The Report committee for David Allen Liu

Certifies that this is the approved version of the following report:

A Demonstration of Applying Alloy to Mechanical Synthesis of Electromechanical Systems

APPROVED BY SUPERVISING COMMITTEE:

Supervisor:

Sarfraz Khurshid

Vijay Garg

A Demonstration of Applying Alloy to Mechanical Synthesis of Electromechanical Systems

by

David Allen Liu, B.S.E.E

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2015

I dedicate this report to my family, which has advocated for my exploration into a Master's degree since I entered undergrad in 2005. To Marie, who has bookended my journey into the graduate school with support, friendship, and adventure to help push me to the completion of this degree.

Acknowledgments

My entrance into graduate school had a rocky start. In 2012, I was unsure about my career direction and unhappy with where I was at in life–little did I know at the time that the missing link was the journey of a graduate degree. Left with a critical decision that year, I decided to drop everything and go full steam ahead into this Master's Degree. Throughout the course of the degree (and this report), I've become a much better engineer and individual after learning how to prioritize tasks, deliver on what is being asked for, and designing for quality.

In addition, during the course of getting this degree I've been thrown many a problem from both the career and finance fronts. Being in graduate school forced me to learn how to focus on the future and less about the short term, as many of the challenges threatened my degree progression along the way. During that time I learned a lot about myself, but I also learned a lot about my friends as well. Many of the people in my life stepped up to help me lend me support when I most needed, and without them I don't know if I would have ever gotten through the program at all.

First and foremost, I would like to thank Dr. Jonathan Sessler. His insight and advice helped me enter the academic world again, and his friendship helped me re-center myself after life events previous to my acceptance into the graduate program. I would like to thank The Intel Corporation for their support during the duration of my internship–they helped provide the environment for me to explore and develop my skills to complete this report. To Lane Holloway, who's advice has

helped me navigate both the academic and industry domains, and who's friendship has kept me from on the road to success. To Cheng Lee, who has provided feedback throughout the last six years to keep my grades, career, and cooking on point.

David Allen Liu

The University of Texas at Austin
December 2015

A Demonstration of Applying Alloy to Mechanical Synthesis of Electromechanical Systems

David Allen Liu, M.S.E
The University of Texas at Austin, 2015

Supervisor: Sarfraz Khurshid

Formal verification methods have traditionally been used in industry for proofs of functional correctness; more recent advances in their use have given rise to additional applications in new domains. Electromechanical systems such as automotive transmissions or robotics have relied heavily on software and mechanical modeling to operate and test the given design; modeling tools that support automated analysis such as Alloy allow formal analysis techniques to apply to this domain, and also enable synthesis. Specifically, Alloy provides a language and tool-set that can abstractly represent and solve for real-world instances, which enable engineers to develop a deeper understanding of the systems they are building.

This goal of the report is to demonstrate the potential of applying the Alloy tool-set for modeling and analysis to assist in the synthesis and operational correctness of software-driven mechanical designs. A survey of literature has been included to demonstrate the foundation of concepts and previous research done in the area, spanning both formal verification and electromechanical design fields. The report also includes two small but illustrative case studies that attempt at mechanical synthesis (or design seeding) using Alloy, and report the abstraction methods and techniques that succeeded in demonstrating design synthesis efforts.

Contents

List of Figures

Chapter 1

Introduction

Formal verification, and its use thereof, has been at a relatively constant use rate in the software world today. From a software testing perspective, formal verification has provided methods of assurance on codebase confidence and operation, allowing software systems to be validated to a general confidence level. In hardware, gate and logic level designs are represented in Boolean expressions and checked with *Satisfiability* (SAT) solvers [1] to determine the logical soundness of a system. The usages presented above in both software and hardware industries have been relatively constant and unvarying; not many usages outside of these two cases provided are well published outside of these realms.

Mechanical systems are typically checked in constraint driven software such as *Solidworks* and other computer assisted design (CAD) software. Designing a mechanical assembly within such programs consists of adding pivots, parallel or orthogonal surface joins, and angle constraints to attempt to define the motion of assembled system. Mechanical simulations are operated on top of these constraint rule sets, and various types of analysis such as *computational fluid dynamics* (CFD) and stress analysis may be run on the generated models for satisfiability purposes. Electromechanical systems such as an automobile bridge the two areas of such domains by requiring the attached software system to have a fundamental understand-

ing of the mechanical operation. As robotics and other forms of electromechanical automation become more prevalent, the logic modeling and mechanical modeling of such systems becomes paramount in the face of an ever more integrated society with such robotics.

The goal of this report is to bridge these seemingly disjoint topics, and provide a process in which to use formal verification techniques via SAT-solving tools to provide a method of seeding and validating mechanical designs and systems. The report looks to use one of the leading academic SAT-based modeling tools, Alloy, and demonstrate generated mechanical designs validity and synthesis viability. Through the use of relational logic and model abstraction in Alloy [2], both software and mechanical systems might be both validated and seeded at simple abstract levels.

Chapter 2

Literature survey

This chapter presents a quick overview of work in the area of formal verification, SAT solving, electro-mechanical systems, and mechanical seeding. The topics covered create the basis for the object set forth in this report.

2.1 Formal Verification

Formal verification, or the structured exhaustive checking of a model, has its roots in mathematics. Originally used to check finite (and sometimes infinite) mathematical models, the conjectures that were presented with axioms and type theory were heavily experimented with during the early 1900s. In the publication *Formally Verified Mathematics*, the tumultuous history of formal verification is detailed from the 1400s to current day. The biggest point of contention was that proving the validity of the solutions required mechanical demonstration of the proof itself, in addition to the problem of proving a "complete" solution [3]. The start of this movement stemmed from the original finding that mechanical proofs could be used; Avigad and Harrison state "one of the most significant advances in mathematical logic around the turn of the 20th century was the realization that ordinary mathematical arguments can be represented in formal axiomatic systems in such a way their correctness can be

verified mechanically, at least in principle." [3] However, the critics to the use of mechanical demonstrations of proofs had relevant concerns. Without a full exhaustive proof documented, the certainty of the solution's completeness is not considered very strong.

