MLVR: Regular Expression-Based Specification for Verified Model Checking of Hardware

by

Gabriel A. Kammer

SB, Computer Science and Engineering, MIT, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2024

Authored by:	Gabriel A. Kammer Department of Electrical Engineering and Computer Science January 19, 2024
Certified by:	Adam Chlipala Professor of Computer Science, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

MLVR: Regular Expression-Based Specification for Verified Model Checking of Hardware

by

Gabriel A. Kammer

Submitted to the Department of Electrical Engineering and Computer Science
on January 19, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Model checking is an approach to verification of finite-state systems which relies on iterating through all possible states and checking whether some condition holds at each state. One challenge with this approach is that in the majority of real-world systems, the number of states to traverse is too large to feasibly fully explore. In this thesis, we present MLVR (Multi-Layer Variable Regexp), a specification language designed for model checking against hardware system implementations. The syntax of MLVR is based on regular expressions, where we specify what traces of inputs and outputs from the system are acceptable. We offer support for variables to be remembered and later recalled, and we allow for treating the values of variables symbolically during model checking. This allows the state space of systems primarily dealing with variable input/output (for example, hardware buses) to be reduced enough that model checking is feasible for formal verification of the system. We provide a simplified language, SLVR (Single-Layer Variable Regexp), with some of the core features of MLVR and formal proofs of correctness for model checking with SLVR, implemented in the Coq proof assistant. The style and structure of the proofs about SLVR provide insight into how proofs of correctness of MLVR might be written, and they demonstrate solutions to some of the technical challenges raised in proving correctness of MLVR.

Thesis supervisor: Adam Chlipala
Title: Professor of Computer Science

Acknowledgments

I would like to thank my advisor, Adam Chlipala, for his guidance and mentorship, from the time that I joined the Programming Languages and Verification Lab as a UROP through to this MEng project. I would also like to thank Hailey Poole, Noah Evans, TJ Machado, and Randy Lober from Sandia National Labs for providing support and assistance on this project and my 6A internship. I thank the community at pika for providing me with countless memories, friendships, and dinners that made the community feel like a home. Last but not least, I would like to thank my family for their love and support and for giving me all of the opportunities that allowed me to be where I am today.

Contents

List of Figures

Chapter 1

Introduction

When designing hardware systems, it is common practice to write a collection of tests to ensure that the system behaves as the designer intended. However, this approach typically makes no formal guarantees about the correctness of the system; it merely increases one's confidence of the correctness of the design. In certain high-risk systems, such an informal assurance is not acceptable, and formal methods instead are preferred.

One of the most straightforward techniques for proving correctness of a system is model checking. This approach involves iterating through every possible state that a finite-state system could be in, inspecting that some correctness condition holds in each state. For small-state-space systems, model checking is typically directly feasible. Unfortunately, as soon as a system incorporates variables or memory storage, the state space general becomes too large to explore via naïve model checking: one would need to check that the correctness condition holds for every possible value of a variable, which would grow exponentially in its length. In order to perform model checking efficiently on such larger-state-space systems, it is necessary to explore entire sets of states at a time, rather than one at a time.[1]

This thesis explores the possibility of using symbolic variable values in order to make model checking possible on systems whose large state spaces are primarily a result of stored variable values. We first construct a regular expression-based simplified language, SLR, for writing specifications without allowing for variable inputs and outputs, to which we match traces of inputs and/or outputs. We use SLR to model-check a simple example problem. This demonstrates the viability of using regular expressions to describe systems.

Next, we extend this language into SLVR, a language which allows for inputs and outputs of variables of a fixed length. We define two different behaviors of this language: First, we define its behaviour given inputs of concrete Boolean values. Second, we define its behaviour given inputs that treat variables symbolically, which is to say that we no longer concern ourselves with the actual values of inputted variables, which by extension drastically reduces our state space. We provide proofs indicating a correspondence between the executions of the symbolic-variable version of the system and the original non-symbolic version.

Finally, we provide a language, MLVR, which is capable of describing systems with multiple input and output streams, variables of variable length being sent from one channel to another, and arbitrary function calls. We show that in order for a regular expression-based language to be this expressive yet still have enough formal guarantees to be useful for model checking, various constraints must be met. Although we do not provide a full proof

of correctness for this language, we discuss the technical challenges surrounding such proofs and show that many of the techniques used for proving correctness of SLVR are applicable to MLVR.

Chapter 2

Regular Expression Specification

In traditional model checking, some correctness criterion is specified as a logical formula and checked against a finite state graph corresponding to the system implementation. In contrast, MLVR directly specifies a finite state graph, which is then compared to the finite state graph of the system implementation. To understand the general behavior of MLVR, we look first at a simplified example language, SLR, over an alphabet Σ, which does not support variables. The syntax of this language is exactly the syntax of regular expressions[1]:

p : SLR ::=

$\mid \varnothing$	(empty set)
$\mid \varepsilon$	(empty string)
$\mid \_$	(any character)
$\mid a$	($a \in \Sigma$)
$\mid \neg p$	(logical not)
$\mid p_1 p_2$	(concatenation)
$\mid p_1 \| p_2$	(logical or)
$\mid p_1 \& p_2$	(logical and)
$\mid p^n$	(repeat p n times)
$\mid p^*$	(Kleene star)

The language Σ represents the type of inputs being passed to the system, for example, bits. For a full description of a system, we specify both a pattern p : SLR and the output we expect when the input trace matches the pattern.

As an example, if we wanted to define a system with bit inputs whose behavior is to output 1 whenever we receive a 1 followed by an even number of 0s, followed by another 1, we would describe this system with the pair, $(\_^*1(00)^*1, 1)$.

[1]Credit to Clement Pit-Claudel, Jason Chen, and Thomas Bourgeat for writing the first iteration of the Coq syntax of this language and implementing most of the functions defined below.

We define a function acc_ε which checks whether a given p : SLR accepts the empty string:

$$\begin{aligned}
\mathrm{acc}_\varepsilon \varnothing &= \bot \\
\mathrm{acc}_\varepsilon \varepsilon &= \top \\
\mathrm{acc}_\varepsilon _ &= \bot \\
\mathrm{acc}_\varepsilon a &= \bot \\
\mathrm{acc}_\varepsilon \neg p &= \neg \mathrm{acc}_\varepsilon p \\
\mathrm{acc}_\varepsilon (p_1 \| p_2) &= (\mathrm{acc}_\varepsilon p_1) \vee (\mathrm{acc}_\varepsilon p_2) \\
\mathrm{acc}_\varepsilon (p_1 \& p_2) &= (\mathrm{acc}_\varepsilon p_1) \wedge (\mathrm{acc}_\varepsilon p_2) \\
\mathrm{acc}_\varepsilon (p_1 p_2) &= (\mathrm{acc}_\varepsilon p_1) \wedge (\mathrm{acc}_\varepsilon p_2) \\
\mathrm{acc}_\varepsilon (p^0) &= \top \\
\mathrm{acc}_\varepsilon (p^n) &= \mathrm{acc}_\varepsilon p \quad (n \neq 0) \\
\mathrm{acc}_\varepsilon (p^*) &= \top
\end{aligned}$$

