UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

Test-case generation for Stream-based specification languages

Testfallgenerierung für strombasierte Spezifikationssprachen

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Gunnar Bergmann

ausgegeben und betreut von
Prof. Dr. Martin Leucker

mit Unterstützung von
Malte Schmitz

Lübeck, den 15. June 2019

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne die Benutzung anderer als der angegebenen Hilfsmittel selbständig verfasst habe; die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe des Literaturzitats gekennzeichnet.

(Gunnar Bergmann)
Lübeck, den 15. June 2019

Abstract TeSSLa is an asynchronous stream-based specification language. Events arrive on multiple input streams and TeSSLa derives output streams out of these.

Software verification is a discipline that checks if software behaves according to a specification, but even software that has been fully verified potentially may not satisfy all expectations if the specification is insufficient or erroneous.

In order to find such mistakes this thesis explores ways of automatically generating test cases for TeSSLa specifications, that act as examples to the human developers. In this thesis we design a full test case generator for TeSSLa. We define a coverage criterion suitable for stream-based languages and additional use case specific constraints, that can be written by the user.

In this thesis we will see how TeSSLa specification are translated into SMT formulas, so that test cases can be found with the help of already existing SMT solvers.

The test case generator is implemented and evaluated.

Kurzfassung TeSSLa ist eine asynchrone strombasierte Spezifikationssprache. Ereignisse treffen an mehreren Eingabeströmen ein und TeSSLa leitet aus diesen die Ausgabeströme ab. Software Verifikation ist eine Disziplin, in der überprüft wird, ob Software sich gemäß einer Spezifikation verhält, aber selbst Software, die vollständig verifiziert wurde, kann möglicherweise nicht alle Erwartungen erfüllen, wenn die Spezifikation nicht ausreichend oder fehlerhaft war.

Um solche Fehler zu finden erforscht diese Arbeit Wege um automatisch Testfälle, die als Beispiele für die menschlichen Entwickler dienen, für TeSSLa-Spezifikationen zu generieren. In dieser Arbeit entwickeln wir einen vollständigen Testfallgenerator für TeSSLa. Wir definieren ein Abdeckungskriterium für strombasierte sprachen, sowie zusätzliche nach Anwendungsfall spezifische Bedingungen, die vom Nutzer geschrieben werden können.

In dieser Arbeit werden wir sehen, wie TeSSLa-Spezifikationen in SMT Formeln übersetzt werden, sodass Testfälle mit Hilfe bereits existierender SMT-Solver gefunden werden können.

Der Testfallgenerator wird implementiert und ausgewertet.

Contents

1. Introduction

Software verification is a process which checks whether a program meets a given specification. Testing is a verification technique, that checks conformance to the specification on certain inputs. Runtime verification (RV) is a testing method, that verifies properties on traces of programs [Leu11]. Commonly the specifications are written as logical formulas, typically in LTL or a related logic. A monitor, that checks the formula on the trace and outputs a verdict, is synthesized from the specification.

Stream runtime verification (SRV) offers an alternative to logic-based specifications. A specification relates a set of input streams to a set of output streams. Stream runtime verification is not just restricted to correctness properties, but can also gather statistical information, e.g. it can count the number of certain values.

The *Temporal Stream-based Specification Language* (TeSSLa)[CHL+18] is such a specification language for SRV in reactive and interactive systems where timing is a critical issue [CHL+18]. Whereas most SRV languages like Lola are *synchronous*, i.e. assume that all streams have simultaneous events, TeSSLa is *asynchronous*. Streams in TeSSLa are sequences of events, where each event consists of a value and a timestamp.

A web IDE allows experimentation with TeSSLa. It can be found at `http://tessla.isp.uni-luebeck.de`.

At each stream events may arrive at different timestamps. These events do not necessarily happen at the same time and are not synchronized by a logical clock.

Instead synchronization has to be done explicitly, for example by accessing the value that was last on a stream relative to the event on other stream: `last(x,y)` allows access to the last value on x whenever y has an event.

One of TeSSLa's design goals is the support of efficient *online monitoring*. That means that traces can be processed while they are generated by a simultaneously running process. For a subset of TeSSLa efficient monitors can be implemented in hardware, e.g. on an FPGA.

All verification techniques can only check if a given program behaves well according to the specification, but errors in the specification are only discovered by humans. If verification fails on a trace, that is actually not wrong, it does not cause much harm in most cases. The error is reported and the trace serves as an example that can be used for tracking down and correcting the mistake.

On the other hand if verification succeeds on an erroneous trace, this error may remain undetected and certain classes of bugs may end up in the product. In order to find such

behavior one may want certain example inputs which highlight what acceptable behavior according to the specification is. Just like in program testing manually assembling such a suite of examples can be tedious and one may accidentally skip certain inputs which one considers unnecessary.

A simple test case can be generated by setting all input streams to the empty stream, but such a case probably does not show interesting properties of the specification. Instead test cases can be derived from the the structure of the code. The goal of this work is the development of automated test case generation for TeSSLa. We will define a suitable criterion for deciding which cases tests should cover. Additional constraints allow users to further restrict the tests and search for error conditions.

For this test case generation a given TeSSLa specification is translated into an equivalent formula that consists of propositional logic and arithmetic constraints. Additional constraints, that define for which interesting cases we search, are added to the formula. We use the SMT solver Z3 for finding a solution to the formula. SMT solvers are specific algorithms, that solve the satisfiability problem for certain logical formulas.

After a theoretical description of criteria, constraints and such a translation from TeSSLa into logical constraints a test case generator `tessla-testgen` is implemented. This test case generator is evaluated on a specific scenario.

1.1. Prior Work

This chapter surveys previous work on test case generation and stream-based languages.

The synchronous stream-based language Lustre has been studied for testing and model checking [HCRP91], [Hal98].

[MHM+95] apply manual black-box testing to synchronous programming languages. The authors describe different models that are used for developing testing criteria, although many test strategies can not be applied to automatic test cases. Tests on a (potentially abstracted) *finite state machine* verify that transitions, states and certain paths work as intended. Test on *predicates* verify application specific behavior, like the mapping of certain inputs to actions. Tests on *data types* test the boundaries of data types, like the highest and lowest value. Tests on *concurrent inputs* check for correct behavior if inputs are simultaneous and of varying order. Such variations happen especially for manual input from user interfaces.

[TFMC94] apply a mixed strategy of random value and extremal value testing on the synchronous language Lustre. The chosen random distribution and the choice of the extremal values are based on the structure of the automaton.

[RNHW98] propose a technique for automated black-box testing, that they implement in the tool Lurette. A supplied observer in the language Lustre compares the input of a reactive system with the output and decides if the property is interesting. This

observer is used for gradually generating a test case. For each cycle of the global clock a formula is created, that relates current state and input to output. This formula is solved with a procedure, that could be called a primitive SMT solver.

Compared to the approach in this work, the formula is not generated at once, but stepwise. The stepwise approach leads to smaller formulas and therefore faster solution, but may not reach certain conditions. Additionally a stepwise approach may be unsuitable for asynchronous language because there is no obvious choice of stepwidth.

1.2. Outline

Chapter 2 on page 5 explains the background knowledge for understanding the following chapters. This chapter explains runtime verification and different languages for writing specification, especially the language TeSSLa which is used throughout the thesis. The chapter also explores different test coverage criteria for traditional programming models, different test case generation concepts and gives an introduction to SMT solvers.

Chapter 3 on page 21 presents coverage criteria for TeSSLa and a language extension for specifying custom criteria. Afterwards it shows, how TeSSLa specifications, custom constraints and coverage criteria can be translated into constrains for an SMT solver.

Chapter 4 on page 41 explains the implementation and its architecture. A core component are different strategies, which determine, how different test cases are assembled into a test-suite with high coverage.

Chapter 5 on page 49 evaluates the implementation by comparing and measuring the different approaches to coverage maximization on two examples.

2. Background

This chapter provides the necessary background knowledge for later chapters and introduces the basic concepts. The basics of testing are explained in section 2.1. Section 2.2 introduces the concept of runtime verification and explains common specification concepts. A focus is on stream-based programming languages, with the examples of Lustre, Lola. Section 2.2 also introduces the language TeSSLa. Section 2.3 explains different coverage criteria, that are used to evaluate test cases on imperative programming languages. Section 2.5 explains SMT solvers in general and the Solver Z3 specifically.

2.1. Testing

Testing is a common *partial verification technique*. *Verification* means that a program is compared to a specification. Testing is *partial* because it only examines a subset of the behaviors. Dijkstra has famously said that "Program testing can show the presence of bugs, but never to show their absence" [BR70, 2].

Nonetheless program testing has shown to detect many common error cases, especially because writing tests for a system is often easier than developing a static verification tool for specific tasks. Tests only run a program on input and check if the behavior is correct.

2.2. Runtime verification

Modern computers often run as *interactive* or *reactive* systems[Ber98]. Both of these system types run and react to inputs continuously. In contrast many classical computation models assume programs that produce a result in finite time. Verification on these programs commonly checks if the result is correct.

Runtime verification (RV) and *model checking* (MC) checks the properties on the run of a system. A run is a sequence of system states, or status information derived from the system states. One might imagine it as an (infinite) log file of the running program.

A run may be infinite.

Definition 2.1 (Run)**.** *Given a finite alphabet* Σ*, a run* r *is a possibly infinite word or trace over* Σ*:*

$$r \in \Sigma^{\infty}$$

Note 2.2:: $\Sigma^{\infty} = \Sigma^{\omega} \cup \Sigma^{*}$ is a possible infinite word, the union of infinite words Σ^{ω} and finite words Σ^{*}.

For many use cases runs can be assumed to be infinite, by extending them with an infinite repetition of *exit-states*.

Definition 2.3 (Execution)**.** *An* execution e *of a system is a* finite *prefix of a run over the alphabet* Σ*,*

$$e \in \Sigma^{*}$$

RV and MC allow infinitely running, reactive systems. The system may have an infinite input sequence.

Model checking (MC) is the art of verifying, that a property holds on all possible and infinite runs. Model checking requires a program, that can be analyzed and turned into a checkable automaton. In the general case model checking is not decidable, because the reachability of an exit-state is equivalent to the halting problem.

Runtime verification is a much simpler task. The property is only checked on a single execution. The execution can be gained by instrumenting the program, so that it outputs the current state and then running the program. Both instrumentation and evaluation can be done automatically.

Formally runtime verification is a word problem:

Definition 2.4 (runtime verification)**.** $e \in \mathcal{L}(\varphi)$ *where* $e \in \Sigma^{\infty}$ *is an execution and* $\mathcal{L}(\varphi)$ *is the language derived from the specification* φ*.*

Definition 2.5 (Monitor)**.** *A* monitor *is a device that reads a finite trace and yields a certain* verdict*[Leu11].*

A verdict is a value from a truth domain. This includes at least the elements **true** and **false**, but often more values for expressing varying degrees of uncertainty are added.

2.2.1. Offline and online RV

Runtime verification can be distinguished into *online* and *offline monitoring*.

In *offline monitoring* the execution is recorded first and verified in a second step. Conversely in *online monitoring* monitor and observed system are running alongside. The

execution is produced incrementally by the system and immediately consumed by the monitor.

Constructing an online monitor may be more complex and for some calculi, especially ones that depend on future states, online monitoring may be impossible.

On the other hand online monitoring has the advantage, that potentially very large executions can be analyzed piecewise without having to store it in between. Some common monitors like the ones presented in [Leu11] can operate within constant memory space.

The related technique *runtime reflection* (RR) uses an online monitor and other RV concepts for detecting and migitating failures in running systems.

2.2.2. LTL

Linear-time temporal logic (LTL) and variations and extension of it are a commonly used logic for writing specifications.

LTL was introduced by Amir Pnueli in 1977 [Pnu77]. It is a logic that extends propositional logic with temporal operators. The input is a potentially infinite word over the alphabet 2^{AP} where AP is the set of the atomic propositions. Examples for temporal operators are **X**(next), **F**(finally) and **U**(until). **X** states, that a property holds in the next step whereas, **F**p states that the property p must be eventually true. p**U**q describes that p must hold until q holds at least once.

There are many variations of LTL with different set of operators.

Limitations

LTL can only express correctness and failure assertions, but can not gather and verify statistical measures. Additionally users may want a monitor, that regularly outputs more detailed information instead of a binary true/false choice.

Writing LTL specifications requires specially trained software engineers.

2.2.3. Stream-based programming

Lustre [HCRP91]

Lustre is an example for a synchronous stream-based programming language. In Lustre every stream is called *flow* and consists of a sequence of values and a sequence of timestamps, the clock.

The clock is not necessarily bound to the outer "real time", but signifies the lowest granularity at which the system can react to inputs. If the normal time is required, the user may use a flow for representing the time.

```
n = 0 -> pre(n) + 1
```

Listing 2.1: Counter in LUSTRE

The flow `n` is a counter, that is initialized with 0 and increases the value on each cycle.

`pre(n)` acts as a memory cell, that stores the last value on `n`, in this case the last counter. `pre(n) + 1` is the last value on `n` increased by `1`. Therefore `n` counts the cycles. Additionally `x -> y` ("followed by") replaces the first value of the flow `y` with the first value from `x`. In Listing 2.1 `0 -> ...` initializes the counter with `0`.

Most operations in LUSTRE are only defined on flows with the same clock.

The operator `X when Y` allows the creation of different clocks, by sampling the flow `X` only when the Boolean flow `Y` is **true**. The example in Listing 2.1 counts the number of clock cycles of the basic clock. Counting the events of a logical clock `B` is possible with

```
n = (0 -> pre(n) + 1) when B
```

Listing 2.2: Counter on custom clock

The operator `current E` allows the usage of flow `E` with a clock, that is faster than the one of `E`. `current E` yields the last value, that E has seen.

[HCRP91] note that LUSTRE is very similar to the temporal logic. This allows users to write reactive and interactive systems in LUSTRE and also specify properties in the same language.

The authors include an *assertion* mechanism in the language.

```
assert not (x and y)
```

specifies that the streams `x` and `y` are never true at the same time.

Definition 2.6 (assert)**.** `assert` φ *describes that an expression* φ *is true in every cycle.*

During normal program creation assertions are meant for improved optimization. For program verification the assertions are checked with model checking algorithms. Common temporal operators can be expressed in LUSTRE.

2.2.4. Lola

Definition

The specification language Lola[dSS+05] is a functional stream computation language for runtime verification.

Just like Lustre Lola is a synchronous stream-based language, but whereas Lustre is designed for the construction of reactive systems, Lola is designed for monitor specification. Lola is not restricted to access to previous events, but may use any value except 0.

