MASTER THESIS
COMPUTING SCIENCE

RADBOUD UNIVERSITEIT

Learning and Model Checking Real-world TCP Implementations

Ramon Janssen
ramonjanssen@cs.ru.nl

Supervisors:
Frits Vaandrager
Paul Fiterău-Broştean

September 7, 2015

Contents

Abstract

In this thesis, active learning of abstract models is combined with model checking, combining the strength of both techniques. Active learning ensures that the model is directly based on the actual implementation, while model checking allows for easy analysis of composed networks of learned models. Utilizing these techniques, models of TCP implementations are inferred, and a network of these models is model checked. As the concrete input alphabet is too large to apply learning directly, a mapper translates the inputs to an abstract domain. To perform model checking on concrete models, this abstraction should be reversed. A mapper language is proposed to make mappers easier to apply, and to make it possible to invert mappers with a constraint solver.

1 Introduction

As our reliance on technology grows, reliability of systems becomes increasingly important. Model checking is one way of verifying the correctness of systems. Once a model of a system is available, it is relatively easy to check whether the required properties hold; algorithms exists to check many kinds of specifications, for example in temporal logic, provided that the model is not too big.

An advantage of model checking is that it easy to write and compose models. Interaction with the actual system is not needed. However, this is also a disadvantage: models are checked for correctness, but how to ensure that these models correspond with the actual behaviour? Ideally, models should be created as part of the development process, but this is not often done in practice. Some progress has been made to obtain models from source code [8, 13, 15]. But such techniques are not always applicable, for example with complex networks or in a black-box setting. In practice, many models are therefore still hand-made, based on specifications or documentation. However, implementation errors and unspecified behaviour are then not modeled. The model itself may also be faulty. These issues undermine the reliability of formal verification on hand-made models.

Another way of creating models is through active learning [3, 7, 20]. This allows to extract models from actual systems, by letting a learner interact with the system under test (SUT) in a black-box setting. The learner follows a learning algorithm, by which it sends inputs to the SUT, and observes the outputs. A model is then inferred which corresponds with these inputs and outputs. This approach ensures that the obtained models correspond with the actual system, and increases the reliability of formal verification. Model checking on learned models has been done [11], but this was restricted to checking just the learned model in isolation. Composition is a strength of model checking, and can be used to scale up to networks. In contrast, checking properties for compositions of systems is harder with test-based approaches. This approach ensures the reliability of test-based approaches, and utilizes the flexibility of model checking.

In practice, inputs and outputs of systems often contains parameters. This makes active learning difficult: the number of inputs and outputs grows exponentially in the number of parameters. The same holds for the number of states: a system using state variables often has a large number of states. Aarts et al. [3] resolved this by mapping the set of concrete parameters to a small range of abstract ones. This reduces the size of the model, such that it can be learned. An abstract model is then learned: the behaviour of the actual system is then described by the combination of this abstract model and the mapping.

These technieques have been applied to model check TCP implementations. Model checking on TCP has been done before, for example based on code [21] or on hand-made models [19]. In this thesis, models of TCP servers and clients are obtained through active learning. This is based on previous work in which a TCP server was modeled, by sending inputs over the network [14], and sections of this article have been re-used for this thesis. The learned models, together with a model of the network, are composed into a setup with multiple communicating TCP systems which is then model checked. This is done with the NuSMV model checker [12]. As parameter abstraction is used to reduce the input alphabet, an abstract model is learned which cannot be model checked directly. For this purpose, the mapping from concrete to abstract parameters is reversed in the model checker. The model then behaves similar to the concrete system, and communication between a TCP client and server is simulated.

One of the contributions of this thesis is that it combines strengths of active learning and model checking in a new way; Learned models are not only model checked, but also composed into networks. At a technical level, the main contribution is the definition of a language to define mappers, such that abstractions are easy to write, and can be automatically reversed during the model checking.

2 Preliminaries

2.1 Mealy machines

We use Mealy machines to model implementations of TCP entities. A Mealy machine is defined as a 5-tuple $\mathcal{M} = (Q, q_0, \Sigma, \Lambda, \rightarrow)$, where

- Q is the set of states
- $q_0 \in Q$ is the initial state
- Σ is the set of input symbols
- Λ is the set of output symbols
- $\rightarrow \subseteq Q \times \Sigma \times \Lambda \times Q$ is the transition relation

The intuitive meaning of a Mealy machine is as follows: At any time, the machine is in a state $q \in Q$, starting in state q_0. If it is in state q, upon receiving input i, it makes a transition $(q, i, o, q') \in\rightarrow$. It then changes state to q' and produces output o. This is also denoted as $q \xrightarrow{i/o} q'$.

If at most one transition can be taken for every combination of input and starting state, the Mealy machine is deterministic. If always at least one such a transition can be taken, it is said to be input-enabled.

2.2 Learning the behaviour of Mealy machines

Angluin [7] described the L*-algorithm for learning a deterministic input-enabled finite automaton describing a system, when inputs can be actively given and it can be observed when the system is accepting. The state of the system is not observed. Niese [16] extended this to learning Mealy machines, in which outputs symbols are observed. This algorithm is used in the LearnLib [27] library, which can be used to learn Mealy machines describing real-world systems. In this setting, LearnLib is called the learner. L* can only learn input-enabled Mealy machines.

The learner sends sequences of inputs to the SUT, and observes all outputs. After each sequence, the learner sends a *reset*-request to the SUT, making it jump to the initial state. By observing multiple sequences, it constructs a hypothesis $\mathcal{H}$, which is a Mealy machine consistent with the observed behaviour. It then asks an equivalence oracle whether this hypothesis describes the behaviour of the SUT correctly. The equivalence oracle is implemented by picking sequences of inputs, and comparing the outputs of the SUT and the hypothesis. If any difference is found, this is a *counterexample* against the hypothesis. This counterexample is returned to the learner, which then builds a new, improved hypothesis. This is repeated until no more counterexample is found. The hypothesis is then accepted as a Mealy machine describing the SUT.

2.3 Abstraction of parameters

Existing implementations of inference algorithms only proved effective when applied to machines with small sets of input symbols. Practical systems like the TCP protocol, however, typically have large alphabets, as inputs and outputs have data parameters of type integer or string. Each combination of parameters is considered a different symbol. The number of states and transitions is typically exponential in the number of parameters. As such, learning a Mealy machine describing a SUT is not feasible, and even if it would be, the resulting number of states and transitions would be too large to describe explicitly.

A solution to this problem was proposed by Aarts et al. in [5]. In this work, the set of concrete values of every parameter is mapped to a small domain of abstract values in a history-dependent manner. Any interaction with the SUT is then done through a mapper component. This component translates abstract inputs to concrete, and concrete outputs to abstract. A mapper component is an actual system that performs the translation, whereas the translation is formally defined by a corresponding *mapper*. If behaviour of the SUT is defined by a concrete Mealy machine $\mathcal{M}$, then $\mathcal{M}$ and the mapper $\mathcal{A}$ together define the abstraction of $\mathcal{M}$, $\alpha_{\mathcal{A}}(\mathcal{M})$. This abstraction is again a Mealy machine, but with an abstract alphabet. Although the SUT still uses concrete inputs and outputs, interaction with an abstraction of the SUT can now be done with abstract symbols through the mapper component. A graphical overview of the learner and mapper component is given in figure 1.

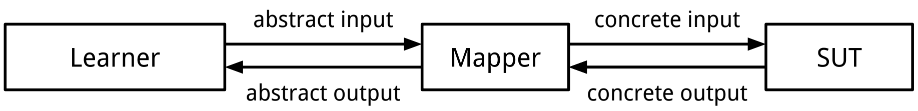


Figure 1: Overview of the learner, the mapper and the SUT

Formally, a mapper $\mathcal{A}$ is defined simply by a deterministic Mealy machine. Suppose that I and O are the concrete input and output alphabets for the SUT, respectively. Similarly, suppose that X and Y are the abstract input and output alphabets for the abstract Mealy machine. Then the mapper has an input alphabet $I \cup O$, and an output alphabet $X \cup Y$. The transition relation is then constrained such that concrete inputs are mapped to abstract inputs, and concrete outputs to abstract outputs. That is, $\forall_{(q,i,o,q') \in \rightarrow}(i \in I \leftrightarrow o \in X) \wedge (i \in O \leftrightarrow o \in Y)$.

Let $\mathcal{M} = (Q, q_0, \Sigma, \Lambda, \rightarrow)$ be a Mealy machine, and $\mathcal{A} = (R, r_0, X \cup Y, \Sigma \cup \Lambda, \rightarrow_{\mathcal{A}})$ be a mapper. The abstraction of $\mathcal{M}$ via $\mathcal{A}$ is then a Mealy machine $\alpha_{\mathcal{A}}(\mathcal{M}) = (Q \times R, (q_0, r_0), X, Y \cup \{\perp\}, \rightarrow_\alpha)$, where $\rightarrow_\alpha$, is defined by the rules

$$\frac{q \xrightarrow{i,o} q', r \xrightarrow{i/x}_{\mathcal{A}} r' \xrightarrow{o/y}_{\mathcal{A}} r''}{(q,r) \xrightarrow{x/y}_{\alpha} (q', r'')} \qquad \frac{\nexists (r, i, x, r') \in \rightarrow_{\mathcal{A}}}{(q,r) \xrightarrow{x/\perp}_{\alpha} (q,r)}$$

The abstraction takes abstract inputs. Translation to a concrete and back to abstract is done according to the first rule. If multiple mapper transitions are legal, it may take any one of them. However, for some concrete outputs there might not exist a corresponding input, in which case $\perp$ is returned according to the second rule.