Now, we define a transition function between states of SLR under inputs a as the derivative function[2]:

$$\begin{aligned}
\partial_a \varnothing &= \varnothing \\
\partial_a \varepsilon &= \varnothing \\
\partial_a _ &= \varepsilon \\
\partial_a a &= \varepsilon \\
\partial_a b &= \varnothing \quad (b \neq a) \\
\partial_a \neg p &= \neg \partial_a p \\
\partial_a (p_1 \| p_2) &= (\partial_a p_1) \| (\partial_a p_2) \\
\partial_a (p_1 \& p_2) &= (\partial_a p_1) \& (\partial_a p_2) \\
\partial_a (p_1 p_2) &= (\partial_a p_1) p_2 \| \partial_a p_2 \quad (\mathrm{acc}_\varepsilon p_1) \\
\partial_a (p_1 p_2) &= (\partial_a p_1) p_2 \quad (\neg \mathrm{acc}_\varepsilon p_1) \\
\partial_a p^0 &= \varnothing \\
\partial_a p^{n+1} &= (\partial_a p) p^n \\
\partial_a p^* &= (\partial_a p) p^*
\end{aligned}$$

We define the derivative function for lists of inputs. For $a \in \Sigma$ and $l \in \Sigma^*$,

$$\partial_{[]} p = p$$

$$\partial_{a::l} p = \partial_l \partial_a p$$

Finally, we define an acceptance function:

$$\mathrm{acc}_l p = \mathrm{acc}_\varepsilon (\partial_l p)$$

In order to perform model checking between a system implementation state $impl_0$ and a specification written in this language $spec_0$, we can exhaustively search the space of pairs

of states ($impl$, $spec$), corresponding to the implementation state $impl_0$ after taking in a certain string of inputs $s \in \Sigma^*$ and a specification state $spec = \partial_s(spec_0)$. For every such distinct pair, we check that the output (or lack thereof) is equal between $spec$ and $impl$.

One last consideration in this language is that there is no guarantee that the set of states reachable by taking derivatives of a given start state is finite. Because there is an equivalence between finite state automata and regular expressions[3], we know that the set of semantically distinct reachable states will be finite, but there may be multiple instances of SLR which are semantically identical. As a trivial example, if we start with $(\_^*)$ and take successive derivatives, we get:

$$(\_^*) \to (\varepsilon\_^*) \to ((\varnothing\_^*) \| (\varepsilon\_^*)) \to ((\varnothing\_^*) \| (\varnothing\_^*) \| (\varepsilon\_^*))$$

All of these are clearly synonymous with $(\_^*)$. In order to prevent this syntactical explosion, we define a reduction function R:

$$\begin{aligned}
R((p_1 \& p_2) \& p_3) &= p_1 \& (p_2 \& p_3) \\
R((p_1 \| p_2) \| p_3) &= p_1 \| (p_2 \| p_3) \\
R(\varnothing \| p) &= p \\
R(p \| \varnothing) &= p \\
R(\varnothing \& p) &= \varnothing \\
R(p \& \varnothing) &= \varnothing \\
R(\varnothing p) &= \varnothing \\
R(p \varnothing) &= \varnothing \\
R(\varepsilon p) &= R(p) \\
R(p \varepsilon) &= R(p) \\
R(p_1 p_2) &= R(p_1) R(p_2)
\end{aligned}$$

We also prove the following theorem stating that this reduction function does not change the semantics of a given pattern:

Theorem reduce_ok : $\forall\ p\ s.\ \mathrm{acc}_s\ p = \mathrm{acc}_s R(p)$.

Every time that we call ∂ during model checking, we apply R (perhaps more than once) on the output, which allows us to keep the state space finite more often. Note that we have no guarantee that we will end up with a finite state space. If the reduction function is not powerful enough to result in a finite state space for some particular example, model checking will simply fail. In practice, this method has been successful in keeping state spaces finite.

Chapter 3

The MANatEE Example

The MANatEE example was provided by Sandia National Labs and is a simple toy problem to test out model-checking approaches. It consists of two buttons, each of which can be pressed or unpressed. There are two layers to the system: first, the button interface reads whether each button is pressed or unpressed at each clock cycle and outputs when a button press is finished. Then, this output is read by the lock section of the system, which checks for a specific "passcode" sequence of button presses and "pulses" when this passcode is received. The system state chart is shown in Fig. 3.1[1].

Although it is possible to write a single regular expression to describe what series of inputs triggers a pulse, this approach is not ideal in practice since the expression would be syntactically large and unintuitive. Instead, we can use one SLR pattern for each type of output we expect from the button interface and then pipe the outputs from these patterns into the input of a second-layer pattern representing the operation of the lock section. Now, instead of a single pattern representing a system, we have a tuple of patterns, each representing a portion of the system.

With this approach, we end up with a readable specification for each section of the system[2]. In the following specification, $\{C, I, G, R\}$ is the alphabet of inputs to the first layer, where C represents both buttons pressed at the same time, R and G represent just the red or green buttons pressed, and I represents neither button pressed. We then get outputs e_{reset},e_{red}, e_{green} and e_{clear}, representing which buttons actions we interpret to have occurred. If none of the patterns are matched at a given time step, then we represent the output of the first layer as e_{none}.

[1] State chart provided by Sandia National Labs.

[2] Credit to Clement Pit-Claudel, Jason Chen, and Thomas Bourgeat for writing the following regular expressions for this example.

