Forging Forge:
Contributions to the Forge Programming Language

Alexander Varga*

May 15, 2020

Abstract

Forge is a programming language for the specification and verification of software systems and systems in general. The author has made many contributions to the early development and maintenance of Forge. This report first outlines some of their major areas of contribution to the core language and then explores some of Forge's interesting new features like support for state transition modelling, bounds-based partial instances, and strategies. Finally it dives into the details and theory of the frameworks put in place for developing conservative transformations of specifications and extensions to the Forge language. In keeping with the spirit of Forge, the author seeks to both prove and test the correctness of the transformations and extensions themselves.

*Department of Computer Science, Brown University. Advised by Tim Neslon.

Contents

1 The Forge Language

Forge[1] is a modelling language which is heavily based on the Alloy [2] modelling language and copies its syntax. Before I tell you what Forge and Alloy are, I'll show you:

```
sig Node { edges: set Node }
pred graph {
    some Node
    all a: Node | some b: Node | b in a.edges
    edges = ~edges
    no iden & edges
}
fun e[x: Node, y: Node] : Node->Node { x->y + y->x }
pred triangle[a: Node, b: Node, c: Node] {
    e[a, b] + e[a, c] + e[b, c] in edges
}
pred mesh {
    graph
    all a, b: Node | e[a, b] in edges => {
        two c: Node | triangle[a, b, c]
    }
    all a: Node | let B = a.edges {
        ^(e[B, B] & edges) = e[B, B]
    }
}
run mesh for 7 Node
```

This document is not intended to be a Forge manual, but I will quickly walk through what the above *means* to give an idea of what Forge can *express*. The first line says that there exists a type of thing called a `Node`, and that it has as a property a set of `edges` to other Nodes. Then a predicate is defined, which is a statement that may or may not be true. This particular predicate is true if and only if the collection of nodes and edges that exist represent an undirected graph. The first line of `graph` requires that some Node exists. The second line uses the language of First-Order Logic to require that all Nodes have an edge to some other Node. The third and fourth lines use a different sort of language to talk about `edges` explicitly as a relation on Nodes. They require that reversing all edges results in the same set of edges, and there are no edges from a Node to itself, respectively. We could have equivalently expressed the third line by saying that for all edges, there exists an edge going the other direction, and here we use these pairs of edges to represent an undirected edge.

After the `graph` predicate comes a function called `e`, which takes two Nodes and returns a tiny relation on Nodes. This relation consists of two edges going either way between the given Nodes, which as we've said represents an undirected edge. The `triangle` predicate takes arguments like a function can. It is true *of* its inputs if there exist undirected edges between all of them, representing a triangle. One may at this point be confused about what I mean when I say that something *exists*. After all, I said a Node *is* a thing, an edge is a *relation* between things, and a `triangle` is some

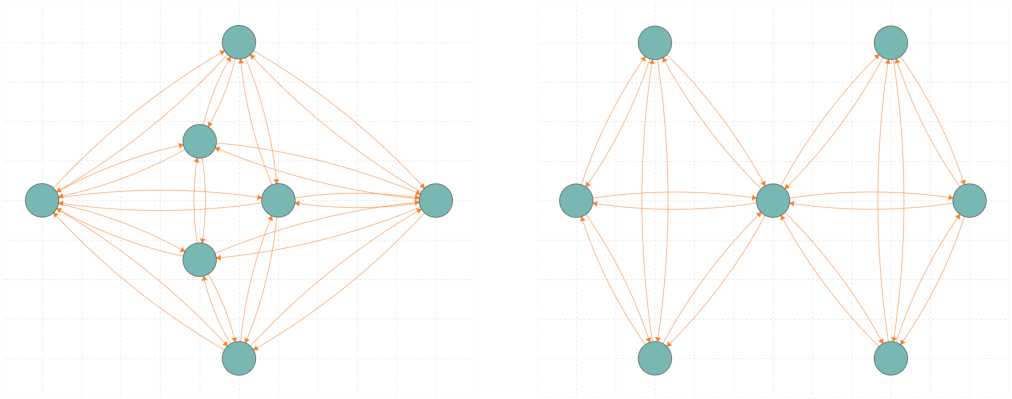
[1] `http://www.forge-lang.com`

collection of things that satisfy some *constraints*. Forge's two main purposes may clear up some of this confusion. Forge's first purpose is to model things; the way in which these things are modelled or represented is up to the specification writer. In this case, we chose to model Nodes as "atomic" entities, edges as explicit relations, and triangles as implicit triples of nodes satisfying a property. Forge's second purpose is to consider many different possible "worlds" – collections of things which may exist in unison – and the relations and properties between and of those things. Before we sink too far into the meaning of existence or the existence of meaning, let's continue.