Additionaly, a mapper component can be used in the reversed direction in a similar manner. Consider the same mapper $\mathcal{A}$, and an abstract Mealy machine $\mathcal{H} = (H, h_0, X, Y, \rightarrow)$. In that case, a concrete Mealy machine may then be simulated. This yields the concretization of $\mathcal{H}$ via $\mathcal{A}$, as $\gamma_{\mathcal{A}}(\mathcal{H}) = (R \times H, (r_0, h_0), \Sigma, \Lambda, \rightarrow_\gamma)$, with $\rightarrow_\gamma$ given by the rule

$$\frac{r \xrightarrow{i/x}_{\mathcal{A}} r' \xrightarrow{o/y}_{\mathcal{A}} r'', h \xrightarrow{x/y} h'}{(r,h) \xrightarrow{i/o}_{\gamma} (r'', h')}$$

Again, it may occur that an abstract output cannot be concretized. Aarts defines a second rule for this situation, again letting the concretization return $\perp$. However, when concretizing an abstracted model, this should not occur.

An abstraction or concretization is implemented by an abstracting or concretizing mapper component, respectively. As the mapper state is also part of the state of the abstraction or concretization, reset messages sent by the learner to the SUT should also reset the mapper state. An overview of directions of translation by mappers and mapper components is shown in figure 2. When mapping an abstact value to a range of concrete values, any one of those concrete values may be used by the mapper component, according to the formal rules for concretization and abstraction. The assumption is made that all these values result in the same abstract behaviour. Ideally, the mapper component should pick useful values to test this assumption. This could be achieved by letting the mapper component pick values randomly, with a distribution that gives a good test coverage. For example, boundary values could be tested, as well as some random values over the entire range.

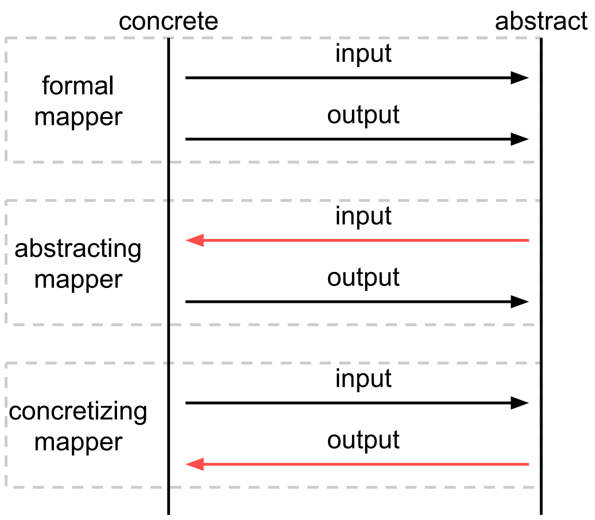


Figure 2: Overview of directions of translation by mappers and mapper components. Red arrows represent directions that are non-deterministic, and possibly non-input-enabled. For this thesis, abstraction is used during learning, and concretization is used during model checking.

Note that the mapper is described by a deterministic input-enabled Mealy machine, which implies that every concrete value is mapped to exactly one abstract value. Translating concrete values to abstract is therefore well defined. However, mapper components need to translate abstract to concrete as well, through the same mapping. The number of abstract symbols is typically smaller than the number of concrete symbols. As such, an abstract value is generally *not* mapped to exactly one concrete value. A mapper component is therefore not deterministic, and possibly not input-enabled, when translating abstact symbols to concrete. If no transition exist for some input, an abstracting mapper component simply returns $\perp$ and does not change state, following the mapper definition. In the resulting abstract state machine, this causes some self-loops with $\perp$ as output, signifying that this abstract input is not enabled. For a concretizing mapper component, such a transition should always exist; otherwise, the abstract Mealy machine creates outputs which cannot be translated, and no concretization can be made. In that case, the mapper makes no sense.

We define $\leq$ as trace inclusion between Mealy machines; $\mathcal{M}_1 \leq \mathcal{M}_2$ denotes that for any sequence of inputs, if $\mathcal{M}_1$ can produce output sequence $o_1, o_2 \ldots$, $\mathcal{M}_2$ can also produce those outputs. A result found by Aarts [2] is that if an abstract Mealy machine $\mathcal{H}$ is learned for a concrete Mealy machine $\mathcal{M}$, it always holds that $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$, so all traces appearing in the abstraction can also occur in the actual system. Additionaly, Aarts has found that $\alpha_{\mathcal{A}}(\mathcal{M}) \leq \mathcal{H}$ implies $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$. As such, $\mathcal{M} \leq \gamma_{\mathcal{A}}(\mathcal{H})$ holds for the concretization of learned models. This is useful for model checking of safety properties, which say something about all traces of a Mealy machine. If such a property holds for $\gamma_{\mathcal{A}}(\mathcal{H})$, the concretization of the learned model, it then also holds for $\mathcal{M}$, the system itself. Model checking can thus be done on the concretization of learned models, and the conclusions can be extrapolated to the SUT.

2.4 Characteristic function

For the transition relation $\rightarrow \subseteq Q \times \Sigma \times \Lambda \times Q$, we define the *characteristic function* $\chi_{\rightarrow} : Q \times \Sigma \times \Lambda \times Q \rightarrow \mathbb{B}$ as $\chi_{\rightarrow}(x) = x \in \rightarrow$. That is, it tells whether a potential transition is legal according to the transition relation. Solving a characteristic function $\chi_{\rightarrow}$ gives the legal transitions. Solving it for given q and i yields the matching transition (q, i, o, q'), i.e. it yields the corresponding output and the new state. Similarly, solving it for given q and o yields the corresponding legal inputs and the new state. In this way, finding transitions of a Mealy machine can be reduced to solving a boolean function. This can be done with constraint solvers or by using binary decision diagrams.

2.5 TCP

Learning and model checking will be applied to the Transmission Control Protocol [24], or TCP. This protocol is used to provide a reliable data stream between two remote applications, through a connection. A connection progresses through several states, first establishing a connection, then sending data bi-directionally, and finally closing the connection in either direction. TCP implementations communicate with their controlling application through system calls, and with the remote TCP implementation through TCP segments, sent over the network. A TCP implementation may be modeled as a Mealy machine, for which the inputs are segments and system calls, and the outputs are also segments and return values to system calls. In our setting, each TCP segment is sent with a single network packet. A simple state diagram is depicted in figure 3.

TCP segments contain some parameters relevant for this thesis. Most importantly, it contains flags used to send control information. The SYN-flag is used to establish a new connection, and the FIN-flag to end the connection (in one direction). The RST-flag resets the connection, terminating it in both directions, whereas the ACK-flag is used to acknowledge received data. Furthermore, all segments contain a sequence number, to ensure the correct ordering of packets, and an acknowledgement number to acknowledge data if the ACK flag is set. The sequence number is incremented for every byte of data sent, and for any SYN and FIN flag sent as well. All data with sequence numbers up to the value of the acknowledgement value are acknowledged.

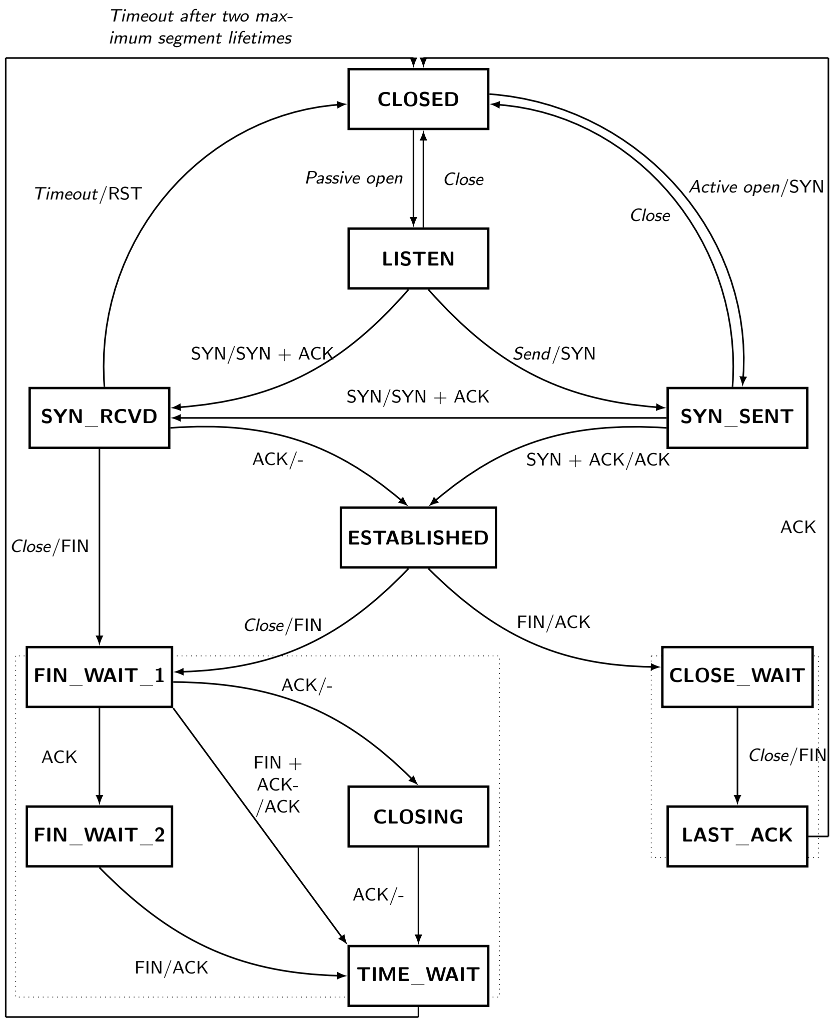


Figure 3: A state diagram describing TCP [1]

An application may act as a TCP client or server: the server starts listening for clients with the *listen*-system call. A client then performs a *connect*-system call, and it sends a segment with a SYN-flag to the server. The server responds with a segment with a SYN and ACK, acknowledging the segment from the client. This segment is again acknowledged by the client, and now a connection is established. An *accept*-system call is done by the server to communicate this connection to the application. Finally, a *close* system call can be done by either side to close one side of the connection, which is communicated with the FIN-flag, which should be acknowledged. Alternatively, two clients can connect to each other by *connect*ing simultaneously. This process is shown in figure 3, but note that this only shows expected inputs. Unexpected inputs may be ignored, make an implementation send a RST-flag, or cause unspecified behaviour.

