|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

|galois|

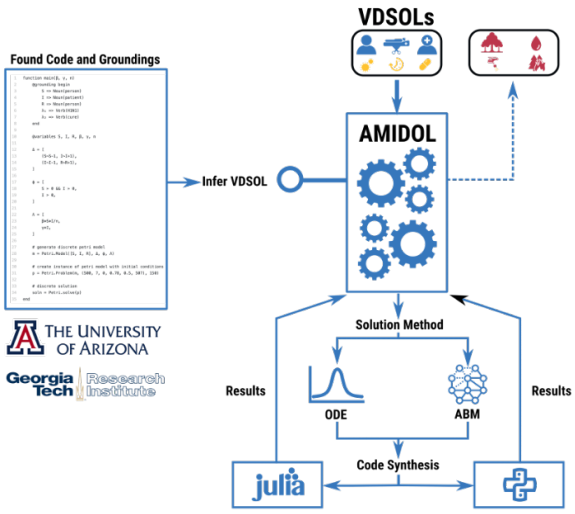


Figure 1: AMIDOL Architecture



## 1. Introduction

Our advancements to AMIDOL have focused on preparing for the upcoming live demo, final code release, and expanding AMIDOL's use of VDSOLs, model composition interface, and the generation of a new "AMIDOL as a Service" (AaaS) API designed to allow other modeling tools to make use of AMIDOL's IR, transformation, and code synthesis capabilities. We have released the next version of AMIDOL at its github site (`https://github.com/GaloisInc/AMIDOL/`) under the BSD 3-Clause "New" or "Revised" License.

Figure 5 shows the current state of the AMIDOL framework. AMIDOL functions as an integrated prototype with the work of our colleagues at GTRI and their SemanticModels.jl development, and Automates from our colleagues at the University of Arizona. AMIDOL currently supports multiple VDSOLs, both graphical and textual, a universal IR, a code synthesis engine capable of generating solutions targeting both ODE solution and Agent-Based Methods (ABMs) in both Python and Julia.

|galois|

# 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 a given 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, such as 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 can 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 in the UI, as we are still developing appropriate primitives for model composition.

### 2.1. VDSOLs for Mathematical Languages

AMIDOL has been extended to include a VDSOL which compiles and translates a system of LaTeXequations into the AMIDOL IR. 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 '\frac{psi}{2}' 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'), assuming identifiers without spaces are atomic. The final step is to convert the system of differential equations into an AMIDOL model, represented in the IR. This is fairly straightforward: variables given as derivitives with respect to time in the equations turn into states and the right-hand side of the equations turns into events and their associated rates. Positive and negative

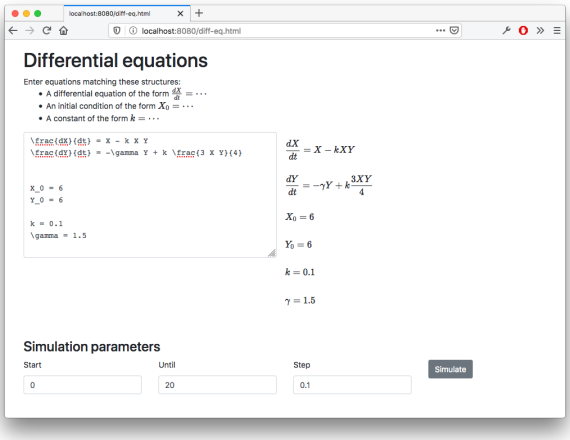


Figure 2: New AMIDOL Differential Equation VDSOL

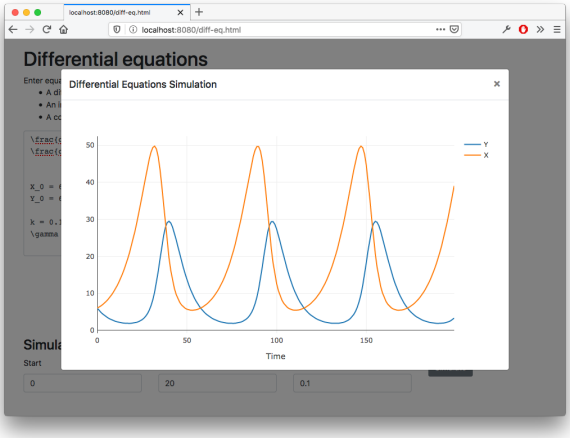


Figure 3: New AMIDOL Results Visualization

rates are matched to infer output predicates.

