other names (of variables/patterns/monitors/defines etc.) defined inside the specification.

- We provide a catalog of most of the popular LTL specification patterns of Dwyer et al. for Spectra.
- Make sure the value of the variables inside the pattern is always uniquely defined! If the value is not unique then synthesized strategies might fail in some environments. If value assignments to pattern variables are contradicting the synthesis will fail.
- Patterns without liveness constraints have no meaning.

Further reading

S. Maoz and J. O. Ringert, GR(1) Synthesis for LTL Specification Patterns. Proc. of ESEC/FSE 2015, pp.96-106, ACM, 2015.

The paper and materials are available from: https://smlab.cs.tau.ac.il/syntech/patterns/

Weight Definitions

The Spectra language supports the definition of integer weights over states and transitions of a reactive system specification. Weights can have a name, an integer value, and a constraint on transitions that the weights apply to.

If not specified, the weight is 0 for all transitions.

Example

The following example defines weights with values -1, 1, -2, and 3 for different constraints over environment and system variables. Notice that some constraints are overlapping, e.g., `carMain & goMain` satisfy the second weight (value 1) and the last weight (value 3) and thus receive the combined weight value 1 + 3 = 4.

```
env boolean carMain;
env boolean carSide;
sys boolean goMain;
sys boolean goSide;

weight -1
  carMain & !goMain | carSide & !goSide;
weight 1
  carMain & goMain | carSide & goSide;
```

```
weight -2
  carMain & !goMain;
weight 3
  carMain & goMain;
```

Remarks

- The constraints of weight definitions can be overlapping (see example). Values associated with satisfied constraints are added up.
- The meaning of weights can be defined in different ways. We show one example in Section Strategy Synthesis with Weights.

Further reading

G. Amram, S. Maoz, O. Pistiner, and J. O. Ringert, Efficient Algorithms for Omega-Regular Energy Games. Proc. of FM 2021.

The paper and materials are available from:
https://smlab.cs.tau.ac.il/syntech/energyefficient/

Quantifiers and Variable Indexes

Spectra language supports existential and universal quantification over logical formulas with the help of two keywords, **`exists`** and **`forall`**, respectively. The quantification acts mainly as a convenience tool (“syntactic sugar”) and is translated to a disjunction or conjunction of several formulas before any further analysis (e.g., synthesis).

Another feature is the usage of integer variables defined in the specification as array indices. This is also a “syntactic sugar” and is translated to a disjunction of formulas in a manner that is presented in the example below.

Example

The following example defines a specification with two assertions:

- A safety assumption `allColorsRepresented` expressing that at any time, each `Color` value is represented in the array at some index.

- A liveness guarantee `alwaysRedAtIndex` expressing that the value of the array at the index defined by `ind` variable should be infinitely often red.

```
type Color = {RED, GREEN, BLUE};

env Color[3] colors;
sys Int(0..2) ind;

asm allColorsRepresented:
    alw forall c in Color. exists i in Int(0..2). colors[i] = c;

gar alwaysRedAtIndex:
    alwEv colors[ind] = RED;
```

This specification is internally translated to the following code:

```
asm allColorsRepresented:
    alw (colors[0] = RED | colors[1] = RED | colors[2] = RED) &
      (colors[0] = GREEN | colors[1] = GREEN | colors[2] = GREEN) &
      (colors[0] = BLUE | colors[1] = BLUE | colors[2] = BLUE);

gar alwaysRedAtIndex:
    alwEv (ind = 0 & colors[0] = RED) |
      (ind = 1 & colors[0] = RED) |
      (ind = 2 & colors[0] = RED);
```

Remarks

- The quantified variable can be either an integer, a boolean, or an enum variable.
- The scope of the quantified variable is naturally limited to the formula that follows **`exists`** or **`forall`** keywords.
- Index arithmetic is supported for ordinary variables, quantified variables, and defines, as long as they are of type integer.

Triggers and Regular Expressions

A Spectra trigger **`trig`** L |=> R intuitively states that whenever a prefix of a computation satisfies L in the next step R should be satisfied. In triggers L and R are regular expressions over states. The semantics of triggers in Spectra is non-overlapping, i.e., when L is satisfied, R is monitored, and only after R is satisfied, L is monitored again for satisfaction (from the step immediately after R was satisfied).