2.2 Problems with Mathematical-only proofs

In consideration of what experts consider "complete" proofs, the mathematical closure offered by mathematically derived proofs were lauded as being the standard of due diligence, but even these beliefs had pitfalls. Avigad and Harrison state in their publication that "a book written by Lecat in 1935 included 130 pages of errors made by major mathematicians up to 1900, and even mathematicians of the stature of J.E. Littlewood have published faulty proofs", and that "Every working mathematician must routinely deal with inferential gaps, misstatements, missing hypotheses, unstated background assumptions, imprecise definitions, misapplied results, and the like." [3] Therefore, even the most mechanically proven examples are subject the mechanism's validity to the problem at hand.

An example of this is the definitive proof of Fermat's last theorem. Until 1995, the proof to verify this theorem was considered incomplete. Fermat originally conjectured this equation in 1637: No three positive integers a, b, and c can satisfy the equation $a^n + b^n = c^n$ for any integer value of n greater than two. In 1994, Andrew Wiles published a proof that had claimed to solve it, but was still plagued by the use of the Kolyvagin-Flach approach, which invalidated his solution. The true solution came a year later in 1995, in which he used his original method to finally have a suitable proof [3]. While this proof's invalid time was limited to about a year, others in the area of group and ring theory lasted significantly longer, such as Daniel Gorenstein's classification of finite simple groups in 1983, which took until 2001 to prove out in a 1,221-page proof by Michael Aschbacher and Stephen Smith [3].

The validity of similar proposed solutions has caused many in the field to look elsewhere for confidence in their equations or models, and created a movement to automate the proof process; specifically solutions other than mechanical proofs. Since mechanical proofs involve exhaustive searching, one of the other issues is the problem statement itself. If one were to express a problem p, and look for every solution to problem p recursively, one would end up with the following problem: "if p is not provable, it will run fruitlessly forever" [3]. In this case, the mechanism for enabling the searching of the proofs cannot cover the fact that the solution set of a problem is not recursive in nature. This method's abilities are limited to only verifying if the proposed solution is correct, and not what other solutions exist based upon the problem's definition. Therefore, a system is needed that can display the proofs based upon the problem definition and other semi-rigid constraint parameters.

2.3 Initial creation of interactive Formal Verification

The initial attempts at making a more intuitive and useful method for using formal verification came in the 1960s with Bruijn's *Automath* [3]. Automath differed from exhaustive solution systems at the time by listing the problem in propositions (or 'categories'), and other novel notations for describing a proof. Additional work was done by Jutting in his 1977 thesis, in which he used "Dedekind cuts" to describe real numbers in an ordered field. Shortly after the work done by Jutting, additional languages and systems were developed to represent the proof checking space, and examples include Andrzej Trybulec's Mizar system, The Boyer-Moore NQTHM theorem prover, Lawrence Paulson's Isabelle, and the Prototype Verification System (PVS) developed by John Rushby, Natarjan Shankar, and Sam Owre [3]. While the verification systems improved, the use of mathematics was still a heavy requirement if one wanted employ its use.

2.4 Formal Verification in Engineering

The implications of formal verification caught the eye of industries outside of mathematics; adoption of its use in hardware and software gained traction as soon the first formal verification languages were developed. In the semiconductor world, the use of the proof language to validate gate logic and decision trees were of great benefit, and companies such as Intel, AMD, and IBM adopted formal verification systems in their workflow to develop validated silicon products. Since gate logic's nature can be described in mathematics, the use of formal verification came naturally to the semiconductor industry. On the other hand, software was a bit of a late adopter, as techniques for representing software in mathematical terms was considered difficult compared to hardware usage.

In the paper *A Survey of Automated Techniques for Formal Software Verification*, the authors D'Silva, Kroening, and Weissenbacher detail the difficulties as abstraction-based, with "predicate abstraction is currently the predominant abstraction technique in software model checking" [4]. Abstraction in hardware proved to be built off of basic gates and simple models, where software provided an incredible challenge with complex code structures and difficult to abstract predicates in great numbers. An example of this difficult would be graph theory; hardware provides a well defined conditional in other building blocks of logic gate conditionals, while graph theory used in software requires careful analysis of the semantics within each of the graph nodes. Automation of this process is still a heavy research area in the verification and validation field today. Innovations in this area, such as the Java Pathfinder (JPF) [5] have attempted to automate the analysis and model creation from the code itself, removing one of the difficulties of profiling each of the node's code behaviors and allowing the code model to be checked [6]. However, difficulties still exist in this area, as described by authors D'Silva, Kroening, and Weissenbacher–"Many existing refinement heuristics may yield a diverging set of predicates", and "Predicate abstraction does not work well in the presence of com-

plex heap-based data structures or arrays." [4] As such, no silver bullets exist from the above methods, since software's nature seems to be the most difficult part of creating abstract software models in general.

2.5 SAT Solvers

Boolean Satisfiability Solvers (or SAT Solvers) require the problem to be described in the format of Boolean formulae; doing so gives the chosen SAT engine clauses in which to find solutions for. Conflict clauses, which are calculated from given Boolean formulae, allow the SAT engine to comb the branches of the search space and reduce the search space for faster proof results [4].

Using standard SAT engines such as MiniSAT or ManySAT provides the ability for integration of SAT functionality into proprietary software or workflows for domain specific usage and integration. Many academic papers exist that expand usages into domains such as Verilog, or code coverage testing systems for software. Examples of such systems would be Yosys, which has higher abstractions of hardware components and a high-level language to provide the necessary workflows for hardware engineers to validate their designs [7].

2.6 The Alloy Modeling Language

The Alloy modeling language is one of the more prominent SAT-based languages used in academia, with courses, books, and research being created with it. Alloy separates itself from other SAT systems by providing relationship modeling first, which improves the workflow by providing easier abstractions that exist in the physical domain. In addition, Alloy's backend using *Kodkod* allows for better model instance discovery, vastly changing the workflow from plain specification of Boolean expressions to discovery of models with loosely constrained systems.

Alloy, which was created by the *Software Design Group* at MIT, introduces

SAT in a slightly different context than other solvers by providing graphing, relational model capabilities, and multiple backend SAT engine options. The proprietary language created for Alloy also provides additional ease in model abstraction by making certain elements easier to represent.

