Insert Title Here

by

Mathew Kuthur James

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2026

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Meta Meta Professor, Cheriton School of Computer Science, University of Waterloo
Supervisor:	Nancy Day Professor, Cheriton School of Computer Science, University of Waterloo
Internal Member:	Meta Meta Professor, Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	Meta Meta Professor, Cheriton School of Computer Science, University of Waterloo
Other Member(s):	Meta Meta Professor, Cheriton School of Computer Science, University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This is the abstract.

Lorem ipsum dolor sit amet

Acknowledgements

I would like to thank all the little people who made this thesis possible.

Dedication

This is dedicated to the one I love.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

background - discuss formal verification, TLA+, Alloy, Dash, model-checking etc.

goal of the project:

1. build a translator for Dash+ and Alloy
2. evaluate performance

Chapter 2

Background

2.1 Alloy

2.2 Dash

2.3 TLA+

Objects whose value do not change with the traces of the model are called CONSTANTS in TLA+. Often, these CONSTANTS are given values in the .cfg file accompanying the .tla file fed into the TLC model checker. These objects correspond to instances generated by the Alloy Analyzer when a Dash+ model is run after translation to Alloy.

2.4 Configuration generation

In a paper [5] by Cunha et. Somson, the authors introduce a tool called Blast, which enables the use of TLA+ to find these objects during the model-checking phase, rather that having the user set specific values manually in the .cfg file.

The .tla file and .cfg file are augmented with annotations read by blast. These annotations determine the type and scope of the configurations to be tested. Blast produces a modified .tla file and a modified .cfg file.

Blast performs the following procedure on the input files:

1. For every CONSTANT C in the original .tla file, Blast adds a VARIABLE V_C in the modified .tla file. (CONSTANTS with no type annotations are ignored by Blast)

2. Blast makes these variables keep the same values during the trace, using the UNCHANGED modifier in TLA+ (equivalent to appending / $V'_C == V_C$ to every transition).

3. Blast adds the type of the variable to the TypeOK relation in TLA+. Blast ignores CONSTANTS whose type is not specified in the annotation, which remain as CONSTANTS in the output files produced by Blast.

4. Blast constructs compound types (such as sets, relations, tuples, et. cetera.) from the annotation, which are constructed using basic types (such as Nat). Blast identifies basic types by the use of all caps identifiers.

5. Blast modifies the Init relation by adding $V_C \in C$, where C is now a CONSTANT.

6. The value of C in the .cfg file is a set whose size is determined by the scope annotations read by Blast.

Chapter 3

Translating Alloy elements to TLA+

Consider the following snippet of Alloy code:

```
sig B
{

}
sig A
{
        f : set B
}
```

Signatures are modelled as sets, whose elements are defined in the .cfg file. The symbols used for the sets are shared by the .tla and .cfg files.

The above code is translated into the following code in TLA+

```
VARIABLES A, B, f

Init == /\ A \in SUBSET A_set
  /\ B \in SUBSET B_set
  /\ f \in SUBSET (A_set \X B_set)

Next == UNCHANGED <<A, B, f>>
```

The A, B and f are variables whose value is determined by the TLC model checker in the Init step, with no changes occurring afterwards. Once a valid configuration is

determined, the Next relation does not change the variables. TLC explores all possible initial configurations.

3.1 Signatures

Signatures can have the following modifiers applied to them:

- subtyping
- multiplicity
- abstraction
- inheritance
- enumeration

Subtyping

```
sig A {}
sig B in A {}
sig C extends A {}
sig D extends A {}
```

In the example above, "in" represents inclusive subtyping - every atom of A may be an atom of B, and every atom of B is an atom of A. Meanwhile, "extends" functions similarly to "in", with the added constraint an atom of A can be either an atom of C or an atom of D or neither. An atom of A cannot be both an atom of C and an atom of D.