### 2.2. Definition of Derived Measures

The AMIDOL comparison UI has been updated 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 limited, but expanding, set of built-in functions. The backend contains a parser for this language as well as an interpreter.

## 3. AMIDOL Demo Instructions

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

In order to characterize AMIDOL's current performance, we executed the benchmarks documented below, measuring over four models, the time to compile the VDSOL to the

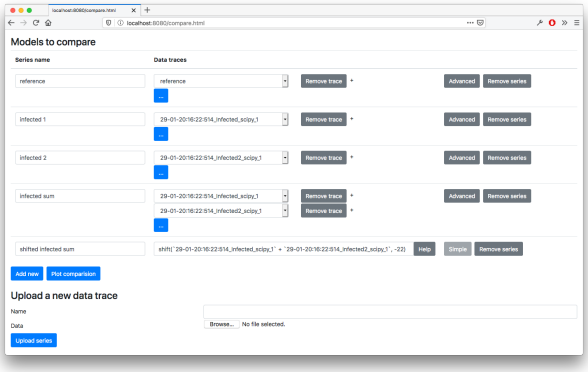


Figure 4: New AMIDOL Model Comparison

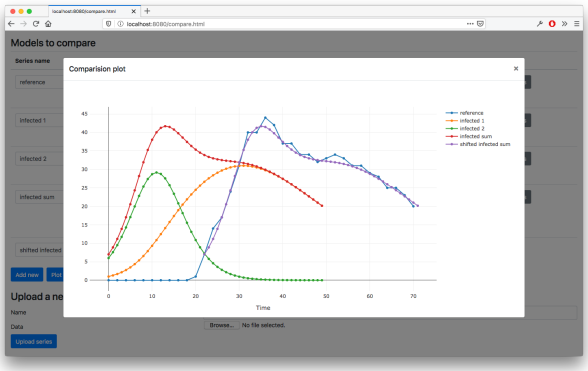


Figure 5: New AMIDOL Model Comparison Results Visualization

AMIDOL IR, the time to translate a Julia AST into a VDSOL (including the time required to search for groundings in the SNOMED ontology), the time to compile the IR to both Python and Julia backends, and the execution times of both backends.

We additionally measured the lines of code generated for each model in both Python and Julia by the AMIDOL code synthesis portion of the compiler. The results are provided below.

| | **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 |

Overall these times demonstrate performance comparable to, or faster than, hand-developed scientific models, with much faster and less error prone development cycles.

# Part II

# Milestone 14

## 5. Final Prototype Development

As part of the final prototype development of AMIDOL, we have worked to both expand the number and types of different VDSOLs supported by AMIDOL; implemented a model composition system that results in a model algebra, capable of composing models serially, in parallel, and substitutionally, even across different VDSOLs; and expanded AMIDOL's ability to operate as a service for external applications.

## 6. AMIDOL as a Service

We have worked to expose AMIDOL's capabilities to external applications via a number of RESTful end points in order to allow for AMIDOL to work not only as a stand-alone application with its own UI, but to accept workloads from external applications seeking to leverage AMIDOL as part of their own tool chain, or as part of a library.

Figure 6 shows the general architecture of the AaaS framework, with input and output APIs, in roughly matched pairs based on their intended use. We discuss those APIs in more detail in the following sections.