With this next predicate, we get to the content of this specification. We wish to reason about and explore triangular meshes like those used in computer graphics. The first line of the `mesh` predicate calls upon the `graph` predicate, so we know that meshiness requires graphiness. Then it says that for every pair of Nodes forming an edge, there are exactly `two`Nodes which form triangles with them. In other words, every edge belongs to exactly 2 triangles. Let's skip the next constraint for now and focus on the last line of the specification. This tells Forge to look for possible worlds satisfying the `mesh` predicate. An important part of every Forge specification, and the subject of much of this report, is its "bounds". There are infinitely many worlds for Forge to consider, so we tell it to consider only those with at most 7 Nodes.

When we run this predicate, Forge finds us a beautiful mesh, or rather the Nodes and edges representing it, which looks like two pyramids with pentagonal bases stuck together at those bases. The resulting shape satisfies our constraints: all edges glue two triangles together with no extra triangles hanging off the main shape. This is the model I had in mind when I wrote this specification, but Forge finds an additional mesh I hadn't considered. It finds two tetrahedra stuck together at one corner... Is this a mesh? According to our specification it is! Since this is not what I had in mind, the specification was refined with the last constraint which we skipped to match my intentions.

The above illustrates the capabilities of the Forge language as well as its limitations. One limitation is simply the need for bounds on the worlds that Forge explores, but that is the price of expressivity. The more important limitation is that Forge can't yet read our minds, and it can be difficult to express what we mean, let alone implement it in software. The only way to gain confidence in the correctness of one's specification is to explore the worlds that Forge finds. The automation of this process is the topic of the last section of this report.



(a) Legal Mesh (b) Questionable Mesh

Figure 1: Two instances satisfying the first iteration of the above specification, rendered in Sterling.

2 Contributions to the Core Language

Forge was developed to be a language for teaching software modelling, and was successfully used in the 2020 offering of the Brown University Computer Science Department's "Logic for Systems" course. Development of the language began the summer before the author joined the team, and had already reached a mostly-functioning point. The author's contributions at the time of writing include helping to get the language to a usable state in time for the course and maintaining the language during the course, as well developing some more novel features. Forge is implemented as a Racket `#lang` which has greatly enabled the incremental and experimental development of the language.

This section lays out some of the authors contributions to the core of the Forge programming language.

2.1 Syntax

Alloy's syntax was recreated with minor tweaks in Racket[2]. The lexer and parser were developed with the "Beautiful Racket"[3] and "brag"[4] packages and languages. These tools output Racket syntax which can readily be converted to Forge's underlying s-expression syntax using Racket's extensive macro-system. Extra care was taken to integrate the language into the DrRacket development environment, to allow for useful error messages and highlighting, bound variable hints, formatting, etc. These features were instrumental in the accessibility of the language for use in "Logic for Systems".

2.2 Visualizer

By default Forge displays models in a new browser-based front-end called Sterling.[5] The new visualizer has a lot of potential for future work enabling various custom visualizations for models. In addition to other minor visualizer features, the author contributed to the evaluator on the Forge side, which allows for the evaluation of expressions and formulas in the context of a model and its formulas and predicates.

2.3 Et Cetera

Additional contributions include functions, let expressions, integers, bug fixes, documentation, testing, code examples, and development processes.

[2] `https://racket-lang.org`
[3] `https://docs.racket-lang.org/br/`
[4] `https://docs.racket-lang.org/brag/index.html`
[5] `https://sterling-js.github.io`

3 States and Transitions

Custom syntax and support for states, transitions, and traces was added to Forge to ease the modelling of stateful systems over time and help users take advantage of some of Forge's optimizations.