[1] Retrieved from `http://www.texample.net/tikz/examples/tcp-state-machine/`. Copyright 2009 Ivan Griffin. Reprinted under the LaTeX Project Public License, version 1.3.

3 Mapper programs

A mapper is formally a deterministic, input-enabled Mealy machine. Often, its behaviour can be easily described in most programming languages, or in logic. This is less trivial for a corresponding mapper component. A mapper component always contains a translation from abstract to concrete, which is non-deterministic and possibly not input-enabled. This may also be done with programming languages that allow randomness, or by returning a range of possible outputs.

In previous research, a mapper component was only used for abstraction of an SUT during learning. This allowed to immediately write the mapper component, instead of defining the corresponding mapper first. The translation could directly be programmed in the direction which was needed, and inverting the translation from abstract to concrete could then be skipped.

However, there are two main problems with this approach. The first, problem is that Java is an extensive, general purpose language. It is therefore difficult to generally prove equivalence between a formal mapper definition, and an implementation of a mapper component. The second, more fundemental problem is that a mapper component is programmed to run some translations from abstract to concrete. There is no simple way of inverting the translation described by a program in an imperative language. One way to base both an abstracting and a concretizing mapper component on the same mapper, is by manually writing the two. This is time-consuming and error-prone: both components should be each others inverses. General imperative programming languages are not designed to achieve this.

This even yields a problem when only implementing one mapper component instead of two; it should still correspond with a formal mapper, so the translation from abstract to concrete should have a functional inverse. It is easy to write mapper components which do not adhere to this restriction.

To solve these problems, a simple language is introduced which is suitable for writing mappers. It is an imperative language, but some restrictions are neccesary to make a mapper behave like a Mealy machine. For example, it should be fully deterministic, and it should have transitions: updating the state is done when translating an input to an output.

Also, as the mapper should also be executable invertedly, there should be a way of finding inputs corresponding to given outputs. This is difficult for a program written in a general-purpose imperative language. Specifically, loops, recursion and function calls, and creating variables at run-time make this into a hard problem. As these features are not neccesary for the mappers used for TCP, these are not supported. As such, all state variables are declared in advance; no other variables are allowed. Additionaly, only inputs and state variables can be read from, and only outputs and new state values can be written. Statements are therefore not dependent on sequentiality, which makes reasoning about the inverse execution easier.

With a mapper in this language, mapper components can be created automatically. Also, the language is simple enough to extract a characteristic function to describe the transitions. This characteristic is used for running the program invertedly, but is also suitable for model checking, as described in section 7. An interpreter (written in Java) reads such a mapper program, and may execute it normally or invertedly.

3.1 Language definition

The Language is defined in extended BackusNaur form (EBNF) [1]. All names within angular brackets are non-terminals, whereas all names within quotes are terminals. Square brackets are used to denote optional expressions, and braces are used to denote expressions that may be repeated.

Program overview

A mapper program comprises a state and any number of mappings, which behave like transitions. Note that this differs from mappers represented by Mealy machines, which only have one transition function. If a mapper translates inputs I to X and outputs O to Y, the corresponding Mealy machine would have input alphabet $I \cup O$ and output alphabet $X \cup Y$. A mapper program defines separate translations from I to X and from O to Y instead. Also, following the definition of a mapper, the state variables of the program are fixed and no new variables can be created at runtime.

⟨*program*⟩ := {⟨*enumDefinition*⟩} [⟨*stateDefinition*⟩] {⟨*mapping*⟩}

The first expression is used to define enumerated types; this can be done simply with the name of the type, followed by a list of values.

⟨*enumDefinition*⟩ := 'ENUM' ⟨*ID*⟩ '{' ⟨*idList*⟩ '}'

⟨*idList*⟩ := ⟨*ID*⟩ idListTl

⟨*idListTl*⟩ := {',' ⟨*ID*⟩}

Identifiers are used for enumeration type names, enumeration values and various other grammar rules.

⟨*ID*⟩ := [A..Z | a..z | _][A..Z | a..z | 0-..9 | _]*

State and mappings

The state is defined as a list of variables, with types and initial values.

⟨*stateDefinition*⟩ := 'STATE' {⟨*varInit*⟩ ';'}

⟨*varInit*⟩ := ⟨*type*⟩ ⟨*ID*⟩ '=' ⟨*expr*⟩

There are a few builtin types, in addition to the enumerated types defined at the start of the program. The flags-type is used for sets of flags. In this way, the flags of a packet can be used as one value instead of separate boolean values for every flag, as the language does not contain array-like data types.

⟨*type*⟩ := 'bool' | 'int' | 'flags' | ⟨*ID*⟩

Mappings have a name, and lists of arguments and outputs. These are all variables with a name and a type. Furthermore, there are statements defining the mapping, and the update of the state. The formal input or output of a mapper is then defined as the tuple of all arguments or all outputs, respectively.

⟨*mapping*⟩	:=	’MAP’ ⟨*ID*⟩ ’(’ ⟨*varDeclList*⟩ ’->’ ⟨*varDeclList*⟩ ’)’ {stmt} [’UPDATE’ {stmt}]
⟨*varDeclList*⟩	:=	⟨*varDecl*⟩ varDeclTl
⟨*varDeclTl*⟩	:=	{’,’ ⟨*varDecl*⟩}

Statements

Statements can either be assignments or if-else-statements. In a mapping, assignments should only be done to the output variables. In state updates, assignments should only be done to state variables. Every variable should be assigned a value exactly once. In case of if-else-statements, this means that both branches should assign to the same variables.

⟨*stmt*⟩	:=	⟨*assign*⟩ | ⟨*ifelse*⟩ | ⟨*stmt*⟩ ’;’ ⟨*stmt*⟩
⟨*ifelse*⟩	:=	’if’ ’(’ ⟨*expr*⟩ ’)’ ’{’ {stmt} ’}’ ’else’ ’{’ {stmt} ’}’
⟨*assign*⟩	:=	⟨*ID*⟩ ’=’ ⟨*expr*⟩ ’;’

Expressions

Expressions may be literals, enumeration values denoted as *type.value*, variables, or operators working on other expressions. All binary operators are right-associative. Also, rules are listed in order of precedence. The types of expression are as expected; arithmetic operators have type $int \times int \rightarrow int$, equality has type $T \times T \rightarrow int$ for all T, etc. The *has*-operator is used to check whether certain flags are set. *f1 has f2* returns a boolean denoting whether all flags of *f2* are set in *f1*. Note that integer literals are non-negative: a negative integer is interpreted as a unary minus operating on a positive integer. This ensures that the expression $x - 1$ is not parsed as $x(-1)$.

⟨*expr*⟩	:=	⟨*exprlit*⟩
	|	⟨*ID*⟩ ’.’ ⟨*ID*⟩
	|	⟨*ID*⟩
	|	⟨*expr*⟩ (’+’|’-’) ⟨*expr*⟩
	|	’-’ ⟨*expr*⟩
	|	⟨*expr*⟩ (’*’|’/’) ⟨*expr*⟩
	|	’!’ ⟨*expr*⟩
	|	⟨*expr*⟩ (’==’|’!=’) ⟨*expr*⟩
	|	⟨*expr*⟩ ’has’ ⟨*expr*⟩
	|	⟨*expr*⟩ ’|’ ⟨*expr*⟩
	|	⟨*expr*⟩ ’&’ ⟨*expr*⟩
	|	’(’ ⟨*expr*⟩ ’)’

⟨*exprlit*⟩	:=	⟨*bool*⟩ | ⟨*int*⟩ | ⟨*flags*⟩
⟨*bool*⟩	:=	'true' | 'false'
⟨*int*⟩	:=	digit+
⟨*digit*⟩	:=	'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
⟨*flags*⟩	:=	'$' {'S' | 'A' | 'F' | 'R' | 'P'}

Furthermore, the mappers may contain C-style block-comments and line-comments, and any form of whitespace may be used to separate tokens.

An example of a mapper language is given in code snippet 1. Note that this program contains only one mapping; a mapper component contains at least two (for SUT inputs and outputs), sharing the state.

Code snippet 1: An example mapper

```
ENUM absDomain {V1, V2}

STATE
 int counter = 0;
 bool isCounting = true;

MAP input(int concIn -> absDomain absOut)
 if (isCounting & concIn == counter) {
   absOut = absDomain.V1;
 } else {
   absOut = absDomain.V2;
 }
UPDATE
 if (absOut == absDomain.V1) {
   counter = concIn + 1;
   isCounting = isCounting;  // unchanged
 } else {
   counter = 0;
   isCounting = false;
 }
```

This mapper translates an integer argument and produces an output of domain *absDomain*, an enumerated type containing two values *V1* and *V2*. The state consists of an integer counter and a boolean value *isCounting*. As long as *isCounting* = *true* and the input follows the *counter* value, *V1* is returned. If another input is given, *isCounting* is set to false and *V2* will always be returned afterwards.

3.2 Interpreting and executing

The interpreter first parses the mapper program and creates an abstract syntax tree. It then checks the types of expressions and variables such that only well-typed programs are accepted. It also checks whether all mappings assign a value to every output.

