|galois|

# AMIDOL Milestone 13 and 14 Report

Eric Davis, Alec Theriault, and Ryan Wright

Galois, Inc.
421 SW 6th Ave., Ste. 300
Portland, OR 97204

## Contents

**1 Introduction** **2**

**I Milestone 13** **2**

**2 Recent Extensions to the AMIDOL Framework** **2**

2.1 VDSOLs for Mathematical Languages . . . . . 2

2.2 Definition of Derived Measures . . . . . 3

**3 AMIDOL Demo Instructions** **7**

**4 AMIDOL Performance for Real-World Systems and Processes** **7**

**II Milestone 14** **7**

**5 Final Prototype Development** **7**

**6 AMIDOL as a Service** **7**

|galois|

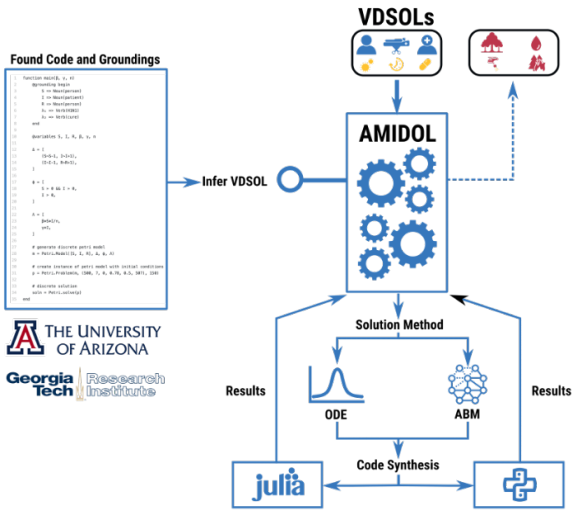


Figure 1: AMIDOL Architecture

6.1 Read/Write Model . . . . . 7
6.2 Infer/Read VDSOL . . . . . 8
6.3 Read/Write Results . . . . . 8
6.4 Transform/Execute Model . . . . . 8

**7 Model Algebras and Transformations 8**
7.1 Composing Models in AMIDOL . . . . . 8
7.2 Substituting Models in AMIDOL . . . . . 9

## 1. Introduction

# Part I

# Milestone 13

## 2. Recent Extensions to the AMIDOL Framework

The UI for comparing results of multiple models has been extended to support a rich language for combining traces. This is particularly useful for constructing derived measures without needing to re-run the model. For instance, this might be used in a situation where an epidemiological model tracks multiple strains of a virus separately, but the hospital data only keeps aggregate data. Another similar case is when the scientist doing modelling is trying to quickly line up peaks of infected populations from model

result and real world data. Our language contains primitives like shift which simplify this sort of transformation.

A completely new VDSOL that is designed to take a system of differential equations written in LaTeX format has been added. The idea behind this new language is to simplify the work of going from a model written in an academic paper to an AMIDOL model. Scientists enter LaTeX equations, constants, and initial conditions in a text box. Beside this input, they get a real-time LaTeX preview of their equations. From there, the model is compiled into the AMIDOL IR so that it can be executed and compared to results from other models (which may have been designed in completely different VDSOLs).

Finally, we are in the process of experimenting with model composition in the backend, with the goal of finding a minimal intuitive language for combining models. To this end, we've been refining our existing state-sharing composition operators and have started to experiment with other techniques revolving around substitution. Most of these changes are still not exposed to end users, since we are not yet sure what a good visual representation would look like.

### 2.1. VDSOLs for Mathematical Languages

Adding a VDSOL for LaTeX input equations: The VDSOL renders LaTeX equations using KaTeX, a fast browser-based Javascript library designed for this purpose. Once a user submits a system of differential equations, the backend tries to parse each line as a differential equation, initial condition, or constant. This step is complicated by the fact that the LaTeX source for equations can sometimes be interpreted in multiple ways (for instance: is '\fracpsi2' the variable 'psi' divided by two, or the product of 'p', 's', 'i', and '0.5'). To solve this ambiguity, we require that implied multiplication include at least a space to separate the factors (eg. 'p s i' vs 'psi'). The final step is to convert the system of differential equations into an AMIDOL model. This is fairly straightforward: variables in the equations turn into states and the right-hand side of the equations turns into events.