A specification relates a set of input streams to a set of output streams. These streams can also contain numeric values, so that some properties like *"the number of a's must always be no less than the number of b's"* can be expressed in Lola, but not in LTL[dSS+05].

```
counter = counter[-1,0] + 1
```

increments the stream `counter` in every step by one. `s[i,c]` is an operator, that accesses a stream at an offset `i` from the current position. The value `c` is the default value for out-of-bounds. In this example the index of `-1` accesses the last value. The counter starts with 0.

Compared to other specification formalisms like LTL, Lola follows a more conventional programming style.

2.2.5. TeSSLa

Like Lola the *Temporal Stream-based Specification Language* (TeSSLa)[CHL+18] is a stream-based specification language for runtime verification.

The main difference between Lola and TeSSLa is that TeSSLa uses *asynchronous streams*. In synchronous languages all the streams have events simultaneously or are at least synchronized by some global clock. Conversely asynchronous ones lift these restrictions. This allows runtime verification to mix sparse and high-frequency streams.

Since asynchronous languages lack a clock for accessing timing information, values arrive at different times, as a discrete sequence of events. Every event consists of a value and a timestamp. Whereas events in Lola are implicitly ordered, TeSSLa supports time as a first-class citizen [CHL+18]. The operator **time** creates a stream of timestamps. The numeric operators like $\leq$ and $=$ can be used to get the order of events. Additionally constants may be added. For example a stream may monitor if a response time happens within a certain interval:

```
error := time(response) > time(request)+5
```

The example of an event counter can be implemented as

```
def counter: Events[Int] := merge(last(counter, x) + 1, 0)
```

In asynchronous languages there are no definite steps and therefore no operator like `pre counter` or `counter[-1,0]`. Instead access to the last element has to be relative to another stream. This is done in the example, where the strictly last value of `counter` is accessed, whenever `x` receives an event.

```
in a: Events[Unit]
in b: Events[Unit]
in errorMargin: Events[Int]

def realErrorMargin = merge(errorMargin, 0)
def diff: Events[Int] := slift(-)(count(a), count(b))
def error: Events[Bool] := slift(>)(diff, realErrorMargin)
out diff
out error
```

Listing 2.3: TeSSLa example code

The code in Listing 2.3 has two input streams `a` and `b` of unit type. This is a type which has only a single instance and streams of type unit only contain events without a payload. An additional stream of integers sets the margin of error.

A new stream `diff` contains the difference between the numbers of events on these streams and another stream `error` signals an error if the difference surpasses the error margin. Both the difference and the error signals are outputs.

Efficiency One of TeSSLa's design goals is the usage in FPGAs, which only accepts specifications, that can be checked without requiring additional memory at evaluation time. Each of the built-in operators works with a memory cell large enough for holding one element. Nonetheless certain implementations in interpreters may support additional types like lists and maps.

Definition

In TeSSLa a stream denotes a sequence of values, where timestamps and event values alternate:

Definition 2.7 (stream definition from [CHL+18])**.** *An event stream over a time domain* $\mathbb{T}$ *and a data domain* $\mathbb{D}$ *is a finite or infinite sequence* $s = a_0 a_1 ... \in (S)_{\mathbb{D}} = (\mathbb{T} \cdot \mathbb{D})^\omega \cup (\mathbb{T} \cdot \mathbb{D})^+ \cup (\mathbb{T} \cdot \mathbb{D})^* \cdot (\mathbb{T}_\infty \cup \mathbb{T} \cdot \{\bot\})$ *where* $a_{2i} < a_{2(i+1)}$ *for all* i *with* $0 < 2(i+1) < |s|$ *[..].*

The set $\mathbb{T}_\infty = \mathbb{T} \cup \{\infty\}$ *extends the time domain with an infinity value. We define that* $\forall_{t \in \mathbb{T}} : t < \infty$

A stream is defined up to a certain timestamp that indicates the *progress*. Events after this progress are not yet recorded by the online monitor.

A stream $s \in \mathcal{S}_\mathbb{D}$ can be interpreted as a function $s : \mathbb{T} \to \mathbb{D} \cup \{\bot, ?\}$. If the stream s has an event $d \in \mathbb{D}$ at a timestamp t, then $s(t) = d$. For all $t \in \mathbb{T}$ after the progress we don't know whether the stream will have an event and set $s(t) = ?$. For all other timestamps, where s has no event $s(t) = \bot$.

The TeSSLa core language is defined by a set of 6 operators.

Definition 2.8 (TeSSLa operators from [CHL+18])**.** *A* TeSSLa *specification* φ *consists of a set of possibly mutually recursive stream definitions defined over a finite set of variables* $\mathbb{V}$ *where an equation has the form* $x := \langle e \rangle$ *with* $x \in \mathbb{V}$*.*

$\langle e \rangle$ ::= **nil**| **unit**| x | **lift**(f)(e, ..., e) | **time**(e) | **last**(e,e) | **delay**(e,e)

operator semantics Let $\mathbb{U} = \{\square\}$ be the unit type, $\mathbb{B} = \{\textbf{true}, \textbf{false}\}$ be the Boolean type.

- **nil** $= \infty \in \mathcal{S}_\emptyset$ is the nil-stream. This is a stream, that never has any events.
- **unit** $= 0 \ \square \ \infty \in \mathcal{S}_\mathbb{U}$ is the unit stream. This stream has a single event at timestamp 0 of type unit and no other events.
- **time** $: \mathcal{S} \to \mathcal{S}_\mathbb{T}, s :=$ **time**(e) maps the events to their timestamps.

$$\forall t \in \mathbb{T} : s(t) = \begin{cases} t & \text{if } e(t) \neq \bot \\ \bot & \text{if } e(t) = \bot \end{cases}$$

- **lift** $: ((\mathbb{D}_1 \cup \{\bot\}) \times ... \times (\mathbb{D}_n \cup \{\bot\}) \to (\mathbb{D}_R \cup \{\bot\})) \to (\mathcal{S}_{\mathbb{D}_1} \times ... \times \mathcal{S}_{\mathbb{D}_n} \to \mathcal{S}_{\mathbb{D}_R})$, $s :=$ **lift**$(f)(e_1, ..., e_n)$ lifts a function f, that is defined on values to a function on streams. The lifted function accepts $\bot$ as inputs if no event happens on a stream, but at least one stream must have an event whenever the output has an event. This prevents **lift** from creating new events.

$$\forall t \in \mathbb{T} : s(t) = \begin{cases} f(e_1(t), ...e_n(t)) & \text{if } \exists i : e_i(t) \neq \bot \wedge \forall i : e_i(t) \neq ? \\ \bot & \text{if } \forall i : e_i(t) = \bot \\ ? & \text{if } \forall i : e_i(t) = ? \end{cases}$$

- **last** : $\mathcal{S}_{\mathbb{D}_a} \times \mathcal{S}_{\mathbb{D}_b} \to \mathcal{S}_{\mathbb{D}}, s := \textbf{last}(a, b)$ looks up the last value on a whenever b has an event.

$$\forall t \in \mathbb{T} : s(t) = \begin{cases} d & \text{if } b(t) \in \mathbb{D}_b \wedge \exists_{t'<t} : (a(t') = d \wedge \forall_{t''|t'<t''<t} : a(t') = \bot) \\ \bot & \text{if } b(t) = \bot \text{ and defined}(s,t), \text{ or } \forall_{t'<t} : a(t') = \bot \\ ? & \text{otherwise} \end{cases}$$

  where $\text{defined}(s,t) := \forall_{t'<t} : s(t') \neq ?$.

  Note that the **last** does not return the current event if a and b have simultaneous events. This means, that $\textbf{last}(x, x)$ looks up the previous value on x, whenever x has an event. The behavior is similar to `pre` from Lustre or the `x[-1, ...]` from Lola.

- **delay** : $\mathcal{S}_{\mathbb{T}\setminus\{0\}} \times \mathcal{S}_{\mathbb{D}} \to \mathcal{S}_{\mathbb{U}}, s := \textbf{delay}(d, r)$ emits a unit event delayed by the time in d. The value in d is read whenever the reset stream r has an event or when s has just emitted an event. If r has an event at t, but $d(t) = \bot$, then the delay resets, but won't emit an event.

  delay can be imagined as having a timer, that is set to $d(t)$, whenever it is reset, and disabled when $d(t) = \bot$.

$$\forall t \in \mathbb{T} : s(t) = \begin{cases} \square & \text{if } \exists_{t'<t} : d(t') = t - t' \wedge \mathit{setable}(s, r, t') \wedge \mathit{noreset}(r, t', t) \\ \bot & \text{if } \mathit{defined}(s,t) \wedge \forall_{t'<t} : d(t') \neq t - t' \wedge d(t') \neq ? \\ \bot & \text{if } \forall_{t'<t} : \mathit{unsetable}(s, r, t') \vee \mathit{reset}(r, t', t) \\ ? & \text{otherwise} \end{cases}$$

  where

$$\begin{aligned} \mathit{setable}(s, r, t') &:= s(t') = \square \vee r(t') \in \mathbb{D} \\ \mathit{reset}(r, t, t') &:= \exists_{t''|t<t''<t'} : r(t'') \in \mathbb{D} \\ \mathit{unsetable}(s, r, t') &:= s(t') = \bot \wedge r(t') = \bot \\ \mathit{noreset}(r, t, t') &:= \forall_{t''|t<t''<t'} : r(t'') = \bot \end{aligned}$$

Helper functions Common helper functions in TeSSLa are `const` and `merge`.

- $\texttt{const}(c)(a) := \textbf{lift}(f)(a)$ where $f(x) := c$ is the constant function.
- $\texttt{merge}(x, y) := \textbf{lift}(f)(x, y)$ where

$$f(a,b) := \begin{cases} a & \text{if } a \neq \bot \\ b & \text{if } a = \bot \end{cases}$$

  `merge` merges events from two streams into one. In the case of simultaneous events, the first one is preferred.

Signals A *signal* in a reactive system is a function $f : \mathbb{T} \to \mathbb{D}$, that assigns a value to each point in time [EH97].

A continuously changing function like $f(x) = \sin(x)$ or $f(x) = x$ can not be implemented in TeSSLa, because TeSSLa only allows streams with discrete events.

Nonetheless TeSSLa can adequately model computations with signals, that either only change at discrete timestamps or can be sampled at these. A signal can be modeled by reporting every change in value as an event. One can access the value of the signal by reading the current or last value on the stream.

The *signal lift* or `slift` is a function like **lift**, but it interprets the input and output streams as signals. It applies the function on the current or last value on the input streams.

For types $\mathbb{D}_x, \mathbb{D}_y, \mathbb{D}_r$, streams $x : \mathcal{S}_{\mathbb{D}_x}, y : \mathcal{S}_{\mathbb{D}_y}$, and function $f : \mathbb{D}_x \times \mathbb{D}_y \to \mathbb{D}_r$:

$$\texttt{slift}(f)(x, y) := \textbf{lift}(f')(x', y')$$

where

$$f'(a, b) := \begin{cases} f(a, b) & \text{if } a \neq \bot \wedge b \neq \bot \\ \bot & \textit{otherwise} \end{cases}$$
$$x' := \texttt{merge}(x, \textbf{last}(x, y))$$
$$y' := \texttt{merge}(y, \textbf{last}(y, x))$$

In a lot of examples infix operators are implicitly signal lifted. For example `a+b` is written in place of `slift(+)(a,b)`.

Lifted functions

There is no specification of the lifted functions or the set of data types $\mathbb{D}$ in [CHL+18]. The listed operators require the existence of a time domain $\mathbb{T}$ and a unit type $\square$. Additionally lifted functions may be partial and input values to functions may be $\bot$ if there is no event on the respective stream. Implementations probably support $\bot$-values by providing an `Option` type.

The following paragraphs describe the lifted functions in the implementation mentioned by [CHL+18].

Types The available basic data types are *integers* ($\mathbb{Z}$), *floating point numbers*, *Boolean's* ($\mathbb{B}$). Another data types like strings and arrays exist in the TeSSLa interpreter, but we restrict the specifications to the simpler types within this thesis.

An `Option[T]` type exists with the variants `Some(t)` and `None` for a `t` $\in$ `T`. Within **lift** `Option` is used for input parameters and return value, where `None` represents $\bot$.

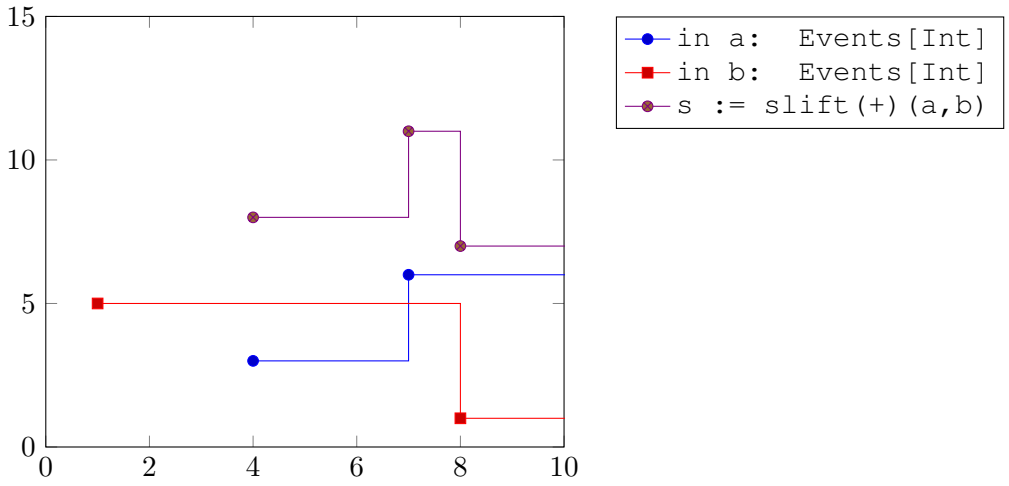


Figure 2.1.: An output signal `s` is computed from signals `a` and `b`. Events that report the change in value are visualized as dots. The signals are shown as functions on the time domain. Note that signals streams are only true signals with a defined value once a first event has been seen. Therefore it is often advised to initialize signals using `merge` and **unit**.

An *object* type exists that allows the creation of more complex structures. Just like objects in other languages, they allow access to members by name. *Tuples* are a special kinds of objects with the names `_1, _2, ...` . The Unit type `()` is the empty object.

Additional types like Maps, Sets and Lists exists. These do not workout with constant memory, and therefore the implementation for FPGAs will not support them.

The language supports generic types in type definitions, functions and macros. These are written in a Scala-like syntax. The generic parameter is put in brackets after the type name: `SomeType[T]`. TeSSLa supports some type inference. Macros are expanded in early stages of the interpreter.