Regular Expressions

The alphabet of regular expressions in Spectra are states. As usual, these are described in expressions over variables. Each state expression is enclosed in square brackets [...]. As an example, `[carA & !greenA]` describes all states where `carA` is true and `greenA` is false.

The operators of regular expressions are:

Operator	**Symbol**	**Example**
concatenation	[...][...]	[true] [carA & !greenA]
union	[...] | [...]	[carB] | [carA]
intersection	[...] & [...]	[carB] & [greenB]
kleene-star (zero-or-more)	[...]*	[carA & !carB]*
plus (one-or-more)	[...]+	[carA & !carB]+
optional (zero-or-one)	[...]?	[carA]?
exact repetitions	[...]{k}	[carA]{4}
at-least-repetitions	[...]{k, }	[carA]{4, }
ranged-repetitions	[...]{k, m}	[carA]{1, 4}
negation	~[...]	~[carA]

Parenthesis may be used to group and structure regular expressions.

Triggers

Triggers start with the keyword `trig` and may appear as assumptions or guarantees. The left side and right side of triggers are regular expressions separated by the trigger operator |=>. The Spectra code below shows three example triggers.

```
sys boolean greenA;
sys boolean greenB;

env boolean carA;

env boolean carB;

asm trig [true]* [carA] |=> [true]{0, 3} [carB];

gar trig [true]* [greenA & greenB] |=> [false];

gar trig [true]* [carA & !greenA]{4} |=> [greenA];
```

The first trigger is an assumption that after seeing a state with `carA` at most 4 states later we are in a state with `carB`. Note that `[true]{0, 3}` matches any state 0 up to 3 times and that the right side of the trigger only matches after the left has matched. Thus, the trigger could match `[carA] [carB]` but not `[carA & carB]`. The trigger matches `[carA] [carB] [carA] [carB]` twice, but matches `[carA] [carB & carA] [carB]` only once.

The second trigger is a guarantee that a state with `greenA & greenB` never occurs. Here, the right side of the trigger is `[false]` and it is thus impossible to match it. Note that this simple case is not presenting a good use of triggers and instead one should write `alw` !(`greenA & greenB`).

Finally, the third trigger expresses that after seeing `carA & !greenA` for exactly 4 consecutive states the next state must satisfy `greenA`.

Further reading

Gal Amram, Dor Ma'ayan, Shahar Maoz, Or Pistiner, and Jan O. Ringert, Triggers for Reactive Synthesis Specifications. Proc. of ICSE 2023.

The paper and materials are available from: https://smlab.cs.tau.ac.il/syntech/triggers

Spectra Tools Analyses

The Spectra Tools implement various analyses of Spectra specifications. These include the synthesis of controllers that guarantee to satisfy specifications.

Strategy Synthesis

Spectra Tools will try to find an implementation that satisfies the GR(1)[2] specification defined by the Spectra document. A GR(1) specification consists of assumptions, which have to be satisfied by the environment, and guarantees, which have to be satisfied by the system, i.e., by the implementation we synthesize.

Assumptions and guarantees can be on initial states (start with optional temporal operator **`ini`**), on all consecutive states (start temporal operator **`alw`**), or on states to visit infinitely often (start with temporal operators **`alwEv`**). We call these three kinds initial, safety, and liveness assumptions and guarantees. If synthesis is successful, the synthesized implementation of the specification ensures:

- If the environment satisfies the initial assumptions then the system satisfies the initial guarantees,
- as long as the environment satisfies all safety assumptions, the system satisfies all safety guarantees, and
- as long as the environment satisfies all liveness assumptions, the system satisfies all liveness guarantees.[3]

In case synthesis is not successful, it is certain that no implementation with the above properties can exist for the given Spectra specification! We call these specifications unrealizable.

[2] For more information on GR(1) see Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, Yaniv Sa'ar: Synthesis of Reactive(1) designs. J. Comput. Syst. Sci. 78(3): 911-938 (2012)
[3] Technically, this condition is called strict realizability and stronger than implication realizability where the satisfaction of all assumptions implies the satisfaction of all guarantees. We chose this semantics because it usually creates implementations that better try to satisfy the guarantees rather than forcing the environment to violate assumptions. See Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, Yaniv Sa'ar: Synthesis of Reactive(1) designs. J. Comput. Syst. Sci. 78(3): 911-938 (2012)

Concrete Controller

[*Right click inside editor or on Spectra file ->*
Spectra -> Synthesize Concrete Controller]

