Team Amalgam

SE390 Requirements Specification

Joseph Hong, Chris Kleynhans, Ming-Ho Yee, Atulan Zaman
{yshong,cpkleynh,m5yee,a3zaman}@uwaterloo.ca

December 13, 2012

Contents

1 Introduction

1.1 Purpose

This document is the requirements specification for Moolloy version 0.3, a computer-based system that will be developed by Team Amalgam. This system will be developed over the course of the fourth-year design project, which consists of the courses SE390, SE490, and SE491.

Furthermore, as the system is built on top of Moolloy, the scope of this document includes the requirements specification for Moolloy. The intended audience for this document will be researchers interested in the *guided improvement algorithm* [1] for multi-objective optimization, or any other relational logic optimization problems.

A software requirements specification describes the behaviour of a software system and not its implementation. In other words, it is concerned with *what* a system does and not *how* it does it. However, our work is to improve the speed and scalability—nonfunctional requirements—of an algorithm, a step-by-step procedure for accomplishing some computation. To resolve this incongruence, Appendix A outlines how the guided improvement algorithm works.

Due to the fact that this is a research project and that we do not know which optimization techniques will be implemented, this requirements specification is subject to change. Specifically, more command-line arguments may be added to control optimization parameters.

This document allows us, and any other researchers, to develop benchmarks and tests for Moolloy. Furthermore, interested researchers may also use this document to write extensions and modify the software.

1.2 Scope

The software product described in this document will be referred to as *Moolloy v0.3*, or *Moolloy* for short. Henceforth, any references to the existing Moolloy versions will be referred to with their version numbers, or as the *original Moolloy*.

Moolloy is an implementation of the *guided improvement algorithm*, which produces Pareto-optimal solutions to general multi-objective optimization problems. Although Moolloy is built on top of a SAT solver, it is not a SAT solver.

Multi-objective optimization is an interest to many fields of science and engineering. In particular, we are interested in problems in aerospace, civil engineering, and software engineering. Many of these problems cannot be

solved by Moolloy v0.2, as the input space is too large. Our work is to optimize Moolloy so it can handle problems of this scale.

Our focus is on optimizing Moolloy and how it calls the SAT solver. We are not concerned with optimizing the SAT solver itself.

1.3 Definitions, acronyms, abbreviations

Definition 1. A solution is said to be *Pareto-optimal* if and only if it is not *dominated* by any other solution. A solution a dominates a solution b if all metrics of a are greater than or equal to their corresponding metrics of b, and there exists some metric of a that is strictly greater than its corresponding metric of b.

Definition 2. The set of all Pareto-optimal solutions is called the *Pareto front*.

Definition 3. A *multi-objective optimization (MOO) problem* is a problem with multiple constraints, as well as multiple goals to optimize over.

Definition 4. An *exact* solution to a multi-objective optimization problem is the Pareto front.

Definition 5. *Discrete* in this document means that there is a countable number of configurations for every problem. This is in contrast to the continuous case. A synonym for discrete is *combinatorial*, but we will only use the former term.

Definition 6. By *general-purpose*, we mean that Moolloy can solve any multi-objective optimization problem, as opposed to a specific one.

Definition 7. *SAT*, or *boolean satisfiability*, is a problem that asks whether a given Boolean formula can be assigned values such that its evaluation is true. In other words, it asks if a given Boolean formula can be satisfied.

1.4 References

Our work is an extension of the original Moolloy, which was described by Rayside, Estler, and Jackson [1].

1.5 Overview

The rest of this document describes the environment, interface, and functionality of Moolloy. We also discuss assumptions about the user, as well

as other constraints and external dependencies. Finally, we describe the functional and nonfunctional requirements for the computer-based system. Appendix A briefly describes the guided improvement algorithm.

2 Overall Description

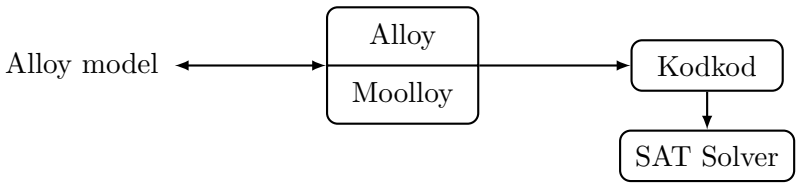

Figure 1: Overview of Moolloy structure.

Moolloy, as an extension of Alloy, is written in Java. It is backed by Kodkod, which in turn makes calls to an external SAT solver. Kodkod and the SAT solver are packaged with Moolloy. As input, Moolloy takes a multi-objective optimization problem modelled in Alloy, and returns the Pareto front as an Alloy instance. Figure 1 depicts the overall structure of Moolloy.

We assume that all users are already familiar with multi-objective optimization, including such terms as *Pareto-optimal* and *Pareto front*. Furthermore, we assume users are familiar with how SAT solvers are used to find these solutions. The user should also be familiar with expressing the problem in Moolloy's domain specific language, as well as interpreting the results.

As Moolloy is implemented in Java, the user will need to have the Java Runtime installed on his or her environment.

3 Specific Requirements

3.1 External Interfaces

Moolloy takes in various command-line arguments as input parameters. Input problems are specified as Alloy models, while solutions are output as Alloy instances.

3.1.1 Command-line arguments

Moolloy has the following command-line arguments:

- `--SingleSolutionPerParetoPoint, -s`
  These two equivalent options specify that only a single solution is computed for each Pareto point. Defaults to False.

- `--LogRunningTimesArg=filename`
  Enables logging of running times to a file. Output will be written to the file `filename`. Defaults to False.