A built-in type was already presented with `Option[T]`. Another built-in type is the type of streams `Events[T]`. Unlike other types, `Events` can not be used within streams, because there is no stream of streams.

Additionally function types `(a: A, b:B) => C` with arbitrary arity exist. These are used in **lift**.

Lifted functions No explicit syntax for lifted functions is given in [CHL+18] or this thesis. Instead we tread lifted functions as composition of functions, values and variables. An expression e is defined as

$$e ::= c \mid v \mid f(e,...,e)$$

where c is a constant, v is a variable symbol and f is a mathematical function. These functions may be partial, for example the division function is not defined for a divisor of

0. We assume that no undefined values are returned from a lifted function. Functions and variables may also be $\bot$. This value indicates the absence of an event.

A safe division

$$f(x,y) = \begin{cases} \bot & \text{if } x = \bot \vee y = 0 \\ \bot & \text{if } y = 0 \\ \frac{x}{y} & \text{otherwise} \end{cases}$$

may be written as the expression

$$ite(isNone(x) \vee isNone(y), None, \\ ite(getSome(y) = 0, None, getSome(x)/getSome(y))).$$

In the syntax used by the TeSSLa implementation from [CHL+18] this function would be implemented as

```
def f(x: Option[Int], y: Option[Int]): Option[Int] :=
  if isNone(x) || isNone(y) then None[Int]
  else if getSome(y)==0 then None[Int]
  else getSome(x)/getSome(y)
```

For convenience the examples allow custom functions and closures, but in tessla-0.7.2 of the TeSSLa interpreter these are never recursive. This means that all of the functions can be eliminated by inlining them.

Built-in operators are the logical operators `&&`, `||`, `!` with their common C-like definition, the numerical operators `+`, `-`, `*`, `/`, `>=`, `>`, `<=`, `<`, `==`, `!=`, an inline-if `ite` : $(c : \mathbb{B}, l : T, r : T) \to T$, the operators `IsNone` : $(Option[T]) \to \mathbb{B}$ and `GetSome` : $(Option[T]) \to T$, as well as the constructors for options `Some` : $T \to Option[T]$ and $None : Option[T]$.

The functions division with `/` and `GetSome` are partial. Over the course of this work it is assumed, that no undefined value is returned from the **lift**-operator. Practically this means, that this work only considers specifications, that never emit undefined behavior because they

a) explicitly check for invariants with `ite`
b) the input to the **lift**-operator is restricted by another operator. For example `filter` removes zeros from the stream, so that a lift never has divide-by-zero errors.
c) application specific knowledge restricts input streams to certain values. For example if the user knows that a certain input stream only contains integers above zero, then a divide-by-zero will never happen. This property needs to be passed to the test case generator through the use of custom constraints.

Otherwise the function is undefined in the same way, undefined behavior works in C. Practically this means that specifications, that contain undefined behavior differ in different implementations. Most importantly the test case generator's implementation

(chapter 4) sets resulting events to arbitrary values, whereas the interpreter in [CHL+18] raises an exception.

In examples `ite` is often written as a ternary `if•then•else`.

This definition of lifted functions is similar, but not equal to the current implementation in the TeSSLa interpreter.

```
def filter[T](events: Events[T], condition: Events[Bool]): Events[T] := {
  def c := merge(condition, last(condition, events))
  def f(e: Option[T], c: Option[Bool]): Option[T] :=
    if isNone(c) then None[T]
    else if getSome(c)
    then e else None[T]
  lift(events, c, f)
}
```

Listing 2.4: filter on streams in TeSSLa

Figure 3.2 shows how a typical filter function is implemented in TeSSLa with a macro. A local stream `c` and a local function `f` are defined within the body of the macro. It is expanded by the TeSSLa interpreter or compiler.

The lifted function `f` accepts two parameters of type `Option` and returns another `Option`. It uses nested `if•then•else` operators to return no event (`None[T]`) which represents $\perp$ if there is no condition or if the condition evaluates to **false**.

2.3. Test coverage criteria

In *White-Box testing*, that is testing where the code is available, test coverage describes which degree of a program's source code is executed by a test suite. Coverage criteria determine which metric is used for measuring the coverage. For imperative programs the typical coverage criteria are defined on the control-flow-graph of the program.

For realistic scenarios a full coverage often is impossible. Therefore a common task in testing is the maximization of the coverage.

2.3.1. Statement coverage

A statement coverage means that every statement of the program must be executed at least once. Of course testing can only show a bug if the erroneous statement is executed. Therefore statement coverage can be assumed as the minimal coverage criterion.

2.3.2. Path coverage

A test-suite has path coverage if every path in the control-flow graph is executed. The values of the variables are ignored. Path coverage subsumes statement coverage, but a full path coverage is often impossible, since loops introduce infinitely many paths.

2.3.3. Decision coverage

For decision coverage the conditions in all the if-statements have to be **true** and **false** at least once.

This means, that every edge in the control-flow-graph is tested at least once.

2.3.4. Condition coverage

For condition coverage every condition every *atom* of each guard has to be set to both **true** and **false** at least once [AOH03].

Note that condition coverage does not subsume decision coverage. This means, that test suite with a certain degree of coverage with condition coverage does not automatically also have decision coverage with the same degree. For example for the code

```
if a && b then
  ...
```

$a =$ **true**, $b =$ **false** and $a =$ **false**, $b =$ **true** has full condition coverage, but not decision coverage.

2.3.5. Decision/Condition coverage

Decision/Condition coverage combines the condition with the decision coverage. Every atom is set to **true** and **false** and the condition evaluates to **true** and **false**.

2.3.6. Modified decision/condition coverage (MC/DC)

Modified decision/condition coverage extends Decision/Condition coverage by requiring, that every atom must contribute to the decision [AOH03].

We say, that an atom c_i *determines* the predicate p if all remaining atoms c_j have values, so that changing the truth value of c_i changes the truth value of p. Conversely if a variable c_i has no influence on the decision it is called *masked*.

For every atom c_i the rest of the test case is chosen, so that c_i determines p. Then c_i is set to **true** and **false** respectively.

2.3.7. Data-Flow-based Coverage

Data-flow-based coverage analyzes the data dependencies in imperative programs. It creates a data-flow-graph with nodes

def(x) when defining or writing the variable **x**

p-use(x) when using **x** in a condition

c-use(x) when using **x** in a computation

Different criteria can be defined on the data-flow graph. These test cases test for dependencies between writes and later usage of variables [CPRZ89].

2.4. Test case generation

Test case generation on imperative programs uses techniques like weakest precondition and symbolic execution for test case generation.

Symbolic execution is a common method for program verification. It can be used for total verification, but also for generating test cases.

In symbolic execution, a program is run, but instead of concrete values, one uses *symbols* [Kin76]. These symbols may be formulas describing the set of possible states. For test-cases, these formulas only describe a subset of possible states. Moreover for test-case generation one can select a specific path and the formula describes an input, which leads to the execution of that path.

A test case is generated by finding a solution to that formula.

2.5. SMT solver

Many common problems can be formulated as satisfiability checking. In a first step the problem is formulated as a set of conjunctive constraints. The solver then checks if there is a solution to those constraints.

SMT solvers can be used for generating test cases. There are good implementations that can quickly find solutions for large SMT formulas. By translating TeSSLa specifications and additional constraints for coverage into SMT formulas we can use existing SMT solver implementations.

SMT solvers are used for test case generation. Symbolic execution or a similar technique is used to turn a path into a formula. The SMT solver is used to find a solution to that formula.

2.5.1. SAT Solver

The *Boolean Satisfiability Problem* (SAT) is a satisfiability problem over propositional logic. A SAT solver searches for an assignment that satisfies a given formula in propositional logic.

Common SAT solvers are based on the *DPLL* algorithm family [DLL62].

Recent developments in the research of SAT solvers lead to the creation of techniques like clause learning and non-chronological backtracking, and improved heuristics for deciding, which variables are chosen for backtracking [BT18]. These improvements made SAT solvers usable for larger formulas.

2.5.2. SMT solver

SMT solvers extend SAT with additional theories, like integer arithmetic, quantifiers, arrays and functions. The exact set of supported theories depends on the used SMT solver.

Older SMT solvers translated the SMT formulas into SAT instances by replacing every integer with a bitvector. Modern solvers on the other hand support arbitrary integers. They use algorithms like DPLL(T) which allow integration of theory-specific constraints into a logical formula. Theory-specific constraints are replaced by Boolean variables and a SAT solver determines the assignment of the variables. The theory specific solvers then set the values (e.g. integers) according to the variable assignment.

2.5.3. Z3

Z3 is a contemporary state of the art SMT solver by Microsoft research [DMB08] and used by many projects.

Z3 supports uninterpreted functions, linear and nonlinear arithmetic on integers and reals, where both are not restricted to a fixed bitsize. Z3 also supports bitvectors, arrays, quantifiers and custom data types. Some of these theory solvers are not complete. This means, that the solver does not always find a solution if one exists, although the given theories often contain classes of problems, for which the problem is still solvable.

Like many other solvers Z3 uses a DPLL-based SAT solver and integrates the other theory-specific solvers into it.

Besides the SMT solver Z3 also contains an optimizer, which solves the max-SMT problem. A max-SMT problem contains a set of constraints in the given theories and an objective function which should be minimized or maximized.

The optimizer supports soft-constraints. Like constraints a given solution should fulfill these soft-constraints but does not have to, if no solution is possible.

These soft-constraints can be used for desired, but not necessary requirements. For example the soft-constraints might be used to define coverage criteria whereas ordinary constraints define the validity of solutions.

3. Test case generation for TeSSLa

In this chapter we will develop all the concepts behind test case generation for TeSSLa and describe a representation as SMT constraints for all of TeSSLa and for all additional techniques we develop in this chapter.

A first section defines a test criterion, that describes properties that a test suite should have. This criterion can be used for automatically choosing tests. Then we introduce a way of defining custom constraints, that restrict the set of possible cases even further. And finally we describe different strategies for generating multiple test cases from a formula, that describes the specification, and a set of constraints that describe the test criteria.

Streams in TeSSLa are infinite sequences of events and the specification defines a relationship between these streams. The formulas for some operators refer to previous timestamps of events. For test cases we can only use streams with a finite number of events. We call these finite sequences up to a certain timestamp t_{max} the *prefix* or *observed prefix* of a trace. We can assume, that the progress has advanced far enough, so that up to t_{max} the stream is completely known, i.e. for all $t < t_{max}$ and all streams s we have $s(t) \neq ?$.

3.1. Test criteria

The generated test cases should show interesting and surprising properties of the given formula. Many TeSSLa specifications have a trivial test case that consists of empty streams only.

Example 3.1: merge: The `merge` function in Listing 3.1 merges events from two streams into one. If `a` and `b` have simultaneous events, a gets preferred.

`merge` has three different cases:

1. x is Some and y is None
2. Both are Some
3. x is None and y is Some

The function never inspects the content of the streams and therefore the detail does not matter, although it is advantageous if the test case generates different values for different streams.

```
def merge[T](a: Events[T], b: Events[T]): Events[T] := {
  def f(x: Option[T], y: Option[Bool]): Option[T] :=
    if isNone(x) then y else x
  return lift(a, b, f)
}

in a: Events[Int]
in b: Events[Int]
def z := merge(a, b)
out z
```

Listing 3.1: The merge function in TeSSLa merges events from two streams into one.

Example 3.2: Traffic light: The state of a traffic light is represented by a stream of integers, where the value 0 symbolizes red, 1 is yellow and 2 is green.

The first set of test cases should contain test inputs, that reach each of these states. Other tests may want to test certain sequences and patterns of the states.

Now the user gets curious and wants to see if there is a sequence, that never reaches yellow.

In this example the values model specific states of an automaton and are used like enum types in many programming languages or nominal scales in statistics. This means, that 2 is not just a larger value than 1, but models a different state of a automaton.

As such the values contain an external meaning that is important to the tests. Automated coverage criteria alone are insufficient and therefore one may need to pass additional constraints to the test case generator.

In order to fulfill both use cases the test case generator contains two sets of criteria. The first set is based on *code coverage criteria* and is checked automatically. The second set contains *custom constraints* and must be provided by the user.

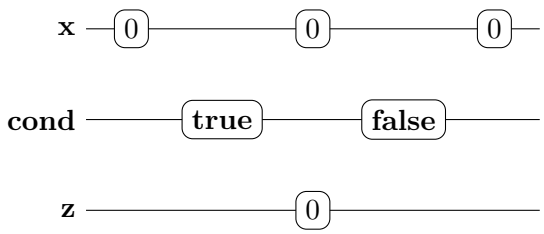
3.1.1. Automated coverage

The test case generator should automatically generate interesting test cases. These test cases should not just use different values, but differ in interesting ways. For example the test suite for the `filter(x, cond)`-function (Figure 3.2) from [CHL+18] should cover three different cases: `cond` is $\bot$, **false** or **true**. The values on `x` don't matter, because they are only forwarded or filtered out, but never inspected or modified. Yet in each of the three cases there should be an event on `x` to show the effect.

```
1  def filter[T](events: Events[T], condition: Events[Bool])
2      : Events[T] := {
3    def c := merge(condition, last(condition, events))
4    def f(e: Option[T], c: Option[Bool]): Option[T] :=
5      if isNone(c) then None[T]
6      else if getSome(c)
7      then e else None[T]
8    lift(events, c, f)
9  }
10
11 in x: Events[Int]
12 in cond: Events[Bool]
13 def z := filter(x, cond)
14 out z
```

Listing 3.2: The filter function in TeSSLa. This definition reuses a previously defined `merge` function.

Therefore a test case may look like



In the case of the `filter` function, the lifted function explicitly handles three distinct cases. Line 5 checks if `condition` is $\bot$ or not. If not line 6 checks for the value of `condition`.

The test suite for the TeSSLa formula therefore also acts as a test on the `filter` function. This observation can be generalized: *An adequate coverage for a* TeSSLa *specification is achieved by maximizing the coverage on the lifted functions.*

The coverage on the lifted functions may be generated with different criteria and by using various techniques.

The lifted functions in the current TeSSLa implementation consist of compositions of function calls. We assume that there are no loops or recursive function calls. Otherwise recursive functions may be turned into nonrecursive ones by limiting the chosen paths to ones with a maximum call depth. Unrolling the recursively defined function up to the chosen depth transforms the recursively defined calculations into nonrecursive ones.

The examples with merge and filter in Listing 3.1 and Figure 3.2 show that the desired test suite approximately matches the cases within the lifted functions. Therefore we choose to maximize the decision coverage on the lifted functions.

The common decision coverage is defined on the control-flow graph of programs in imperative languages, where branches are gated by Boolean expressions. TeSSLa functions use nested expressions only, but these are not just restricted to Boolean values.