A concrete controller synthesized for the specification Rover.spectra is shown below:

```
state INI[initial], S0, S1;

INI -> S1 {gravityDetected:false, found:false} / {speed:0, mode:IDLE};
INI -> S1 {gravityDetected:false, found:true} / {speed:0, mode:IDLE};
S0 -> S0 {gravityDetected:true, found:false} / {speed:0, mode:MEASURE};
S0 -> S0 {gravityDetected:true, found:true} / {speed:0, mode:MEASURE};
S1 -> S1 {gravityDetected:false, found:false} / {speed:0, mode:IDLE};
S1 -> S1 {gravityDetected:false, found:true} / {speed:0, mode:IDLE};
S1 -> S0 {gravityDetected:true, found:false} / {speed:0, mode:IDLE};
S1 -> S0 {gravityDetected:true, found:true} / {speed:0, mode:IDLE};
```

The synthesized automaton has three states. Each line denotes a transition consisting of the source state, the target state, the variable values chosen by the environment, and the variable values chosen by the system in response.

Controllers have to accept any assignment to environment variables that satisfies the assumptions in every state. Thus, concrete controllers can become quite large. As an example, a controller for specification Elevator.spectra has more than 2000 transitions.

Symbolic Controller

[*Right click inside editor or on Spectra file ->*
Spectra -> Synthesize … Symbolic Controller]

Spectra Tools allows for efficient representation of controllers as a formula of allowed transitions. These formulas are represented by a special data structure called BDDs. Spectra Tools store symbolic controllers in a machine readable format in a subfolder /out/ relative to the specification.

Currently, two variants of symbolic controllers. Both handle controllers with thousands of states and transitions where concrete controller synthesis would reach its limits. The

“Just-in-time” variant is typically faster and more memory efficient than the “Static” variant as it does not create a monolithic transition relation in one BDD.

Further reading

Shahar Maoz, Ilia Shevrin: Just-In-Time Reactive Synthesis. ASE 2020: 635-646

The paper and materials are available from: https://smlab.cs.tau.ac.il/syntech/jits/

Strategy Synthesis with Weights

Specifications with weight definitions are treated as bounded energy games. The system starts with an initial energy credit within a bound given by the user. The values of weights are accumulated in every step. Accumulation that exceeds the bound is truncated to the bound. The system loses when the accumulated value becomes negative.

Further reading

S. Maoz, Or Pistiner, and J. O. Ringert, Symbolic BDD and ADD Algorithms for Energy Games. SYNT 2016: 5th Workshop on Synthesis (with CAV'16).

The paper and materials are available from: https://smlab.cs.tau.ac.il/syntech/energy/

Counterstrategy Synthesis

In case a specification is unrealizable some environment strategy that satisfies all assumptions can prevent any system from satisfying all guarantees. It might be helpful to compute and inspect an example of such an environment strategy.

As an example, consider the following unrealizable specification of an elevator:

```
type Floors = Int(0..3);
sys Floors elevatorLocation;
env Floors request;

gar startOnGroundFloor:
  elevatorLocation=0;
```

```
gar moveOneFloorAtATime:
  alw (next(elevatorLocation) = elevatorLocation+1) |
      (next(elevatorLocation) = elevatorLocation-1);

gar eventuallyHandleOpenRequests:
  S_responds_to_P_globally(request  = elevatorLocation,
                                    request != elevatorLocation);
```

Concrete Counterstrategy

[*Right click inside editor or on Spectra file ->*
Spectra -> Synthesize Concrete Counter Strategy]

A concrete counterstrategy demonstrates behavior of the environment that forces the system to violate at least one guarantee:

The following is a counterstrategy where in every state the environment chooses a request and the system can chose a next elevatorLocation. The last guarantee of the specification to eventually ensure `request=elevatorLocation` cannot be satisfied:

```
state INI[initial], S0, S1, S2, S3;

INI -> S3 {request:2} / {elevatorLocation:0};
S3 -> S2 {request:0} / {elevatorLocation:1};
S2 -> S3 {request:1} / {elevatorLocation:0};
S2 -> S0 {request:1} / {elevatorLocation:2};
S1 -> S0 {request:0} / {elevatorLocation:2};
S0 -> S1 {request:0} / {elevatorLocation:3};
S0 -> S2 {request:0} / {elevatorLocation:1};
```

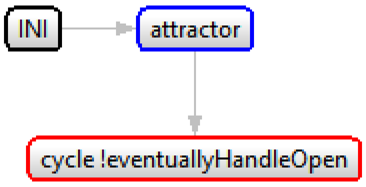
With this counterstrategy, the environment forces the system to violate the last guarantee by always setting a request that cannot be handled by the system in one step (system can only move one floor at a time).