Analysis of the signature hierarchy shows that A is the base signature, thus the atoms are explicitly defined for A in the .cfg file.

```
// in the .cfg file
A_set == {"a0","a1","a2"} // the default size of a signature is 3
```

```
// in the .tla file
VARIABLES A, B, C, D

TypeOK == /\ A \in SUBSET A_set
  /\ B \in SUBSET A_set
  /\ C \in SUBSET A_set
  /\ D \in SUBSET A_set

sig_hierarchy ==
/\ B \subseteq A
/\ C \subseteq A
/\ D \subseteq A
/\ C \intersection D = {}

Init == TypeOK /\ sig_hierarchy
Next == UNCHANGED <<A, B, C, D>>
```

The use of the + operator enables inclusive subtyping from multiple parents. The translation reuses the general logic for inclusive subtyping.

```
sig X {}
sig Y {}
sig Z in X + Y {}

sig_hierarchy ==
/\ Z \subseteq X \union Y
/\ C \subseteq A
/\ D \subseteq A
// X \intersection Y = {} is not specified explicitly because this is implicitly true
```

Multiplicity

The multiplicities applied to signatures are:

- none

- lone
- one
- some

The translation of these multiplicities follow the scheme described in the Fields subsection.

Abstraction

When a signature is made abstract, all its atoms belong to extensions. In the TLA translation, the sets associated with abstract signatures are defined in the .cfg file with explicit atoms, but the variable associated with them is subject to the constraint of being an empty set.

Enumeration

Enumeration fixes the possible values of the signature and orders them. Enums are logically equivalent to creating an abstract sig, with several one-sigs extending the abstract sig. The translation method for enums reuses the methods for one-sigs and abstract sigs.

```
enum X {A, B, C}
```

is equivalent to

```
abstract sig X {}
one sig A, B, C extends X {}
```

In addition, enums include an implicit import of the ordering module.

3.1.1 Fields

In addition to the default "set" multiplicity, fields have the following multiplicities:

- none
- lone
- one
- some

There are two possible translation schema for these multiplicities - using set comprehension, and using the inbuilt "Cardinality" function. Though the latter is more straightforward, the use of Cardinality is costlier than set comprehension in terms of computation time for running the models.

```
sig B
{

}
sig A
{
 f : one B
}

// using set comprehension
multiplicity_constraint ==
/\ \E x \in f : TRUE
/\ \A x \in f : \A y \in f : x = y

// using cardinality
multiplicity_constraint ==
/\ Cardinality(f) = 1
```

The general translation of multiplicities using set comprehensions can be constructed from two elementary predicates, which are then composed to form the TLA+ translations of multiplicities:

```
// S is a binary relation
no_elements(S) == // needs correct expression with testing
more_than_one_element(S) == // needs correct expression with testing


sig B
{

}
sig A
{
 f : one B
 g : some B
 h : set B
 i : none B
 j : lone B
}


multiplicity_constraint ==
/\ ~no_elements(f) /\ ~more_than_one_element(f)
/\ ~no_elements(g)
/\ TRUE // no constraints on h
/\ no_elements(i)
/\ ~more_than_one_element(j)
```

In addition to the basic multiplicities, the modifier "disj" can be applied

Multi-relations

Multi-relations consist of the following form of relations:

```
sig A {}
sig B {}
sig C{
 f : A -> B
}
```

Alloy supports multi-arity fields, though fields with an arity of more than 3 are discouraged (reference - Alloy read-the-docs).

In both TLA+ and Alloy, cartesian products of sets are flat, i.e. A x B x C is a set containing tuples of the form (a,b,c). The translator constructs such sets when translating multi-relations from Alloy.

```
sig A {}
sig B {}
sig C {
 f : set A -> set B
}

VARIABLES A, B, C, f

Init == /\ A \in SUBSET A_set
  /\ B \in SUBSET B_set
  /\ C \in SUBSET C_set
  /\ f \in SUBSET (C_set \X A_set \X B_set)

Next == UNCHANGED <<A, B, C, f>>
```

The logic used to translate multiplicities is applied as before. Certain combinations of multiplicities which result in unsatisfiable models in the original Alloy code, will result in translations which produce an error when run in TLC.

3.1.2 Multiplicities

Explain the four multiplicities with two basic types, the use of quantifiers over cardinality as optimizations and set comprehension.

This section needs to be folded into previous sections since multiplicities apply to fields and signatures, and is context-dependent elsewhere.