As the mapper language is imperative, executing mappings from concrete to abstract is quite straightforward. An overview of the data flow is given in figure 4. At the start of the interpretation, it creates a state defined by the initial values of all state variables. When executing a mapping, a valuation for the arguments should also be provided. Given such valuations, expressions may then be reduced to values. The mapping contains assignments to output variables, which create an output-valuation. The update of a mapping contains assignments to state variables, changing the state. A mapping runs its update after determining the output. Obviously, if-else-statements execute either branch, depending on the value of the condition. The interpreter provides an interface for other programs to execute mappings in this way, and retrieve the results. This interface is used by the learner to translate incoming responses from the SUT. Additionaly, there is a timeout-mapping which does not have any arguments or results, but only updates the state.

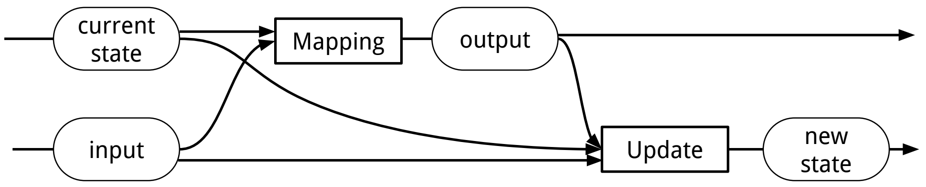


Figure 4: Graphical overview of executing a mapping. The current state and input are used to determine the output. Then, the current state, input, and output may be used to update to the new state.

3.3 Semantics of expressions

The semantics of the language are quite simple when executing a mapping in the normal direction. In sequentially executed programs, evaluated variables can be substituted by their value. But when a mapper is evaluated invertedly, expressions may contain input variables, which are not assigned a value beforehand. As such, some expressions cannot be reduced. The evaluation therefore does not yield a single value, but a more complex expression containing variables. In general, it is not neccesary to substitute any variable by its value. The semantics of expressions are therefore simply a translation from syntax to a similar mathematical expression.

$[\![lit]\!] = lit$ where *lit* is an integer, boolean, flag or enumeration literal.

$[\![x]\!] = x$ where variable x is an input or state variable.

$[\![-expr]\!] = -[\![expr]\!]$

$[\![expr1 + expr2]\!] = [\![expr1]\!] + [\![expr2]\!]$ (and similar for other operators)

For a known valuation, expressions may then be reduced by substituting variables, and by solving the mathematical expressions. As expressions may only contain input and state variables, expressions can always be reduced to a single variable when executing in the normal direction.

3.4 Extracting the characteristic function

The interpreter can extract a characteristic function representing the transition described by a mapping. Recall that formally, $\chi_{\rightarrow} : Q \times \Sigma \times \Lambda \times Q \rightarrow \mathbb{B}$. We represent this function as a boolean expression, based on the statements of the program. This expression is built recursively from the statements of the mapping. As may be expected from the formal definition, this expression contains inputs, outputs, variables in the current state, and variables in the new state. Evaluating this expression for some transition (q, i, o, q') then yields a boolean, denoting whether this transition is legal.

As the result of a mapper program does not depend on the order of statements, the characteristic of multiple statements is simply the conjunction of the individual characteristics:

$\chi_{\mathit{stmt1;stmt2;}}(q, i, o, q') = \chi_{\mathit{stmt1}}(q, i, o, q') \wedge \chi_{\mathit{stmt2}}(q, i, o, q')$

The characteristic of an assignment to an output is a simple equality between the output and the expression. Caution should be taken in the update of the state: the left hand side of an assignment contains variables in the new state q', whereas state variables in expressions are in the current state q.

$\chi_{\mathit{id=expr;}}(q, i, o, q') = (\mathit{id} = \mathit{expr})$ (with id a variable in o or q')

For if-else-statements, either branch can be taken. If the first branch is taken, the branching condition should be true. For the second branch, it should be false. Additionaly, the characteristic of the branch that is taken should hold.

$\chi_{\textbf{ if } (\mathit{cond}) \{\mathit{branch1}\} \textbf{ else } \{\mathit{branch2}\}}(q, i, o, q')$
$= ([\![\mathit{cond}]\!] \wedge \chi_{\mathit{branch1}}(q, i, o, q')) \vee (\neg[\![\mathit{cond}]\!] \wedge \chi_{\mathit{branch2}}(q, i, o, q'))$

In this way, a characteristic expression for both the mapping and the corresponding update is found. The total characteristic is simply the conjunction of the two.

As an example, the characteristic expression of the mapping in code snippet 1 is given. The mapping contains a single if-else-statement. Its condition gives the characteristic expression $(\mathit{isCounting} \wedge \mathit{concIn} = \mathit{counter})$. The first branch gives the equality $\mathit{absOut} = \mathit{V1}$, and the second branch gives $\mathit{absOut} = \mathit{V2}$; The complete mapping is then the disjunction of two branches:

$\chi_{\mathit{input}}(\mathit{isCounting}, \mathit{counter}, \mathit{concIn}, \mathit{absOut}, \mathit{isCounting}', \mathit{counter}')$
$= ((\mathit{isCounting} \wedge \mathit{concIn} = \mathit{counter}) \wedge \mathit{absOut} = \mathit{V1})$
$\vee (\neg(\mathit{isCounting} \wedge \mathit{concIn} = \mathit{counter}) \wedge \mathit{absOut} = \mathit{V2})$

The state update yields a similar expression, containing state variables of the current state and the new state. Combining these expressions then gives the total characteristic function describing the transitions of this mapping.

3.5 Executing mappers invertedly

The program flow for executing a mapper invertedly is similar to normal execution, as shown in figure 5. The difference is that the role of inputs and outputs of a mapping is swapped; outputs are supplied, and matching inputs are returned. This is done by first extracting the characteristic of the mapping. All expressions are reduced as far as possible, substituting all outputs and state variables. This yields a boolean expression on the input variables. A constraint solver is then used to find input values satisfying this expression, and these are returned. The characteristic function can also be used to find the new valuation of state variables. However, the update-function can always be run in the normal direction, equivalent to normal execution, which is faster than using a constraint solver. If no transition exists, the constraint solver will find no input, and the update is not executed; this is useful for learning, as described in section 2.3.

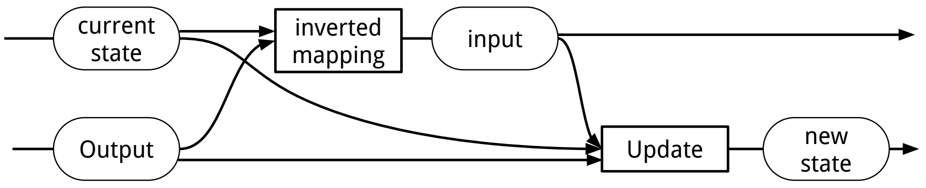


Figure 5: Graphical overview of executing a mapping invertedly. The program flow is similar to executing normally, but the outputs are provided and inputs are returned.

For this purpose, the library Choco [26] has been used. This is a high-performance constraint solver written in Java. Choco supports integer variables with arbitrary ranges. Boolean variables are handled as special cases of integers, with domain {0,1}. Enumerated values are also handled as integers: for each enumerated type, the values are represented as integer constants. Choco also supports sets, and as such, flags can be represented with sets of integers. Every type of flag is then represented by an integer constant.

Choco does not offer a constraint type for satisfaction of arbitrary boolean expression trees. However, it offers enough constraint types to simulate every operator in the mapper language separately. To create an expression tree, an intermediate variable is created for every operator node. For example, the expression $x + y$ is made by creating a variable *tmp* for which the constraint $tmp = x + y$ holds. *tmp* then represents the expression, and can be used in other constraints. The expression $z * (x + y)$ is then again represented as $tmp2$, for which $tmp2 = z * tmp$.

Choco can tell whether solutions exists. If multiple solutions exists, Choco returns one of them. There is no control over which of these solutions is returned. As described in section 2.3, the constraint solver should ideally pick boundary values with a significant chance. However, this cannot be done with most constraint solvers.

As an example, the mapper in code snippet 1 can be run in the inverted direction. First, consider the case where *isCounting* = *true*. If *V1* is expected as output, the first branch of the mapper should be evaluated. As such, *concIn* will have value *counter*. If *V2* is expected, any other value for *concIn* is allowed.

Now consider the case when *isCounting* = *false*. The first branch can then never be taken. Then there is no solution when expecting output *V1*, and any input satisfies expected output *V2*.

3.6 Antlr

Parsing of the language is done with the Antlr 4 [22]. This is a Java-library, which reads a dialect of EBNF-grammar, and which can then generate parser code for a given grammar. The EBNF-grammar as described in this section is thus used directly (in Antlr-syntax) to ensure a correct parser. The generated code can then be called to obtain a parse tree, used in the interpreter and also for generating mappers components used in model checking, as described in section 5.2. Antlr is able to handle left-recursion and rule precedence automatically, which simplifies parsing expressions and reduces the size of the parse tree.

4 Learning Setup

4.1 Connecting the Learner

With the mapper language available, mappers can be constructed to learn models of TCP implementations. In that case, the SUT is either the server or the client in the TCP communication. These SUTs are from now on called the TCP entities. Both client and server have been modelled with LearnLib in separate, but similar, experiments. In a real life TCP connection, there are always two TCP entities participating. When learning a model of one TCP entity, the other is simulated by the learner and mapper component, which can send and receive network packets. The mapper component is the interpreter of the mapper language, together with the mapper program it executes. In this mapper program, actions, incoming segments, outgoing segments and timeouts are all mapped through different mappings, so a small wrapper is used to call the appropriate mapping. The SUT also communicates with its application layer through system calls, also controlled by the learner. Whereas the mapper component implements mapping between abstract and concrete messages, we introduce the adapter to transform these concrete messages to actual actions performed by the system. This can be either creating TCP segments and sending it over the network, or commanding the application layer to perform a system call. The adapter also listens for response packets on the network and infers the respective concrete outputs, which it delivers to the learner. The adapter is also responsible for detecting system timeouts; the SUT is not guaranteed to respond with a TCP segment, so after a short time, the abstract output TIMEOUT is concluded.