Consider the following sig declaration:

```
sig MyState {
    field1: ...
    field2: ...
}
```

3.1 State Predicates

We can write a state predicate parameterized over `MyState` to succinctly express properties of `MyState`s. Such a predicate implicitly takes an additional argument `this` of type `MyState`. Inside such a predicate, fields of `this` can be referenced directly like `field1` instead of `this.field1`. The full field can be referenced like `@field2`. As an example, a state predicate like:

```
state[MyState] myInit[a: A] {
    a in field1
    no @field2 & iden
}
```

turns into a normal predicate like:

```
pred myInit[this: MyState, a: A] {
    a in this.field1
    no field2 & iden
}
```

3.2 Transition Predicates

Transition predicates are very similar to state predicates but take two implicit arguments: `this` and `this'`. Fields of `this'` can be accessed like `field1'`. We can now easily describe how a transition should update a state. For example, a transition predicate like:

```
transition[MyState] addA[a: A] {
    a not in field1
    field1' = field + a
    field2' = field2
}
```

turns into:

```
pred addA[this, this': MyState, a: A] {
    a not in this.field1
    this'.field1 = this.field1 + a
    this'.field2 = this.field2
}
```

Note that in contrast to imperative languages, we must explicitly declare that `field2` doesn't change between states with the line `field2' = field2`. Without this line, `field2` could change to any value whatsoever. Since this is a common error, transition predicates raise "Underspecified Field" errors if any of the paramterized sig's fields are absent.

3.3 Traces

Traces are a succinct way to express sequences of state transitions. They take advantage of Forge's optimizations to effectively hard-code in a sequence of states. Because of this, they are able to generalize to non-linear timelines. A tree-like trace over a state `S` can be defined and run like:

```
trace<|S, myInit, myTran, myTerm, myInv|> MyTrace: tree { ... }
run<|MyTrace|> { ... }
```

- `S`: The parameterized state sig.
- `myInit`: A state predicate true of all initial states.
- `myInit`: A transitions predicate true of all transitions.
- `myTerm`: A state predicate true of all terminal states.
- `myInv`: A state predicate true of all states.
- `tree`: A strategy describing the timeline.

The above ultimately expands to something like:

```
one sig T extends TraceBase {}
facts[T] T_pred {
    some tran => {
        S    = tran.S+S.tran
        init = tran.S-S.tran
        term = S.tran-tran.S
    } else (lone S and init = S and term = S)
    all s: init          | myInit[s]
    all s: S, s': s.tran | myTran[s, s']
    all s: term          | myTerm[s]
    all s: S | myInv[s]
    ...
}
pred T_fact { all t: T | T_pred[t] }
inst T_inst { tran is tree }
run { T_fact ... } for T_inst
```

4 Partial Instances

As mentioned above, a crucial part of any Forge specification is the bounds placed on it. We call the mathematical object composed of all relevant formulas and bounds a relational logic "problem". When a Forge specification is run, it is converted into a problem and sent to the Kodkod [6] relational logic solver. We already showed that simple integer upper bounds on sigs can be given, but bounds are much more flexible than this. Kodkod can handle arbitrary lower and upper bounds on relations. This hypothetically lets us explicitly say which tuples in a relation like `edges` must, may, or mustn't exist, and this is exactly what Forge instances allow.

As a running example, let's say we've designed a fragment of a mesh and would like to find full meshes that complete it with a limited number of Nodes. We wish to specify an upper bound on `Node` which identifies specific node atoms and a lower bound on `edges` defining the edges of our fragment. Since all Nodes are effectively equivalent and interchangeable in models, there's no way to define, or single out, any single atom with formulas alone. Sure we could define new `one sig`s extending `Node`, but we shouldn't have to change the signature of our relational logic problem just to explore specific examples or run tests. This is where Forge instances can make our lives easier:

```
inst myFragment {
    Node in a + w + x + y + z + (n1 + n2 + n3)
    edges ni a->(w+x+y+z) + w->x + x->y + y->z + z->w
}
run mesh for myFragment
```

The constraints in instance declarations differ from those in formulas in several ways, but effectively act as an imperative sublanguage acting on bounds rather than formulas. Both lines in the instance above are assignments; their RHS is evaluated using the aforementioned evaluator, and then used to update the bounds of the LHS. The first constraint uses `in` to set an upper bound for `Node` and the evaluator makes up new nodes with the names given. These names are then used in the second constraint, which uses the inverse `ni` to define a lower bound on edges.