Symbolic Counterstrategy (JVTS)

[*Right click inside editor or on Spectra file* ->
Spectra Add-ons -> GR(1) Counter Strategy Generator ->
Generate Symbolic Counter Strategy]

When concrete counterstrategies get too large it might help to display a high-level abstraction of counterstrategies consisting of symbolic nodes. The nodes are either

- attractors: the environment forces the system out of the concrete states contained in these nodes, or
- cycles: the system can stay in a cycle of concrete states forever but violates at least one liveness guarantee.



An example JVTS for the unrealizable elevator specification is shown on the right.

Remarks

- Concrete counterstrategies must provide a transition/state for every possible choice of the system and can thus get very large.

Further reading

A. Kuvent, S. Maoz and J. O. Ringert, A Symbolic Justice Violations Transition System for Unrealizable GR(1) Specifications. FSE'17, 2017.

The paper and materials are available from: https://smlab.cs.tau.ac.il/syntech/jvts/

Unrealizable Core Computation

[*Right click inside editor or on Spectra file* ->
Spectra Add-ons -> Cores -> Find an Unrealizable Core]

When a specification is unrealizable, some subset of the guarantees cannot be satisfied together. Usually, a minimal unrealizable subset of guarantees is small[4] and studying it may help to understand reasons for unrealizability. Due to minimality, removing any guarantee from a core makes the combination of guarantees remaining in the core realizable. However, It is not always the case that removing a guarantee from the core makes the whole specification realizable. Sometimes multiple reasons for unrealizability exist.

Example

In the above example of the unrealizable elevator specification an unrealizable core consists of the two last guarantees. Together these make the specification unrealizable and removing any of them makes the specification realizable. Removing the first allows the system to immediately jump to the requested floor. Removing the second does not require the system to move to a requested floor.

```
type Floors = Int(0..3);
sys Floors elevatorLocation;
env Floors request;

gar startOnGroundFloor:
  elevatorLocation=0;

gar moveOneFloorAtATime:                                                  _
  alw (next(elevatorLocation) = elevatorLocation+1) |                     _
        (next(elevatorLocation) = elevatorLocation-1);                    _

gar eventuallyHandleOpenRequests:                                         _
  S_responds_to_P_globally(request  = elevatorLocation,                   _
                                    request != elevatorLocation); _
```

Further reading

S. Maoz and R. Shalom, Unrealizable Cores for Reactive Systems Specifications. Proc. of ICSE 2021.

[4] There is not necessarily a unique minimal unrealizable subset, i.e., minimality might be local.

The paper and materials are available from:
https://smlab.cs.tau.ac.il/syntech/unrealcores/

Well-Separation Check

[*Right click inside editor or on Spectra file* ->
Spectra Add-ons -> Well-Separation ->
Diagnose Well-Separation of Environment]

Well-separation is a property of the environment specification, i.e., the assumptions in a Spectra document. An environment is well-separated iff it cannot be forced to violate its assumptions from any reachable state. The property of well-separation is desirable.

In case an environment is non-well-separated a possible implementation of a system can force the environment to violate some assumption. Once the environment violates an assumption the system implementation does no longer have to satisfy the guarantees (see conditions described in Strategy Synthesis).

Example of a non-well-separated environment specification

The following example shows a set of safety and liveness assumptions, which constitute a non-well-separated environment. There are multiple ways how a system can force the environment to violate at least one of its assumptions.

```
env boolean atStation;
env boolean cargo;
sys {STOP, FWD, BWD} mot;
sys {LIFT, DROP} lift;

asm findStat: // always possible to find a station
  alwEv (atStation);
asm samePos: // same station position when stopped
  alw (mot=STOP -> next(atStation)=atStation);
asm liftCargo: // lifting clears sensor
  alw (lift=LIFT -> next(!cargo));
asm dropCargo: // dropping senses cargo
  alw (lift=DROP -> next(cargo));
asm clearCargo: // backing up clears cargo
```

```
  alw (mot=BWD -> next(!cargo));
```

For a specific example see the next section. For more examples and a formal definition, see the materials under Further reading.