The adapter runs on the side of the learner whereas the SUT runs on a different (virtual) machine. Therefore, the adapter cannot perform system calls directly. System calls need to be performed by an application, and therefore there is an application running the SUT. Like the adapter, it only listens to learner commands, and performs the corresponding system calls. For this purpose, the application running the SUT also makes a direct socket connection with the adapter. Additionaly, this connection is also used to communicate which port should be used to send and receive packets; After every reset, this port is changed to prevent connections from previous runs from interfering.

With that said, we present in figure 6 the framework implemented to learn segments of the TCP implementation. On the learner side, we use *LearnLib* [27], a Java based learning library. LearnLib provides the Java implementation of the L* based learning algorithm. A Python adapter based on *Scapy* [10] is used to craft and send packets and retrieve response packets. Communication between the mapper component and adapter is done over sockets. With virtualization, the experiments can be run on a personal PC by running the learner on a virtual machine and the SUT on the host, or vice versa.

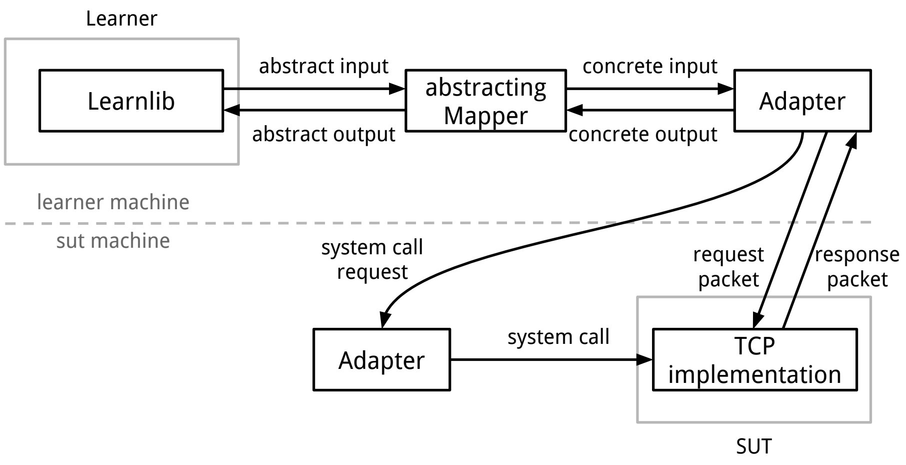


Figure 6: Overview of the experimental setup

The LearnLib implementations of L* and the observation pack algorithm [17] are used for building model hypotheses. Equivalence testing was done with random traces.

4.2 Application Layer Choices

The mapping from abstract to concrete values and vice versa are trivial for the messages to and from the application layer, as they contain no parameters; the mapper is just the identity function. According to the rfc [25], the system calls needed to start a connection are *listen* and *accept* for the server, and *connect* for the client. To terminate a connection, both sides need a *close*. In the input alphabet, we have used the corresponding messages *listen*, *accept*, *connect*, *close connection* and *close server*. These messages need to be interpreted as actions for the application running the server. However, the descriptions of these messages in the rfc abstract from many implementation details, and these inputs do not all trivially translate one on one to system calls. As such, choices have to be made in this implementation regarding the actual actions performed.

Specifically, the rfc standard does not describe how sockets are used while establishing connections. All modern operating systems use a dedicated socket to represent the listen state. When a client connects, a new socket is created for this connection. This connection then proceeds to the next state from the listen state, while the listening socket remains in the listen state. In this way, many connections may be established with one listening socket. I have chosen to use at most one listening socket and one connection socket at any time. If a connection socket still exists when a new connection socket is created, the old connection socket is discarded and all operations are performed on the new socket. There are two reasons why we do not allow multiple connections at any time:

- It would become unclear which connection *close connection* is supposed to close, or over which connection a TCP segment should be sent.
- Every connection would have an own state, so the state diagram of multiple parallel connections would become needlessly complex. If the behaviour of individual connections can be presumed independent, this would not add any information.

The state diagram as described in the rfc mentions only one *close* message, but this does not describe the behaviour of socket implementations accurately; the listening socket and connection socket can be closed individually, and both show different behaviour. Closing the listening socket does not close any connections, but prevents any new connections from being created. Closing the connection socket terminates the connection, but does not influence the listening socket. As such, we use the two messages *close connection* and *close server* instead of a single *close* message.

Another implementation detail is that the *accept* and *connect* system calls are blocking. The thread calling these cannot do any other system calls until the connection is established, for which additional inputs are needed. It is questionable what the behaviour of the application should be when the learner asks for the input sequence *listen*, *accept*, *close server*. The application may not be able to close the socket if it is still blocked by the *accept*. This input sequence would then cause the same behaviour as when the *close server*-input would not be asked. Additional insight in the behaviour of the server is therefore gained if the application is able to process any inputs when an *accept*-call is still being done. For this reason, we have chosen to process the *accept*-input in a different thread, such that additional inputs can still be processed in parallel. If an *accept* is asked when this additional thread is still handling another, older *accept*, the last *accept* is ignored. The same is done for *connect*.

The messages *close connection* and *close server* simply call a closing system call on their socket. *close server* is only in the input alphabet of the server. Additionaly, for the server the listening socket is already bound to an address with a *bind* system call at the start of a learning sequence. The connection socket of the client is also bound to an address with a *bind*. In practice, the client address is generated only upon establishing a connection, but for learning, it is practical to already have the address available before starting to send an input sequence. The actions as described above should allow traces which follow the transitions as described in the state diagram in the rfc, as well as traces deviating from it.

The system calls may return error messages to the application. These are not included in the output alphabet for simplicity, as each transition may then have two outputs: one as a tcp segment over the network, and one error message. These inputs could be combined such that each output in the abstract model becomes a tuple of outputs from these two channels. However, this would delay the learning, as the learner can now proceed with the next input as a TCP segment arrives; it would then also have to wait for error messages, which are expected to appear less often, leading to more timeouts. Also, it is unclear how error messages from delayed system calls *accept* and *connect* should be handled.

4.3 Practical problems

The general setup as described in this section works, and can be used to learn model of TCP implementations. However, there are some technical issues that also need to be solved. These have to do mostly with the interface between the learner and the SUT.

As TCP should provide a reliable data stream even on unreliable networks, packets are retransmitted if they are not acknowledged properly. The learner cannot handle such retransmits; it will erroneously match the retransmit to the input that was sent just before. Running a trace should be finished before the first retransmits arrive, and the length of traces should thus be short enough. In practice, this problem limits the size of the abstract input alphabet; The number of abstract inputs determines the level of detail of the model, and the length of input sequences needed to learn it. Alternatively, retransmits could be detected and left out, but the SUT might also send two actual responses with the same parameters, and actual responses would then be discarded.

Another issue is the TIMEOUT output, which denotes that the SUT does not respond. The adapter concludes this output if it does not receive any network output within a certain period. If this period is too short, the adapter may conclude a TIMEOUT before a response from the SUT arrives. Due to the issues with retransmits, this period should also not be too long. The minimal waiting time depends on the SUT implementation, but also on the virtual machine and on the virtual network. The delay could be shortened by giving all relelvant processes priority in their operating systems. Different hardware gives different minimal waiting times, but typical timeout values are in the range of 100 to 200 milliseconds.

Furthermore, there are issues with sending requested packets. Usually, TCP packets are only sent through sockets; the network stack manages this. But the learner requests single packets, not neccesarily according to the protocol. Although a library such as Scapy can do this, the network stack does not know about this. If it then receives a response, it will not recognise it as a registered connection, and it will attempt to close the apparent connection by sending a packet with a reset-flag. A firewall should be used to prevent these resets, while still allowing the packets from Scapy.

Ideally, a reset from the learner should also reset the adapter and the application running the SUT. From the side of the learner, packets are not send through sockets, so no sockets are created and none have to be closed. The SUT, however, creates many sockets during learning, and these would accumulate if they are not closed properly. This is done by sending a packet with a reset-flag, and by closing any created sockets on the side of the SUT. As the learner tries many combinations of inputs, these sockets are closed in many different states. With the application netstat [9], it is observed that not all sockets close properly during the learning in this way. Some sockets remain in the listen state, but the cause for this was not found. After too many sockets have been created, creating any new sockets becomes slow, and eventually the learning cannot continue. A work-around was to make restarts of the SUT application possible during learning.

Many of the mentioned issues manifest as non-determinism: retransmitted or missed responses depend on timing, and this varies per trace. One input trace might therefore yield different output traces, as these issues might or might not occur for every attempt. By default, LearnLib assumes that the SUT is deterministic. A wrapper was added to the mapper, which checks determinism; if an input trace was already learned before, the output trace should be equal to earlier observations. Upon detecting non-determinism, the learning is stopped.

5 Mapper implementation

5.1 Abstraction

As mentioned previously, the mapper component has several mappings. Actions simply use the identity function, and they do not update the mapper state. TCP segments from the learner to the SUT and vice versa do need a mapping. The parameters used in this translation are shown in table 1. The mappings are based on the work of Aarts et al. in [5]. Like in their work, both inputs and outputs are generated based on the sequence number, acknowledgement number and flags found in each TCP segment.

Abstract inputs from the learner are *valid* and *invalid*. A *valid* input is an input that can be sent according to the protocol. This mapper is an intuitive interpretation of the rfc and practical experience. It is not trivial to define when a connection is established in terms of inputs and outputs. As such, small mistakes may exist in the initial mapper. However, these may be detected with the model checking, and the mapper can be improved to fix these issues.