The Alloy language reduces language elements to SAT, and leaves the SAT language semantics out of the modeling process. Alloy's specification structuring using signatures allows for object-oriented relations with abstract, extends, and module to be mixed with SAT-style declarations such as *fact*, *pred* (predicate), *check*, and *assert*. Further relation operators exist within the language to tie in the language pieces together to represent models in a more intuitive manner. Thus, one may represent real world models with greater ease than SAT-solving languages on their own. With the ability to generate instances, representing real world elements with predicates and relationships becomes possible within Alloy's language, enabling constraint and instance solving. Looking from the direction of mechanical engineering, constraints are used in a similar fashion for programs such as *Solidworks*, where part assemblies and movements are constrained against one another. The constraint engine assists in helping the assembly remain locked by assumed rules, which allows additional iteration of the design to be done on top of them. In this way, Alloy is somewhat of a bridge between the two realms of formal verification and mechanical systems.

2.7 Software Architecture in Mechanical Systems

In both consumer and industrial electronics today, the use of electromechanical systems is commonplace; micro controllers and embedded systems control things such as the door locks on a car to the engine operation of a car, and robotics and similar systems assemble the vast majority of objects we purchase. After 1971, the adoption of embedded systems grew quickly, and the subsequent technological movement has pushed many analog systems to become automated or controlled by

various types of embedded systems.

One of the areas of research that grew with the increased use of embedded systems was the firmware or software that resided on the chip itself. The "brains" of the system had basic functionality, and it grew in scope and size with the production of smaller silicon and increased performance requirements. However, the need for controlled systems has increased with every passing year, garnering more complex code to control more complicated or precise mechanical systems. Well-known examples of this transition from full mechanical to electromechanical exist in transportation–the automobile. The automobile went from a completely mechanical operation of the engine to embedded controlled in as little as one decade, giving way to better performing and more reliable engine operation. At the core of this movement however, was a fundamental understanding of the operation of the engine itself. Modeling the operation of the engine had to be researched to create the software and firmware that runs the embedded system for the engine–hence the abstraction of a mechanical system was used in this case. While many of the modeled parameters involved chemistry and physics, often the mechanical constraints were implemented via state machines with Mealy and Moore variants. While the behavior of some mechanical models were easy to describe, the complex nature of what could and what could not be done was an area of study that companies such as Honda and Bosch pioneered in the late 1980s. Patents created by both detailed strategies such as fuzzy logic and conditional state machines, allowing for defined mechanical control and verification capabilities.

Other well-known examples of such systems are in robotics and assembly line automation. While industrial automation typically has a more controlled environment, keeping electromechanical control systems in check still requires the same fundamental concepts of abstract mechanical modeling within software in order to keep a system in check. As systems have increased in complexity, best practice methods and conventions have yet to be made in the area. In many cases, the hard-

ware and mechanical technology is outpacing the research of software modeling of the said systems.

2.8 Mechanically Seeded Systems

One of the first widely reported uses of software to generate mechanical designs was NASA's Evolutionary Antenna Synthesis. The concept of evolved antennas first was an area of deep investigation in the 1990s from researchers such as Michielssen, Altshuler, Linden, Haupt, and Rahmat-Samiis [8]. By using genetic algorithms to determine optimal designs, an antenna design was determined through evolutionary combing, which penalizes 'design branches' that have sub-par performances. Iterative development over time can eventually lead to a more optimal design than static analysis by a human, and can be done unsupervised as a plus [8]. Stochastic hill climbing is the characteristic that is used by the optimization process (in contrast to a stochastic gradient descent), and the usefulness of such a process assists in the direction of design combing that the system takes while unsupervised. Examples of antenna solutions yield designs of unexpected nature, yet provide optimal performance properties that seem to defy conventional design logic in antenna design [8].

The caveat to such processes originate in the problem's description–the optimization process isn't one of low-cost nature. The constraints and calculations surround the algorithm for stochastic hill climbing took decades of work by researchers to develop, and is one of the reasons that the research has not exited the area of antennas easily. Bound by both metrics and models that must be found first before judging a design's performance, modeling of designs within the physical realm are only known for certain domains, others have remained stagnate for years. In order to provide a solvable solution, a known model of moderate accuracy must first be created, and then path options (which also must be modeled) must be created to traverse algorithmically [8]. This is just one of the may reasons such mechanically

seeded design systems are not commonplace; exhaustive research usually needs to be done to describe the problem domain before such a process may be applied.

2.9 Formal verification and Mechanical Synthesis

Formal verification is occasionally used to verify mechanical systems, but is more used on software or gate-level hardware designs in today's industries. In some industries, formal verification is applied after a design becomes finalized, but never the other way around. The large rift between formal verification and mechanical design have stood for years, but advancements today may actually close the gap to a different workflow altogether.

Software's use of formal verification focuses primarily on the architecture abstractions that naturally come from common tasks arising in programming. Logic structures, graphical flows, and hierarchical mappings are the most modeled subjects, as they provide mission essential systems the robustness needed for business process or similar qualifications. In many cases however, the emergence of physical instances (sometimes known as resource counts) assist in describing software operation on hardware systems. These instances pave the way to describe the inherent physical aspects of a software system that can be operated on with the correct constraints.

Mechanical seeding takes the two crossovers described previously to provide a generative solution to creating mechanical systems. By using software predicates as mechanical constraints, the operational nature and motion of a mechanical system may be proven through formal verification techniques and syntax. Evidence of possible linkage between the software and mechanical world have been conjectured at a primitive level with Alloy in previous work, which conducted research on linking the design of a Dual-Clutch Transmission firmware design by using mechanical predicates in Alloy [9]. The findings within the work demonstrate the possibility of generating software architecture from mechanical predicates, and associate the find-

ings to the rule-based nature of software architecture itself. While the findings list the possibility of generating software architectures from mechanical configurations, the reverse process has not been found true yet; the transitive nature of architecture constraints to mechanical configurations is not known to be injective mapping [9].

Mechanical seeding, or mechanical design configuration generation, is an area that has been researched in several ways, namely those using cost functions with stochastic hill climb or descent techniques. Thus, the usage of SAT-solvers and formal verification tools to assist in mechanical design is a prime area of research since sparse literature exists in the topic domain, and even fewer examples exist in the area of SAT-solvers and formal verification tools.