A simple coverage criterion requires that the condition in each `ite` expression is at least once **true** and once **false**. For the filter example a first case may set `isNone(c)` in line 5 to **false** and `getSome(c)` in line 6 to **false**, but for the second event `isNone(c)` is **true** whereas `getSome(c)` is *undefined*. We have not specified how we handle undefined values, but often SMT solvers are allowed to fill in *any* value that fits and consequently satisfy the constraint meant to force a coverage.

For other many specifications this criterion means that a full test suite consists of only a single test case with two events. In the first event every condition for every `ite` is set to **false** and in the second event it is set to **true**, even when we can't see the results of intermediate computations, because another `ite` chooses to return the value from the other branch of the expression tree.

As we have seen, the simple criterion is insufficient as a coverage criterion. We need a coverage criterion which does never depend on undefined behavior.

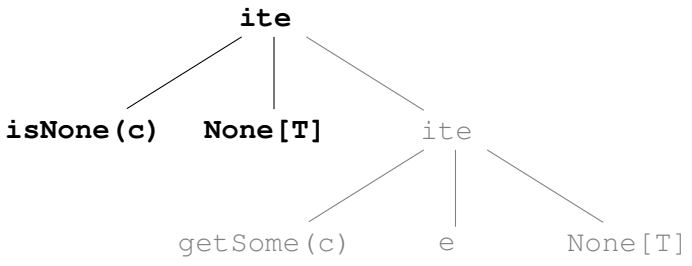
We say that a subexpression is *masked out* if it does not contribute to the value of the expression. More formally for an expression $\texttt{ite}(cond, a, b)$ if $cond$ evaluates to **true** then b is masked out, otherwise a is masked out. For $e := f(e_1, ..., e_m)$ all subexpressions $e_1, ..., e_m$ are masked out if e is masked out.

We may define masking slightly differently. For example we may also include the effect of Boolean operators and say that for $a \wedge b$ the subexpression b is masked out when a is **false** and so forth.

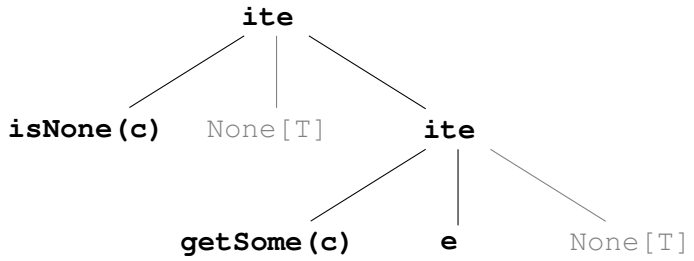
We now define decision coverage on the data-flow graph of the expression. Decision coverage demands that each subexpression is once **false** and not masked out and once **true** and not masked out.

For the filter example we get:

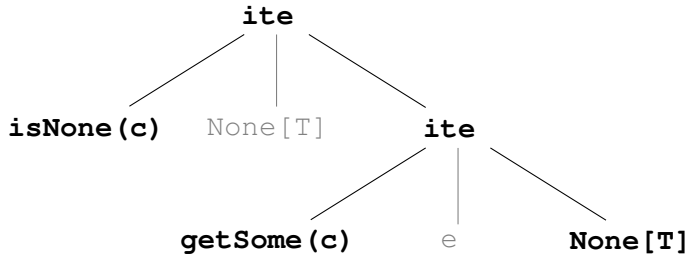
1. The first `ite` at line 5 is set to **true**:



2. The first `ite` at line 5 is set to **false** and the second at line 6 to **true**:



3. The first `ite` at line 5 is set to **false** and the second at line 6 to **false**:



These three test cases are the ones that we have chosen at the beginning of the section.

We called the coverage criterion decision coverage, because it is equivalent to decision coverage on an equivalent imperative program, if we translate an expression of the form

$$e := \texttt{ite}(cond, a, b)$$

into an imperative program

def cond′ := *cond* // *evaluate condition first*
if cond′ **then**
  e := a
else
  e := b

In the example with the simpler criterion we have seen, that the coverage may depend on undefined values. Decision coverage does not suffer from this problem. In subsection 2.2.5 we have allowed partial functions in expressions only if a lifted function never returns an undefined value. The computations that would lead to undefined behavior are masked out by other functions. For example in the definition of `filter getSome(c)` is masked out if `isNone(c)` evaluates to **true**. It does only considered for test coverage if `c` is not None and therefore `getSome` is defined.

3.1.2. Custom constraints

In some cases like Example 3.2 on page 22 the automatic coverage we defined in 3.1.1 we may want to search for test cases

- in which the system will eventually reach state red (0)
- that contain a sequence red → yellow → green:

Yet decision coverage only tests for these requirements if a decision within the specification contains it as a goal.

Additionally we may want to restrict test cases to the valid states 0,1 and 2. This restriction can not be expressed with the coverage criteria.

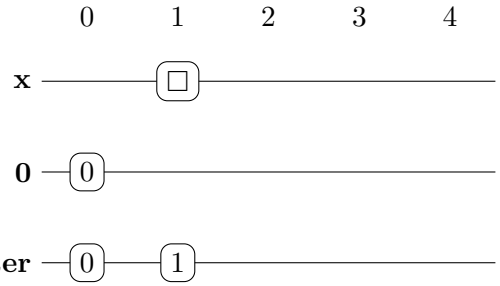
This other example is an event counter.

```
in x: Events[Unit]
def counter: Events[Int] := merge(lift1(last(counter, x), inc), 0)
out counter

def inc(a: Option[Int]) := Some(getSome(a) + 1)

def merge[T](x: Events[T], y: Events[T]): Events[T] :=  {
  def f(a: Option[T], b: Option[T]) :=
    if isNone(a) then b else a
  lift(x,y,f)
}

def lift1[T, V](x: Events[T], f: (Option[T]) => Option[V]) := {
  def ff(a: Option[T], b: Option[Unit]) := f(a)
  lift(x, nil[Unit], ff)
}
```

`merge` is used to initialize the counter to 0.

There is only a single `ite` in the counter, that in the `merge`. It is used to initialize the counter to zero.

The test case



reaches full decision coverage. For timestamp 0 `isNone(a)` evaluates to **true**. Conversely on timestamp 1 `isNone(a)` evaluates to **false**.

We would like to show, that the counter can count to higher numbers than 1.

As we have seen, decision coverage is insufficient for these cases. Yet more complex coverage criteria won't help here, because these specifications have use case specific constraints.

Instead we develop a mechanism, that allows users to pass custom constraints to the test case generator.

These custom constraints restrict the set of possible test cases. No generated test case will break them. This means, that passing contradicting constraints may lead to formulas that are unsatisfiable.

Inverting a constraint allows us to search for counter examples or proof that a formula has no solution, at least for the finite case. For example we may ask if there is an input, that does never initialize the counter to 0. Such an input does not exist and therefore the test case generator fails to find one.

These constraints must be written in some sufficiently expressive calculus.

Most of the time one wants to express that a certain property will eventually hold or that it will never hold.

We introduce the operator `some`. `some` expresses that a given stream has an event.

Definition 3.3 (some)**.** *Let $s \in \mathcal{S}_{\mathbb{D}}$ be a* TeSSLa *stream. Let $t \in \mathbb{T}_{\infty}$ be a timestamp so that s is known until t: $\forall t' < t : s(t) \neq ?$*

Then `some` *can be defined as*

$$[\![\textbf{some}\ \ \mathtt{s}]\!] \equiv \exists t' < t : s(t) \neq \bot$$

Now that there is the `some` operator, that acts as an existential quantifier, one might want to define the dual universal quantifier. A hypothetical `full` quantifier has no useful meaning, because there is no sense in specifying that a stream has always events. Instead the `empty` operator asserts that a stream has no events.

Definition 3.4 (empty)**.**

$$[\![\textbf{empty}\ \ \mathtt{s}]\!] \equiv \neg[\![\textbf{some}\ \ \mathtt{s}]\!]$$

Despite the name, a stream may eventually have events after the observed prefix. This happens if additional inputs occur, but may also appear if there is still a pending `delay`. Unfortunately this property makes it possible to generate event-free test cases for TeSSLa-specifications that are not empty when extended to an infinite trace.

For example the code

```
in d: Events[Int]
def z := delay(d, d)
```

with constraints `empty z` and `some d` can be fulfilled by an event on `d` at t_n on every prefix trace, but there is no solution for the infinite case, because `z` will fire at $t_n + d(t_n)$.

These two operators can be combined with TeSSLa formulas to produce arbitrarily complex criteria.

Theorem 3.5 (Disjunction of exists operators)**.** *Let $a \in \mathcal{S}_{\mathbb{D}_A}, b \in \mathcal{S}_{\mathbb{D}_B}$ be streams. Then the logical disjunction of* `exists a` *and* `exists b` *is*

$[\![$**some** `a`$]\!] \vee [\![$**some** `b`$]\!] \equiv [\![$**some** `merge(const(□,a), const(□,b))`$]\!]$

`merge` combines the streams into a single stream. If an event occurs on either `a` or `b` then there is also an event on the merged stream. The calls to `const` replace the values of the streams with the unit value □, effectively ignoring the values and making the merge work on differently typed streams.

Theorem 3.6 (Conjunction of exists operators)**.** *Let $a \in \mathcal{S}_{\mathbb{D}_A}, b \in \mathcal{S}_{\mathbb{D}_B}$ be streams. Then the logical conjunction of* `some a` *and* `some b` *is*

$[\![$**some** `a`$]\!] \wedge [\![$**some** `b`$]\!] \equiv [\![$**some** `slift(f)(a, b)`$]\!]$

where $f(x, y) = \square$.

The signal lift implements signal semantics in TeSSLa, by applying a function to the last seen values on the streams. `slift` does not emit events until both argument streams had an event. Therefore the constant function f only emits an event if and only if there has been at least on event on each stream.

Note 3.7: Conjunction as separate assertions:

$[\![$**some** `a`$]\!] \wedge [\![$**some** `b`$]\!]$

can be replaced with two separate assertions `some a` and `some b` for test case generation, because all generated test sequences need to fulfill all of the listed constraints.

Assertions in TeSSLa specifications

For simplicity we would like to write the constraints as annotations into the TeSSLa specifications. This allows us to keep specification and additional constraints in a single place.

```
# assert some z
# assert empty merge(a, b)
```

These assertions are treated as comments by the TeSSLa compiler. This allows us to add annotations, that are specific for test case generation, without breaking the specifications for evaluation with the TeSSLa interpreter.

3.1.3. Extensions for Boolean streams

Now that the logic can specify existence and non-existence of events, it would be beneficial to constrain possible values as well.

We will see that TeSSLa streams and the existential quantifiers are enough to specify properties of values.

Another built-in operator is `exists`. `exists z` specifies, that a Boolean stream z contains at least one event with value **true**. `exists` can be expressed as combination of `some` and a `filter` function. Instead of the binary `filter` we use a unary one, that takes a Boolean stream as input and removes all **false** events.

For a $p \in \mathcal{S}_{\mathbb{B}}$ the constraint

```
# assert exists p
```

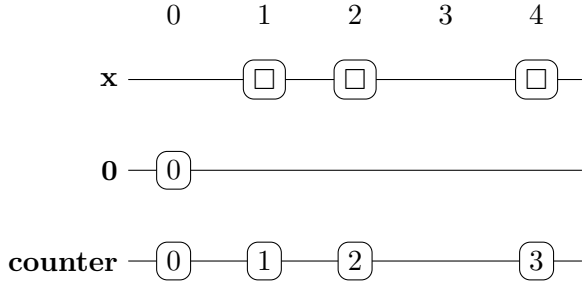
can be expanded into

```
def exists(z:Events[Bool]) := lift1(f)(z) where {
    def f(x:Option(Bool)) := ite(getSome(x), x, None[Bool])
}
# assert some exists(p)
```

These operator allows the specification of arbitrary conditions. If we return to the example with the event counter in Listing 3.1.2 an additional constraint

```
# assert exists counter==3
```

enforces a test case, that has some event with value 3, for example:



A counter itself can be used to specify even more properties. If a stream `z` should have at least two events, it may be combined into:

```
# assert exists count(z)>=2
```

For convenience the dual operator `all`

```
# assert all p
```

is provided and defined as

$[\![\texttt{all}\ \ p]\!] \equiv [\![\neg\texttt{exists}\neg p]\!] \equiv [\![\texttt{empty exists}(\neg p)]\!]$.

`all` enforces that a condition holds for every event on the stream, but does not imply the existence of events. This means, that a given `all` constraint can be trivially fulfilled with an empty stream. It can be used to further constrain the domain of allowed values on streams.

For example

```
in x: Events[Int]
# assert all x>=0
```

restricts the domain of inputs to unsigned integers.

3.1.4. Alternatives

The language for custom constraints uses `some`, `empty` operators and TeSSLa formulas. Alternative approaches may be considered. *linear time logic* (LTL) would support a broad set of operators. **F** (finally) is very similar to approximately to `some`. LTL would directly allow mixing of operators from propositional logic with temporal operators in any order, whereas the formulation in subsection 3.1.2 requires the existential operators on the left and needs to perform all other operations in TeSSLa on streams.

LTL and other approaches like regular expressions would directly support specifying sequences whereas implementing automata in TeSSLa is somewhat cumbersome.

Yet the current approach greatly benefits from its simplicity. With only two additional operators the expressivity of TeSSLa can be used for describing constraints. Additionally TeSSLa streams are very different from the word-based definitions of LTL and regular expressions. Extending these with operators that distinguish between existence of events and operators over the values within these events complicates the alternatives even more.

Finally the current mechanism only describes minimal requirements. Additional features can be built in TeSSLa and provided as a library or implemented as syntactic sugar in the test case generator.

3.1.5. Coverage strategies

SMT solvers get a formula as input and determine a single possible assignment for it. We don't want a single solution, but a set of solutions, that reaches the highest possible degree of coverage. Even a degree of 100% is not possible for many specifications. Ideally few test cases can reach the maximal coverage.

We can translate the coverage criteria into a set of SMT constraints. Adding this whole set to the constraints to the formula that the specification translates to forces the solver

to find a single solution, that fulfills all of the constraints, i.e. reaches 100% coverage. 100% coverage is not always possible much less in a single example. Many of the formulas become unsatisfiable, so that the solver can no longer find a valid test case at all.

There are four strategies for generating a test suite.

Simple strategy The *simple strategy* just ignores the coverage criteria and searches for a single test case.

Hard strategy As an alternative to the simple strategy one may just set all the constraints and either find a full coverage or no solution at all. We call this the *hard strategy*.

These strategies are not particularly helpful for most specifications. Instead we want to get multiple different test cases.