- `--NoAdaptableImprovement`
  This option specifies that no adaptable minimum improvement at each step should be used. Defaults to False.

- `--LogPrintHeaders`
  This option outputs the header in the log. Defaults to False.

Additional command-line arguments may be added to the external interface to control optimization parameters. For example, flags may selectively enable optimization features. Because this is a research project and we do not know which techniques will be implemented, we cannot concretely specify the arguments at this time.

3.1.2 Input model

The input model, specified as an Alloy file, is passed to Moolloy as the last command-line argument. The file follows the Alloy grammar [2], with a few Moolloy specific additions to allow the definition of objectives and to run the corresponding optimizations. Objective blocks are supported to define a series of metrics and whether they should be maximized or minimized. Inst blocks are also available to allow the creation of a set of typescopes which can be referred to in the run command as the bounds of the configuration space. This new bounds block can be referenced wherever a typescope is required.

```
Additions to existing grammar rules
    paragraph ::= objectiveDecl
    typescope ::= name
    cmdDecl ::= [name ":"] ["run"|"check"] [name|block]
                scope "optimize" name

Objective Declaration
    objectiveDecl ::= "objective" name
                      "{" metric ["," metric] "}"
```

```
    metric ::= ["maximize"|"minimize"] name

Inst Declaration
    instDecl ::= "inst" name "{" typescope ["," typescope] "}"
```

For examples of Moolloy input files see the test cases provided in our test plan document.

3.1.3 Output instance

Moolloy returns the solution as an Alloy instance. Alloy instances are expressed as XML files [3] with a predefined schema [4]. In this case, solutions are generated as files with the name `alloy_solutions_N.xml` where `N` is the solution number. The XML files can be imported into the Alloy Analyzer, and visualized as a graph or as a tree.

3.2 Functional Requirements

Moolloy takes in a multi-objective optimization problem as an Alloy model, a format described in Section 3.1.2. It applies the guided improvement algorithm to compute the Pareto front of the specified problem. These solutions are converted to an Alloy instance, expressed as XML, as described in Section 3.1.3.

If the `SingleSolutionPerParetoPoint` flag is specified, then Molloy will compute only one solution for each Pareto point. If this flag is not specified, Moolloy will use its default settings, which will return all solutions that result in that Pareto point.

If the `LogRunningTimesArg` flag is specified, then Molloy will log running times to the specified file. By default, this flag is not set.

If the `NoAdaptableImprovement` flag is specified, then Moolloy will not use the adaptable improvement optimization, which was exists in the original version of Moolloy. This optimization uses exponential steps when searching for better solutions. By default, this flag is not set.

If the `LogPrintHeaders` flag is specified, then Moolloy will include column headers in the log files. By default, this flag is not set.

3.3 Performance Requirements

We are concerned with two performance requirements: speed and scalability. These two are related, as significant improvements in speed will allow Moollloy to scale up and handle larger inputs.

3.3.1 Speed

Moolloy's algorithm works by making multiple calls to a SAT solver for each multi-objective problem it must solve. These calls can be classified as SAT and UNSAT calls, where SAT calls can be satisfied, and UNSAT calls cannot be satisfied. UNSAT calls are very expensive, while SAT calls are less expensive, since they terminate once a solution is found.

Our goal is to improve the speed at which Moolloy solves a problem. This allows users to solve our current problems much faster. To accomplish this goal, we aim to reduce the number of SAT and UNSAT calls.

From Rayside et al.'s paper [1], the twenty-five variable, four metric knapsack problem takes approximately forty-five hours to complete, in the process requiring 1,938 calls to the SAT solver, where 814 were UNSAT. Our goal is to reduce this time to under twenty-four hours.

3.3.2 Scalability

Because solving large problems is very expensive, there are many real-world problems that simply cannot be solved at this time. We aim to optimize Moolloy so that such problems are solvable.

3.4 Design Constraints

As Moolloy is currently implemented in Java, the new version must also be implemented in Java. Furthermore, to maximize the utility of this program, Moolloy must be able to build and run on any platform that supports Java.

A Guided Improvement Algorithm

This appendix outlines, at a high-level, how the *guided improvement algorithm* works. Rayside et al. [1] provide a more formal and detailed description in their paper.

First, Moolloy finds a solution that satisfies the constraints. Once this solution is found, Moolloy finds a new solution that dominates the previous one. Moolloy continues this process of finding new solutions until no other dominating solution can be found—this last solution is on the Pareto front, by definition.

Moolloy continues this process with new starting solutions (that are not dominated by any previously discovered solution) until all solutions on the Pareto front are found.

Thus, Moolloy makes a large number of calls to the underlying SAT solver. These are extremely expensive calls, so the goal for optimizing Moolloy is to reduce the number of calls to the SAT solver.

References

[1] D. Rayside, H.-C. Estler, and D. Jackson, "The Guided Improvement Algorithm for Exact, General-Purpose, Many-Objective Combinatorial Optimization," Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep. MIT-CSAIL-TR-2009-033, Jul. 2009.

[2] *Core Alloy4 Syntax*, Software Design Group, Massachusetts Institute of Technology.

[3] *Alloy Analyzer*, Software Design Group, Massachusetts Institute of Technology. [Online]. Available: `http://alloy.mit.edu/alloy4/xmlformat.html`.

[4] *Alloy XSD schema*, Software Design Group, Massachusetts Institute of Technology. [Online]. Available: `http://alloy.mit.edu/alloy4/xmlformat.html`.