Chapter 3

Case Study

3.1 Objective

Two small but illustrative case studies have been created to simulate industry usage of formal verification systems for mechanical synthesis and design solving. The first case study lists the conversion of a model to a synthesis-capable mechanical system, and the second is an abstracted resource system. The goal of both case studies is to demonstrate software workflows of formal verification tools (such as Alloy) applied to known mechanical architectures and real world configurations and to link the two design processes into a workflow. Each case study will list the setup requirements that one would need to setup a problem for solving, along with the changes needed to the model that will lead to a synthesis-solvable solution. An analysis of the model data is given to understand the necessary parameters for a suitable initial model. Finally, results will be presented and reviewed for analysis of the report's initial conjecture on formal verification and mechanical synthesis.

3.2 Examples

This report focuses on small but illustrative examples that will help demonstrate key usages within the mechanical design realm. An example of existing mechanical configuration will be used first to demonstrate the model changes needed for synthesis. The second example will take a general model with cost constraints and run a full synthesis effort on it. At the end of each example, the results and lessons learned will be discussed.

3.2.1 Bicycle Example

The first example employs the mechanical model of a mechanical configuration–the bicycle. In setting up the following example, several key attributes of the problem domain are unique. For one, the bicycle and its possible configurations have been seen for well over a century, showing the maturity of the design [10]. The optimum configurations of such bike gearing components are calculated by many pro cycling teams per day, allowing each rider a customized setup to the course and their riding behavior [10]. Configuration solving is typically done by using heuristics to calculate the course difficulty and probable gearing ratios needed to be competitive in the race, while also including other variables based upon race conditions and the rider themselves. Using this information, various gearing configurations are generated to be selected by the lead bike technician. Final decisions on configuration types are filtered and reduced based upon overall configuration weight and sprint capabilities.

Gear ratio calculations have provided one means to measure the overall configuration of a bicycle drivetrain [11]. With the use of common physics equations on mechanical advantage [reference], configurations can be quantified and ranked with charts that display optimal efficiency at speed increments [10]. Results of the analysis are then paired with the rider's ride preferences (i.e. higher or lower cadence) and primed to be assembled on the bike itself. After assembly, final weight tuning must be done on the bike to meet weight regulations imposed by racing officials or

organizations. Thus, most weight savings is only done via rotating mass, as bikes must all meet or exceed minimum weight regulations. Tradeoffs between efficiency of gearing setup and matching of the setup to a given course is then balanced by the mechanic, as to give optimum performance in race conditions for each rider.

The example presented below allow demonstrates an existing model created demonstrate the model checking abilities in Alloy. By systematically changing parts of the model, loosened constraints in the right manner lead to synthesis-capable models which can find desired mechanical instances. Details on some of the necessary changes to Alloy code are described with code listings, and diagrams cover the generated instances.

In order to represent the bicycle configuration in Alloy, several initial steps are required. The first step is to lay out the basic atoms that represent the individual components of the bike. There exist several key variables in the model in solving for optimal configuration: cassette size, chainring size, and chainring count. The choice of granularity is of great importance; too large of a component can prevent the solver from accommodating a design necessity, too small and it may provide too many degrees of freedom for the solver to realistically handle. In the case of this problem, the main atoms that are necessary are the *Chainrings*, the *Crank*, the *Cassette*, the *Drivetrain*, and *Bicycle*. Next, the linking of the relations between these atoms are next. Defining an atom relation and set ownership is defined in each of the atom signatures in Alloy. The following code is an example of such a model that would be generated by a user.

Code Listing and Changes Required

Using the code in *Listing A.1*, the results of the instance solving are displayed graphically to the user with relational arrows describing the solution. Utilizing the Alloy solver, multiple solutions may be cycled through to gain insight into what

design freedoms were possible with the given relations and constraints. Next, one may look to thin the result pool and provide a more substantiated solution to test in the physical domain. In order to do so, additional predicates and constraints may be added to guide the solver to relevant mechanical configurations needed. *Figure 3.1* below demonstrates one of the instances found by the Alloy solver.

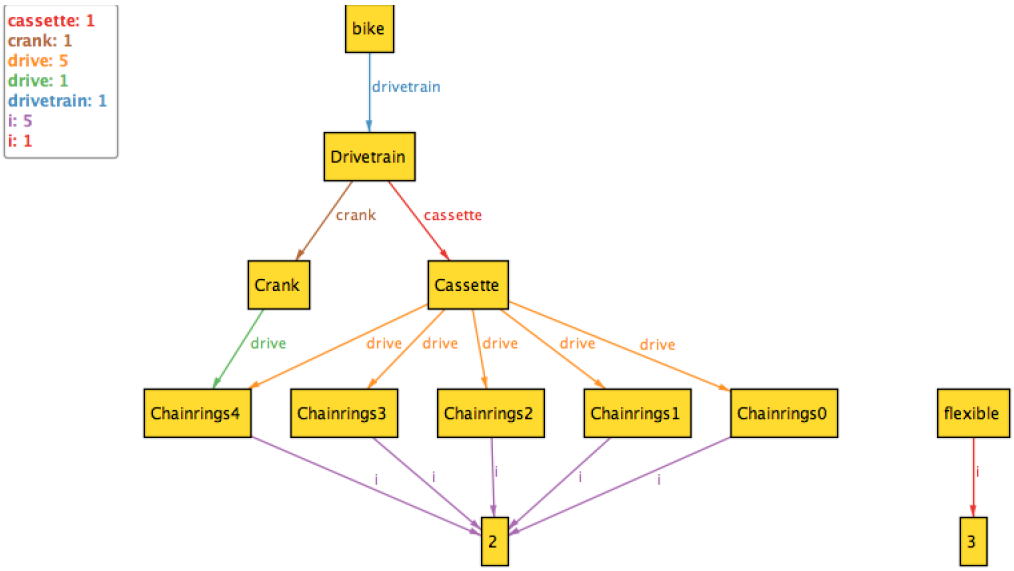

Figure 3.1: An example of a resulting instance for *Bike*

In the figure, the discrete atom counts called out by the *run* command, asking for only *one* bike and Drivetrain in the solution, but asking for freedoms on other variables in the solution. The *fact enforceComparison* runs some of the initialization code, which contains various the two variables to solve for and seeded starting values. In this case, the *Chainring* counts have been initialized to five for a starter design; subsequent code will look to solve for these values. Noticeable is the floating flexible atom, which currently does not shared constrains with the design. The grade value of **i** for each chainring will eventually link into the flexible atom and constraint.