The instance sublanguage can only express constraints that can be enforced purely by bounds, which leads to some interesting limitations. Nevertheless, they can still accomplish a great deal and help maintain separation of concerns between the formulas and bounds of a problem. Instances have been effectively use to enable students of "Logic for Systems" to test their specifications against partial[6] instances. Refer to the Forge documentation for a full listing of supported constraints.

[6]The `exactly` keyword can be used to tell Forge to assert that an instance is fully specified/concrete.

5 Strategies

5.1 Bounds Breaking

When modelling with Forge, certain patterns present themselves which can be expressed as common relational properties like reflexivity, injectivity, or linearity. Strategies allow Forge users to easily annotate relations with such properties and let Forge enforce them. In addition to convenience, strategies provide performance improvements by operating on bounds instead of just using formulas, whenever possible. In fact, these are the performance optimizations that the aforementioned traces take advantage of. When a trace on states is declared to follow a linear sequence, this sequence can be hard-coded into the bounds of the problem to reduce problem size. The author has developed a framework for the definition and use of strategies that can be soundly combined and composed.

Several unmentioned strategies have been implemented like `tree` and `acyclic`, as well partial variants of these and inverse and variadic strategy combinators. The interface to using strategies is built into instances and integrates with their other bounds-constraining operations. Please refer to the documentation for a full listing and descriptions.

Whenever possible, a strategy should be enforced on a relation by constraining bounds. However, there are cases where this isn't possible and the strategy should fall back to a default implementation using purely formulas, effectively acting as a simple macro. We call the act of constraining the bounds of a relation "breaking" the relation, in reference to symmetry breaking. When might a strategy be forbidden from breaking? A simple case is when that relation has already been broken. Consider a binary transition relation on some state sig which has been declared linear and has been constrained to some arbitrary relational constant. If this relation is broken to be linear again with a *different* linear relational constant, we may end up with an unsatisfiable problem that was otherwise satisfiable, which is unacceptable. The obvious solution is to limit strategies to one break per relation. However, another solution which Forge uses when possible is to combine breaks together. The property of a relation being both a tree and a cotree (the inverse of a tree) is equivalent to the property of being linear, so a tree break and a cotree break on the same relation can be replaced by a single linear break.

Now let's consider a trickier way in which breaking multiple relations could go wrong. Consider two sigs A and B as well as two relations $f : A \to B$ and $g : B \to A$ declared to be functions:

```
sig A { f : set B }
sig B { g : set A }
run {} for f is func, g is func, #A=2, #B=2
```

In addition to the necessary formulas, the `func` strategy can break the bounds of f by choosing some $a : A, b : B$, putting $a \to b$ in the lower bound of f, and removing $a \to b'$ from the upper bound of f for all $b' \neq b$. This seems complicated, but formalizes precisely the statement "assume without loss of generality that $f(a) = b$"! But now lets consider what happens if we do the same for g, and "assume without loss of generality that $g(d) = c$". The problem is that we *do* in fact lose generality. Consider the formula `no a: A | g(f(a)) = a` under the possible resulting bounds. Depending on our choice of $a, c : A$, $b : d : B$ and the resulting bounds, this formula may or may not be satisfiable. This means such an optimization is incorrect, and only one of f or g can be broken.

The solution we arrived at is to annotate breaks with a sort of interface we call a "break-graph" describing how they can be soundly composed. While a full account of the semantics, algorithm,

and proof of correctness for break-graphs and their composition is beyond the scope of this report, I'll attempt to give a high-level overview of it's flavor.

5.2 Problem Equivalence

First, let's step back and consider the comparison of relational logic problems in general. Let's say we've defined some transformation on problems, be it a simple macro or a complex optimization, and we wish to prove it "correct". For our purposes, a transformation is correct if the original and resulting problems are equivalent, but what does "equivalent" mean in this context? If a transformation only acts on formulas, equivalence testing is easy because we can compare the original and resulting formula sets *within the same bounds*. In fact, this is exactly how we test the equivalence of two predicates inside of Forge. If p and q are two predicates over the same set of signatures and relations, we can test their equivalence up to the bounds in Forge by simply running a `check` on $p \leftrightarrow q$. These predicates are equivalent if and only if no counterexample models are found.