### 2.2. Definition of Derived Measures

Updating the comparison UI to support a richer mathematical language for derived measures: In order to implement this, we've moved the work of combining data traces from the UI to the backend. The language for describing derived measures supports translations, linear distortions, component-wise arithmetic, as well as a couple built-in math functions. The backend contains a parser for this language as well as an interpreter. One of the challenges here is around how to interpolate when combing data traces whose measures had different time ranges or step sizes.

|galois|

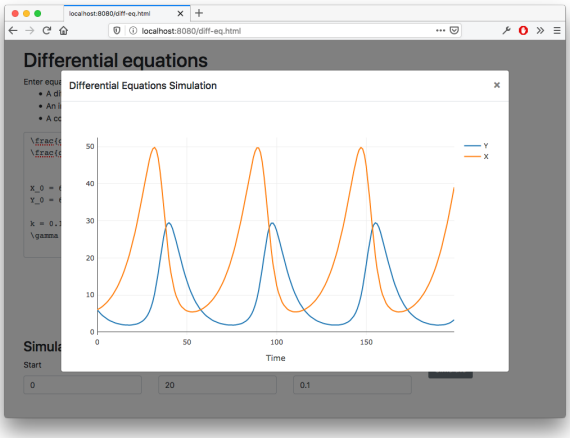


Figure 2: AMIDOL Architecture

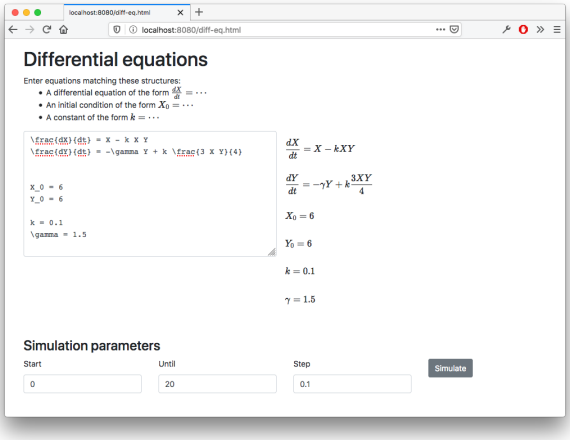


Figure 3: AMIDOL Architecture

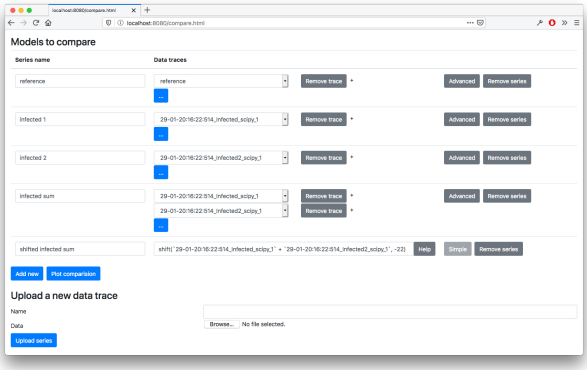


Figure 4: AMIDOL Architecture

|galois|

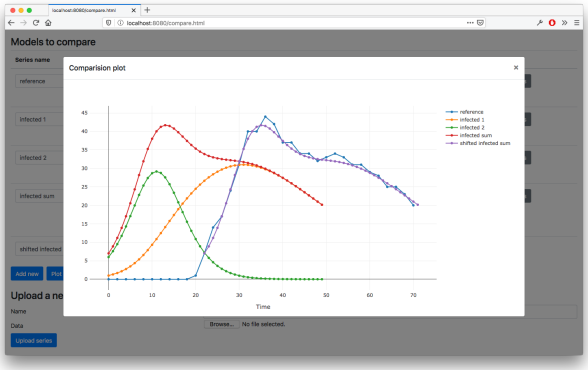


Figure 5: AMIDOL Architecture

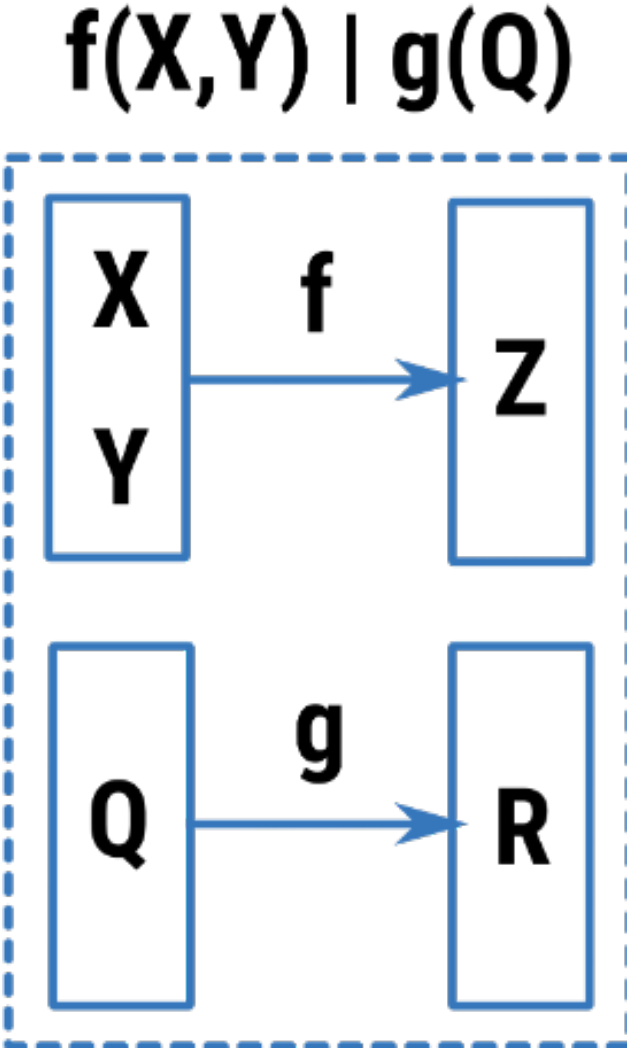


Figure 6: AMIDOL Architecture

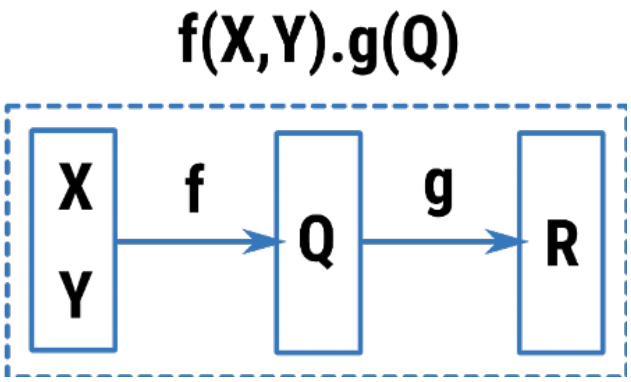


Figure 7: AMIDOL Architecture

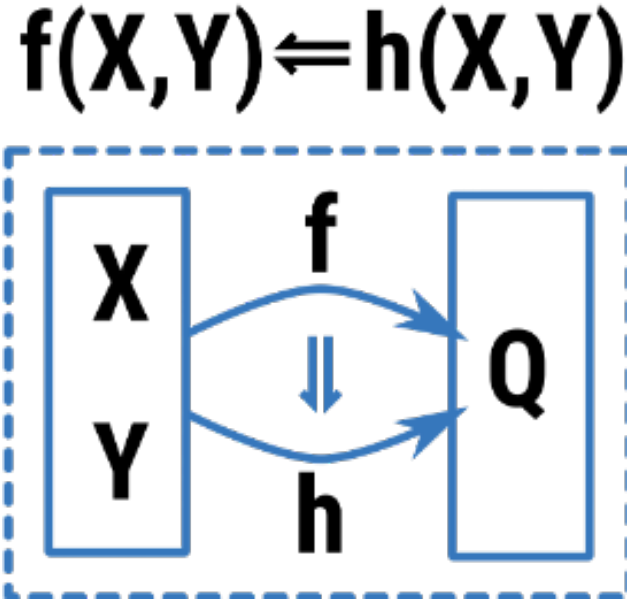


Figure 8: AMIDOL Architecture

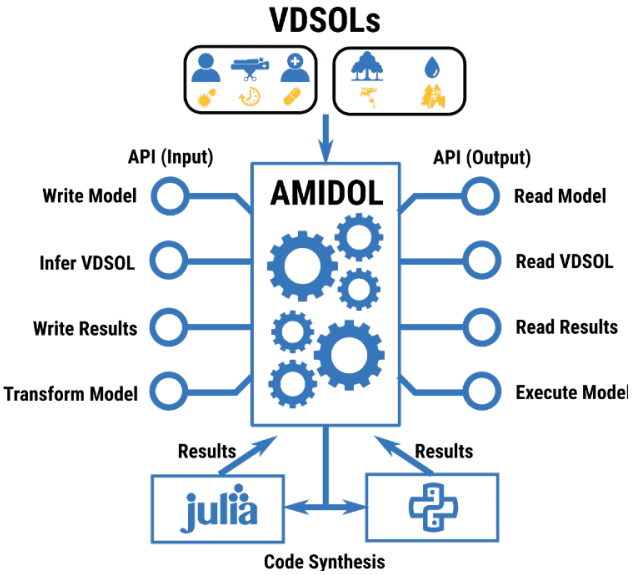


Figure 9: AMIDOL as a Service

## 3. AMIDOL Demo Instructions

## 4. AMIDOL Performance for Real-World Systems and Processes

| | Model | SIR | SIIR | Predator-Prey | SIR with vital dynamics |
|---|---|---|---|---|---|
| *Time* | Graph VDSOL to IR compilation | 1.7ms | 1.7ms | 1.1ms | 1.6ms |
| | Julia to Graph VDSOL (including ontology grounding search) | 1.8s - 6s | 2.1s - 6s | n / a | n / a |
| | IR compilation to Python backend | 190ms | 160ms | 30ms | 160ms |
| | IR compilation to Julia backend | 24ms | 25ms | 15ms | 52ms |
| | Execution of Python backend output | 2.5s | 2.4s | 1.1s | 2.5s |
| | Execution of Julia backend output | 9.8s | 12s | 7.5s | 7.0s |
| *Lines of Code* | Python backend output | 40 | 45 | 39 | 47 |
| | Julia backend output | 44 | 62 | 51 | 60 |

# Part II

# Milestone 14

## 5. Final Prototype Development

## 6. AMIDOL as a Service

### 6.1. Read/Write Model

Summary – Allow external applications to read and write models to AMIDOL's IR.