Moving the simulation towards a synthesis-capable model, the attribute of

flexible design needs to be integrated into the atoms and relations of the bike simulation. Looking at the code and the resultant visualized solutions, the place to add the flexible attribute would be the signature of *chainrings*, as they give a bike more capabilities in combating various inclines and race speeds. *Listing A.2* contains the code that will be changed. In both cases, a simplification of the code will be made to remove the integer listing, and add the *flexible* attribute.

In *Listing A.2*, an attempt at trying to discretize the attributes of the *Chainrings* and *flexible* has created a complicated model. While synthesis requires some type of cost function or calculation to be present, trying to enforce the relation with integers or other numerical values creates an Alloy model that has numerical properties, but is unable to solve for them. By abstracting the *flexible* atom to more of a first-order relation, the Alloy model is more open to find instances than with numerical properties existing.

After making changes the above changes to *Listing A.2*, *Listing A.3* lists updated and simplified signature code that makes progress toward a synthesis-capable model.

This change allows for the atom of *flexible* to be adapted and used in the solver as a pivoted and solvable parameter when explicitly defined. Other changes to the code include facts and relational constraints for the changes made to the above atoms.

Additional constraints are added to prevent overlapping of relations, and several changes are made to the *noOverlapRings* and *fullyAssembleBike* to connect the atoms in a meaningful way. In the example, some of the key features of a model capable of synthesis are already in place. First is the command *#(Chainrings)* $\leq$ *5* in *initComparison*, which gives the a simple minimum with flexibility to the SAT-solver within Alloy. Later enforcing the final running of the system with *for 9 but exactly 1 bike, 1 Drivetrain* gives the rest of the 'hard' constraints, but leaves the rest of the atom relations up to solver to find instances for *Listing A.4*.

Finally, the changes to the bike.als file are then re-run in the Alloy solver, and instances such as what is shown in *Figure 3.2* can be created.

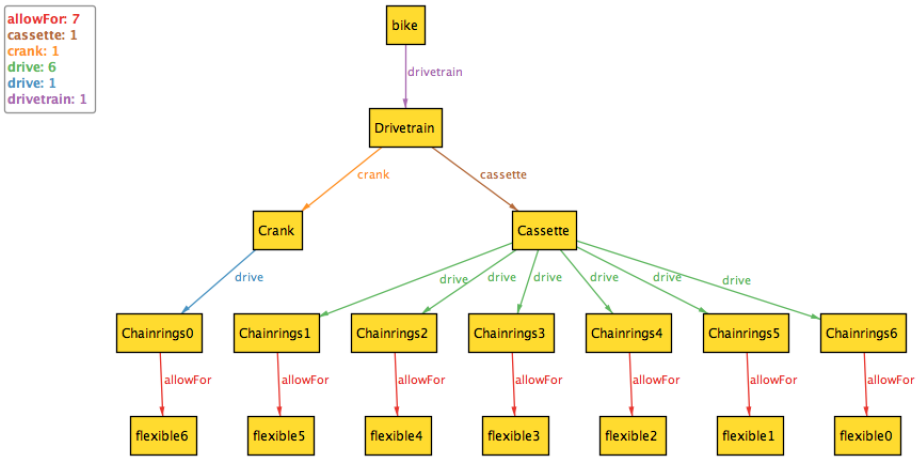

Figure 3.2: Resultant instance after changes for *flexibility*

In this generated instance, if a configuration is required that meets a certain flexibility criteria (created through experiments and heuristics), explicitly stating the necessary amounts through the # operator in Alloy gives a quick, easy to understand syntax with results that are easy to understand and tinker with. As shown by this example, some boilerplate is necessary to setup the initial atoms and relations for the bike. A small set of atoms, predicates, and facts make up the base mechanical configuration, but leave the constraints loose enough to provide the synthesis capabilities to find optimal solutions for one's racing configuration.

Results and Lessons learned

With this example, the most prominent observation from the experiment is that some real-world models are capable of synthesis, and require small changes to the Alloy abstractions to do so. In dealing with synthesis models, it seems the hardest part of the effort is to come up with the correct abstraction for the given problem. Leaving a proper set of constraints tied to relations can be infuriating at times, giving

either an over-constrained model or one that is seemingly incapable of looking like a real-world instance.

When looking to make changes to a model, one must first look at the granularity of the items and leave the pivoting synthesis values as single atoms. Basic relations must tied that explain instantiation and atom ownership, but not over-constrain to the point where Alloy is not free to create instances. This means defining predicates (and not facts) that give greater than or less than directives with the # symbol, and using the final run options to define if exact counts of atoms need to be in a given solution. Generally, one must start with a loosely constrained and defined model, and slowly introduce restrictions until the majority of instances return valid designs.

Alloy as a tool for developing synthesis models seems to be adequate, as the language is flexible enough to accommodate atoms and relations paired with a syntax that allows for flexible synthesis exploration. Small changes may be visualized through the relational viewer, which can help the user guide the model towards a synthesis-capable state.

3.2.2 Resource Management Synthesis

The second example provides a more complicated problem with two interlocked attribute values, requiring a balancing of the two for optimal configuration. In many types of mechanical configurations, various types of tradeoffs are balanced and selected to provide an optimum configuration. This can be from a load balancing operation on a server, to staff and resource balancing within a company or factory. Abstractions of these problems lend themselves to the same overall structure, and is one that may be synthesized with systems such as Alloy with the right strategies.

The basis of resource management modeling and solving lies in the defining of *work units* and *resources*. Each of these abstractions is linked with the opposing one in some fashion, giving way to many combinations of resource and work

configurations that may not be optimal. By allowing the Alloy solver to look for configurations that satisfy the system of constraints, the guesswork can be taken out of the modeling process.

For a *work unit*, it can stand in for raw manpower, kilowatt-hours, parallel assembly lines, or manufacturing plants. Resources can be respectively money, energy, or other combinations of abstracted and measured items. The goal of such modeling is to provide a trade-off abstraction for the system to solve for, allowing the end user or management to decide upon a configuration to move forward with.

For this example, a staff and resource management model will be created and solved in Alloy. The example simulates an additional complexity over the bike simulation in *Section 3.2.1*, where a *penalty* for the *flexible* attribute is introduced. Since most properties in the physical world incur significant penalties for instantiation, the model can be adapted and extended upon to model many other types of configurations in engineering and design.

Code Listing

The code in *Listing A.5* demonstrates the properties of synthesis by defining a predicate that describes numerical requirements of atoms (resources) in order to be instantiated. This *cost* provides Alloy the satisfiability requirement to change the given amount of work units and comb through trees of configuration combinations.