If a connection is established, a valid sequence number is equal to the last acknowledgement received from the SUT. A valid acknowledgement number acknowledges the last sequence number received, so it is equal to that number plus one. If no connection is established yet, any sequence number can be picked as a fresh value, and the acknowledgement number is zero. All other values are invalid.

state variable	domain
sutSeq	integer
learnerSeq	integer
learnerSeqProposed	integer

concrete parameter	domain
flagsIn	flags
concSeqIn	integer
concAckIn	integer
flagsOut	flags
concSeqOut	integer
concAckOut	integer

abstract parameter	domain
absSeqIn	{valid, invalid}
absAckIn	{valid, invalid}
flagsIn	flags
absSeqOut	{current, next, zero, fresh}
absAckOut	{current, next, zero, fresh}
flagsOut	flags

Table 1: The domains of all mapper parameters and state variables.

By observing the responses from the SUT, the mapper concludes what the current sequence numbers from learner and SUT should be. Firstly, the SUT may send acknowledgement numbers with value zero. This may occur when no connection is established according to the SUT, or when it sends a reset. As such, *zero* is an abstract output. Whenever the SUT sends a non-zero acknowledgement number, this value can be based on the sequence number of the learner. When acknowledging the last sequence number sent, it is equal to that number plus one. This is represented by the abstract output *next*. Furthermore, the last packet may be rejected, and the previous sequence number may then be acknowledged with abstract output *current*. In all other situations, the acknowledgement number is translated to *fresh*.

For the sequence number of the abstract output, we use a similar approach. *next* is used to represent a sequence number that is expected according to the last acknowledgement number sent by the learner. *current* is when a previously received number is re-used. *zero* and *fresh* are again used for all other values.

5.2 Mapper program

The mapper program used to learn a models is shown in code snippet 1, 2, and 3. This mapper works for both the client and the server. It is denoted in mathematical representation for readability. Furthermore, unchanged state variables should be explicitly mentioned in the mapper language. This has been omitted in these code snippets, as well as the identity mapping for flags.

The state comprises three integer variables: the current sequence numbers of the SUT and learner, and a freshly proposed sequence number of the learner.

$sutSeq, learnerSeq, learnerSeqProposed : \{0..2^{32}\}$

The latter is used when the learner can pick a fresh value. If it is acknowledged, the learner can continue with that sequence number. Otherwise it is concluded that the sequence number is rejected by the SUT, and it can be reset such that new fresh values can be picked. Furthermore, we use the constant value **not_set** $= -3$ to denote that a state variable does not have a meaningful value. This is also the initial value of all state variables.

Code snippet 1 The mapper program for translating abstract TCP segments from the learner to concrete segments of the SUT.

```
map REQUEST($flagsOut, concSeqOut, concAckOut \rightarrow absSeqOut, absAckOut$)
    if $learnerSeq =$ not_set $\vee$ $learnerSeq = concSeqOut$ then
        $absSeqOut = valid$
    else
        $absSeqOut = invalid$
    end if
    if $(sutSeq =$ not_set $\wedge\ concAckOut = 0) \vee (sutSeq \neq$ not_set $\wedge$
$concAckOut = sutSeq + 1)$ then
        $absAckOut = valid$
    else
        $absAckOut = invalid$
    end if
end map
update
    if $learnerSeq =$ not_set then
        $learnerSeqProposed = concSeqOut$
    else
        $learnerSeqProposed =$ not_set
    end if
end update
```

With these mappers it was possible to learn models, but with the model checker, some errors in the mapper were found as described in section 7.6. Traces were found with *invalid* inputs which should actually be *valid* according to the specifications. One of the errors was that the acknowledgement number is always expected to be the last sequence number received, plus one. This is only the case if the packet containing that sequence number also contains control information, i.e. a SYN or FIN flag. The change to the mapper which fixes this can be seen in code snippet 4

An additional mistake was found in recognising when the learner and SUT are connected. Establishing a connection seemed to function properly, but closing a connection was not. If the connection was already closed with a reset according to the SUT, it could send reset-packets with an acknowledgement number zero. The learner did not recognise that the connection was terminated, and expected its sequence number to be acknowledged. This could be fixed by updating the mapper according to code snippet 5.

Code snippet 2 The mapper program for translating concrete TCP segments from the SUTto abstract segments to the learner.

```
map Response($flagsIn$, $concSeqIn$, $concAckIn \rightarrow absSeqIn$, $absAckIn$)
    if $concSeqIn = sutSeq + 1$ then
        $absSeqIn = next$
    else if $concSeqIn = sutSeq$ then
        $absSeqIn = current$
    else if $concSeqIn = 0$ then
        $absSeqIn = zero$
    else
        $absSeqIn = fresh$
    end if
    if $concAckIn = learnerSeq + 1 \vee concAckIn = learnerSeqProposed + 1$ then
        $absAckIn = next$
    else if $concAckIn = learnerSeq$ then
        $absAckIn = current$
    else if $concAckIn = 0$ then
        $absAckIn = zero$
    else
        $absAckIn = fresh$
    end if
end map
update
    if $flagsIn$ has $R \vee (learnerSeqProposed \neq$ not_set $\wedge\ concAckIn \neq learnerSeqProposed + 1)$ then
        // upon receiving reset, or if a fresh seqNr from the learner is rejected
        $sutSeq = learnerSeq =$ not_set
    else if $learnerSeqProposed =$ not_set $\vee\ concSeqIn = sutSeq + 1$ then
        // if seqNr matches, or if a fresh seqNr from the learner is accepted
        $sutSeq = concSeqIn$
        $learnerSeq = concAckIn$
    else if $flagsIn$ has $S$ then
        // a SYN means that the SUT picked a fresh seqNr
        $sutSeq = concSeqIn$
        if $concAckIn \neq 0$ then
            $learnerSeq = concAckIn$
        end if
    end if
    $learnerSeqProposed =$ not_set
end update
```

Code snippet 3 The mapper program for translating timeout responses, i.e. when the SUT does not respond. As there are no parameters to translate, the mapping itself is empty, but it still updates the state.

```
map Timeout
end map
update
    $learnerSeqProposed =$ not_set
end update
```

Code snippet 4 The correction to the validity of acknowledgement numbers, in the *Response*-mapping. The update of *sutSeq* is now only done conditionally.

```
map RESPONSE
    ...
end map
update
    ...
    if $(\mathit{flagsIn}$ has $\$S) \wedge (\mathit{flagsIn}$ has $\$F)$ then
        $\mathit{sutSeq} = \mathit{concSeqIn}$
    end if
    ...
end update
```

Code snippet 5 The correction to closing connections with a reset, in the *Request*-mapping.

```
map REQUEST
    ...
end map
update
    if $\mathit{flagsOuthas}\$R$ then
        $\mathit{learnerSeqProposed} = \mathit{sutSeq} = \mathit{learnerSeq} =$ not_set
    else if $\mathit{learnerSeq} =$ not_set then
        $\mathit{learnerSeqProposed} = \mathit{concSeqOut}$
    end if
end update
```

6 Learned models

With the mapper described in section 5, model of an Ubuntu 14.04 server and client have been learned. The result is shown in appendices A and B. For the parameters, the flags SYN, FIN, RST are used, as well as combinations of these together with ACK, and ACK without other flags. The *close server* input is not included; the learner showed some non-determinism with this input. As mentioned in section 4.3, larger input alphabets were harder to learn due to longer input sequences. Invalid inputs are therefore not learned, as they were less important for the model checking, as described in section 7.

7 Model Checking TCP Entities

7.1 NuSMV

NuSMV [12] is a tool for checking specifications for state-based models. In NuSMV, a state is modeled as a valuation of variables, each with a finite number of possible values, such as integers in a certain range, or an enumerated set of symbolic constants. A NuSMV-model describes variables and their domains, allowed initial states, and the possible transitions to new states. These transitions are defined as invariant predicates on the current state and the next state; Given the current state, all next states for which these invariants hold are legal. This allows for non-determinism; if multiple states are legal, NuSMV will check them all (if necessary).

Once specified, a model can be checked for specifications in temporal logics such as CTL and LTL [23]. If a specification does not hold, the tool will produce a counterexample if possible. For safety properties that do not hold, a counterexample should always be given. It can also be used to find deadlock states. Additionaly, NuSMV can be used to step through a model interactively. This can be used to observe the behaviour of the model and to debug it.

7.2 Simulating TCP Entities

For this thesis, the behaviour of TCP entities as found by active learning is model checked with NuSMV. This is done by composing a models of two TCP entities, and simulating communication between the two. The Mealy machines learned with *LearnLib* [27] are directly translated to NuSMV-models by a script. However, these models take abstract inputs and produce abstract outputs, which are not necessarily from the same domain. This makes them unsuitable for model checking, since an abstract output from one TCP entity cannot be used directly as an input by the other TCP entity. Therefore, the concretization of these Mealy machines is model checked instead. For this purpose, the mappers used during learning have been translated to NuSMV-modules as well.

7.3 Global Overview

The global structure of the NuSMV-model as used to model check the communication between two TCP entities is displayed in figure 7. In this model, the mapper component is used for concretization. It is split into the *input abstractor* and the *output concretizer*.