3.2 Expressions/Formulae

a, b and c are n-ary relations, f is a functional relation, r, r1 and r2 are binary relations, s is a set, e1 and e2 are expressions

Alloy	TLA+	Comments
b[a]	tla	box join
a.b	tla	dot join
a -¿ b	tla	cartesian product
s ¡: a	tla	domain restriction
s :¿ a	tla	range restriction
a & b	tla	intersection
r1 ++ r2	tla	relational override
#a	Cardinality$a$	cardinality
a + b	a	
union b	union	
a - b	tla	difference
=	=	equality (= has another meaning as well in context)

3.3 Facts

3.4 Predicates/Functions

Chapter 4

Translating Dash+ elements to TLA+

A Dash+ model is written with Alloy expressions and code embedded inside Dash structures. The Dash structures are used to model a transition system. Since TLA+ supports transitions natively, the dash elements are translated directly to TLA+, without an intermediate Alloy step.

Consider the following Dash model:

```
state root
{
 env event ee {}
 event ei {}
 default state s1{
  conc a {
   default state s1a1 {}
   state s1a2 {}
  }
  conc b {
   default state s1b1 {}
   state s1b2 {}
  }
 }
 state s2 {
```

```
 conc p {}
 conc q {}
}
trans t1 {
 from s1
 goto s2
}
trans t2 {
 on ee
 from s1a1
 goto s1b2
 send ei
}
trans t3 {
 from s1
 goto s1b1
}
trans t4 {
 from s1
 goto s1a1
}
trans t5 {
 from s1
 goto s1a2
}
}
```

This model is used to demonstrate the translation of following features of Dash:

- State hierarchies
- Transitions
- Concurrency
- Big-step small-step semantics
- Internal and Environmental Events

4.1 State Hierarchy

The Dash model shown above has the following state hierarchy:

```
state root
{
 default state s1 {
  conc a {
   default state s1a1 {}
   state s1a2 {}
  }
  conc b {
   default state s1b1 {}
   state s1b2 {}
  }
 }
 state s2 {
  conc p {}
  conc q {}
 }
}
```

There are three types of states in Dash:

- Leaf states (s1a1, s1a2, s1b1, s1b2, p, q)
- OR states (root, a, b)
- AND states (s2, s1)

During translation, every state is associated with a TLA definition, consisting of a set of strings. Each leaf-state is associated with a singleton set consisting of a string derived from the fully qualified name of the leaf-state, while non-leaf states are associated with sets made from the union of the sets associated with the leaf states that compose the non-leaf states.

```
_root_s1_b_s1b1 == {"root/s1/b/s1b1"}
_root_s1_b_s1b2 == {"root/s1/b/s1b2"}
_root_s1_a_s1a2 == {"root/s1/a/s1a2"}
_root_s1_a_s1a1 == {"root/s1/a/s1a1"}
_root_s2_q == {"root/s2/q"}
_root_s1_b == _root_s1_b_s1b1 \union _root_s1_b_s1b2
_root_s2_p == {"root/s2/p"}
_root_s1_a == _root_s1_a_s1a1 \union _root_s1_a_s1a2
_root_s2 == _root_s2_p \union _root_s2_q
_root_s1 == _root_s1_a \union _root_s1_b
_root == _root_s1 \union _root_s2
```

Since TLA+ requires that a definition exists before it is used, the translator sorts the states in such a manner that a state is translated only after all the child states of that state is translated. Since the fully qualified names of a state is unique, no two states will be translated to TLA+ definitions of the same name.

4.2 Dash variables

The following variables are present in the .tla translation:

- _conf - this is a set of leaf state strings, representing the states that a snapshot is in
- _trans_taken - this is a string that holds the name of the transition taken to arrive at the current snapshot
- _stable - this is a boolean value that is TRUE iff the current snapshot is stable
- _scopes_used - this is a set of leaf state strings, representing the scopes used in the current big step
- _events - this is a set of events (represented as strings) that are active in the current snapshot

```
VARIABLES _trans_taken, _conf, _scopes_used, _stable, _events

\* string literal representations of transitions taken, which are the values taken by t
```

```
_root_ei == "root/ei"
_root_ee == "root/ee"
_environmental_events == {_root_ee}
_internal_events == {_root_ei}

\* string literal representations of transitions taken, which are the values taken by t
_taken_root_t2 == "root/t2"
_taken_root_t1 == "root/t1"
_taken_root_t4 == "root/t4"
_taken_root_t3 == "root/t3"
_taken_root_t5 == "root/t5"
_none_transition == "[none]"

\* state constraints
_valid_conf(arg) == arg \in _root_s2_p \union _root_s2_q \union _root_s1_a_s1a1 \union
_valid_trans_taken(arg) == arg \in {_taken_root_t2,_taken_root_t1,_taken_root_t4,_taken
_valid_scopes_used(arg) == _valid_conf(arg)
_valid_stable(arg) == arg \in BOOLEAN
_valid_events(arg) == arg \in SUBSET (_environmental_events \union _internal_events)

_valid_unprimed == _valid_conf(_conf) /\ _valid_trans_taken(_trans_taken) /\ _valid_sco

_valid_primed == _valid_conf(_conf') /\ _valid_trans_taken(_trans_taken') /\ _valid_sco

_Init == _valid_unprimed
/\ _conf = _root_s1_a_s1a1 \union _root_s1_b_s1b1
/\ _stable = TRUE
/\ _scopes_used = {}
/\ _trans_taken = _none_transition
/\ _events \intersection _internal_events = {}

_Next == _valid_primed /\ _small_step
```

Each transition and event has a formula with its fully qualified name as a string, which is used to assign values to _trans_taken and _events, respectively. In addition to existing transitions, there is a default "[none]" string to represent stutter steps, where no transitions are taken. The use of square brackets prevent naming clashes with other transition names.

Since TLA+ requires that any formula involving a primed variable v start with a clause either of the form $v' \in exp$ or $v' = exp$, the formula _valid_primed is the first clause of _Next. This allows a separation between the parts of the translation concerned with restricting the variables to valid values (_valid_primed), and the part of the translation that concerned with ensuring the variables values are correct (_small_step).

This is a departure from the translation to Alloy, and is required since TLA+ explores the set of all possible values for a variable. Without a validity constraint, the state space becomes infinite, and TLC cannot be run on the translation.

Since the first snapshot has no prior transition, the value assigned to _trans_taken in _Init is _none_transition. Since the first snapshot is always stable, the value assigned to _stable is TRUE, and to _scopes_used is the null set. Since internal events are generated by transitions, and no transition has been taken yet, there is a clause that states that no internal events are present in _events.

The initial value of _conf is determined by the following algorithm, executed during the translation.

- Let $f(S)$ be the set of leaf states to be in, when starting at S as the root state. $f(root)$ is the initial value of _conf, where $root$ is the root state of the model.
- $f(S) = S$ if S is itself a leaf state, with no children
- $f(S) = C_d$ if S is an OR state, where C_d is the default child state of S
- $f(S) = \bigcup_{i=1}^{n} f(C_i)$ if S is an AND state

A possible alternate choice for such a translation involves the use of string literals for all states (leaf and non-leaf), with transitions written in such a way that the value of conf is always legal within the state hierarchy of the model. In such a scenario, the state of the model can be determined using only the set membership operator. The number of states that TLC needs to explore is a function of the semantics of the input model, and unaffected by the choice of representation of the states.

However, the amount of memory required to represent a particular configuration in the original scheme is less than or equal to the amount of memory needed to represent that configuration in the alternate scheme. All leaf states contribute a string literal to the conf set in both schemes, while non-leaf states contribute a string literal only in the alternate scheme. Since the memory used is lesser in the original scheme, we chose to base the translator design on it, instead of on the alternate scheme.