General atom boilerplate is required to provide the initial constraints needed for sensible results. One-to-one mapping between atoms is necessary to prevent acyclic relations and overlapping maps of atoms. The two facts restricting this are listed in *Listing A.6*.

In order for the resources to be tied to work units, a one-to-one property is employed with restrictions with *consume*, enforcing that all consumable resources are unique to each other. An over-constraint to the system is commented out to allow for floating and unused resources. An over-constrained system would enforce

that all resources be used up – a slightly different use case for the end user.

The *connected* fact enforces that all resources and flexible units belong to the set owned by *staffStrength* and *staffCost*. By enforcing this property, the sets may be easily compared by some further predicate or higher relation by the end user.

The *nosharedconsume* fact assists in making 'floating' *flexibleUnits* not shared consumes, preventing any overlap of consumed resources. Once these constraints are in place, the rest of the synthesis logic may be pursued.

The most important part of the synthesis effort is to provide predicates and running conditions that allow Alloy to go through the right pathways for seeding a design. Two predicates below are taken from *Listing A.5* and shown in *Listing A.7*

The following predicates enforce that any solution should attempt to find solutions with the following restrictions met. In the example, *currentMarket* is defined that all flexible units must consume 1-3 resources per invocation. This can be changed based upon the current 'market' rate, and simulates the requirements in a first order manner. The *show* property enforces the discrete number of work units. If reversed, one could find the necessary work units to complete a given task with restricted resources. Again, the adaptability of the simulation has been provided by specific boilerplate atom relations and arrangements of predicates.

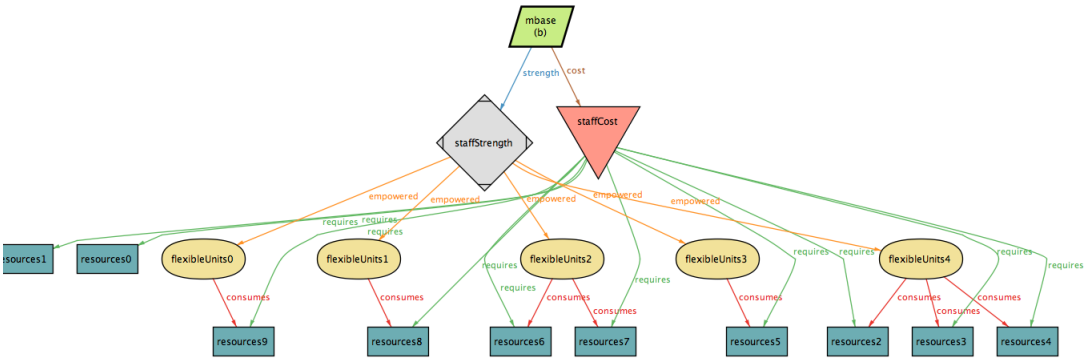

Figure 3.3: Instance of a double constrained system

Figure 3.3 demonstrates one of the instances found based upon the con-

strained code created in *Listing A.5*. Noticeable in this figure is the adherence to the exact amount of resources given in the *run* function, and the *option* by the constrained system to either use or ignore the resources while attempting to grant the work units requested by the end-user. In addition, the predicate of *currentMarket* is easily seen in the diagram as followed, with each *flexibleUnits* taking either 1-3 resources per invocation. Cycling through different instances provide varied combinations that meet the predicate's satisfiability; additional restrictions can thin down the returned instances if necessitated.

Thus, the following Alloy code allows for the solving of a double-constrained system with some explicit requirements, and synthesizes a result that meets the needs of the end-user. Although somewhat of a small example, additional relations and predicates may be stacked upon this example for more complicated mechanical or configuration problems.

Taking synthesized instances into the physical world

One of the advantages to using Alloy is the XML output generated for instances found by the SAT solver. Once a suitable instance is found, one has several options to pipe in the resultant instance into some form of physical controller or generator. The suitable instance is first put out to XML, which can then be read and parsed by the system via the user's choice of programming languages. This can be anything from Python, Javascript, Perl, or a similar language. Next, the parsed output of the XML is read to obtain the generated atoms to be seeded into a physical form. This can be done by having a work order generated from the list, assignments invoiced, or factory commands to build the work units with resources. Thus, a link directly from the suitable instance to the final physical configuration can be made from a synthesis-capable model in Alloy.

For the purposes of this report, Python will be used to demonstrate the XML parsing of Alloy's instance output. The XML output and associated code for parsing

is demonstrated below:

In *Listing A.8*, the Alloy XML output is seen to be a tree of fields and associated atoms that make up the resultant instance. Utilizing XML parsing capabilities of modern programming languages, extracting the relevant instance information should be straightforward in this case.

Using the following code in *Listing A.9*, the final three variables contain the configuration and usage relations of the resources and work units together. The outputs of this script can then be piped to whatever software or platform controls the physical configuration of the given problem domain (*Listing A.10*).

By extracting the simple relations and set of resources and units generated by the Alloy XML output, scripts and other software can easily test out, deploy, or recommend mechanical configurations to the end user. Other places of use may be through the use of test oracles; the results of the generated instances can be tested and fed back to the system if one were to devise a more intricate cost system into the design.

Results and Lessons learned

Working from one of the classic examples of balancing, it can be seen that Alloy is fully capable of providing synthesis-capable models and is able to provide designs that exist in the physical world. The resource vs. work unit system can be modeled as this singular entity, or as one in which each attribute contains its own instance of resource vs. work units. Adding further relations and atoms can also assist in building more representative mechanical models, if required.

Where some models fall short of representation, several options exist to allow for better cost functions and numerical injection. Alloy is built upon open source code, and can be interacted through with an API in addition to the Alloy language. Thus, domain specific cost functions can be subbed in for the *flexibleUnits*, giving the SAT-solver and Alloy capabilities to find mechanical solutions that exit first-order

relations but maintain simplicity of calculation.

What worked well in this example was the suitability of Alloy for synthesis workflow. Taking the XML output of the generated instances, an end-to-end system could be automated to deliver mechanically synthesized designs from an Alloy model, thus improving upon the concepts provided in this report. While the XML output is easy to parse and use, a more powerful way to use Alloy would be to use the Java API to generate and extract the instances. While the Alloy syntax may be more comfortable for some users, a more streamlined and powerful synthesis effort could be done if one were to do the entire effort in the Java API.