Non-Well-Separated Core Computation

[Right click inside editor or on Spectra file ->
Spectra Add-ons -> Well-Separation ->
Compute Non-Well-Separation Core]

In addition to checking well-separation, Spectra Tools can also compute a minimal subset of the assumptions that still make an environment specification non-well-separated.

The following highlighting shows a non-well-separated core. The two first assumptions make the environment specification non-well-separated. Any system can stop the motor once it is away from a station. Then the environment has to violate the first assumption of always eventually finding a station.

```
asm findStat: // always possible to find a station
  alwEv (atStation);
asm samePos: // same station position when stopped
  alw (mot=STOP -> next(atStation)=atStation);
asm liftCargo: // lifting clears sensor
  alw (lift=LIFT -> next(!cargo));
asm dropCargo: // dropping senses cargo
  alw (lift=DROP -> next(cargo));
asm clearCargo: // backing up clears cargo
  alw (mot=BWD -> next(!cargo));
```

Further reading

S. Maoz and J. O. Ringert, On Well-Separation of GR(1) Specifications. Proc. of ESEC/FSE 2016, pp.362-372, ACM, 2016.

The paper and materials are available from:
https://smlab.cs.tau.ac.il/syntech/separation

Complete Example Specifications

Blink5.spectra

The complete specification is realizable and well-separated.

```
spec Blink5

env boolean atStation;
sys boolean light;

define blink := next(light)!=light;

counter blinks(0..5) {
      // initially no blinks
      blinks=0;
      // blinking is turning light on when it was off
      inc: atStation & blink;
      // reset counter when not at station
      reset: !atStation;
      // blink exactly 5 times
      overflow: false;
}

// if we are not blinking do blink
gar alw atStation & blinks < 5 -> blink;
```

Docking.spectra

The complete specification is realizable and well-separated.

```
import "DwyerPatterns.spectra"

spec Docking

env boolean dockRequest;
sys boolean ready;
sys boolean docking;

asm alw next(dockRequest) -> ONCE(ready);

// stay ready if seen dockRequest since ready
gar alw (dockRequest SINCE ready) -> next(ready);

gar alwEv ready;

gar S_responds_to_P_globally(docking, dockRequest);
```

Elevator.spectra

The complete specification is realizable and well-separated.

```
import "DwyerPatterns.spectra"

spec Elevator

type Floors = Int(0..10);
env Floors elevatorLocation;
env Floors request;
sys {UP, DOWN} command;

define
  requestedUp := request > elevatorLocation;
  requestedDown := request < elevatorLocation;
  openRequest := request != elevatorLocation;

// maintain requests while not answered
asm alw elevatorLocation != request -> next(request) = request;

// instantiate response pattern for eventually reaching the
// requested floor
asm S_responds_to_P_globally(elevatorLocation = request,
      requestedUp & command = UP);

asm S_responds_to_P_globally(elevatorLocation = request,
      requestedDown & command = DOWN);

asm moveRightDirection:
  alw (command=UP -> next(elevatorLocation) >= elevatorLocation) &
      (command=DOWN -> next(elevatorLocation) <= elevatorLocation);

gar eventuallyHandleOpenRequests:
  S_responds_to_P_globally(request=elevatorLocation, openRequest);
```

ElevatorUnrealizable.spectra

The complete specification is unrealizable and well-separated.

```
import "DwyerPatterns.spectra"

spec ElevatorUnrealizable

type Floors = Int(0..3);
sys Floors elevatorLocation;
env Floors request;

gar startOnGroundFloor:
  elevatorLocation=0;

gar moveOneFloorAtATime:
  alw (next(elevatorLocation) = elevatorLocation+1) |
      (next(elevatorLocation) = elevatorLocation-1);

gar eventuallyHandleOpenRequests:
  S_responds_to_P_globally(request  = elevatorLocation,
                                    request != elevatorLocation);
```

This assumption is a fix of unrealizability:

```
asm maintainRequestsIfNotAnswered:
  alw elevatorLocation != request -> next(request) = request;
```