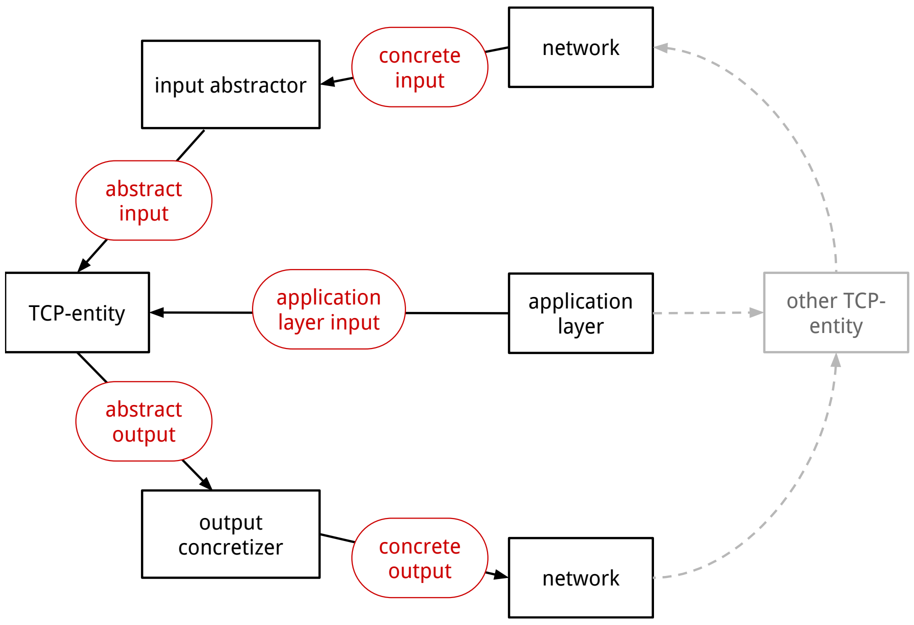


Figure 7: A schematic overview of the NuSMV-model. For simplicity, only half of the setup is shown in detail, as the model is symmetric and the other half is similar. Each rectangle is an instance of a NuSMV-module. The arrows represent the data flow. NuSMV-variables are not mentioned explicitly in this overview, but each arrow contains at least one variable, and modules may also contain variables.

A TCP entity may receive an input, either from the application layer or over the network. Concrete inputs from the network are first translated to abstract inputs by the input abstractor. Upon receiving an abstract input, the TCP entity will change its state and produce an abstract output. The output concretizer then translates this abstract output to a concrete one. Alternatively, a timeout may be given and no concrete output is produced (but the mapper state is still updated). If there is a concrete output, the network will then copy it to the input of the other TCP entity. This process will then repeat, until either side produces a timeout and communication stops. Whenever a TCP entity does not already have an input from the network, the application layer may choose to provide it with a system call input. This also implies that both entities may receive an input in parallel. Situations where both entities get a system call at the same time are thus also considered, as well as situations in which a system call is done when the other entity is handling a network input.

Concrete sequence and acknowledgement numbers in the real implementations are 32-bit numbers, but this makes the state space larger than necessary. During model checking, integers are therefore unsigned integers in the small range, from zero to seven. No information is lost because of this, as the learner assumes that different numbers behave similarly anyhow, if they are mapped to the same abstract value.

Abstract numbers are represented by symbolic constants. Flags are encoded as 'word' variables, which are simply vectors of bits. In theory, flags are also translated from abstract to concrete and vice-versa, but as this translation is simply the identity function, so the same variable is used for both abstract and concrete.

When no input or output is used, the corresponding variables are set to a default value. For symbolic constants, the constant NONE is used, whereas integers get the value -3 and flags are set to the zero-vector. We cannot rely on these values alone to determine whether an input or output is given. For example, when default concrete values are set, this may be either because of the absence of an output, or because of a timeout-output. Also there can be only one type of input at the same time: either a TCP segment, or a system call from the upper layer. For this reason, there are meta-variables denoting what the input and output for both TCP entities is: a TCP segment, a system call, a timeout, or NONE when the input or output is not set. Initially, all input and output types are NONE and all values are set to default. NuSMV also requires a model to explicitly state when a variable remains unchanged. As such, when no input is provided to a TCP entity or to a mapper module, its state is unchanged.

In addition to the modules mentioned above, NuSMV requires a main module. This module creates instances of all other modules, and all variables that need to be shared among modules.

7.4 Model Details

7.4.1 TCP entities

The NuSMV module used for TCP entities is shown in code snippet 2. Two modules are generated; their general structure and interface with the network is the same, but the states and transitions may differ. The structure of the models of the TCP entities are direct translations from the corresponding abstract Mealy machines. As such, the NuSMV code for states and transitions is generated from the dot-files describing the Mealy machines. This module controls the values of the abstract outputs, containing flags and a sequence and acknowledgement number. It also sets the output type.

Code snippet 2: The module of a TCP entity. The generated lists of states and transitions are omitted due to their length. All code shown is manually written and generated code is inserted at the indicated positions.

```
MODULE TCP1(actionInput, flagsInput, seqInput,
   ackInput, flagsOutput, seqOutput, ackOutput,
   inputType, outputType)
  VAR
```

```
    state : {
      states are enumerated here
      , ERROR};
  ASSIGN
    init(state) := s0;
    next(state) := case
      state = ERROR: ERROR;
      transitions are enumerated here
      inputType = NONE : state;
      TRUE: ERROR;
      esac;
    next(flagsOutput) := case
      flags outputs are enumerated here
      TRUE: 0b_00000;
      esac;
    next(seqOutput) := case
      sequence number outputs are enumerated here
      TRUE: NONE;
      esac;
    next(ackOutput) := case
      acknowledgement number outputs are enumerated here
      TRUE: NONE;
      esac;
    next(outputType) := case
      next(flagsOutput) != 0b_00000 & next(seqOutput)
         != NONE & next(ackOutput) != NONE: PACKET;
      inputType != NONE: TIMEOUT;
      TRUE: NONE;
      esac;
```

The state of a TCP entity is a symbolic constants, having the domain of states as described in the corresponding Mealy machines. Obviously, this module updates its state according to the Mealy machine, and the initial state is respected.

When no input is given, all outputs are set to their default value, and the state remains unchanged. If an input is given, but there is no transition matching the input, the state is set to an error-value. In the error-state, the model does not make any transitions or give any output anymore. This occurs when this input was not included in the input alphabet during learning. When checking a specification, one should check that this error-state is not reached: in that case, the specification cannot be checked for the inputs which are not learned.

7.4.2 The network and application layer

There are two network instances, one for each direction of network traffic. These instances simply move the output of one side to the input of the other side. The network modules are also suited for modeling an unreliable network by not moving output to input, or by tampering with variables. The network itself contains no state variables. There is a separate module for the application layer. It observes the inputs from the network, and if there is none, it may give a system call as input.

7.4.3 Mapper

The mapper is composed of three modules: the mapper state, an input abstractor and an output concretizer. The mapper state simply lists all state variables which are also used for active learning. It also sets the initial values accordingly, and it ensures that the state remains unchanged if necessary. A reference to the mapper state is given to the input abstractor and output concretizer, as they need to read and set state variables. When an input or output is given, the state variables are set by the input abstractor or output concretizer, respectively. The modules receive the relevant input or output types, to know when they should make a transition.

The construction of the input abstractor and output concretizer follows directly, from the transition relation of the mapper. Obviously, the mapper should be the same as the mapper used during the learning of the Mealy machine. The characteristic of the transition relation is extracted from the mapper as described in section 3. As it is a simple boolean expression, it is directly usable in NuSMS, apart from some small differences in syntax. This transition relation is translated to NuSMV-syntax and set as an invariant. In this way, when the current state and either the concrete or abstract symbol is defined, this invariant automatically constraints the corresponding other symbol and the next state. In this way, NuSMV properly handles all legal transitions.

7.5 Checking specifications

The properties to check are written in LTL. LTL formula encode properties about paths; it is satisfied if it holds for all possible paths. It is a superset of propositional logic, and a relevant operator is the *always*-operator, G. The formula $G\psi$ specifies that ψ should hold in any point in time.

The specification to be checked is that invalid inputs should never occur at either side. In case they do, there is something wrong in the implementation, assuming that the learned models and the mappers are correct. This property can be described by the formula

$$G(\mathit{absSeqInput1} \neq \mathit{INV} \wedge \mathit{absSeqInput2} \neq \mathit{INV} \wedge \mathit{absAckInput1} \neq \mathit{INV} \wedge \mathit{absAckInput2} \neq \mathit{INV})$$

7.6 Results

The specification has been checked, for both the situation with one client and one server, and for two clients. It was not satisfied, so invalid inputs could occur. As the model NuSMV provides a counterexample, these could be analysed to find the cause of the invalid input. Three main causes were found.

1. Errors in the mapper: The mapper should capture the notion of validity, but as described in section 5.2, this is done manually and might be incorrect. The model checking helped in finding these errors, and after fixing them, these issues were all solved.
2. Errors in the learned models: Re-running the trace in the learner-setting showed that the real implementation did not show these invalid inputs.
3. The notion of validity is not general enough. During learning, only sequential traces are analysed, but this cannot be generalised to concurrent systems.

The second issue shows an important difference between theory and practice for learning. The learning algorithm appeared to erroneously join states which could be reached after opening and closing a connection. As such, connections could be closed multiple times in the model, but not in the real implementation. As a result, multiple closing packets with a FIN flag could be sent instead of only one. The first packet closes the connection, making any further FIN-packets unexpected, hence invalid. The equivalence oracle should find any difference between a model hypothesis and the SUT, but it does not. As described in section 4, only random traces were used during these experiments. In some cases, the erroneous output only appears in traces of length seven or more. Then there is only a very small chance of finding these errors with random traces. They are found with the model checker, and these traces could be used as a counterexample for the hypothesis. In this way, these errors could be fixed, but this is a tedious approach if not automated. As such, not all errors found in this way are fixed.