4.3 Transitions

Transitions are modelled as boolean formulae in TLA+. In Dash, the transitions are defined inside the state from which the transitions is taken. In the translation, all transitions are defined at the top-level. Each transition has the following:

- (implicit) - the 'from' state
- goto - the 'to' state
- when - the guard, translated as an Alloy expression
- do - the action, translated as an Alloy expression
- on - the triggering event
- (emitting event - todo: fill this up post discussion)

Consider a simple Dash model:

```
state root
{
 default state s1 {

 }
 trans t1
 {
  from s1
  goto root
 }
}
```

This is translated into TLA+ as:

```
 VARIABLES _trans_taken

 _taken_root_t1 == "root/t1"
 _none_transition == "[none]"
```

```
_pre_root_t1 == ((_conf \intersect [root/s1]) /= {})
_post_root_t1 == (_trans_taken' = _taken_root_t1) /\ (_conf' = ((_conf \ {root/s1}) \u
_root_t1 == _pre_root_t1 /\ _post_root_t1

_some_transition == _pre_root_t
_some_pre_transition == _root_t
```

Each transition is translated into the following elements:

- _taken_[transition-name]
- _pre_[transition-name]
- _post_[transition-name]
- _enabled_[transition-name]
- _[transition-name]

The transition is translated into a formula named after the fully qualified name of the transition, and is split into a pre-condition and a post-condition. The pre-conditions are used in _some_pre_transition, which is the disjunction of all the preconditions, used to test if any transition is enabled.

4.3.1 Transition Taken

The _taken_[transition-name] refers to a definition of a string that represents the transition, obtained from the fully qualified name of the transition. The fully qualified name depends on the location of definition of the transition in the model, and is unique to the transition.

When the transition is taken, the _trans_taken variable takes on the value of the string defined by _taken_[transition-name]. This does not affect the result obtained after running TLC on the translated model, and serves primarily as an aid to analyze traces produced by TLC. Since no transition is taken until the _Next formula is true, the initial value of the _trans_taken variable is _none_transition, which defines a standard string that represents no transition.

```
VARIABLES _trans_taken

_taken_root_t == "root/t"
_none_transition == "[none]"

_all_trans_taken == {_taken_root_t,_none_transition}
_TypeOK == (_trans_taken \in _all_trans_taken)

_Init == _TypeOK /\ (_trans_taken = _none_transition)
```

4.3.2 Stability

(Assumption: Big step and small step semantics explained in background)

Each snapshot is either stable or unstable, marking the start and end of a big step. The stability of the current snapshot is required to define the preconditions of transitions, while the postconditions determine the stability of the following snapshot. This is tracked using a boolean variable called _stable.

```
VARIABLES _stable

_TypeOK == (_stable \in BOOLEAN)

_Init == (_stable = TRUE)
```

4.3.3 Scopes

Assumption: the term "scope" is explained in background

The scopes used up to the current snapshot is tracked using the _scopes_used variable. This variable has the same type as the _conf variable, since there is a one-one correspondence between the set of scopes and the set of transitions. The _scopes_used variable is used when defining the preconditions to ensure that within a big step, only transitions in

orthogonal scopes are taken. The value of the variable in the next snapshot is defined in the postconditions of the transitions.

```
VARIABLES _scopes_used

_all_scopes_used == _all_conf

_TypeOK == (_scopes_used \subseteq _all_scopes_used)

_Init == _TypeOK /\ (_scopes_used = {})
```

4.3.4 Stability after the transition

To check whether a transition will be enabled for a given value of _scopes_used, a parameterized formula is associated with each transition, called _enabled_[transition-name]. When the next snapshot is stable, none of the transitions are enabled, which is represented by the _next_is_stable formula. This is defined as the conjunction of the negations of the _enabled_[transition-name] formulae associated with each transition.

```
_enabled_root_t(_arg_scopes_used) == // insert logic here

// arguments change to include events later

_next_is_stable(_arg_scopes_used) == ~(_enabled_root_t(_arg_scopes_used))
```

4.3.5 Stutter and small step

A small step consists of the disjunction of all the transition formula, along with a stutter formula, where none of the variables change. Assumption: TLA+'s approach to stutter is explained, as is the separate mechanism to track inherent stutter in TLA+ vs explicit stutter. A stutter happens only when no other transitions are possible, which is determined using the _next_is_stable formula.

```
 _some_transition == _pre_root_t
 _some_pre_transition == _pre_root_t
 _stutter == (_trans_taken' = _none_transition) /\ UNCHANGED <<_conf,_stable,_scopes_us
 _small_step == _some_transition \/ (_stutter /\ (~_some_pre_transition))
```