NonWellSep.spectra

The complete specification is realizable and non-well-separated.

```
spec NonWellSep

env boolean atStation;
env boolean cargo;
sys {STOP, FWD, BWD} mot;
sys {LIFT, DROP} lift;

asm findStat: // always possible to find a station
  alwEv (atStation);
asm samePos: // same station position when stopped
  alw (mot=STOP -> next(atStation)=atStation);
asm liftCargo: // lifting clears sensor
  alw (lift=LIFT -> next(!cargo));
asm dropCargo: // dropping senses cargo
  alw (lift=DROP -> next(cargo));
asm clearCargo: // backing up clears cargo
  alw (mot=BWD -> next(!cargo));
```

Rover.spectra

The complete specification is realizable and well-separated.

```
spec Rover

env boolean gravityDetected; // whether sensors detect gravity
env boolean found; // whether direction was found after seek
sys Int(0..7) speed; // the speed set by the system
sys {SEEK, MEASURE, IDLE} mode; // current system mode

asm inBeginningNoGravity:
  gravityDetected = false;
// once gravity is detected it does not go away
asm
  alw gravityDetected -> next(gravityDetected);
asm
  eventuallyDetectGravity: alwEv gravityDetected;

gar mode = IDLE; // start in IDLE mode
gar // if no gravity then no speed and be IDLE
  alw gravityDetected = false ->
           next(speed = 0) & next(mode) = IDLE;
gar alw next(mode)=SEEK -> next(speed) != 0;
gar doSomethingUseful:
  alwEv mode != IDLE;

/** monitor whether system is expecting finding after SEEK */
monitor boolean expectFinding {
  !expectFinding;
  alw next(expectFinding) = (mode=SEEK | expectFinding & !found);
}

asm alwEv !expectFinding;
```

TrafficLight.spectra

The complete specification is realizable and well-separated.

```
spec TrafficLight

env boolean carMain;
env boolean carSide;
sys boolean goMain;
sys boolean goSide;

asm alwEv carSide;
asm alwEv carMain;

gar alw !(goMain & goSide);
gar alwEv carSide & goSide;
gar alwEv carMain & goMain;

weight -1
  carMain & !goMain | carSide & !goSide;
weight 1
  carMain & goMain | carSide & goSide;
weight -2
  carMain & !goMain;
weight 3
  carMain & goMain;
```

DiningPhilosophers.spectra

The complete specification is realizable and well-separated.

```
import "DwyerPatterns.spectra"

spec DiningPhilosophers

type State={FREE, LEFT, RIGHT};
sys State[5] chopsticks;
env boolean[5] eatRequest;

// Initially there are no requests
asm initialNoRequests:
    forall i in Int(0..4). !eatRequest[i];

// Initially the chopsticks are not taken from either side
gar initialChopsticksFree:
    forall i in Int(0..4). chopsticks[i]=FREE;

// Eating philosopher stops requesting for chopsticks
asm eatingDoesntRequest:
    alw forall i in Int(0..4). (chopsticks[i]=RIGHT &
chopsticks[(i+1)%5]=LEFT) -> next(!eatRequest[i]);

// Keep requesting if one of the chopsticks is taken by neighbor
asm keepRequesting:
    alw forall i in Int(0..4). (eatRequest[i] & (chopsticks[i]=LEFT |
chopsticks[(i+1)%5]=RIGHT)) -> next(eatRequest[i]);

// Right philosopher gets left chopstick once freed
gar getLeftChopstick:
    alw forall i in Int(0..4). (eatRequest[i] & chopsticks[i]=FREE)
-> next(chopsticks[i]=RIGHT);

// Left philosopher gets right chopstick once freed
gar getRightChopstick:
    alw forall i in Int(0..4). (eatRequest[i] &
chopsticks[(i+1)%5]=FREE) -> next(chopsticks[(i+1)%5]=LEFT);

// Each philosopher must eat infinitely often
gar S_responds_to_P_globally(chopsticks[0]=RIGHT &
chopsticks[1]=LEFT, eatRequest[0]);
```

```
gar S_responds_to_P_globally(chopsticks[1]=RIGHT &
chopsticks[2]=LEFT, eatRequest[1]);

gar S_responds_to_P_globally(chopsticks[2]=RIGHT &
chopsticks[3]=LEFT, eatRequest[2]);

gar S_responds_to_P_globally(chopsticks[3]=RIGHT &
chopsticks[4]=LEFT, eatRequest[3]);

gar S_responds_to_P_globally(chopsticks[4]=RIGHT &
chopsticks[0]=LEFT, eatRequest[4]);
```