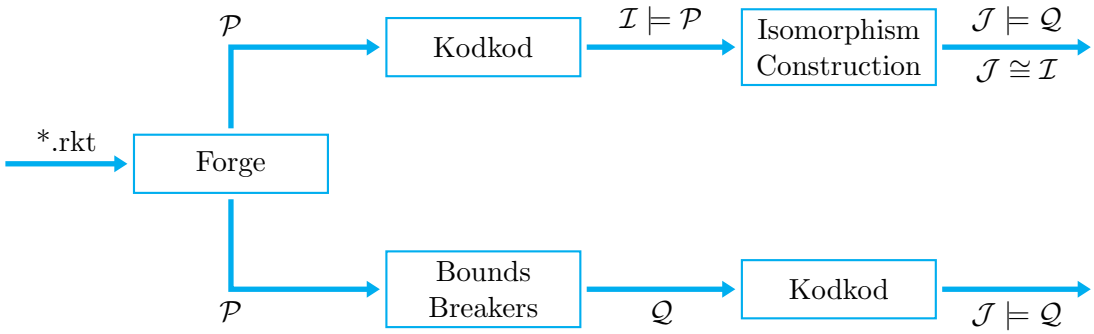
On the other hand, a general transformation may in fact change the bounds of a problem. Checking the equivalence of two problems with different bounds is trickier and must be done *outside* of Forge. What went wrong with the example breaking both f and g above is that the original and resulting problems were not equisatisfiable. However, the true test for problem equivalence is more more general: problems $\mathcal{P}$ and $\mathcal{Q}$ are equivalent if and only if they satisfy the same instances up to isomorphism. Testing this for two problems of unknown origin is the topic of the next section, but for now we return to the correctness of Forge's strategy composition.

5.3 Proof Sketch of Strategy Correctness

A full proof of correctness of Forge's strategy framework would require both proofs that every strategy meets the criteria promised by its break-graph (yet to be explained) as well a proof that given the correctness of all strategies, their composition is always correct. We sketch a proof of the latter here.

Let $\mathcal{P}$ be some problem and $\mathcal{Q}$ be the problem resulting from breaking some of its relations. We say that the transformation done by the strategy framework is sound if for all instances $\mathcal{I}$ satisfying $\mathcal{P}$, there exists an isomorphic instance $\mathcal{J}$ satisfying $\mathcal{Q}$. Completeness, the converse of this, follows simply from the correctness of the strategies and the fact that $\mathcal{Q}$ is stricter than $\mathcal{P}$ in that it has tighter bounds, so $\mathcal{J}$ itself satisfies $\mathcal{P}$.

We can imagine the soundness condition as imposing a hypothetical execution phase – after, lexing, parsing, macro-expansion, problem construction, solving – where the strategy framework must construct from some $\mathcal{I}$ satisfying $\mathcal{P}$ an isomorphic $\mathcal{J}$ satisfying $\mathcal{Q}$. This hypothetical phase never actually needs to run, but it's existence means that instead of solving the transformed problem, we could have just solved the original problem and converted it into something the transformed problem might have returned.



In other words, given any model $\mathcal{I}$ satisfying $\mathcal{Q}$'s formulas, we must be able to permute the atoms of $\mathcal{I}$ to fit inside the bounds imposed by $\mathcal{Q}$. This permutation must be constructed for each sig by one of the strategies breaking a relation over it, taking into account the sigs in that relation which have already been permuted. Break-graphs serve as an interface to specify the ways in which permutations from certain sigs can be percolated to permutations on other sigs. The strategy composition algorithm chooses a maximal subset of strategies such that, during the hypothetical Isomorphism Construction phase, permutations can be effectively percolated across all sigs. All other strategies are enforced in the default way with formulas. The example with f and g fails precisely because of a cycle in dependencies between the f and g, deadlocking the isomorphism construction phase.

A break-graph has the sigs of the relation it breaks as nodes, with undirected edges between them. The guarantee that a strategy makes via its break-graph is that given

1. An assignment to the relation satisfying the strategies formulas
2. Permutations for any set of sigs such that no two are connected by an edge

the strategy can construct permutations for the *remaining* sigs such that the assigned relation under these permutations fits within the bounds set by the strategy. The strategy composition algorithm uses these break-graphs to construct a dependency graph on the relations, and then selects a maximal set of relations such that the restriction of the dependency graph to the set is acyclic. This guarantees that there is some topological sort on the relations defining an order in which the isomorphism construction phase can build a permutation.