Putting it all together, for a basic model with simple transitions and states, we have:

```
VARIABLES _trans_taken, _conf, _scopes_used, _stable

\* State literals, represented as sets of strings. Leaf-states become strings and non-l
_root_s2 == {"root/s2"}
_root_s1 == {"root/s1"}
_root == _root_s1 \union _root_s2

\* string literal representations of transitions taken, which are the values taken by t

\* string literal representations of transitions taken, which are the values taken by t
_taken_root_t == "root/t"
_none_transition == "[none]"

\* parameterized formulae to check if transitions are enabled
_enabled_root_t(_arg_scopes_used) == TRUE

\* negation of disjunction of enabled-formulae
_next_is_stable(_arg_scopes_used) == ~(_enabled_root_t(_arg_scopes_used))

\* Translation of transition root/t
_pre_root_t == ((_conf \intersect [root/s1, root/s2]) /= {})
_post_root_t == (_trans_taken' = _taken_root_t) /\ (_conf' = ((_conf \ {root/s1,root/s2
_root_t == _pre_root_t /\ _post_root_t

\* Small step definition
_some_transition == _pre_root_t
_some_pre_transition == _pre_root_t
```

```
_stutter == (_trans_taken' = _none_transition) /\ UNCHANGED <<_conf,_stable,_scopes_use
_small_step == _some_transition \/ (_stutter /\ (~_some_pre_transition))

\* type restrictions on variables
_all_conf == _root_s1 \union _root_s2
_all_trans_taken == {_taken_root_t,_none_transition}
_all_scopes_used == _all_conf
_TypeOK == (_conf \subseteq _all_conf) /\ (_stable \in BOOLEAN) /\ (_scopes_used \subse

\* initial values for variables
_Init == _TypeOK /\ (_conf = {}) /\ (_stable = TRUE) /\ (_scopes_used = {}) /\ (_trans_

\* Next relation
_Next == _TypeOK /\ _small_step
====
```

4.4 Events

This is a draft subject to rewrites:

This list is an outline of what to explain:

- The events variable
- mapping the events to string forumale
- standard formulae for the set of internal events and the set of env events
- the use of filters using set intersection, to refer to internal events and environmental events in the transitions
- The policy used to clear the active events after big steps
- The mechanism used to prevent runaway cascades of repeated small steps taken endlessly
- non-determinism of which small step is taken in the event of multiple possible ones, though this is to be done in the background, and referred to here

4.5 Actions

Actions are translated from Alloy, and form boolean expressions which are integrated into the transition postconditions.

4.6 Guards

Actions are translated from Alloy, and form boolean expressions which are integrated into the transition preconditions.

4.7 Notes (Ignore)

fundamental features of dash:

description of a state hierarchy description of transitions initial constraints

States: control states OR states AND states

OR and AND states have multiple child states in an OR state, the system is in one or more of its child states

question: is it exclusive? no paper clearly says one or more

priority is given for transitions that leave the parent state over those that leave the child state

what does this mean? if there is one transition that leaves to a different control state, what then?

AND state - in all of them, and their sub-states are independent

questions: AND state out of date, how t deal with citations for new stuff?

- conc

question: enforcing the addition of multiple states or inference from just the leaf states

transition templates and add-ons - punt

A model - is it the description, or the series of snapshots?

Transitions take the model from one snapshot to the next

is there a canonical order to snapshots?

stable snapshot - no enabled transitions enabled transition - one which has met guards and Events

unstable snapshot - not stable

big step - list of transitions from one stable snapshot to the next small step - a single transition from an unstable snapshot big step is also a list of small steps

Big step maximality: at most one transition per concurrent state

question: if big step maximality is enabled, is there a guarantee that all big-steps will be of finite length?