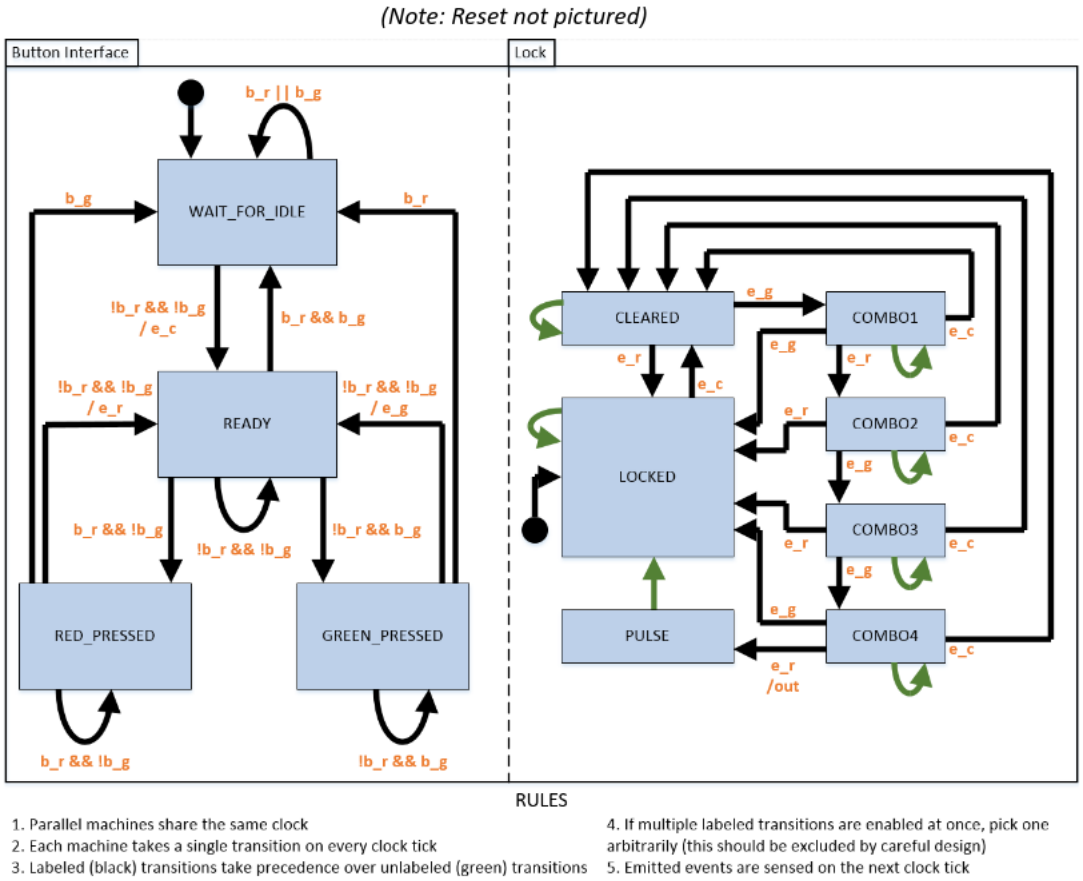

Figure 3.1: Stateflow chart of the MANatEE system.

$$
\begin{aligned}
waveform\_to\_button := [& \\
& (\_^{*}C, e_{reset}); \\
& (\_^{*}IRR^{*}I, e_{red}); \\
& (\_^{*}IGG^{*}I, e_{green}); \\
& (\_^{*}I(\neg I)(\neg I)^{*}I, e_{clear}); \\
& ((\neg I)^{*}I, e_{clear})].
\end{aligned}
$$

button_to_output := $[(.^{*}e_{clear}e^{*}_{none}e_{green}e^{*}_{none}e_{red}e^{*}_{none}e_{green}e^{*}_{none}e_{green}e^{*}_{none}e_{red}(\neg e_{clear}), \text{pulse})]$.

We can then write an implementation of this system in the hardware design language Kôika[4] and prove that the specification and the implementation are equivalent using model checking.

Chapter 4

Symbolic Variables

One construct that SLR is not able to handle well is the notion of variables. Suppose we have a system that is intended to take in as input some string of inputs and then send that same string back as output. Depending on the length of the input string, we might be able to write a SLR pattern describing this system, but it would require listing all possible values of input strings, which is exponential in the length of input. We would like to add support for describing and model-checking such systems using variables instead.

In extending the regular-expression language to allow for expressing variables as inputs and outputs, we cannot simply add a constructor that matches and stores input strings as variables to be recalled later as outputs when matched to a pattern. To see why such a constructor leads to issues, suppose we naïvely keep the same definition of SLR as before and add one additional constructor:

$$\mid V(x, l) \quad \text{(remember the next } l \text{ input characters as variable } x\text{)}$$

We immediately run into issues with examples such as the following:

$$V(x, 2) \| (\_V(x, 1))$$

Should the variable x that we remember be both of the next two characters or just the second one? This and various other forms of ambiguity force unavoidable constraints on our specification language. Some other examples of specifications that would generally not be expressible with most methods of implementing variables include:

- Wait for an input string to repeat itself after n characters, then pulse.
- Receive as input an n-character string encoding a number l, then receive as input a variable v of length l, then output v.
- Receive as input an n-character variable v, then start outputting v, but stop outputting partway if receiving any input characters of value 1.

The approach that MLVR takes is to write regular expressions that match not only input values but also output values, and give a specified scope within which that variable value is live, which prevents variable-value ambiguity. In this formulation, a user would describe

an entire system's inputs and outputs as one large expression. The model checker would then take derivatives of it based on whatever output the implementation returns for a given input, and check whether the new state is still acceptable (for a definition of acceptable which will be described later). We note that this means our specification language no longer guarantees that for every possible set of inputs, there will be an acceptable output, nor that an accepting output value is unique. This gives users the flexibility to not fully constrain their specifications, but they must take care to describe the expected output behavior for all input edge cases.

Since the inputs to the specification now correspond to both inputs and outputs of the system, we can no longer treat the inputs as $s \in \Sigma$. Instead, we accept inputs $s \in \Sigma^n$, where it is assumed that the number of input and output channels remains constant and that their input/output alphabets are the same. In order to express patterns of multiple streams of characters, we create a two-layered definition for our language (along with a definition of lengths of variables, allowing for variable-length variables[1]):

l : Len ::=
| n (constant length $n > 0$)
| x_i (value of ith stream of variable of name "x")

p : FLVR ::=
| $\varnothing$ (empty set)
| ε (empty string)
| $\_$ (any character)
| a ($a \in \Sigma$)
| $p_1 p_2$ (concatenation)
| $p_1 \| p_2$ (logical or)
| p^n (repeat p n times)
| p^* (Kleene star)
| x_i (recall value of ith stream of variable of name "x")

p : MLVR ::=
| $\varnothing$ (empty set)
| ε (empty string)
| $\_$ (any character)
| $p_1 p_2$ (concatenation)
| $p_1 \| p_2$ (logical or)
| p^n (repeat p n times)
| p^* (Kleene star)
| let $x : l = p_1$ in p_2 (introduce variable of name "x" of length l : Len in p_2, as long as its value matches p_1)
| $[p_1 | p_2 | ... | p_n]$ (match first stream against p_1 : FLVR, second stream with p_2 : FLVR, etc.)

[1] We currently restrict variables to strictly positive lengths due to some quirks in dealing with zero-length variables, but it would be possible to add support for zero-length variables.

In writing a specification, a user must also provide a function $a2l : \Sigma^* \to \mathbb{Z}$. This is used to determine what the length should be of a variable-length variable. As an example, a system with bitstring inputs and outputs might convert a string of bits into binary and then take the value of that binary string.

The derivative functions for FLVR and MLVR are generally quite similar to the derivative function for SLR, plus additional handling of variables.