6 Testing Problem Equivalence

The last section described a fairly complex problem transformation and outlined a proof of its correctness based on the equivalence of the original and resulting problems. In this section, we consider the case of comparing two problems $\mathcal{P}$ and $\mathcal{Q}$ (over a common set of sigs and relations) of unknown origin for equivalence. We must find a way to compare problems from outside of Forge while knowing nothing of the contents of these problems. Even when we do know the origin of the problems and have a supposed proof of their equivalence, this approach is useful for exhaustively double checking their equivalence in case the correctness proof or implementation of the transformation is incorrect.

As stated above, $\mathcal{P}$ and $\mathcal{Q}$ are equivalent when they have the same satisfying instances up to isomorphism. Letting `Forge(_)` be a function that returns a generator of insances satisfying some problem, a naive algorithm for testing problem equivalence might be:

```
def equivalent(P, Q):
    for I in Forge(P):
        someJ = False
        for J in Forge(Q):
            if isomorphic(I, J): someJ = True
        if not someJ: return False
    for J in Forge(Q):
        someI = False
        for I in Forge(P):
            if isomorphic(J, I): someI = True
        if not someI: return False
```

This would certainly work, but is both compute and memory intensive, and requires the development of a custom isomorphism detection algorithm. Here we lay out an approach that allows us to utilize Forge and Kodkod's existing solving capabiltites. In order to do this, we need some way of reifying the meta-level concept of instance isomorphism from *outside* of Forge into something *inside* of Forge. We accomplish this with characteristic formulas.

6.1 Characteristic Formulas

Given an instance $\mathcal{I}$, we wish to produce a characteristic formula $\chi(\mathcal{I})$ which fully describes $\mathcal{I}$ and can be added to a problem. When we say that a formula should "fully describe" an instance, what we really mean is that $\mathcal{I}$ should be the unique instance satisfying $\chi(\mathcal{I})$, up to isomorphism:

$$\mathcal{J} \models \chi(\mathcal{I}) \Leftrightarrow \mathcal{J} \cong \mathcal{I}$$

Let $\{A_0, ..., A_n\}$ be the sigs of $\mathcal{I}$, let $A_i^{\mathcal{I}}$ be the assignment of each A_i in $\mathcal{I}$, and let $m_i = |A_i^{\mathcal{I}}|$ be their cardinalities. Let $\{r_0, ..., r_l\}$ be the relations of $\mathcal{I}$ and let $r_i^{\mathcal{I}}$ be the assignment of each r_i in $\mathcal{I}$. We define:

$$\chi(\mathcal{I}) := \exists a_{0,0}...a_{0,m_0} : A_0, ..., a_{n,0}...a_{n,m_n} : A_n \Bigg(\bigwedge_{0 \leq i \leq n} \Bigg(A_i = a_{i,0} + ... + a_{i,m_i} \wedge \bigwedge_{j \neq k : A_i} a_{i,j} \neq a_{i,k} \Bigg) \wedge \bigwedge_{0 \leq i \leq l} r_i = \texttt{reify}(r_i^{\mathcal{I}}) \Bigg)$$

This creates an existential formula with a quantifier variable for each atom of $\mathcal{I}$ which exactly defines all sigs and relations of $\mathcal{I}$. $\texttt{reify}(r_i^{\mathcal{I}})$ expresses the relational constant $r_i^{\mathcal{I}}$ in terms of the quantifier variables $a_{i,j}$. We don't prove the correctness of this formulation here (i.e. that $\mathcal{J} \models \chi(\mathcal{I}) \Leftrightarrow \mathcal{J} \cong \mathcal{I}$), but we do test this at the end of the section.

6.2 Finding Isomorphic Instances

With this tool, we're ready to tackle problem equivalence. We only focus on one direction as the converse is the same. We wish to show that all instances satisfying $\mathcal{I}$ also satisfy $\mathcal{J}$ up to isomorphism:

$$\forall \mathcal{I} \Big(\mathcal{I} \models \mathcal{P} \Rightarrow \exists \mathcal{J} \Big(\mathcal{J} \cong \mathcal{I} \text{ and } \mathcal{J} \models \mathcal{Q} \Big) \Big)$$