Additionally, visualization options that explain the synthesis effort would be of great use. Guiding a model towards synthesis in Alloy is difficult enough; facilitated model refining with visualizations or a pre-built framework would greatly enhance the synthesis effort.

Chapter 4

Conclusion

The use of formal verification methods and tools has previously been locked to specific domains – those that involve modeled states, and those that looked to verify algorithm and software correctness. In both cases, the most difficult parts of using formal verification methods is to abstract the correct atoms, and provide the right types of predicates and constraints to solve for one's desired solution. In the case of synthesis, the process is further complicated by providing the right set of freedoms in the model to guide relevant instance discovery.

The results of the case studies provided in this report demonstrate a process that may be used to generate mechanical or physical instances via synthesis. While relatively simplistic, building upon the base model outlined in this report can allow for expanded capabilities of synthesis if additional costs and relations are modeled appropriately.

In the case studies listed in this report, the *flexibility* atoms are assumed to be measured by the end-user in the user specific domain, and are of known scalar values. In the event that the attribute modeled is not reducible to scalar value, then additional work may be required to bring the model to a solvable state. Options include interacting with the Alloy Application Programming Interface (API) itself, or by using modifications of Alloy such as **Alloy*** [12] which allow for higher order

constraints than what the base Alloy tools can provide. Analysis of domain specific cost functions seems to be the biggest priority in synthesis, with the constraint and setup next in difficulty.

The use of Alloy for synthesis allows for a more extendable and programmatic way of developing synthesis tools, as the built-in module capabilities and namespacing assist the end users with many ways of sharing and building upon each other's work. The open source code, API, and syntax helped the synthesis efforts of the experiment possible, and further work may be pursued with the groundwork laid by this report. Additionally, the software workflow of post-instance processing can be connected to the domain specific systems for bringing one's resulting instance into the physical realm, with many flexible tools and languages capable of assisting the effort.

Design synthesis has been one of the less researched areas within formal verification, and mechanical design synthesis is an even less common topic within design synthesis. However, the techniques and concepts derived from the case studies in this report demonstrate that Alloy and similar software may be used to generate such models with a bit of effort. By first breaking the problem into specific atoms, constraints may be tied together with enough room to allow synthesis to occur by creating either scalar cost atoms or adapting higher order relations to be used for certain attributes.

In previous research mentioned in the literature review, software synthesis and electromechanical synthesis were known applications of formal verification systems such as Alloy. However, the mechanical synthesis capabilities were previously completed with one-off custom systems or software, and not created with the same systems as that did software or electromechanical synthesis. The results of the case studies in the report demonstrate that Alloy's language and abstraction capabilities allow for all types of synthesis, with mechanical synthesis requiring domain-specific cost functions to be created into scalar atoms.

Based on the results of the report, efforts to further the developments of mechanical synthesis in Alloy should be placed on creating a module or framework for Alloy that assists in boilerplate and discrete predicate setup for the end-user. Additionally, domain specific cost functions should be approximated before solving for designs, which will allow for mechanical design synthesis to be achievable in Alloy.

4.1 Future Work

The results of this report show promising results in the synthesis front using Alloy and similar formal verification tools. Since Alloy is open source and is easily extendable, the next step projected for developing a better method of synthesis would be developing an Alloy module to handle some of the required steps found in this report. Once the module is developed, the focus should be shifted into either developing new methods of creating cost functions into atoms, or utilizing the Java API that powers Alloy to allow a custom cost function to be used for a predicate listing.

Other options would be to utilize the higher-order system of **Alloy*** to attempt to create a still abstracted framework or module for mechanical synthesis, or a combination of the two options together.

Appendix A

Code Listings

Listing A.1: bike.als

```
open util/integer
sig flexible { i: Int }{ int[i] = 3 }
sig Chainrings { i: Int }{ int[i] = 2 }
sig Cassette { drive: set Chainrings }
sig Crank { drive: lone Chainrings }
pred initComparison {
        //Chainrings.i > flexible.i
        #(Chainrings) >= 5
}
fact enforceComparison {
        initComparison
        one flexible
}
fact one2one {
        cassette.~cassette in iden
        univ.cassette = Cassette
        crank.~crank in iden
```

```
        univ.crank = Crank
}
fact noOverlapRings {
        all disj a,b : Chainrings | a !=b
}
sig Drivetrain {
        cassette: one Cassette,
        crank: one Crank
}
sig bike {
        drivetrain: one Drivetrain
}
pred fullyAssembleBike {
        all b: bike, d: Drivetrain | d in b.drivetrain
        all cs: Cassette, c: Chainrings | c in cs.drive
}
run fullyAssembleBike for 8 but exactly 1 bike, 1 Drivetrain
```

Listing A.2: Original *flexible* and *Chaingrings* listing

```
sig flexible { i: Int }{ int[i] = 3 }
sig Chainrings { i: Int }{ int[i] = 2 }
```

Listing A.3: Updated *flexible* and *Chaingrings* listing

```
sig flexible {}
sig Chainrings { allowFor: flexible }
```

Listing A.4: Updated bike.als with synthesis changes

```
open util/integer
```

```
sig flexible {}
sig Chainrings { allowFor: flexible }
sig Cassette { drive: set Chainrings }
sig Crank { drive: lone Chainrings }
pred initComparison { #(Chainrings) >= 5 }
fact enforceComparison { initComparison }
fact one2one {
        cassette.~cassette in iden
        univ.cassette = Cassette
        crank.~crank in iden
        univ.crank = Crank
        allowFor.~allowFor in iden
        univ.allowFor = flexible
}
fact noOverlapRings {
        all disj a,b : Chainrings | a !=b
        all c: Crank, cs: Cassette, cr: Chainrings
                | cr not in (c.drive & cs.drive)
}
sig Drivetrain {
        cassette: one Cassette,
        crank: one Crank
}
sig bike {
        drivetrain: one Drivetrain
}
pred fullyAssembleBike {
```

```
        all b: bike, d: Drivetrain | d in b.drivetrain
        all cs: Cassette, c: Chainrings, cr: Crank
                | c in cs.drive || c in cr.drive
}
run fullyAssembleBike for 9 but exactly 1 bike, 1 Drivetrain
```