The third issue is a more fundemental one. The notion of validity currently defines what should be sent to the TCP entity, in case of the learning setup by the learner. During model checking, it is about what should be sent to the model by the other side, i.e. the other model and the network. However, in the setting of learning, this notion is simpler than during model checking. Learning only occurs sequentially, and every output is a response to the preceding input. The model checker also checks concurrent situations, in which the two entities send a packet at the same time. The received packet is then not a response to the sent packet, as the packets crossed on the network. To prevent these invalid inputs, a third category is therefore needed, next to only *valid* and *invalid*: outputs that should not be sent as a response to the previous input, but that might occur due to these concurrency issues. Network issues such as packet loss are not modeled for this thesis, but might have the same problems. To exclude these issues in a quicker but more restricting way, non-sequential traces can be excluded. This can be done by adding an invariant which prevents concurrency, or with the following LTL-formula that only looks at traces without concurrent inputs:

$G\neg((\mathit{inputType1} = \mathit{PACKET} \ \vee \ \mathit{inputType1} = \mathit{ACTION}) \wedge (\mathit{inputType2} = \mathit{PACKET} \ \vee \ \mathit{inputType2} = \mathit{ACTION})$
$\rightarrow$
$G(\mathit{absSeqInput1} \neq \mathit{INV} \wedge \mathit{absSeqInput2} \neq \mathit{INV} \wedge \mathit{absAckInput1} \neq \mathit{INV} \wedge \mathit{absAckInput2} \neq \mathit{INV})$

However, this prevents checking some states which are possible in practice. In particular, the situation with two clients can hardly be model checked, as both entities should get a *connect*-input at the same time to establish a connection.

8 Future work

When combining learning of abstract models and model checking, some practical problems arise. The mapper language solves many of these, but there is room for improvement. For example, the learner now picks concrete values with a constraint solver, when translating abstract inputs to concrete. Constraint solvers usually pick these values deterministically, and this does not give a good test coverage. This is a common problem in test case generation [6], and progress in this field could also improve learning with abstraction. Also, the behaviour of TCP protocols is also hard to describe as a Mealy machine due to timing and re-transmission of segments. Learning models in a less restrictive way might help modeling these aspects, such as labeled transition systems [28]. Ideally, learning of timed automata should be possible to perfectly infer the behaviour of TCP implementations.

Another field of improvement is on equivalence testing. The experiments in this thesis could be repeated with improved equivalence testing algorithms. For example, Lee & Yannakakis [18] designed an algorithm that should find counterexamples more efficiently by looking at states in the hypothesis.

The learned behaviour of TCP implementations is now mainly restricted to establishing and closing connections, where the possibility of finding bugs is limited. By improving the learning setup, more aspects of the behaviour may be modeled, and more interesting results might be found for TCP. The abstract input alphabet with only *valid* and *invalid* parameters is also somewhat limited. Simpler input parameters may be used, which are less open to interpretation, similar to the current abstract output alphabet. The logic to determine the validity of inputs could then be expressed in the specifications, based on these simple inputs, instead of directly through the mapper. Ideally, mappers could even be constructed in an automated way by the learner as well [4].

The learning and model checking setup could also be applied to other protocols and systems. It has proven hard to connect the SUT and the learner, in such a way that a Mealy machine could be learned. For other systems, the learned models may be more reliable, and increasing the number of abstract inputs might be more feasable.

Furthermore, many interesting ways of combining learning, testing and model checking are possible. Model based testing, for example, focuses on generating only input traces neccesary to test the specifications. This can also be done in the setting of learning and model checking. Testing traces which are counterexamples to the specifications is now done manually, so another improvement is to automate this, and to improve the equivalence oracle in this manner.

9 Conclusions

Active learning and model checking can complement each other in terms of strengths and weaknesses. The distance between an abstract model and an error-prone implementation is a fundemental problem, which can be tackled by using learned models. An abstract alphabet with *valid* and *invalid* input parameters was used, and models for TCP clients and servers were learned.

The resulting models have been model checked to verify that no *invalid* parameters were used. Model checking on learned systems worked well, and composing models into networks was succesful. This does rely on the correctness on the learning algorithm. In particular, equivalence testing with random input sequences proves ineffective. Model checking provides another way of finding relevant counterexamples, but reliability of the learned models is important nonetheless: satisfaction of a specification cannot be extrapolated the the real system if the model is not correct.

Using abstract models for model checking gives some additional problems compared to using concrete models, as abstract models cannot be composed into networks directly. We have solved these problems by defining a mapper language, which can translate both from concrete to abstract and vice versa. This technique should be generalisable to learning and model checking many other systems with abstraction.

Model checking TCP implementations has proven feasible, but difficult. practical problems make it difficult to scale up the input alphabet, and the resulting models. No bugs in the implementations have been found, but tackling these problems may increase the effectiveness of model checking. Within this thesis, the abstract inputs for TCP are now expressed in terms of valid and invalid. If events happen sequentially, this is sufficient for model checking, but to model check concurrent systems more extensively, a broader notion is needed.

References

[1] ISO/IEC JTC 1. Information technology Syntactic metalanguage Extended BNF, stage 90.60. Technical Report 14977:1999, Geneva, Switzerland, 1996.

[2] Fides Aarts. *Tomte: bridging the gap between active learning and real-world systems.* PhD thesis, Radboud Univeriteit Nijmegen, 2014.

[3] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. Automata learning through counterexample guided abstraction refinement. In *FM 2012: Formal Methods*, volume 7436 of *LNCS*, pages 10–27. Springer Berlin Heidelberg, 2012.

[4] Fides Aarts, Falk Howar, Harco Kuppens, and Frits Vaandrager. Algorithms for inferring register automata. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 202–219. Springer, 2014.

[5] Fides Aarts, Bengt Jonsson, Johan Uijen, and Frits W. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, 46(1):1–41, 2015.

[6] Bernhard K Aichernig and Percy Antonio Pari Salas. Test case generation by ocl mutation and constraint solving. In *Quality Software, 2005.(QSIC 2005). Fifth International Conference on*, pages 64–71. IEEE, 2005.

[7] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, November 1987.

[8] Thomas Ball and Sriram K Rajamani. The s lam project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, pages 1–3. ACM, 2002.

[9] Fred Baumgarten, Matt Welsh, Alan Cox, Tuan Hoang, and Bernd Eckenfels. Netstat man page. `http://linux.die.net/man/8/netstat`.

[10] Philippe Biondi. Scapy documentation. `www.secdev.org/projects/scapy/doc/`, 2010. version 2.1.1.

[11] Georg Chalupar, Stefan Peherstorfer, Erik Poll, and Joeri De Ruiter. Automated reverse engineering using lego. In *8th Usenix Workshop on Offensive Technologies (WOOT 2014). Usenix*, 2014.

[12] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002.

[13] James C Corbett, Matthew B Dwyer, John Hatcliff, Shawn Laubach, Corina S Păsăreanu, Ro Bby, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448. IEEE, 2000.

[14] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Learning fragments of the TCP network protocol. In Frédéric Lang and Francesco Flammini, editors, *Formal Methods for Industrial Critical Systems - 19th International Conference, FMICS 2014, Florence, Italy, September 11-12, 2014. Proceedings*, pages 78–93, 2014.

[15] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186. ACM, 1997.

[16] Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model generation by moderated regular extrapolation. In *Fundamental Approaches to Software Engineering*, pages 80–95. Springer, 2002.

[17] Malte Isberner and Bernhard Steffen. An abstract framework for counterexample analysis in active automata learning. In *Proc. ICGI*, 2014.

[18] David Lee and Mihalis Yannakakis. Testing finite-state machines: State identification and verification. *Computers, IEEE Transactions on*, 43(3):306–320, 1994.

[19] Lars Lockefeer, David M Williams, and Wan J Fokkink. Formal specification and verification of tcp extended with the window scale option. In *Formal Methods for Industrial Critical Systems*, pages 63–77. Springer, 2014.

[20] Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next generation learnlib. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 220–223. Springer, 2011.

[21] Madanlal Musuvathi, Dawson R Engler, et al. Model checking large network protocol implementations. In *NSDI*, volume 4, pages 12–12. Citeseer, 2004.

[22] Terence Parr. *The definitive ANTLR 4 reference: building domain-specific languages.* Pragmatic Bookshelf Raleigh, 2013.

[23] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[24] J. Postel. Transmission control protocol. RFC 793 (Standard), sep 1981. Updated by RFCs 1122, 3168.

[25] J. Postel (editor). Transmission Control Protocol - DARPA Internet Program Protocol Specification (RFC 3261), September 1981. Available via `http://www.ietf.org/rfc/rfc793.txt`.

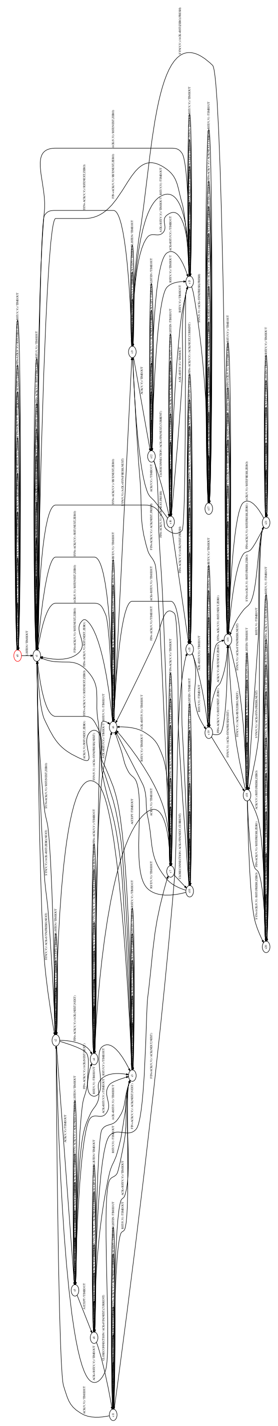
[26] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco3 Documentation.* TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.

[27] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009.

[28] Michele Volpato and Jan Tretmans. Active learning of nondeterministic systems from an ioco perspective. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 220–235. Springer, 2014.

Appendices

A Server model



B Client model