We can ask Forge to give us all instances $\mathcal{I}$ satisfying $\mathcal{P}$ and then must check if some isomorphic $\mathcal{J}$ exists satisfying $\mathcal{Q}$. It suffices to ask Forge if $\mathcal{Q} + \chi(\mathcal{I})$ is satisfiable:

$$\Big(\mathcal{J} \cong \mathcal{I} \text{ and } \mathcal{J} \models \mathcal{Q} \Big) \Leftrightarrow \Big(\mathcal{J} \models \chi(\mathcal{I}) \text{ and } \mathcal{J} \models \mathcal{Q} \Big) \Leftrightarrow \Big(\mathcal{J} \models \mathcal{Q} + \chi(\mathcal{I}) \Big)$$

Our final theorem is:

$$\mathcal{P} \equiv \mathcal{Q} \Leftrightarrow \Bigg(\forall \mathcal{I} \Big(\mathcal{I} \models \mathcal{P} \Rightarrow \exists \mathcal{J} \Big(\mathcal{J} \models \mathcal{Q} + \chi(\mathcal{I}) \Big) \Big) \text{ and } \forall \mathcal{J} \Big(\mathcal{J} \models \mathcal{Q} \Rightarrow \exists \mathcal{I} \Big(\mathcal{I} \models \mathcal{P} + \chi(\mathcal{J}) \Big) \Big) \Bigg)$$

Our final algorithm is[7]:

```
def equivalent(P, Q):
    for I in Forge(P):
        if not Forge(Q+chi(I)): return False
    for J in Forge(Q):
        if not Forge(P+chi(J)): return False
    return True
```

Delving one layer deeper into testing tests, we should convince ourselves that our χ implementation itself is correct. We can do this by noting that $\mathcal{I} \cong \mathcal{I}$, so it must be the case that all $\mathcal{I}$ which satisfy $\mathcal{P}$ also (uniquely) satisfy $\mathcal{P} + \chi(\mathcal{I})$.[8] With that, we have properly Forged Forge.

[7] In practice, we do in fact implement this from a scripting language, as opposed to Racket. This helps avoid stateful bugs by encapsulating each run of the solver in its own process.

[8] In fact, if χ is correct, this is a good way to test the quality of Forge's symmetry breaking.

7 Conclusions

This completes an overview of the author's contributions to the early development of the Forge language, some of Forge's interesting features, and the higher-order testing frameworks in place.

There is significant room for growth and experimentation of the language, enabled by Racket's flexible language-development facilities. Possible avenues of exploration include the definition and improvement of new strategies, improved integration of strategies with instances, automatic conversion of predicates to instances, an auto strategy utilizing staged evaluation and BDDs, improved coordination between strategies and Kodkod's symmetry breaking, encodings of algebraic data types, and modal logic syntax for traces. Ways in which the existing code base could be improved include making it more declarative so that specifications can be treated more readily as relational logic problems as in this report and reducing redundancy of representations of code through better use of stage polymorphism.

I'd like to express my appreciation to my advisor Tim Nelson for his guidance and inspiration, the rest of the Forge team for their support, and the other "Logic for Systems" TAs and students for their feedback. They have all been a joy to work with.

References

[1] Theophilos Giannakopoulos et al. "Towards an Operational Semantics for Alloy". In: *FM 2009: Formal Methods*. Ed. by Ana Cavalcanti and Dennis R. Dams. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 483–498. ISBN: 978-3-642-05089-3.

[2] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012. ISBN: 0262017156.

[3] Vajih Montaghami and Derek Rayside. "Extending Alloy with Partial Instances". In: June 2012, pp. 122–135. DOI: 10.1007/978-3-642-30885-7_9.

[4] Vajih Montaghami and Derek Rayside. "Staged Evaluation of Partial Instances in a Relational Model Finder". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Ed. by Yamine Ait Ameur and Klaus-Dieter Schewe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 318–323. ISBN: 978-3-662-43652-3.

[5] Ilya Shlyakhter. "Generating effective symmetry-breaking predicates for search problems". In: *Discrete Applied Mathematics* 155.12 (2007). SAT 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing, pp. 1539–1548. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2005.10.018. URL: http://www.sciencedirect.com/science/article/pii/S0166218X06004604.

[6] Emina Torlak and Daniel Jackson. "Kodkod: A Relational Model Finder". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Orna Grumberg and Michael Huth. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 632–647. ISBN: 978-3-540-71209-1.