Requirements – Working to supplement AMIDOL with searchable model database, supporting a query interface for model properties and UUIDs for models.

### 6.2. Infer/Read VDSOL

Summary – Currently supports VDSOL inference from AMIDOL IR or Julia code with groundings.

Requirements – Allows retrieval of VDSOLs by UUID and query interface for VDSOL properties.

### 6.3. Read/Write Results

Summary – Allow external applications to read and write to the AMIDOL results database.

Requirements – AMIDOL is implementing a graph-based results database, including mechanisms to index into the database to write external results (from "data" or an "external model").

Suggestions – Measure indexing by outside applications is an open question. We need a clear definition of measure equivalence. What does it mean for two measures to be the same? At a basic level, they have the same sigma-algebraic structure, over borel sets that are proxies for the same space. What does this mean in practice?

### 6.4. Transform/Execute Model

Summary – Allow external applications transform a model in AMIDOL, and to execute a model in AMIDOL.

Requirements – Valid transformation set. This is essentially a model transformation algebra.

## 7. Model Algebras and Transformations

Models form a group with the operation "." (serial composition). Closed Associative Identity element The "pass through" model which immediately yields its inputs without changing them. Inverse element is a bit harder. Intuitively you need a model which "undoes" g() yielding no output, and passes through the inputs to f().

### 7.1. Composing Models in AMIDOL

Composition via state sharing is very powerful: we've built prototypes that combine birth-death processes, the SIRS model, Lokta-Volterra predator prey, and population flow models. Some of the challenges we've identified in this space:

Naming shared states, especially when multiple states are being combined. The most promising avenue is to find a way to force users to specify names.

It should be possible to provide some automated assistance about which compositions make sense or don't based on units / types / ontological categories