If we treat the problem no longer as a satisfiability problem, but as an optimization problem, we can formulate an objective function, that grows with the number of satisfied coverage constraints.

Soft strategy Fortunately Z3 supports optimization problems and also simplifies this kind of problem by providing *soft assertions*. These can be used like ordinary constraints, but they use the optimizer to fulfill many, but not necessarily all of the constraints.

Soft constraints are suitable for iterative generation of more test cases. In each run only the coverage constraints from the set of unresolved constraints are used as soft asserts. The soft strategy maximizes the number of fulfilled constraints per run.

Using soft asserts forces the implementation to switch to an optimizer for the whole formula, not just for some constraints. This makes the constraint-solving slower and sometimes impossible.

Iterative strategy For the iterative strategy we take the idea from the soft strategy and iteratively search for more tests, but we return to the classical, more efficient solver.

The iterative strategy sets a single of the coverage constraints in each run, and generates a test case for that constraint. Afterwards that coverage constraint is removed from the set and test case generation continues with the next unfulfilled constraint.

Otherwise if no solution was found for that particular constraint, we know that no possible test case can fulfill that constraint and remove it from the set.

This approach generates a maximal coverage, but may need to generate two test cases for each `ite`-operator. In practice a lot fewer are necessary, because some constraints are satisfied by previous cases, even when they were not set as constraints.

Comparison

The simple strategy ignores the coverage constraints. It can be used if only user-defined constraints are required.

Soft and iterative strategy both support iterative generation of multiple test cases. The soft strategy shows many interesting properties of the specifications in few examples, yet the need for an optimizer probably reduces performance. The iterative strategy supports iterative generation with the ordinary solver. It is probably the most versatile and most promising strategy.

The hard strategy only works if a complete coverage can be reached within a single test case. This is rarely the case and becomes impossible even with simple TeSSLa specifications, for example those that contain `merge(x,x)`. Merge is defined by

```
1 def merge[T](x: Events[T], y: Events[T]): Events[T] := {
2   def f(a: Option[T], b: Option[T]) :=
3     if isNone(a) then b else a
4   lift(x,y,f)
5 }
```

The lifted function `f` is only evaluated if at least one of the streams `x` and `y` has an event. `isNone(a)` in line 3 only evaluates to **true** if only the second stream has an event. Since both input streams of the merge are the same stream, they always have simultaneous events. This means, that the `isNone(a)` in line 3 never evaluates to **true** and therefore a full decision coverage becomes impossible. The iterative and soft strategy can skip a coverage constraint without generating a test case for it, but the hard strategy fails.

3.2. Representation as SMT formula

3.2.1. Stream and event representation

The test case generation only generates finite prefixes up to a finite $n \in \mathbb{N}$. Therefore we can assume, that $s(t_i) \neq ?$ for all $t_i \leq t_n$ where $t_n \in \mathbb{T}$ is the last timestamp in the generated prefix.

In TeSSLa a stream s consist of a discrete sequence of events. In [CHL+18] this sequence is defined as a sequence of $t_i \cdot d_i$ pairs, where $t_i \in \mathbb{T}$ is the timestamp and $d_i \in \mathbb{D}_s$ is the datum.

For the test case generation a different representation is more suitable. Every stream might be imagined as a sequence of n values. All streams are synchronized by a global set of timestamps and streams without an event at a timestamp contain $\perp$.

The test case might be imagined as a grid.

t_1	t_2	t_3	t_4	t_5	t_6		...	t_n
	0		42					-13
		true						
□								

Each of the streams has n slots, where each slot may contain an event, but does not have to. The slots determine the order, that events have, but the timestamps $t_1, ..., t_n$ are not fixed at start of generation. Instead they can be freely chosen by the SMT solver. Outside of these slots no events are allowed to happen.

Now a finite and quantifier-free formula is derived from the TeSSLa specification by unrolling the recursive formulas, that define the streams, n times.

Definition 3.8. *Let $\mathcal{T} = \{t_1, t_1, ..., t_n\}$ with $\mathcal{T} \subset \mathbb{T}$ and $t_1 < t_2 < ... < t_n$ be the global set of timestamps.*

For all streams s and all $t' \in \mathbb{T} : (t' \leq t_n \wedge s(t') \neq \bot) \Rightarrow t' \in \mathcal{T}$

Since $\mathcal{T}$ contains all the timestamps where events may happen up to t_n and only timestamps with events are of interest, $\mathbb{T}$ and $\mathcal{T}$ are often used synonymously within this thesis.

The chosen representation keeps SMT formula generation simple, because the order of events in different streams is implicit. Other representations require explicit constraints which restrict the order of events between streams.

SMT solvers don't necessarily support Option-types or $\bot$-values. In order to circumvent this limitation every stream s is represented by a sequence of n pairs $(s_{v,i}, s_{d,i}) \in \mathbb{B} \times \mathbb{D}_s$. $\mathbb{B} = \{\textbf{true}, \textbf{false}\}$ is the Boolean type with the usual interpretation. The first value $s_{v,i}$ is a Boolean flag that determines if the event is valid. The second value $s_{d,i}$ contains the datum of the event and is undefined if $s_{v,i}$ is **false**.

Definition 3.9. *The prefix of a stream s is defined as the tuples $s_v \in \mathbb{B}^n$ and $s_d \in \mathbb{D}_s^n$, where*

$$s(t_i) = \begin{cases} s_{d,i} & \textit{iff } s_{v,i} = \textbf{true} \\ \bot & \textit{iff } s_{v,i} = \textbf{false} \end{cases}$$

$s_{v,i}$ describes if an event happens on stream s at timestamp t_i. When thought of as the `Option`-type, $s_{v,i}$ is the type tag, that distinguishes between `Some` and `None`.

$s_{d,i}$ contains the datum if $s_{v,i}$ is set or is left undefined.

$$s_{d,i} = \begin{cases} s(t_i) & \text{iff } s(t_i) \neq \bot \\ \textit{undefined} & \text{iff } s(t_i) = \bot \end{cases}$$

undefined does not describe a specific bottom value, but is similar to *undefined behavior* in C. A stream may still set $s_{d,i}$ to a specific value but operators must not depend on any specific values of streams that are used as inputs. In most cases the undefined values are left unrestricted, so that the SMT solver can freely a suitable value.

3.2.2. Operator representation

The operators relate input to output streams. Each of the operators is defined by a set of constraints. The constraints, that relate types are not translated into an SMT formula but used by the formula generator for choosing the right variable types.

For reasons of efficiency and sometimes decidability it is advised to keep the constraints small and use formulas from a sufficiently simple theory, even when the SMT solver supports more complex ones with direct translation.

The following operators use the $\forall$-Quantor in the constraints, but only with rank 1 and only when the bound variable has a finite and sufficiently small domain. An implementation should use a `for`-loop to unroll the quantified constraint into a set of quantifier-free constraints.

Definition 3.10: $s = \textbf{nil}$**:**
nil creates an empty stream without any events.

$$\forall i : s_{v,i} = \textbf{false}$$

Definition 3.11: $s = \textbf{unit}$**:**
unit creates a stream, that has a single event at $t_1 = 0$ and is empty otherwise.

$$\begin{aligned} \mathbb{D}_s &= \{\square\} \\ t_1 &= 0 \\ s_{v,1} &= \textbf{true} \\ s_{d,1} &= \square \\ \forall i > 1 : s_{v,i} &= \textbf{false} \end{aligned}$$

$\mathbb{D}_s$ contains a single element which may be represented by any value, such as 0 or the empty record.

Definition 3.12: $s = \mathbf{time}(e)$**:**
time(e) creates a stream, that maps every event on stream e to the timestamp of that event.

$$\begin{aligned} \mathbb{D}_s &= \mathbb{T} \\ \forall i : s_{v,i} &= e_{v,i} \\ \forall i : s_{d,i} &= t_i \end{aligned}$$

Definition 3.13: $s = \mathbf{last}(a, b)$**:**
last(a, b) creates a stream, that yields the last value of stream a, whenever b has an event.

Let $m \in \mathbb{B}^n$ be a helper variable. It memorizes if an event has previously occurred on a.

$$\begin{aligned} \mathbb{D}_s &= \mathbb{D}_a \\ m_1 &= \mathbf{false} \\ \forall i > 1 : m_i &= m_{(i-1)} \vee a_{v,(i-1)} \\ \forall i > 1 : s_{d,i} &= \begin{cases} a_{d,i-1} & \text{for } a_{v,i-1} = \mathbf{true} \\ s_{d,i-1} & \text{else} \end{cases} \\ \forall i : s_{v,i} &= b_{v,i} \wedge m_i \end{aligned}$$

last memorizes the value last seen on a in $s_{d,i}$, even when s has no event. This is valid, because the $s_{d,i}$ is *undefined* in the stream definition given previously and therefore may be set to an arbitrary value. m memorizes if an event has already occurred on a. $s_{v,i}$ is set to **true**, if there is an event on b and there was already one on a.

Without helper variables m, the last line in the formula could be written in the form

$$\forall i : s_{v,i} = b_{v,i} \wedge (\bigvee_{j<i} a_{v,j}).$$

These constraints grow with the size of n. Large clauses are hard to solve for SMT solvers. By introducing the helper variable and iteratively memorizing the result of the last step in it, we get a maximal constraint size, that is independent from the used n. Of course the number of constraints still grows linearly with n.

Definition 3.14: $s = \mathbf{lift}_1(f)(e)$**:**
$\mathbf{lift}_1$ is the unary lift.

$$\begin{aligned} \forall i : s_{v,i} &= e_{v,i} \wedge (f(e_{d,i}) \neq \bot) \\ \forall i : (e_{v,i} \wedge f(e_{d,i}) \neq \bot) &\Rightarrow (s_{d,i} = f(e_{d,i})) \end{aligned}$$

Definition 3.15: $s = \mathbf{lift}_2(f)(a, b)$**:**

$\mathbf{lift}_2$ is the binary lift.

$$\forall i : s_{v,i} = (a_{v,i} \vee b_{v,i}) \wedge (f(a(t_i), b(t_i)) \neq \bot)$$
$$\forall i : s_{v,i} \wedge f(a_{d,i}, b_{d,i}) \neq \bot \Rightarrow s_{d,i} = f(a_{d,i}, b_{d,i})$$

Both lift-functions are currently not particularly interesting. The exact implementation depends on the definition of the lifted functions.

Definition 3.16: $s = \mathbf{delay}(d, r)$**:**
The **delay** operator waits for a specified amount of time, before raising an event. It has two operands: d contains the delay time, that needs to pass before **delay** triggers an event. r is the reset event, for setting and resetting the timer to the value in d.

$$\mathbb{D}_s = \{\square\}$$
$$\mathbb{D}_w = \mathbb{T} \setminus \{0\}$$

We introduce helper variables $\mathbf{isReset} \in \mathbb{B}^n$ and $\mathbf{fireAt} \in (\mathbb{T} \cup \{\bot\})^n$. **fireAt** can be seen as a timer, that memorizes at which timestamp the delay should "fire", i.e. at which time there should be an event on s.

isReset determines if the timer **fireAt** is reset.

$$\forall i : \mathbf{isReset}_i = r_{v,i} \vee (\mathbf{fireAt}_i = t_i)$$

Use **fireAt** to memorize, at which timestamp the **delay** will emit an event on s. Set it to $\bot$ if no event is expected.

$$\mathbf{fireAt}_1 = \bot$$

$$\forall i > 1 : \mathbf{fireAt}_i = \begin{cases} t_{i-1} + d_{d,i-1} & \text{for } \mathbf{isReset}_{i-1} \wedge d_{v,i-1} \\ \bot & \text{for } \mathbf{isReset}_{i-1} \wedge \neg d_{v,i-1} \\ \mathbf{fireAt}_{i-1} & \text{else} \end{cases}$$

Just like we did for **last** we use the helper variable to avoid large constraints.

Stream s fires whenever the timestamp has come:

$$\forall i : s_{v,i} = (\mathbf{fireAt}_i = t_i)$$

Additionally s is not allowed to skip the timestamp without firing:

$$\forall i : (\mathbf{fireAt}_i = \bot) \vee (t_i \leq \mathbf{fireAt}_i)$$

Since $t \geq 0$ for all $t \in \mathbb{T}$, one can use a negative value like -1 for representing $\bot$ in the SMT formula.

The TeSSLa-specification requires that all inputs are larger than 0. In order to avoid undefined behavior all values on d need to be constrained to values larger than zero:

$$\forall i : (d_{v,i} \Rightarrow (d_{d,i} > 0))$$

The given constraints allow for pending delays after the input. This has the confusing consequence, that a test case for a specification with assertion `assert empty z` might still lead to an event on `z` at some timestamp $t_{n+1} > t_n$.

For avoiding this an additional constraint might force a hypothetical $\mathbf{fireAt}_{n+1}$ to $\bot$.

$$\begin{aligned}
\bot &= \mathbf{fireAt}_{n+1} \\
&= \begin{cases} t_{n+1-1} + d_{d,n+1-1} & \text{for } \mathbf{isReset}_{n+1-1} \wedge d_{v,n+1-1} \\ \bot & \text{for } \mathbf{isReset}_{n+1-1} \wedge \neg d_{v,n+1-1} \\ \mathbf{fireAt}_{n+1-1} & \text{else} \end{cases} \\
&= (\mathbf{fireAt}_n = \bot \vee \mathbf{isReset}_n \wedge \neg d_{v,n})
\end{aligned}$$

Using this additional constraint on every **delay** means that a `empty` assertion really means, that no event will occur on that stream. It also means, that no event will ever happen after t_n for any stream, and force test case generator and interpreter to have equivalent output for any prefix.

Constraining **delay**s also reduces the set of viable test cases and even the set of testable programs. Programs with infinite output traces become impossible to test.

3.2.3. Types

Integers and Booleans are directly represented by SMT solvers. This may later include additional types.

Tuples and objects are unpacked into a sequence of variables. For example a tuple of type a `(Int, (Bool, (), Int)` gets unpacked into three variables $a_0 \in \mathbb{Z}$, $a_1 \in \mathbb{B}$, $a_2 \in \mathbb{Z}$. Equality constraints between data types are translated into a conjunction of equality constraints over the members. This means, that the unit type is not represented in the resulting formula at all. A stream of type unit consists only in valid flags, but has no actual datum, and also the equality constraints on the datum $s_{d,i}$ are only presented as a **true** constant to the SMT solver.

The `Option[T]` type is represented like a tuple $\mathbb{B} \times$ `T`. This representation is equal to those of events.

3.2.4. Lifted functions

The lifted functions are translated into adequate Z3 constraints. Most of these operators for integer arithmetic and Boolean logic have a direct counterpart in Z3. The only exceptions are operations that access or set parts of composed types.