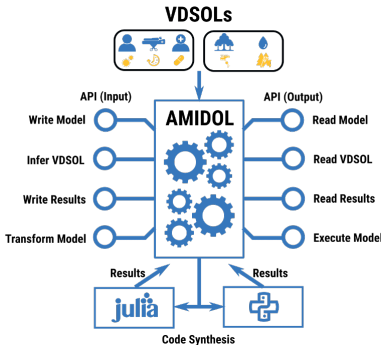


Figure 6: AMIDOL as a Service

### 6.1. Read/Write Model

The `Read/Write Model` API end points are designed to allow external applications to read and write models directly to AMIDOL's IR. This allows the import and export of models into AMIDOL for further API calls. To support this we have been exploring future work including a searchable model database, and the support of a query interface for finding models by their properties.

### 6.2. Infer/Read VDSOL

The `Infer/Read VDSOL` API end points are designed to import found code, generating an inferred VDSOL, or to read an existing VDSOL from AMIDOL. Imported code takes the form of Julia code, with grounding information supplied by AutoMATES. The Read end point allows for retrieval of VDSOLs already in AMIDOL's database. We have been exploring future work to also allow retrieval using a query system over VDSOL properties.

### 6.3. Read/Write Results

The `Read/Write Results` API end point is designed to allow external applications to read and write to the AMIDOL results database. AMIDOL is implementing a graph-based results database, supporting model comparison, and including mechanisms to index into the database to write data, traces, or external model results. This will allow users to access and compare measures generated by AMIDOL, and submit new data or measures for comparison. 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. We are exploring what this means in practice as part of our future work.

### 6.4. Transform/Execute Model

The `Transform/Execute Model` API end point allows external applications to compile and transform models in AMIDOL, including through model composition, and then execute compiled models.

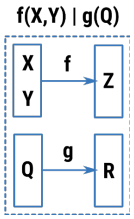


Figure 7: Parallel Model Composition in AMIDOL

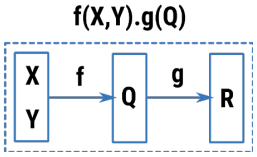


Figure 8: Serial Model Composition in AMIDOL

## 7. Model Algebras and Transformations

In order to support the `Transform/Execute Model` API end point, and enable users to compose models generated in one, or multiple, VDSOLs, we have implemented a model algebra to allow for serial, parallel, and substitional model composition. Models in AMIDOL are composed via the process of **state sharing** a powerful primitive which unifies the values of states across models, joining them. While many unanswered questions exist on state sharing workflows for users, the current AMIDOL framework allows for parallel and serial composition using this mechanism.

Parallel composition is accomplished by sharing no state variables, resulting in the diagram shown in 7. This has the effect of taking the union of input and output objects of both morphisms, and treating the new parallel composition as a morphism between these unions. While seemingly simple, this construct allows for a richer set of serial compositions by altering the apparent input and output model signatures of the new composed models to potential match other models with which it can then be composed.

Serial composition, shown in Figure 8, is accomplished by taking the disjoint union of two open models in AMIDOL's IR, $P : X \to Y$, and $Q : Y \to Z$ resulting in $P + Q : X \to Z$, sharing all similar states in the objects $Y$ belonging to $P$ and $Q$.

Lastly, AMIDOL implements substitutional composition using the double-push out rule shown in Figure 9, implementing the double-push out graph rewriting mechanism. Sub-

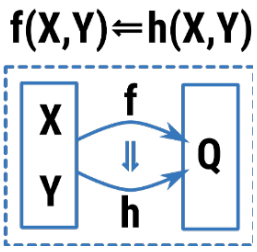


Figure 9: AMIDOL Architecture

stitution 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). The substitional transformation requires a user to give an example of the initial model, the model invariants, and the resulting new model to be obtained via substition. While semantically valid, it presents some possible issues as it is entirely 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 $S + I + E + R$. To address this issue we are focusing our future work on providing a mechanism for building up symbolic expressions alleviating the pain of "refactoring" after substitution.