Sub-models being composed need to be pretty general to account for different ways in which we may want to specialize them through composition. One idea to help deal with

this is using substitution to introduce new states into existing models.

### 7.2. Substituting Models in AMIDOL

Substitution of subgraphs (of events and states) with other compatible subgraphs Provides a mechanism for quickly swapping out sub-models used in a bigger model (for instance: changing an SIR model into an SEIR one). Required inputs: the states to remove, the events (between these states) to remove, the new model to introduce, and a mapping from the removed states to the states in the new model. Challenge: it is possible for subgraph substitution to accidentally "break" some other part of the model. Consider the case of swapping an SIR sub-model for an SEIR sub-model: elsewhere in the larger model, uses of $S + I + R$ probably indicated "total population" and should be replaced with $S + I + E + R$. Solution idea: provide a mechanism for building up symbolic expressions $total_{population} = S + I + R$ that can be used multiple times in the bigger model. This alleviates the pain of "refactoring" after substitution. Solution idea: bring to the users attention all locations in the model where there are references to state variables that are part of substitution Challenge: sometimes, we want to be able to do multiple substitutions in parallel. For instance, we want to replace every instance of a sub-model with some other sub-model. In this case, model substitution turns into what looks like a structural graph query problem.

One useful take of substitution involves introducing new states into existing models. For instance: a rate in a model starts by being approximated using a constant but eventually needs to be turned into a proper state (the model is being run with parameters where the constant assumption no longer holds - think a pendulum that goes from small-angle swings to large-angle ones). Challenge: this sort of transformation seems painfully low-level - is there some bigger operations that generalizes this nicely?