3.2.5. Custom assertions

For supporting custom assertions, it suffices to express `some s` and `empty s` for a stream $\mathtt{s} \in \mathcal{S}_{\mathbb{D}}$ in a flattened specification.

Let $e \in \mathbb{B}^n$ be a set of helper variables. e_i memorizes if an event has occurred on the stream `s` at some point at or before t_i.

$$\begin{aligned} e_1 &= s_{v,1} \\ \forall i > 1 : e_i &= e_{i-1} \vee s_{v,i} \end{aligned}$$

For `some` an event must have occurred at or before t_n:

$$e_n.$$

For `empty` an event must not have occurred

$$\neg e_n.$$

All other operators are reduced to `some` and `empty`, using the unary filter function from subsection 3.1.2.

3.2.6. Coverage criteria

Automated coverage is achieved by maximizing decision coverage on lifted functions. For each expression $e := ite(cond, l, r)$ new variables $e_{left} \in \mathbb{B}^n$ and $e_{right} \in \mathbb{B}^n$ are introduced. These variables describe, that the respective branch had an influence on the result, and are constrained by

$$\begin{aligned} \forall i : e_{left,i} &= parent_i \wedge cond_i \\ \forall i : e_{right,i} &= parent_i \wedge \neg cond_i \end{aligned}$$

where $parent \in \mathbb{B}^n$ indicates, that the ite-expression has an influence on the result or in other words is not masked out by another *ite*.

If e is in the left or right branch of another *ite*-expression b, then $parent = b_{left}$ and $parent = b_{right}$ accordingly. Otherwise if there is no such b, then there is a $s := \mathbf{lift}(..)(..)$, that controls if the expression is evaluated. In this case $\forall i : parent_i = s_{v,i}$.

The decision coverage requires, that each decision has been taken once. Therefore additional variables $e_{onceLeft}, e_{onceRight} \in \mathbb{B}^n$ are introduced with constraints

$$\begin{aligned} e_{onceLeft,0} &= e_{left,0} \\ \forall i > 1 : e_{onceLeft,i} &= e_{onceLeft,i-1} \wedge e_{left,i} \\ e_{onceRight,0} &= e_{right,0}. \\ \forall i > 1 : e_{onceRight,i} &= e_{onceRight,i-1} \wedge e_{right,i}. \end{aligned}$$

4. Implementation

The previous chapter explains the concepts behind test case generation for TeSSLa and gives a translation from TeSSLa specifications into logical formulas, that can be solved by a common SMT solver like Z3.

The test case generator tessla-testgen has been implemented on the defined concepts.

There is an existing implementation of a TeSSLa interpreter [CHL+18]. tessla-testgen reuses this interpreter for parsing the code and transforming the high level data structure into a core language TesslaCore, that allows simpler transformation of code.

The implementation of tessla-testgen closely resembles the concepts from chapter 3. It generates a test suite with decision coverage by using the strategies and supports custom constraints as annotations on the code.

4.1. Technology

tessla-testgen is implemented in the programming language Scala. This allows the inclusion of the TeSSLa interpreter from [CHL+18] as a library for parsing the TeSSLa code and simplifying into the code representation TesslaCore. TesslaCore is a reduced set of TeSSLa code, which is similar to operations in [CHL+18]. Additional instructions are present from older versions of TeSSLa or provided for convenience. TesslaCore provides TeSSLa specifications in normal form, i.e. specifications that don't have nested expressions. Additional TeSSLa features like macros are fully expanded in TesslaCore.

4.2. Custom assertions

The TeSSLa compiler can not directly parse the custom assertions from subsection 3.1.2. Nonetheless the TeSSLa interpreter can parse the specification, because these assertions are written as comments.

At first tessla-testcase reads the specification into memory. It parses the assertions with a regular expression. Like many regular expression engines the one used by scala supports the extraction of *capture groups*. The regex for parsing

```
\s*#\s*assert\s+(\S+)\s+(.*)
```

uses one group for extracting the quantor and another for the stream definition. The later reads until the end of the line and allows reading of complex stream definitions like

```
# assert exists count(x)>=5
```

tessla-testgen expands the assertions on Boolean streams into TeSSLa streams and the core quantors `some` and `empty`. tessla-testgen expands the assertions on Booleans into TeSSLa streams according to the rules in subsection 3.1.2. The previous example gets expanded into

```
def zzz0 = lift(count(x)>=5, nil[()],
  (c:Option[Bool], _:Option[()]) =>
    if getSome(c) then c else None[Bool])

out zzz0
```

and a constraint `some zzz0`. `zzz0` is a fresh identifier. The stream is set as an output so that it can be mapped to a stream in TesslaCore, even after *name mangling* within the TeSSLa interpreter has changed the name of the stream.

The simplified quantor and stream, in this case `some zzz0` are memorized and passed to the SMT formula generator. The expanded specification is parsed by the TeSSLa interpreter and transformed into TesslaCore.

4.3. Architecture

The generator consists of 4 major components.

PlainTessla is an intermediate representation of TeSSLa code, that is more suitable for test-case generation. PlainTessla shares many similarities with TesslaCore with the primary difference, that all function calls are inlined. This mechanism does not allow recursively defined functions, but we have assumed that these will never occur in the specifications.

PlainTessla represents the code in a total of 8 type constructors:

`Nil, Time, Last, Delay, Merge, Lift, Unit`

Most operators, that TesslaCore support, with the exception of `merge` are rewritten in terms of lifted functions and other operators.

Additionally lifted functions are represented by:

`Constant, Input, Alias, Application`

`Input` is reading values from a stream by copying the content.

`Alias` is copying a value. This type constructor is not necessary and can be replaced in an optimizer pass, but having `Alias` simplifies the passing of arguments into inlined functions.

`Constant` represents a constant.

`Application` represents the application of a builtin operator. Most of these operators like + or == simply state a relation between input and output. `ite` is a builtin operator with some special treatment for code coverage.

Another type `Specification` contains all the simplified TeSSLa code in flat representation with the PlainTessla operators.

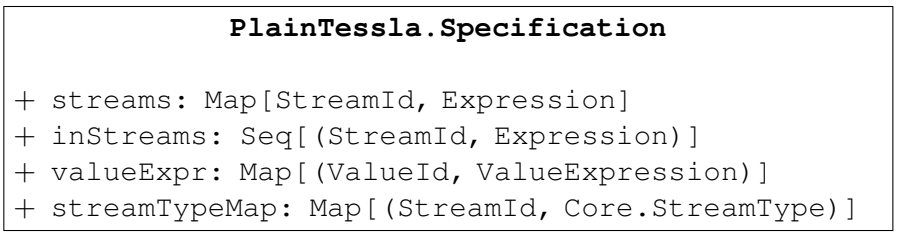


Figure 4.1.: `PlainTessla.Specification` is a datastructure which contains a simplfied representation of the TeSSLa code. `StreamId` and `ValueId` are typesafe indices to expressions

TypeInference is used to infer the types on all the expressions and streams. While TesslaCore already assigns types to the streams, it does not determine types for the lifted expressions. Expansion of operators, that are represented in TesslaCore, but not in PlainTessla does not add types to the streams. Some of these operators like `slift` and `default` are present in nearly every practical TeSSLa specification.

This module defines types and type variables and provides the functionality for unifying them.

For now the type inference is far from optimal. A future implementation will probably remove the type inference from tessla-testgen and rely on type information from the TeSSLa compiler.

GenZ3 is the module, which does all of the constraint generation. The most important methods of the interface are *run* and *isSat*. `isSat` checks if a given formula with the assertions is satisfiable. This allows an early check, for distinguishing unsatisfiable inputs from those with unsatisfiable coverage criteria.

The constructor `GenZ3` translates the specification into an SMT formula using the constraints from section 3.2.

The method *build* constructs the SMT formula. *build* primarily pattern matches on all the built-in operators and generates the constraints. The implementation matches the description in section 3.2 on page 32. Most operators of lifted expressions contain direct counterparts in Z3. Constraints are created with the Java Z3 api. The code is very similar to the one in section 3.2. For example the constraints in section 3.2 for operator **time**(e) are

$$\begin{aligned} \mathbb{D}_s &= \mathbb{T} \\ \forall i : s_{v,i} &= e_{v,i} \\ \forall i : s_{d,i} &= t_i \end{aligned}$$

The code is

```
1  case Time(ref) => {
2    assert(out.ty.innerType == TypeInference.Int)
3
4    val tyRef = streams(ref)
5    for (i <- eventIdx) {
6      constrain(
7        ctx.mkEq(out.valid(i), tyRef.valid(i)),
8        ctx.mkEq(out.data(i)(0), times(i))
9      )
10   }
11 }
```

Line 2 checks if the type has been correctly determined by the type inference. The $\forall$ quantor is replaced with a for-loop over all the $1 \leq i \leq n$ (line 4). The method `constrain` adds some constraints, here two equality constraints.

Additionally the dependency tree in each lifted expression is traversed for each **lift** to generate the conditionals, that detect if an if statement has been masked out, as well as the conditionals that describe that the if has evaluated to **true** and **false**.

run is the method, that generates a test-case. It sets the coverage constraints, runs the solver, converts Z3's model into a test-case for tessla, and updates the set of yet unresolved constraints.

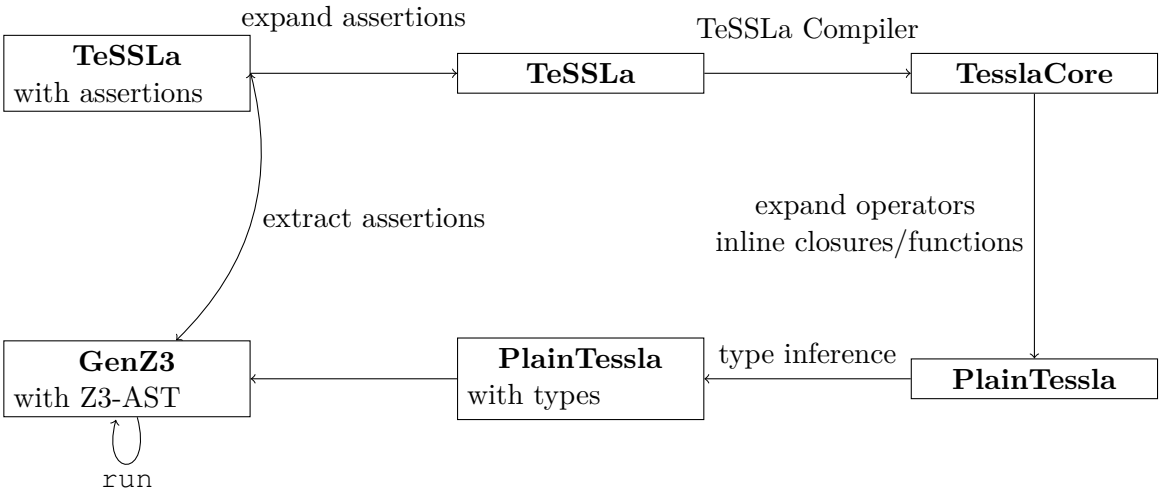
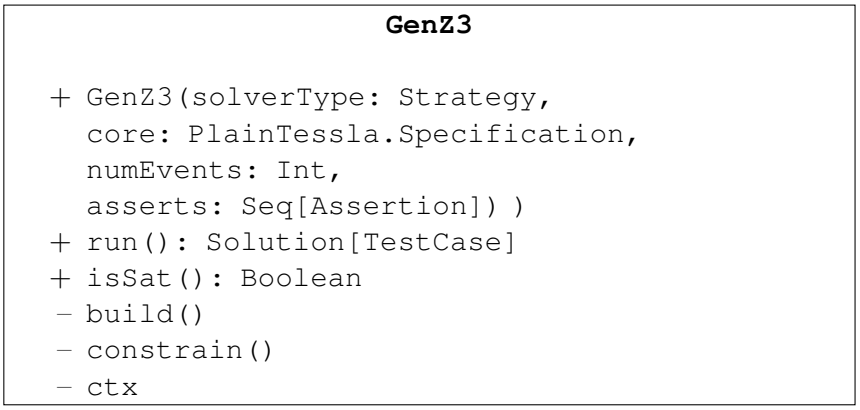


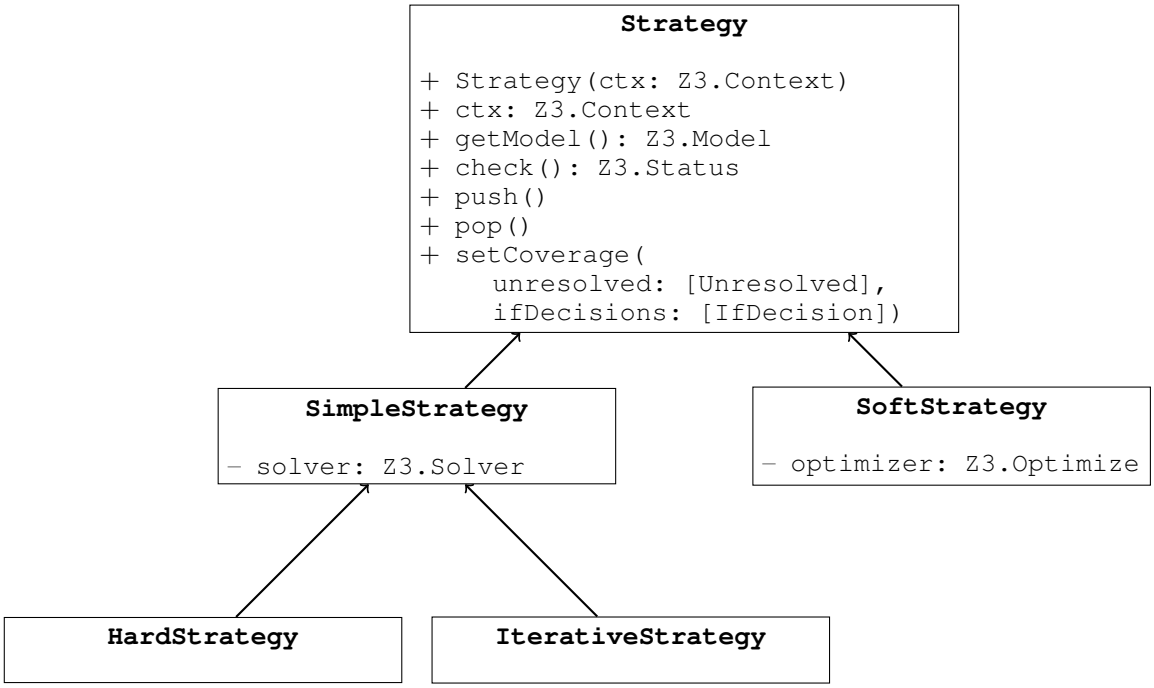
Figure 4.2.: Operation chart for turning TeSSLa code into a datastructure for Z3.