In the following equalities, the notation $p/(x \leftarrow a)$ means to append the value a to any reference to x in p. More specifically, given a_i the ith element of a, we substitute x_i with $a_i x_i$. Similarly, the notation $p/(x \leftarrow \varepsilon)$ represents the termination of x, meaning that we substitute x_i with ε and substitute (let $y : a_1 a_2 ... a_k x_i = p_1$ in p_2) with (let $y : (a2l(a_1 a_2 ... a_k)) = p_1$ in p_2).

$$
\begin{aligned}
\partial_a x_i &= \varnothing \\
\partial_a(\text{let } x : 1 = p_1 \text{ in } p_2) &= p_2/(x \leftarrow a)/(x \leftarrow \varepsilon) && (\text{acc}_a p_1) \\
\partial_a(\text{let } x : 1 = p_1 \text{ in } p_2) &= \varnothing && (\neg \text{acc}_a p_1) \\
\partial_a(\text{let } x : l + 1 = p_1 \text{ in } p_2) &= \text{let } x : l = \partial_a p_1 \text{ in } p_2/(x \leftarrow a) \\
\partial_a(\text{let } x : y_i = p_1) \text{ in } p_2) &= \varnothing \\
\partial_a[p_1|...|p_n] &= [\partial_a p_1|...|\partial_a p_n]
\end{aligned}
$$

As an example, suppose we start with the following pattern.

$$\text{let } x : 2 = \_^* \text{ in } [y_1|x_0]$$

If we take the derivatives sequentially for inputs (0,1) followed by (5,7), we would obtain the following patterns:

$$
\begin{aligned}
\partial_{(0,1)} \ (\text{let } x : 2 = \_^* \text{ in } [y_1|x_0]) &= \text{let } x : 1 = \_^* \text{ in } [y_1|0x_0] \\
\partial_{(5,7)} \ (\text{let } x : 1 = \_^* \text{ in } [y_1|0x_0]) &= [y_1|05]
\end{aligned}
$$

We provide a similar acceptance function as the one from SLR, starting with the function that checks for acceptance of the empty string and taking consecutive derivatives to check if a pattern accepts a given nonempty input.

For model checking, our approach of matching both inputs and outputs against a pattern presents a new challenge. In order to catch an error in model checking, we need an error condition for when a given pair of inputs and outputs does not match the specification. Since we treat inputs as a potentially infinite stream, we must check whether there exists any future input which still matches the current pattern or, equivalently, whether the current pattern is semantically equivalent to $\varnothing$. The definition of a function to perform such a check is nontrivial, and in fact it is the reason for dropping the logical not ($\neg$) and logical and (&) constructors which were present in the original regular-expression language.

We first define a helper function L^F over patterns of FLVR which returns all of the lengths of inputs accepted by the pattern as a list of values $(a, b) : \mathbb{N} \times \mathbb{N}$, each representing that for all $n \in \mathbb{N}$, there exists an input of length $a + bn$ that is accepted by the pattern. We also define a similar function L^M over patterns of MLVR. We supply L with an additional argument which is a map m of lengths of fixed-length variables representing a pattern's context.

$$
\begin{aligned}
L^F_m \varnothing &= [] \\
L^F_m \varepsilon &= [(0,0)] \\
L^F_m \_ &= [(1,0)] \\
L^F_m a &= [(1,0)] \\
L^F_m (p_1 \| p_2) &= L^F_m p_1 \cup L^F_m p_2 \\
L^F_m (p_1 p_2) &= L^F_m p_1 \overset{\oplus}{\times} L^F_m p_2 \\
L^F_m p^0 &= [(0,0)] \\
L^F_m p^{n+1} &= L^F_m p \overset{\oplus}{\times} L^F_m p^n \\
L^F_m p^* &= [(0,0)] \cap \prod_{(a,b) \in L^F_m p}^{\oplus} \left([(0,a)] \cup \bigcup_{1 \le k \le b} [(ak, b)] \right) \\
L^F_m x_i &= [(m(x), 0)] && (x \in m) \\
L^F_m x_i &= [] && (x \notin m)
\end{aligned}
$$

$$
\begin{aligned}
L^M_m \varnothing &= [] \\
L^M_m \varepsilon &= [(0,0)] \\
L^M_m \_ &= [(1,0)] \\
L^M_m (p_1 \| p_2) &= L^M_m p_1 \cup L^M_m p_2 \\
L^M_m (p_1 p_2) &= L^M_m p_1 \overset{\oplus}{\times} L^M_m p_2 \\
L^M_m p^0 &= [(0,0)] \\
L^M_m p^{n+1} &= L^M_m p \overset{\oplus}{\times} L^M_m p^n \\
L^M_m p^* &= [(0,0)] \cap \overset{\oplus}{\prod_{(a,b) \in L^M_m p}} \left([(0,a)] \cup \bigcup_{1 \le k \le b} [(ak, b)] \right) \\
L^M_m \ (\text{let } x_i : n = p_1 \text{ in } p_2) &= [(n,0)] \overset{\oplus}{\times} L^M_{m \leftarrow (x:n)} p_2 \qquad (\exists (a,b) \in L^M_m p_1, \exists k \in \mathbb{N}, a + bk = n) \\
L^M_m \ (\text{let } x_i : n = p_1 \text{ in } p_2) &= [] \qquad (\forall (a,b) \in L^M_m p_1, \forall k \in \mathbb{N}, a + bk \neq n) \\
L^M_m \ (\text{let } x_i : y_j = p_1 \text{ in } p_2) &= [] \\
L^M_m [p_1 | ... | p_n] &= \textstyle\prod_{k=1}^{n} L^M_m p_k
\end{aligned}
$$

Various operators used above are defined here[2]:

[2]Credit to TJ Machado for his implementation and proof of correctness of the $\sqcap$ function.

$$
\begin{aligned}
(a_1, 0) \oplus (a_2, b_2) &= [(a_1 + a_2, b_2)] \\
(a_1, b_1) \oplus (a_2, b_2) &= \bigcup_{k=0}^{b_1 - 1} [(a_1 + a_2 + k b_2, b_1)] && (b_1 \neq 0)
\end{aligned}
$$

$$
l_1 \overset{\oplus}{\times} l_2 = \bigcup_{x \in l_1, y \in l_2} x \oplus y
$$

$$
\begin{aligned}
(a, b_1) \sqcap (a, b_2) &= [(a, lcm(b_1, b_2))] \\
(a_1, b_1) \sqcap (a_2, b_2) &= [] && (gcd(b_1, b_2) \nmid (a_1 - a_2)) \\
(a_1, 0) \sqcap (a_2, b_2) &= [] && (a_2 > a_1) \\
(a_1, b_1) \sqcap (a_2, 0) &= [] && (a_1 > a_2) \\
(a_1, b_1) \sqcap (a_2, b_2) &= [(a_1 + \frac{(a_2 - a_1)(extgcd(b_1, b_2)[2])b_1}{gcd(b_1, b_2)} \bmod lcm(b_1, b_2), \\
&\qquad lcm(b_1, b_2))] && (a_2 > a_1) \\
(a_1, b_1) \sqcap (a_2, b_2) &= [(a_2 + \frac{(a_1 - a_2)(extgcd(b_1, b_2)[1])b_2}{gcd(b_1, b_2)} \bmod lcm(b_1, b_2), \\
&\qquad lcm(b_1, b_2))] && (a_1 > a_2)
\end{aligned}
$$

$$
\begin{aligned}
\textstyle\bigsqcap_{x_1, x_2, \ldots} f &= f(x_1) \sqcap f(x_2) \sqcap \ldots \\
\overset{\oplus}{\prod_{x_1, x_2, \ldots}} f &= f(x_1) \overset{\oplus}{\times} f(x_2) \overset{\oplus}{\times} \ldots
\end{aligned}
$$