CONCURRENCY: says that a small step is a single transition

EVENT Lifeline: events last till the next big-step

variable lifeline: variable changes are allowed to cascade through small steps

priority of transitions: leaving parent has priority over leaving child

question: can this affect which transitions are taken at all? yes, cancelling

Frame problem: what si the equivalent in TLA+

non-determinism: same env input given in a big step can result in different ending snapshots

priority consistency

proving all of this formally

meta-question: does all this need full explanation

IMPORTANT: if nothing mentioned variable unchanged

Chapter 5

Translating LTL properties to TLA+ invariants

5.1 Commands

Chapter 6

Evaluation

The evaluation of the translator is broadly divided into three sections: Correctness, performance of the model, and performance of the translator.

The translator produces a tla+ file from a .dsh file, which can be analyzed statically for complexity metrics, such as the number of lines of code. This can be compared to the number of lines required for a manual translation for specific models, to observe how efficient the translator is at capturing the semantics of the model.

Through the analysis of simple models, certain optimization techniques are revealed and implemented, in an iterative process. However, a weakness of such a system is overfitting to certain kinds of models, whereby optimizing for small, hand-written models may have undesirable effects on the efficiency of other types of models.

The correctness of the translation is a binary correct/not-correct. Since the space of all possible models cannot be reasoned about, a determination of correctness is made via:

- custom unit tests of known models
- fuzz-testing using randomly generated models

To check for specific instances, a pair (model, instance) is preprocessed into model', which is a model that tests specifically for that instance. Model' is translated and tested for the only property present.

TODO: find an analogue

The following metrics are used to judge the performance of the model:

- time taken to execute, as compared to the time on the Analyzer
- clock cycles to execute (usually tightly correlated with time)
- trace lengths generated during execution
- state space of the translated model, in comparison to the original model in the Analyzer

TODO: formalize the notion of specific model types in terms, and discuss how the performance is affected

Chapter 7

Related Work

A paper [3] by A Cunha compares the performance of Alloy and TLA+ implementations of the spanning tree algorithm (used in cite Blast paper). The authors constructed a model of the algorithm by hand for both languages, and compared the performance of the TLA+ implementation with the Alloy implementation. They concluded that in general, TLA+ outperforms Alloy when dealing with simple configurations with large scopes, while Alloy is more suitable for models with complex configurations whose validity takes a non-trivial amount of work by the model-checker to prove. The findings of this paper were limited to a single model, written manually in both languages. The thesis involves comparing Alloy models to their machine-translated TLA+ models.

Chapter 8

Conclusion

Summary of the thesis and possible future work.

References

[1] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, revised edition, 2016.

[2] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, MA, United States, second edition, 2002.

[3] Nuno Macedo and Alcino Cunha. Alloy meets tla+: An exploratory study. 03 2016.

[4] Jose Serna, Nancy Day, and Shahram Esmaeilsabzali. Dash: declarative behavioural modelling in alloy with control state hierarchy. *Software and Systems Modeling*, 22, 08 2022.

[5] Paul Somson and Alcino Cunha. Verifying Multiple TLA+ Configurations with Blast. In *2025 IEEE/ACM 13th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 135–145, 2025.

APPENDICES

Appendix A

Matlab Code for Making a PDF Plot

A.1 Using the Graphical User Interface

Properties of Matab plots can be adjusted from the plot window via a graphical interface. Under the Desktop menu in the Figure window, select the Property Editor. You may also want to check the Plot Browser and Figure Palette for more tools. To adjust properties of the axes, look under the Edit menu and select Axes Properties.

To set the figure size and to save as PDF or other file formats, click the Export Setup button in the figure Property Editor.

A.2 From the Command Line

All figure properties can also be manipulated from the command line. Here's an example:

```
x=[0:0.1:pi];
hold on % Plot multiple traces on one figure
plot(x,sin(x))
plot(x,cos(x),'--r')
plot(x,tan(x),'.-g')
title('Some Trig Functions Over 0 to \pi') % Note LaTeX markup!
legend('{\it sin}(x)','{\it cos}(x)','{\it tan}(x)')
hold off
```

```
set(gca,'Ylim',[-3 3]) % Adjust Y limits of "current axes"
set(gcf,'Units','inches') % Set figure size units of "current figure"
set(gcf,'Position',[0,0,6,4]) % Set figure width (6 in.) and height (4 in.)
cd n:\thesis\plots % Select where to save
print -dpdf plot.pdf % Save as PDF
```