Main module performs the expansion of custom assertions and the directs the expansion into PlainTessla. It also directs the iterative generation of multiple test cases, by repeatedly calling *run*.

Figure 4.2 shows how the input files are transformed into SMT formulas. Assertions are parsed from the file and expanded to TeSSLa code. The interpreter from [CHL+18] is used to turn this code into the TesslaCore representation. Inlining and expansion transforms TesslaCore to the simpler representation PlainTessla and type inference is used to determine the types. *GenZ3* translates PlainTessla and the assertions into a SMT formula. Repeated calls to *run* set different coverage constraints and generate the test suite.

Strategies The classes, that inherit from Strategy implement different strategies for coverage maximization.



The class `Strategy` and its derivates support different strategies for coverage generation. A detailed description of the different strategies is in section 3.1.5 on page 30. The class `Strategy` acts primarily as an abstraction over the SMT solver `Solver` and the SMT optimizer `Optimize` from Z3. Most of the methods are set accordingly.

The only exception is the method `setCoverage`, which sets the constraints for the coverage.

The tessla-testgen implementation provides a total of 4 different strategies for test coverage maximization.

The coverage strategy is chosen with the command-line flag `--strategy`. Some of the coverage strategies support interactive generation of multiple test cases.

The interactive mode can be enabled with the command line flag `--interactive`. It outputs one test-case for each line on the standard input (stdin). If used from the command line this mode allows to produce a new input at key-press. Additionally, primarily for measuring the time, a full set can be produced with the flag `--generateAll`.

For interactive mode, testgen maintains a set of all constraints, that have not been fulfilled at least once. After each test case it removes all the fulfilled from this set.

Output tessla-testgen outputs the input streams in the same format that is used by TeSSLa interpreter. Additional debug output can show the streams in a grid-based layout, like the one used in section 3.2. For the example with the filter function the output of all streams contains lines, that look like

```
#times:          0       1       2       3       4
c:                                false           true
x:               14                               9
$11:                                              false
c$10:                             false           true
z$1:                                              9
fresh@0:                                  false   false
```

The normal input look like this:

```
0: x = 14
2: c = false
4: c = true
4: x = 9
```

The output was generated with the hard strategy for $n = 5$.

The first line `#times` shows all the timestamps for $t_1, ...t_n$. The lines c and x show the input of the stream, z\$1 is the output stream. The output contains additional lines for many other internal streams and values in lifted functions, that is only relevant for debugging. The full output contains even more lines, but these were removed from this excerpt.

Indeed this output is similar to the one we defined in subsection 3.1.1. All three cases are covered, although x does not always have an input that shows whether the event has been removed or left on the stream.

5. Evaluation

5.1. Evaluation time

Since solving SMT formulas is NP-hard, there may be an exponential increase in run time and formulas may become infeasible at a certain size.

Two examples are prepared to measure the time it takes to generate the test cases for different n and different strategies.

The first example specifies signal decoder and is probably representative of many common use cases for specification, especially when monitoring is used on hardware specific cases. This example is evaluated for two time domains, one where $\mathbb{T} = \mathbb{N}$ and one where $\mathbb{T} = \mathbb{R}$, because linear constraints on reals belong to the class of convex problems, which are much easier to solve.

The second example introduces nonlinear computations which are much harder to solve, especially for the optimizer. This example is designed to push the optimizer of Z3 to its limit and probably show a difference between the strategies.

5.1.1. Example 1 (Linear arithmetic)

The first example is a signal decoder. A digital signal is transmitted as a sequence of events and encoded by frequency. The monitor reads the signals and translates them into a binary sequence. A signal is measured within an interval. The example uses an interval length of 10, which in an implementation with $\mathbb{T} = \mathbb{N}$ allows for maximal 10 events.

The input is a stream of unit events. The frequency of events encodes binary numbers. If there are at least 8 events within a fixed interval, the transmitted number is 1. Between 3 and 5 spikes the number is 0. If the transmitter is off, no events will be received. Faulty input sequences, e.g. caused by a noisy channel are indicated by an event on the stream `decodeErr`.

An excerpt of the code can be seen in Listing 5.1 on the next page. The full code is shown in section A.1. Input it received on the stream `wire`. An interval begins with an event. A timer is started to reset the counter and trigger the output at the end of the interval. This output is either a valid binary number on `decoded` (if `freq` is in the mentioned ranges) or an error event is raised on `decodeErr` otherwise.

```
in wire: Events[Unit]
def isInterval: Events[Bool] = ...
def firstInInterval:Events[()] := filter(wire, cond) where {
  def cond := merge(true, last(!isInterval, wire))
}

def timer := delay(const(interval, firstInInterval),
                   firstInInterval)
def freqCounter := resetCount(wire, timer)
def freq := last(freqCounter, timer)

def decoded := lift1(freq, ...)
def decodeErr := lift1(freq, ...)
out decoded
out decodeErr
```

Listing 5.1: Excerpt from the example code

This example is evaluated by measuring the run-time of the test case generation.

The four different generation strategies *simple, soft, soft-all and iterative-all* are compared. Simple and soft generate a single test case, whereas *soft-all* and *iterative-all* generate test cases, until the coverage is maximal. The *hard* strategy is impossible, because requiring full coverage leads to a contradiction. The matching *iterative* strategy, that does only generate a single case is excluded because it only differs from the *simple* strategy by a single constraint.

For the test case generation only the time for generating the constraints and running Z3 is measured. This excludes the overhead, that parsing, code expansion and type inference introduces. Especially the last one uses a suboptimal algorithm.

The measured data is plotted in Figure 5.1 on the facing page. The data is the mean of 10 runs.

For $n = 10$ the generated formula contains 1850 variables, primarily Booleans, and 1742 constraints. Yet the actual difficulty of the problem can not easily determined from these numbers. A single constraint can be quite large and translation into normal form may increase the number of constraints. On the other hand manual inspection of the formula shows that many constraints are eliminated by the reprocessing steps of Z3 or in the first steps of the DPLL-Solver. In general most of the formulas, that TeSSLa specification translate to seem to be rather simple to solve.

While the simple case without coverage criteria quickly produces results even for large event sizes, the other techniques do grow in run-time, but do not become infeasible for sizes up to 100 events. The resulting formulas are very large, yet Z3 quickly finds a solution to them.

This grows the confidence that test case generation works for realistic cases.

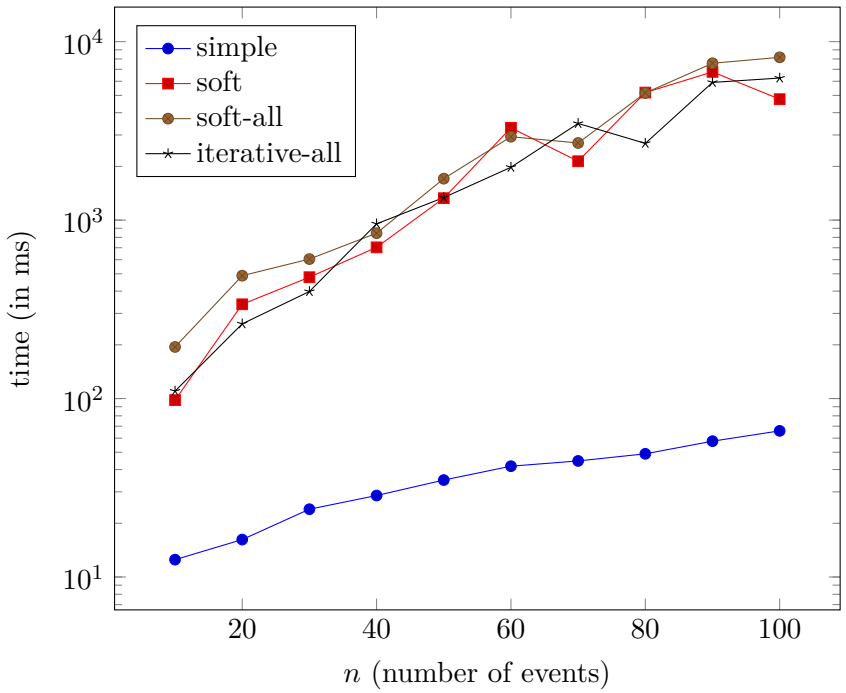


Figure 5.1.: Time in ms of test case generation with $\mathbb{T} = \mathbb{N}$

An example for a full test-suite is shown in Table 5.1 on the next page. In contains five different test cases. The first test case (case 0) contains no event. Some of these tests are redundant. For example case 0 is included in the prefix of case 1 up to $t_i = 15$ and case 1 is included in case 2. In comparison the soft strategy can generate a test-suite with only two cases. It is shown in Table 5.2 on page 53. Despite requiring fewer runs of the SMT solver, the soft strategy was not faster than the iterative strategy.

Case 0		Case 1		Case 2	
Input	Output	Input	Output	Input	Output
		16: wire	26: timer	13: wire	23: timer
		24: wire	26: freq = 3	17: wire	23: freq = 3
		25: wire	26: decoded = 0	21: wire	23: decoded = 0
				26: wire	36: timer
				34: wire	36: freq = 3
				35: wire	36: decoded = 0
				36: wire	

Case 3		Case 4	
Input	Output	Input	Output
0: wire	10: timer	0: wire	10: timer
10: wire	10: freq = 0	1: wire	10: freq = 6
15: wire	10: decodeErr	2: wire	10: decodeErr
17: wire		4: wire	21: timer
30: wire		5: wire	21: freq = 8
31: wire		6: wire	21: decoded = 1
		7: wire	
		11: wire	
		12: wire	
		13: wire	
		14: wire	
		15: wire	
		16: wire	
		17: wire	
		20: wire	
		21: wire	

Table 5.1.: This test-suite was generated with the iterative strategy for $n = 20$. It shows the timestamp, the stream and the value for each event. Values of type unit are omitted.

Case 0		Case 1	
Input	Output	Input	Output
0: wire	10: timer	1: wire	11: timer
11: wire	10: freq = 0	2: wire	11: freq = 5
19: wire	10: decodeErr	4: wire	11: decoded = 0
20: wire	21: timer	5: wire	22: timer
22: wire	21: freq = 3	6: wire	22: freq = 9
23: wire	21: decoded = 0	12: wire	22: decoded = 1
24: wire	32: timer	13: wire	
25: wire	32: freq = 6	14: wire	
26: wire	32: decodeErr	15: wire	
31: wire		16: wire	
32: wire		17: wire	
33: wire		18: wire	
34: wire		19: wire	
36: wire		20: wire	
38: wire		22: wire	

Table 5.2.: In comparison to Table 5.1 on the facing page the soft strategy can find a full coverage in only 2 cases.

Real time domain Linear programming is computationally simpler on reals than on integers. Linear programming in reals has algorithms that can find a solution in polynomial time, whereas linear integer programming is NP-complete. Therefore constraint generation on real numbers may find a solution quicker. The implementation in chapter 4 does not yet support reals and the time domain $\mathbb{T}$ is defined as integer. Yet it would be beneficial to see if constraints with linear real programming solves the problems quicker. Therefore all integers from the code are replaced with reals. The computation time is measured over 5 runs.

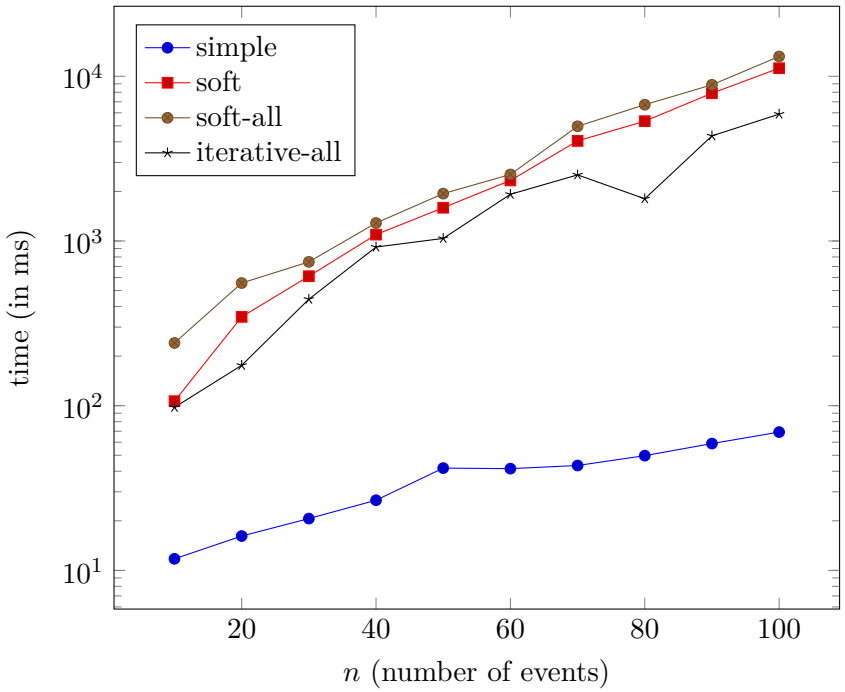


Figure 5.2.: Time in ms of test case generation with $\mathbb{T} = \mathbb{R}$

Switching to real numbers did not improve test case generation times. In fact the times are larger for the strategies using the optimizer. Timing with the iterative strategy is slightly better than without, although these may be influenced by external factors, since the measurements were not taken on the same day.

The inclination of the curves in Figure 5.1 and Figure 5.2 does not allow a conclusively answer whether the computation time grows exponentially or subexpontentially. The data indicates that growth is low enough that test case generation becomes feasible for large n. Larger specifications may probably reach a critical point earlier, but test case generation with the approach from this thesis probably remains feasible for these specifications.

5.1.2. Example 2 (Quadratic arithmetic)

A second examples shows the limits for more complex formulas.

```
in x: Events[Int]

def y = (x-4)*(x+9)
out y
```

The specification just calculates a quadratic polynomial with the roots 4 and 9.

The constraint `all y==0` forces the solver to search for roots of the polynomial. Additional constraints enforce, that two roots are found and that these two differ.

```
def y0 := last(y,y)

# assert all y==0
# assert some y
# assert some y0
# assert all x!=last(x,x)
```

The result of shows the time it took to find a solution. The soft strategy with the optimizer is much slower than the simple strategy with the solver. Due to the high calculation time only three cases $n \in \{5, 10, 15\}$ are evaluated with the optimizer and these values are averaged over only two runs. For larger values no time is determined. Yet the high difference in computation time shows that the optimizer become infeasible for specifications with nonlinear calculations. The time for $n = 15$ is much lower than the time for $n = 10$. Solving but especially optimizing SMT constraints depends heavily on backtracking. Sometimes the optimizer quickly finds a good solution while other times it has to try many different values. This makes timing occasionally unpredictable, especially for the optimizer.