Finally, we can check that there exists some string which a given p accepts by checking that $L^M_{[]} p$ is nonempty. Unfortunately, since variables can be of variable length dependent on the function `a2l` that a user supplies, we cannot guarantee that this function finds all possible acceptable lengths, as we may not yet know the exact length of any given variable. In practice, this means the specification language can only model-check systems that only have fixed-length variables, since variable-length variables will always be determined not to accept any inputs. We discuss potential solutions to this problem in section 8.1.

Chapter 5

The TINCAN Example

The TINCAN Bus example is a second example problem provided by Sandia National Labs. It describes a shared bus protocol which should be implemented by a transmitter. The protocol has various sequential steps, including arbitration with the other users of the bus, sending the destination address, sending the message length, performing a cyclic redundancy check (CRC), and receiving an acknowledgement (ACK) and a flow control transmission from the receiver. If all of these steps go correctly, only then may the transmitter transmit their message (along with another CRC and another ACK). A diagram of the phases is shown in Fig. 5.1[1].

This example demonstrates the utility of providing support for variable introduction and recall in writing regular expression-based specification. Using this feature, we can keep track of the current message we want to send, and try to send it multiple times if the first attempt fails. However, we notice a few places where MLVR is not quite expressive enough to fully describe the intended behavior of TINCAN. In particular, the CRC is a more complicated function of various inputs and is not suited to be expressed in regular expression syntax. One possible addition to the MLVR syntax would be to add a constructor for specific user-defined Gallina functions such as the CRC function. The user would also have to provide some proof that the functions behave identically to a given component of the implementation system, although the details of how this would be done have not yet been fully explored.

If we omit the CRC section, we can start to write a specification for the rest of TINCAN. From a transmitter's perspective, it reads two input bitstreams: a stream of messages that it ought to send and a stream of readouts from the bus. It also has one output bitstream corresponding to its output to the bus. Therefore, when describing this system, we should write a MLVR expression that accepts tuples of three bits as inputs. Let us arbitrarily define the 0th bit as the stream of messages, the 1st bit to be readouts from the bus, and the 2nd bit to be the output to the bus. Let us also define $a2l(b = b_1 b_2 ...) = 8 \cdot \overline{b_1 b_2 ...}_2$, that is, in order to get a variable length from a string of bits, we read those bits as a binary number and multiply by eight. Our first attempt might be something like the following (using $\langle addr \rangle$

[1] TINCAN Bus Protocol and diagram provided by Sandia National Labs.

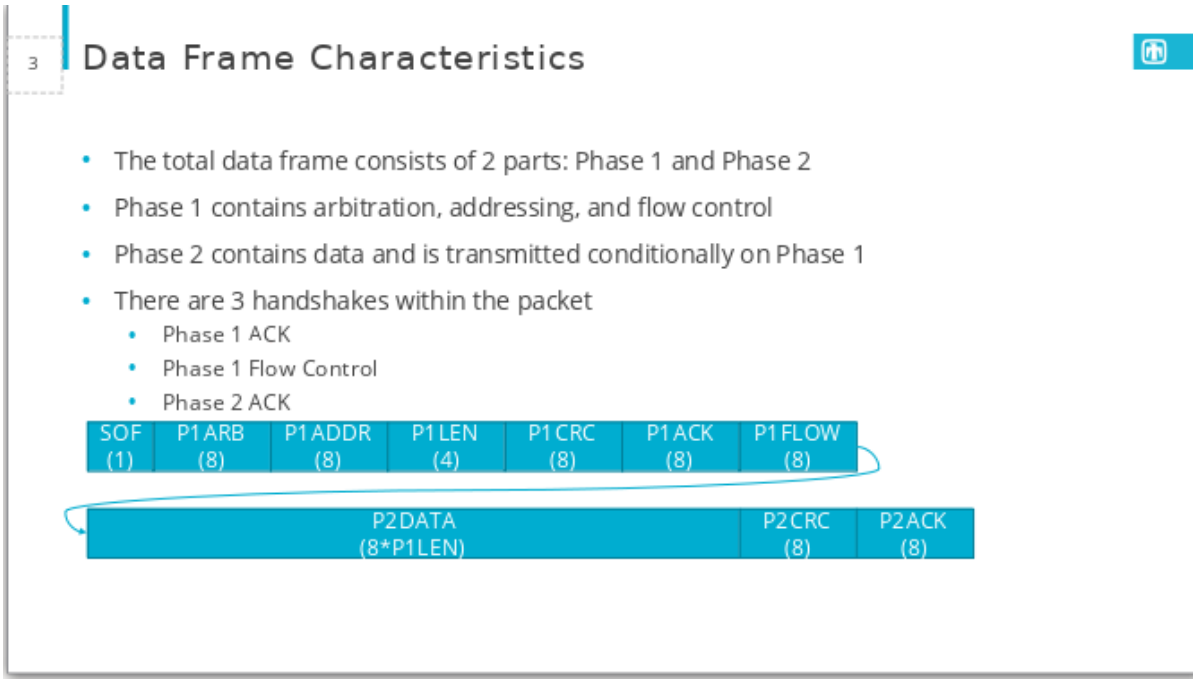

Figure 5.1: TINCAN Bus Protocol Diagram.

as a placeholder for our personal address):

$([0\|\_\|0]^*$	(wait for message)
$[1\|\_\|\_]$	(message input begins)
let $a : 8 = [\_\|\_\|0]^*$ in	(receive message destination address)
let $l : 4 = [\_\|\_\|0]^*$ in	(receive message length)
let $m : l = [\_\|\_\|0]^*$ in	(receive message content)
$[\_\|0\|0]^{200}$	(wait until bus is quiet)
$[\_\|\_\|1]$	(start of frame)
$[\_^8\|\langle addr\rangle\|\langle addr\rangle]$	(arbitration)
$[\_^8\|\_^8\|a_0]$	(transmit address)
$[\_^4\|\_^4\|l_0]$	(transmit message length)
$[\_\_\|00\|00]$	(wait for ACK - 3 lines)
$[\_^4\|(11\_\_\|\|\_11\_\|\|\_\_11)\|0^4]$	
$[\_\_\|00\|00]$	
$[\_\_\|00\|00]$	(wait for FLOW - 3 lines)
$[\_^4\|(11\_\_\|\|\_11\_\|\|\_\_11)\|0^4]$	
$[\_\_\|00\|00]$	
$[\_^*\|\_^*\|m]$	(transmit message)
$[\_\_\|00\|00]$	(wait for ACK - 3 lines)
$[\_^4\|(11\_\_\|\|\_11\_\|\|\_\_11)\|0^4]$	
$[\_\_\|00\|00]$	
$)^*$	