Listing A.5: mbase.als

```
sig flexibleUnits { consumes: set resources }
fact one2one {
        consumes.~consumes in iden
        //Comment this out to allow for floating
                unused resources
        //univ.consumes = resources
}
sig resources{ //The modeled resource }
sig staffStrength { empowered: set flexibleUnits }
sig staffCost { requires: set resources }
sig mbase{
        strength: one staffStrength,
        cost: one staffCost
}
fact connected {
        all m: mbase, s: staffStrength | s in m.strength
        all s: staffStrength, f: flexibleUnits |
                f in s.empowered
        all m: mbase, sc: staffCost | sc in m.cost
        all sc: staffCost, r: resources | r in sc.requires
}
```

```
fact nosharedconsume {
        all disj f1, f2: flexibleUnits |
                f1.consumes != f2.consumes
}
pred currentMarket {
        //consumes multiple resources
        all f: flexibleUnits | #f.consumes =< 3
                && #f.consumes >= 1
        //all f: flexibleUnits | #f.consumes = 1
}
pred show(b: mbase) {
        currentMarket
        #flexibleUnits = 5
        //#resources < 10
}
run show for 9 but 1 mbase, exactly 10 resources
//run show for 9 but 1 mbase
```

Listing A.6: boilerplate properties

```
fact one2one {
        consumes.~consumes in iden
        //Comment this out to allow
                for floating unused resources
        //univ.consumes = resources
}
fact connected {
        all m: mbase, s: staffStrength | s in m.strength
        all s: staffStrength, f: flexibleUnits
```

```
                | f in s.empowered
        all m: mbase, sc: staffCost | sc in m.cost
        all sc: staffCost, r: resources | r in sc.requires
}
fact nosharedconsume {
        all disj f1, f2: flexibleUnits
                | f1.consumes != f2.consumes
}
```

Listing A.7: important predicates

```
pred currentMarket {
        //consumes multiple resources
        all f: flexibleUnits | #f.consumes =< 3
        && #f.consumes >= 1
        //all f: flexibleUnits | #f.consumes = 1
}
pred show(b: mbase) {
        currentMarket
        #flexibleUnits = 5
        //#resources < 10
}
run show for 9 but 1 mbase, exactly 10 resources
//run show for 9 but 1 mbase
```

Listing A.8: XML Example

```
<sig label="this/flexibleUnits" ID="4" parentID="2">
   <atom label="flexibleUnits$0"/>
   <atom label="flexibleUnits$1"/>
```

```
    <atom label="flexibleUnits$2"/>
    <atom label="flexibleUnits$3"/>
    <atom label="flexibleUnits$4"/>
</sig>

<field label="consumes" ID="5" parentID="4">
    <tuple> <atom label="flexibleUnits$0"/>
          <atom label="resources$9"/> </tuple>
    <tuple> <atom label="flexibleUnits$1"/>
          <atom label="resources$8"/> </tuple>
    <tuple> <atom label="flexibleUnits$2"/>
          <atom label="resources$7"/> </tuple>
    <tuple> <atom label="flexibleUnits$3"/>
           <atom label="resources$6"/> </tuple>
    <tuple> <atom label="flexibleUnits$4"/>
          <atom label="resources$5"/> </tuple>
    <types> <type ID="4"/> <type ID="6"/> </types>
</field>
```

Listing A.9: XML to Physical Configuration example in Python

```
import xml.etree.ElementTree as ET
tree = ET.parse('test_xml_2.xml')
root = tree.getroot()

recordatoms = False
consume_list = []
for items in tree.iter('field'):
    # Look specifically for the consumes subtree
```

```
    if items.attrib['label'] == 'consumes':
        recordatoms = True
    else:
        recordatoms = False
    if recordatoms:
        for second_items in items.iter():
            if (second_items.tag == 'atom'):
                # Get the resource or work unit
                consume_list.append(
                                second_items.attrib['label'])

resources = set()
units = set()
for i in range(0,int(len(consume_list)/2)):
    resources.add(consume_list[i*2+1])
    units.add(consume_list[i*2])
```

Listing A.10: Final Configuration list in Python

```
consume_list:
        ['flexibleUnits$0',
        'resources$9',
        'flexibleUnits$1',
        'resources$8',
        'flexibleUnits$2',
        'resources$7',
        'flexibleUnits$3',
        'resources$6',
        'flexibleUnits$4',
```

```
        'resources$3',
        'flexibleUnits$4',
        'resources$4',
        'flexibleUnits$4',
        'resources$5']

resources:
        {'resources$3',
         'resources$4',
         'resources$5',
         'resources$6',
         'resources$7',
         'resources$8',
         'resources$9'}

units
        {'flexibleUnits$0',
         'flexibleUnits$1',
         'flexibleUnits$2',
         'flexibleUnits$3',
         'flexibleUnits$4'}
```

Bibliography

[1] *MiniSat*. URL `http://www.minisat.se`.

[2] *Alloy*. URL `http://alloy.mit.edu/alloy/`.

[3] Jeremy Avigad and John Harrison. Formally verified mathematics. *Communications of the ACM*, 57(4):66–75, April 2014.

[4] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7), July 2008.

[5] *Java Pathfinder*. URL `http://babelfish.arc.nasa.gov/trac/jpf/`.

[6] NASA. *NASA Ames Research Center*. 2005. URL `http://babelfish.arc.nasa.gov/trac/jpf`.

[7] Clifford Wolf. *Yosys Open SYnthesis Suite*. 2012. URL `http://www.clifford.at/yosys/`.

[8] Al Globus, Greg Hornby, Derek Linden, and Jason Lohn. Automated antenna design with evolutionary algorithms. *American Institute of Aeronautics and Astronautics*, 2006.

[9] David Liu and Olamide Kolawole. *Modeling of Dual-Clutch and Semi-Automatic Transmission Logic in Alloy*. April 2013. URL `https://github.com/triskadecaepyon/DCT_Alloy_Model`.

[10] Chester R. Kyle and Frank Berto. The mechanical efficiency of bicycle derailleur and hub-gear transmissions. *Human Power: Technical Journal of the IHPVA*, 1(52), 2001.

[11] Sheldon Brown. *Sheldon Brown's Gear Calculator*. 1998. URL `http://www.sheldonbrown.com/gears/`.

[12] *Alloy*: A Higher-Order Relational Constraint Solver*. URL `http://alloy.mit.edu/alloy/hola/`.