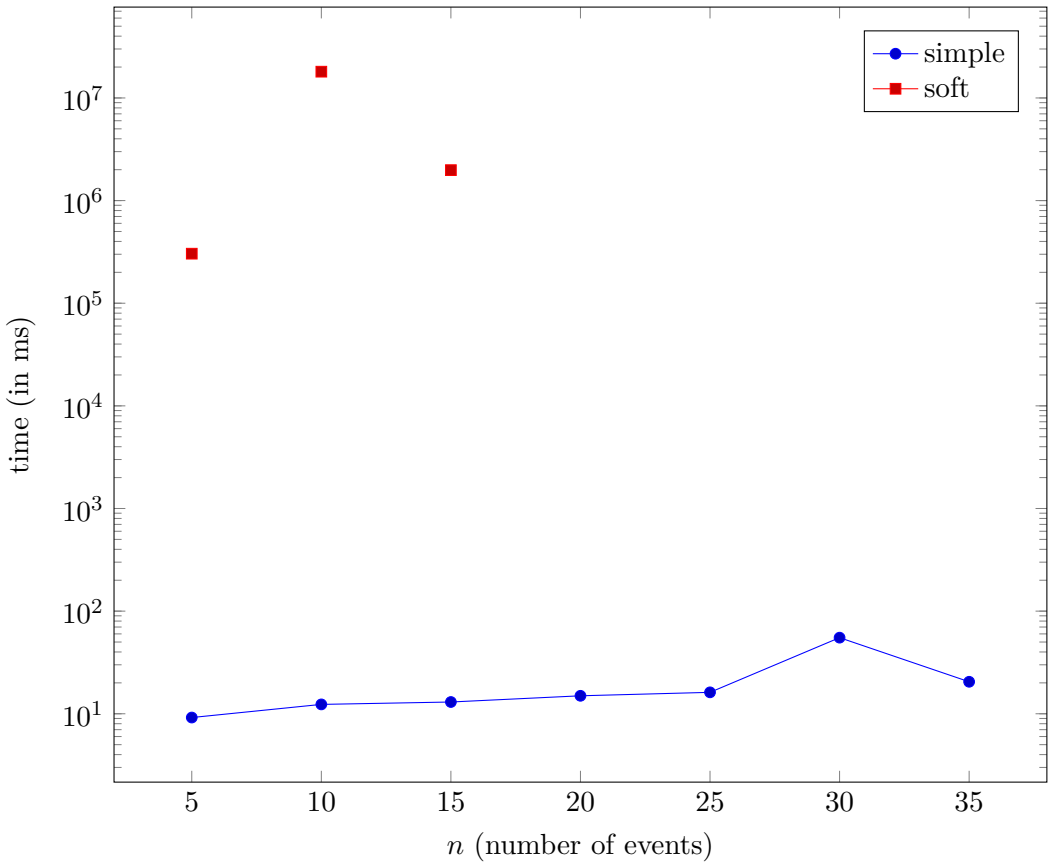


Figure 5.3.: Time in ms of test case generation with $\mathbb{T} = \mathbb{N}$. The soft strategy is much slower than the simple one.

5.2. Decidability and Complexity

The decidability and complexity of test case generation greatly depends on the complexity of the lifted functions. Nonetheless it makes sense to study the complexity of the test case generation with sufficiently restricted lifted functions.

SMT formula generation in section 3.2 only outputs constraints of constant size and each operator gets translated into a constant number of constraints per timestamp. For a specification of size m and n timestamp the resulting formula has a size of $\mathcal{O}(n \cdot m + l \cdot n))$ where l is the factor of the lifted formulas.

In order to evaluate the complexity of the TeSSLa formula itself, we formulate the decision problem as the satisfiability problem on TeSSLa specifications with custom assertions [subsection 3.1.2].

For the TeSSLa fragments $\text{TeSSLa}_{\text{bool}}$ and $\text{TeSSLa}_{\text{bool+c}}$ and all TeSSLa formulas with lifted functions, that are restricted to propositional logic and linear arithmetic, the re-

sulting formula only contains constraints with Boolean arithmetic and linear arithmetic (for $\mathbb{T} = \mathbb{R}$) or linear integer arithmetic (for $\mathbb{T} = \mathbb{N}$).

The satisfiability problem on linear integer programming and propositional logic is decidable and NP-complete. This means that for most TeSSLa formulas, that do not introduce more complex constraints through lifted functions, a test case can be found. While NP-hard is still a complexity class, that makes sufficiently large problems infeasible it is still relatively easy compared to other techniques that requre EXPTIME or a larger complexity class. Advancements in SMT solver technology made it possible to find solutions for many formulas.

5.3. Summary

The data indicates that test case generation remains feasible for larger specifications with linear arithmetic and Boolean constraints. The strategies that are based on the solver still works for the quadratic example.

In all examples the iterative strategy is at least as fast and often much faster than the alternative strategies, except for the simple strategy. The hard strategies can not be used at all.

This means that the iterative strategy is probably the most useful strategy. It can efficiently find a test suite with maximal coverage. It is based on the solver, which means that it can still be used to find solutions for constraints from theories, that are harder to solve e.g. quadratic programming and it is also open for extended possible extensions. For example Z3's solver supports theories for certain data types like arrays and sets. By implementing support in tessla-testgen these theories could be used with the iterative strategy without much change.

Besides the iterative strategy the simple one can be used if code coverage is not required. This strategy has all the advantages of the iterative strategy, but is even faster. It can be used to efficiently proof certain properties on specifications or otherwise find a counter example.

6. Conclusion and Outlook

In this thesis we have developed an algorithm for test-case generation for TeSSLa specification. For that we have seen a translation of TeSSLa into an SMT constraints. We have shown, that it is possible to translate all TeSSLa operators, including **delay**, into SMT constraints, that only consist of propositional logic and linear arithmetic. We have developed criteria for choosing interesting test cases for TeSSLa specifications.

The tool *tessla-testgen* has been implemented according to the theoretical concepts and experiments with this tool have shown that automatic test case generation is possible and feasible for a presumably certain specification.

While the evaluated example allows for quick generation, it is questionable, to which degree the example is representative of common use cases. One of TeSSLa's design goals is the usage on hardware, which restricts the specifications to those that that can be evaluated in constant memory. Additionally the application in runtime verification may indicate that specifications are primarily concerned with testing certain sequences of events rather than performing complex computations. Therefore it may be assumed, that comparable use cases primarily make use of propositional logic and linear arithmetic and that more advanced theories occur rarely.

6.1. Outlook

Future work may extend this thesis into three major directions. The first is an empirical evaluation of test case generation for different specification languages. TeSSLa is a very young and rarely used specification language. In general asynchronous stream-based languages are rare. Consequently only few specifications and programs in such languages exist. Without such empirical work most development is restricted to conjecture about the applicability and plausibility of new concepts.

The second direction defines new coverage criteria and compares different criteria. This thesis assumes, that decision coverage on the lifted functions is a sufficient coverage criterion on streams. One may use more detailed coverage criteria like MC/DC on the lifted functions or define entirely different criteria, that add specific constraints for certain operators. Additionally one may want to define global constraints, that add constraints to groups of operators or even the whole specification.

The third direction of potential future work refines strategies. These may be different strategies like incremental generation, similar to the ones [RNHW98] did for Lustre. These generate inputs for by iteratively choosing the next step from the current state.

Such an approach would speed up the generation process by a huge margin. The second example in subsection 5.1.2 the formula contains quadratic computations, that are much slower to find a solution for, at least with the soft strategy. With the incremental generation it suffices to find a solution for the polynomial twice. Afterwards the remaining constraints are fulfilled by empty streams.

Another advantage of the incremental approach is a new strategy for maximizing coverage with few test cases. Like with the iterative strategy only a single constraint is set every time, but it can be fulfilled for a small n. By extending this test case up to n_{max} a single example can satisfy multiple coverage constraints. Similar to the soft strategy this approach fits multiple cases in a single example, but it can use the ordinary SMT solver.

The current approach encodes the whole specification as an SMT formula. In imperative languages the computation path is often predefined and a technique like symbolic execution is used to generate a formula, that describes the input. This idea may be transferred to TeSSLa specifications. Instead of encoding the whole formula and then setting the coverage constraint one may choose all the decisions and prune all the masked expressions from the formula. This approach keeps the formulas simpler then they are currently. It may be combined with other improvements, for example with the incremental strategy.

A. Appendix

A.1. Example 1 (Linear arithmetic) (5.1.1)

```
# 1: Hi-frequency: >=8 in interval
# 0: Lo-frequency: 3<=x<=5 in interval
in wire: Events[Unit]
def interval: Int = 10

def isInterval: Events[Bool] = default(merge(const(true, wire),
                                             const(false, timer)), false)
def firstInInterval:Events[()] := filter(wire, cond) where {
  def cond := merge(true, last(!isInterval, wire))
}

def timer := delay(const(interval, firstInInterval), firstInInterval)
def freqCounter := resetCount(wire, timer)
def freq := last(freqCounter, timer)

out wire
out timer
out freq

def decoded := lift1(freq, f) where {
  def f(x: Int) :=
    if x>=8 then
      Some(1)
    else if x>=3 && x<=5 then
      Some(0)
    else
      None[Int]
}

def decodeErr := lift1(freq, f) where {
  def f(x: Int) :=
    if x<8 && x>5 then
      Some(())
    else if x<3 then
      Some(())
```

```
    else
      None[()]
}

out decoded
out decodeErr

# Could be in a standard library

def resetCount[T, B](x:Events[T], reset: Events[B]) := c where {
  def f(a: Option[Int], r: Option[()]) :=
    Some(if isNone(r) then
      getSome(a) + 1
    else if isNone(a) then
      0
    else
      1)
  def c: Events[Int] = merge(lift(last(c,x), const((), reset), f), 0)
}

def lift1[T, V](x: Events[T], f: (T) => Option[V]) := {
  def ff(a: Option[T], b: Option[Unit]) := f(getSome(a))
  lift(x, nil[Unit], ff)
}

def prev[T](x: Events[T]) : Events[T] = last(x,x)

def filter[T](events: Events[T], condition: Events[Bool]): Events[T] :=
    lift(events, c, f) where {
  def c := merge(condition, last(condition, events))
  def f(e: Option[T], c: Option[Bool]): Option[T] :=
    if isNone(c) then None[T]
    else if getSome(c)
    then e else None[T]
}
```

Bibliography

[AOH03] Ammann, Paul ; Offutt, Jeff ; Huang, Hong: Coverage criteria for logical expressions. In: *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* IEEE, 2003, S. 99–107

[Ber98] Berry, G erard: *The foundations of Esterel.* 1998

[BR70] Buxton, John N. ; Randell, Brian: *Software engineering techniques: report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969.* NATO Science Committee, 1970. – 65–68 S.

[BT18] Barrett, Clark ; Tinelli, Cesare: Satisfiability Modulo Theories. Version: 2018. http://dx.doi.org/10.1007/978-3-319-10575-8_11. In: Clarke, Edmund M. (Hrsg.) ; Henzinger, Thomas A. (Hrsg.) ; Veith, Helmut (Hrsg.) ; Bloem, Roderick (Hrsg.): *Handbook of Model Checking.* Cham : Springer International Publishing, 2018. – DOI 10.1007/978–3–319–10575–8_11. – ISBN 978–3–319–10575–8, 305–343

[CHL+18] Convent, Lukas ; Hungerecker, Sebastian ; Leucker, Martin ; Scheffel, Torben ; Schmitz, Malte ; Thoma, Daniel: TeSSLa: Temporal Stream-based Specification Language. In: *CoRR* abs/1808.10717 (2018). http://arxiv.org/abs/1808.10717

[CPRZ89] Clarke, Lori A. ; Podgurski, Andy ; Richardson, Debra J. ; Zeil, Steven J.: A formal evaluation of data flow path selection criteria. In: *IEEE Transactions on Software Engineering* 15 (1989), Nr. 11, S. 1318–1332

[DLL62] Davis, Martin ; Logemann, George ; Loveland, Donald: A machine program for theorem-proving. In: *Communications of the ACM* 5 (1962), Nr. 7, S. 394–397

[DMB08] De Moura, Leonardo ; Bjørner, Nikolaj: Z3: An efficient SMT solver. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems* Springer, 2008, S. 337–340

[dSS+05] d'Angelo, Ben ; Sankaranarayanan, Sriram ; Sánchez, César ; Robinson, Will ; Finkbeiner, Bernd ; Sipma, Henny B. ; Mehrotra, Sandeep ; Manna, Zohar: LOLA: Runtime monitoring of synchronous systems. In: *12th International Symposium on Temporal Representation and Reasoning (TIME'05)* IEEE, 2005, S. 166–174

[EH97] Elliott, Conal ; Hudak, Paul: Functional Reactive Animation. In: *International Conference on Functional Programming*, 1997

[Hal98] Halbwachs, Nicolas: Synchronous programming of reactive systems. In: *International Conference on Computer Aided Verification* Springer, 1998, S. 1–16

[HCRP91] Halbwachs, Nicholas ; Caspi, Paul ; Raymond, Pascal ; Pilaud, Daniel: The synchronous data flow programming language LUSTRE. In: *Proceedings of the IEEE* 79 (1991), Nr. 9, S. 1305–1320

[Kin76] King, James C.: Symbolic execution and program testing. In: *Communications of the ACM* 19 (1976), Nr. 7, S. 385–394

[Leu11] Leucker, Martin: Teaching runtime verification. In: *International Conference on Runtime Verification* Springer, 2011, S. 34–48

[MHM+95] Müllerburg, Monika ; Holenderski, Leszek ; Maffeis, Olivier ; Merceron, Agathe ; Morley, Matthew: Systematic testing and formal verification to validate reactive programs. In: *Software Quality Journal* 4 (1995), Nr. 4, S. 287–307

[Pnu77] Pnueli, Amir: The temporal logic of programs. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)* IEEE, 1977, S. 46–57

[RNHW98] Raymond, Pascal ; Nicollin, Xavier ; Halbwachs, Nicolas ; Weber, Daniel: Automatic testing of reactive systems. In: *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)* IEEE, 1998, S. 200–209

[TFMC94] Thévenod-Fosse, Pascale ; Mazuet, Christine ; Crouzet, Yves: On statistical structural testing of synchronous data flow programs. In: *European Dependable Computing Conference* Springer, 1994, S. 250–267