The issue with this specification is that it is not complete. We have only written out the expected behavior if all of the arbitrations, acknowledgements, and other signalling go exactly as intended. If anything doesn't go as expected, we would need to stop sending bits to the bus immediately and wait to try again later. Unfortunately for us, MLVR does not give us the flexibility to express this concisely and legibly. We could try to implement an if/else-like statement in MLVR in the following way. Suppose we want to write (if p_c then p_t else p_f). We could do something like $(p_c p_t \| (\neg p_c) p_f)$, except that we are no longer able to use the $\neg$ (logical not) constructor in MLVR[2]. As such, we would need to write out the negation of all branching subexpressions by hand, which is impractical and would result in large and unreadable pattterns.

[2]We cannot re-add a negation constructor to MLVR because we would be unable to decide whether strings of a particular length would be accepted by patterns using the negation constructor, which would mean that we would not be able to tell whether they are valid or not.

Chapter 6

Single-Stream Language

Due to the difficulties and unresolved issues in MLVR as described in the previous chapters, we provide SLVR, a toy version of MLVR, which only adds support for basic fixed-length variables and a single stream of Boolean input. We use SLVR to demonstrate what the core proofs of correctness might look like for a more expressive language such as MLVR and how such a language might be used for model checking. First, we define the language as:

p : SLVR ::=

$\mid \varnothing$	(empty set)
$\mid \epsilon$	(empty string)
$\mid \_$	(any character)
$\mid a$	($a \in \Sigma$)
$\mid p_1 p_2$	(concatenation)
$\mid p_1 \| p_2$	(logical or)
$\mid p^*$	(Kleene star)
$\mid$ let $x : l$ in p	(introduce variable of name x and length $l \in \mathbb{Z}^+$ in scope p)
$\mid x$	(recall value of variable name x)
$\mid \text{Sym}(i)$	(recall value of variable introduced i variables ago).

Of note, we have an additional constructor Sym which represents a symbolic variable value which will only be used in symbolic model checking. It should not be used when writing an actual specification. As such, we also specify a well-formedness condition for SLVR patterns, checking that they do not contain any Sym constructors or unintroduced variable names.

We define a derivative function ∂ and an accept function acc in a similar fashion as in the previous languages. Next, we also define an alternative symbolic derivative function δ, which instead of taking just concrete Boolean values as inputs, takes a variant which can either take a concrete Boolean value or the symbolic introduction of a variable of a particular length or the repetition of a previously introduced symbolic variable value (referenced in de-Bruijn index style) (with an option to express that the value being inputted does not actually match the original value of the variable):

s : Inp :=
| b (Boolean bit, where $b \in \{0, 1\}$)
| v_l (newly introduced variable of length l)
| r_i (recall the ith most recently introduced variable)
| $\bar{\mathrm{r}}_i$ (incorrectly recall the ith most recently introduced variable)

In the following definitions, δ_a is treated as the symbolic derivative with respect to any $a \in \mathrm{Inp}$, δ_b is treated as the symbolic derivative with respect to a Boolean bit b, δ_{v_l} is the symbolic derivative with respect to a new variable, and δ_{r_i} is the symbolic derivative with respect to a recalled variable. Any symbolic derivative not specifically listed below should be interpreted as $\varnothing$.

$$
\begin{aligned}
\delta_a \varnothing &= \varnothing \\
\delta_a \varepsilon &= \varnothing \\
\delta_b \_ &= \varepsilon \\
\delta_b b &= \varepsilon \\
\delta_b b' &= \varnothing \quad (b \neq b') \\
\delta_a (p_1 \| p_2) &= (\delta_a p_1) \| (\delta_a p_2) \\
\delta_a (p_1 p_2) &= ((\delta_a p_1) p_2 \| \partial_a p_2) \quad (\mathrm{acc}_\varepsilon p_1) \\
\delta_a (p_1 p_2) &= (\delta_a p_1) p_2 \quad (\neg \mathrm{acc}_\varepsilon p_1) \\
\delta_a p^0 &= \varnothing \\
\delta_a p^{n+1} &= (\delta_a p) p^n \\
\delta_a p^* &= (\delta_a p) p^* \\
\delta_{\mathrm{v}_l} \text{let } x : l \text{ in } p &= p/x \\
\delta_{\mathrm{v}_l} \text{let } x : l' \text{ in } p &= \varnothing \quad (l \neq l') \\
\delta_{\mathrm{r}_i} \mathrm{Sym}(i) &= \varepsilon \\
\delta_{\mathrm{r}_i} \mathrm{Sym}(j) &= \varnothing \quad (i \neq j)
\end{aligned}
$$

Here, p/x means that we substitute all occurrences of x inside p with $\mathrm{Sym}(0)$, and all occurrences of $\mathrm{Sym}(k)$ with $\mathrm{Sym}(k+1)$.

Next, we define an accepts function that behaves identically to the accepts functions in previous examples.

Lastly, we provide a definition of equivalence between concrete and symbolic input strings. To do this, we end up needing to express symbolic inputs in a tree-like data structure to represent different contexts in which variables are live:

t : sym_tree ::=
| Node(t_1, t_2)
| Leaf$(s : Inp, t)$
| Empty

We provide a 'flatten' function that flattens instances of sym_tree into lists:

$$\begin{aligned}
\text{flatten}(\text{Node}(t_1, t_2)) &= \text{flatten}(t_1)^\frown flatten(\text{t}_2) \\
\text{flatten}(\text{Leaf}(s, t)) &= \text{s}^\frown flatten(\text{t}) \\
\text{flatten}(\text{Empty}) &= []
\end{aligned}$$

Then we define a helper function *substr* that takes in an optional list of concrete Boolean values, a sym_tree, and a list of introduced variables, and outputs the optional substring of the concrete values that would be left if we matched the symbolic tree to the beginning of the concrete list, under the listed introduced-variable context, or $\emptyset$ if the symbolic tree does not match any initial substring of the concrete input:

$$\begin{aligned}
substr(\emptyset, t, v) &= \emptyset \\
substr(c, [], v) &= c \\
substr(c, Node(t_1, t_2), v) &= substr(substr(c, t_1, v), t_2, v) \\
substr(b^\frown c, Leaf(b, t), v) &= substr(c, t, v) \\
substr(b^\frown c, Leaf(\neg b, t), v) &= \emptyset \\
substr(b_1^\frown b_2^\frown ... b_l^\frown c, Leaf(v_l, t_2), v) &= substr(c, t_2, (b_1^\frown ...^\frown b_l)^\frown v) \\
substr(c, Leaf(v_l, t_2), v) &= \emptyset && ((\text{length } c) < l) \\
substr(c, Leaf(\overline{v}_l, t_2), v) &= \emptyset && ((\text{length } c) < l) \\
substr(v[k]^\frown c, Leaf(r_k, t_2), v) &= substr(c, t_2, v) \\
substr(c, Leaf(r_k, t_2), v) &= \emptyset && (\text{c doesn't start with } v[k]) \\
substr(v[k]^\frown c, Leaf(\overline{r}_k, t_2), v) &= \emptyset && (\text{length } v[k] = n) \\
substr(b_1^\frown ...^\frown b_n^\frown c, Leaf(\overline{r}_k, t_2), v) &= substr(c, t_2, v) && (\text{length } v[k] = n)
\end{aligned}$$

With this, we can finally express concrete/symbolic input equivalence:

$$substr(c, t, []) = [] \longleftrightarrow t \sim c$$

With these definitions of concrete derivatives and acceptance and symbolic derivatives and acceptance, we provide proofs of two theorems relating the definitions. The first theorem states that for all well-formed patterns p, if p accepts concrete input *conc*, then there exists an equivalent symbolic input *sym* which is accepted by pattern p. An approximate theorem statement is as follows:

Theorem sym_ok_wrong : $\forall$ *conc* p. $\text{acc}_{conc} p \rightarrow \exists$ *sym*. $(sym \sim conc \wedge \text{acc}_{sym} p)$.

The second theorem states that for all well-formed patterns p, if p accepts symbolic input *sym*, then p accepts every concrete input *conc* equivalent to *sym*. An approximate theorem statement is as follows:

Theorem conc_ok_wrong : $\forall$ *sym* p. $\text{acc}_{sym} p \rightarrow (\forall$ *conc*. $sym \sim conc \rightarrow \text{acc}_{conc} p)$.

However, due to the differences in types between sym_tree and raw lists of symbolic inputs, we instead need to loosen the statement of sym_ok and conc_ok to:

Theorem sym_ok : $\forall\ conc\ p.\ \text{acc}_{conc}p \rightarrow \exists\ sym\_t.\ (sym\_t \sim conc \wedge \text{acc}_{(\text{flatten}\ sym\_t)}p)$.

Theorem conc_ok : $\forall\ sym\ p.\ \ \text{acc}_{sym}p \rightarrow \exists sym\_t.\ (\text{flatten}\ sym\_t = sym) \wedge (\forall\ conc.\ sym\_t \sim conc \rightarrow \text{acc}_{conc}p)$.

Now, we would like to be able to prove that that for all other pairs of related concrete and symbolic systems such that these same two theorems hold, equivalence between symbolic systems implies equivalence between concrete systems. More precisely, suppose we are given an arbitrary system Sys and acceptance functions (sys_acc : Sys$\rightarrow \{0,1\}^* \rightarrow \{\top, \bot\}$) and (sys_sym_acc : Sys$\rightarrow \text{Inp}^* \rightarrow \{\top, \bot\}$). Suppose also that we prove the following two theorems (identical to the ones we proved about SLVR):

Theorem sys_sym_ok : $\forall\ conc\ p.\ \text{sys\_acc}_{conc}p \rightarrow \exists\ sym\_t.\ (sym\_t \sim conc \wedge \text{sys\_acc}_{(\text{flatten}\ sym\_t)}p)$.

Theorem sys_conc_ok : $\forall\ sym\ p.\ \ \text{sys\_acc}_{sym}p \rightarrow \exists sym\_t.\ (\text{flatten}\ sym\_t = sym) \wedge (\forall\ conc.\ sym\_t \sim conc \rightarrow \text{sys\_acc}_{conc}p)$.

Then we would like to prove that the following theorem holds:

Theorem model_check_ok : $\forall\ sys\ p.\ (\forall\ sym.\ \text{sym\_acc}_{sym}p \leftrightarrow \text{sys\_sym\_acc}_{sym}sys) \rightarrow (\forall\ conc.\ \text{acc}_{conc}p \leftrightarrow \text{sys\_acc}_{conc}sys)$.

We run into issues proving this statement, however, because we can have multiple different sym_trees which flatten to the same list of symbolic inputs, but which have different meanings. This prevents us from being able to make generalizable statements about acceptance definitions. As an example, consider the following two patterns.

$$\text{let } x : 1 \text{ in } (\text{let } y : 2 \text{ in } xy)$$

$$\text{let } x : 1 \text{ in } (\text{let } y : 2 \text{ in } x)x$$

Both patterns would accept the following symbolic input list:

$$[\text{v}_1;\ \text{v}_2;\ \text{r}_1;\ \text{r}_0]$$

The interpretation of this list is different in each of the two patterns, though. In the first pattern, the r_0 input refers to the variable introduced by v_2, whereas in the second, the r_0 input refers to the variable introduced by v_1. We discuss this issue and possible solutions in section 8.4.

Chapter 7

Single-Stream Proof Sketch

In order to make it easier to work with definitions of acceptance in proofs, we first define an equivalent inductive definition of the accept function, for both the concrete and symbolic definitions of acceptance. Here, we keep the original Coq code used in the source code to make more clear how we have structured the inductive nature of these definitions.

```
Inductive accepts_ind : pattern → list bool → Prop :=
| accepts_Epsilon : accepts_ind Epsilon []
| accepts_Atom : ∀ a, accepts_ind (Atom a) [a]
| accepts_Any : ∀ a, accepts_ind Any [a]
| accepts_Seq : ∀ p1 p2 b1 b2,
                        accepts_ind p1 b1 →
                        accepts_ind p2 b2 →
                        accepts_ind (Seq p1 p2) (app b1 b2)
| accepts_Or_l : ∀ p1 p2 b, accepts_ind p1 b → accepts_ind (Or p1 p2) b
| accepts_Or_r : ∀ p1 p2 b, accepts_ind p2 b → accepts_ind (Or p1 p2) b
| accepts_Repeat_0 : ∀ p, accepts_ind (Repeat p) []
| accepts_Repeat_S : ∀ p b bs,
                        accepts_ind p b →
                        accepts_ind (Repeat p) bs →
                        accepts_ind (Repeat p) (app b bs)
| accepts_IntroVar : ∀ name len p v b,
                        length v = S len →
                        accepts_ind (subst_full_var p name v) b →
                        accepts_ind (IntroVar name len p) (app v b)

Inductive sym_accepts_ind : pattern → list sym_input → Prop :=
| sym_accepts_Epsilon : sym_accepts_ind Epsilon []
| sym_accepts_Atom : ∀ a, sym_accepts_ind (Atom a) [Bool a]
| sym_accepts_Any : ∀ a, sym_accepts_ind Any [Bool a]
| sym_accepts_Seq : ∀ p1 p2 b1 b2,
                        sym_accepts_ind p1 b1 →
                        sym_accepts_ind p2 b2 →
                        sym_accepts_ind (Seq p1 p2) (app b1 b2)
| sym_accepts_Or_l : ∀ p1 p2 b, sym_accepts_ind p1 b → sym_accepts_ind (Or p1 p2) b
```

```
| sym_accepts_Or_r : ∀ p1 p2 b, sym_accepts_ind p2 b → sym_accepts_ind (Or p1 p2) b
| sym_accepts_Repeat_0 : ∀ p, sym_accepts_ind (Repeat p) []
| sym_accepts_Repeat_S : ∀ p b bs,
                         sym_accepts_ind p b →
                         sym_accepts_ind (Repeat p) bs →
                         sym_accepts_ind (Repeat p) (app b bs)
| sym_accepts_IntroVar : ∀ name len p str,
                         sym_accepts_ind (intro_symvar p name true) str →
                         sym_accepts_ind (IntroVar name len p) (NewVar len :: str)
| sym_accepts_SymVar : ∀ index, sym_accepts_ind (SymVar index) [RepeatVar index true]
```

We also provide a proof of equivalence between these definitions and the derivative-based definitions:

```
Lemma accepts_ind_ok' : ∀ p b,
    (wf p) = true →
    (accepts p b = true ↔ accepts_ind p b)).

Lemma sym_accepts_ind_ok' : ∀ p b names,
    sym_wf p = true →
    (sym_accepts p b = true ↔
        sym_accepts_ind p b).
```

Here, `wf` and `sym_wf` are well-formedness conditions on patterns to check that they don't have references to unintroduced variables.

Each of these two equivalence proofs is done by induction on pattern p. These two lemmas can then be used in the proofs of two theorems found in chapter 6: sym_ok and conc_ok. The general strategy is to convert all derivative-based accept functions into inductive-based definitions, and then perform induction over the structure of the inductive acceptance definitions.

Chapter 8

Conclusions and Future Work

The work in this thesis demonstrates that regular expression-based specification can be used to write human-readable specifications that can be used for verified model checking. More work needs to be done to make this strategy viable for model checking with symbolic variables, both in writing proofs of correctness of the existing specification languages in this work and in extending the languages to be expressive enough to describe a more diverse set of systems. The greatest limitation of our approach is the constraint that one needs a decidable function to check whether there exists an input that matches against a given pattern, which poses a significant challenge to providing even basic constructors such as the logical not ($\neg$) and the logical and (&). Here we provide a list of future work that could make our approach useful in practice.

8.1 Variable-Length Variables

Although MLVR already offers a constructor for variables whose lengths depend on the value of other variables, this feature could not be used in practice to model check a system because we have no way of checking whether there exist any inputs that match a given pattern mentioning variable-length variables. One promising possibility would be to remove the variable reference constructor from the first layer (FLVR), and instead have a constructor for referencing variables in the top layer, where while a particular variable x is being recalled, every stream has to either be outputting the value of one of the streams of x, or be a pattern that accepts only strings of length 1, which will be repeated for the length of x. We might have an expression like $[\mathrm{v}_2|\_]_x$, which would mean that in the zeroth stream, we accept the value at the second stream of variable x, and in the first stream, we accept any string of the same length as x.

This solution leaves a few issues to be resolved. We still cannot check whether a given pattern in MLVR accepts any inputs of a particular length, but we can check that a pattern in FLVR accepts an input of a particular length. We can then nearly check whether there exists an input at all which a given pattern in MLVR accepts, except that in the constructor (let $x : l = p_1$ in p_2) we need to be able to tell whether p_1 accepts inputs of a given length l. We may resolve this by forbidding references to variable-length variables inside the first MLVR expression in let constructors. This approach has not yet been implemented in Coq.

8.2 Arbitrary Function Calls

As mentioned in chapter 5, it would be helpful to allow users to specify their own functions, which they would be able to reference within an MLVR pattern. These functions would take one or more variables' values and return a string to be used in the MLVR. More work needs to be done to understand how the support for such functions would affect theorems relating to the correctness arguments in model checking.

8.3 Conditional Branching

As mentioned in chapter 5, MLVR currently lacks any sort of branching constructor which depends on the success of matching on a particular subpattern. The issue with adding such a constructor is that in order to check whether such a branching constructor accepts some input, we need to check whether it's possible for any string to not match against the pattern being branched on. Unfortunately, this is a difficult condition to check, and we have not found a way to do it yet for the full MLVR language.

8.4 More Expressive Symbolic Input Language

In order to complete the equivalence proofs between symbolic and concrete model checking for the SLVR language, we would need to use a more expressive language for the symbolic inputs. Such an input language would need to incorporate information about the context in which a variable is being referenced. One possibility would be to add one additional constructor to the input language:

s : Inp :=	
$\mid b$	(Boolean bit, where $b \in \{0, 1\}$)
$\mid \mathrm{v}_l$	(newly introduced variable of length l)
$\mid \mathrm{r}_i$	(recall the ith most recently introduced variable)
$\mid \bar{\mathrm{r}}_i$	(incorrectly recall the ith most recently introduced variable)
$\mid$ EndCtx	(Match with the end of a SLVR pattern with a let constructor)

This would eliminate ambiguity about the semantics of variable references described at the end of chapter 6. How this would affect the symbolic derivative function is an open question as it is not currently possible to check whether a given subpattern was previously inside a let constructor. It may be necessary to extend the SLVR definition to indicate variable contexts.

8.5 Finishing Proofs of Correctness

Currently, we only have the majority of the proofs of equivalence between symbolic and concrete executions of SLVR. We would like to provide full proofs of equivalence between symbolic and concrete executions of MLVR. This would likely involve a reworking of how

we treat variable lengths in symbolic inputs. For the proofs relating to SLVR, we could use a natural number to represent the length of each variable we wanted to input, but with variable-length variables, it would become more involved to check whether lengths match up between concrete and symbolic inputs.

We would also like to create a symbolic-variable step function for Kôika and prove equivalence between this version and the original Kôika semantics. This would allow us to model check real systems with MLVR.

References

[1] E. M. C. Jr., O. Grumberg, and D. Peleg, *Model Checking*. Cambridge, MA: The MIT Press, 1999.

[2] S. Owens, J. Reppy, and A. Turon, "Regular-expression derivatives re-examined," *Journal of Functional Programming*, vol. 19, no. 2, pp. 173–190, 2009. DOI: 10.1017/S0956796808007090.

[3] M. Sipser, *Introduction to the Theory of Computation*. PWS Pub. Co, 1997, ch. 1.3.

[4] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, "The essence of bluespec: A core language for rule-based hardware design," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 243–257, ISBN: 9781450376136. DOI: 10.1145/3385412.3385965. [Online]. Available: https://doi.org/10.1145/3385412.3385965.