Semantic Program Analysis for Scientific Model Augmentation

Christine Herlihy, Kun Cao, Sreenath Reparti, Erica Briscoe, and James Fairbanks

Georgia Tech Research Institute, Atlanta, GA

Abstract

`SemanticModels.jl` is a system for extracting semantic information from scientific code and reconciling it with conceptual descriptions to build a knowledge graph. This knowledge graph represents the connections between elements of code (variables, values, functions, and expressions) and elements of scientific understanding (concepts, terms, relations), and can be reasoned over to facilitate several metamodeling tasks, including model augmentation, synthesis, and validation. We present a category theory-based framework for defining metamodeling tasks and extracting semantic information from model implementations, and show how `SemanticModels.jl` can be used to augment scientific workflows in the epidemiological domain.

1 Introduction

`SemanticModels.jl`facilitates several metamodeling tasks by detecting and exploiting the implicit relationships between the semantically rich, natural language-based representations of scientific knowledge found in academic papers, and the relatively semantically sparse, but modular and precise representations of such knowledge found in code. These relationships comprise a knowledge graph. We demonstrate how this knowledge graph can be reasoned over to support metamodeling tasks that may be exploratory, iterative and/or inter-disciplinary in nature. Our software can augment scientific workflows in support of a wide range of objectives, from state space exploration and hypothesis generation to evidence-based policy-making.

1.1 Motivation

Progress in science comes from adapting and extending models from prior work to address new problems, but current scientific research workflows make this difficult. This is due in part to the fact that scientific papers are not sufficiently high fidelity representations of conceptual, semantic, and mathematical models required to provide complete transfer of information between and among domains and researchers.

In addition, the nature of scientific inquiry often lends itself to highly tailored, procedural scripts that are primarily intended to produce and record results, while objectives such as modularity take a backseat. Scientific code contains a large amount of sophisticated domain knowledge that is known to the author of a code, but not expressed within the programming language. Such semantic modeling information includes rules and constraints imposed by the physical phenomena being modeled. For example, 1) stochastic systems are modeled with probability values, which are constrained to be between 0 and 1; 2) unitful measurements must obey the laws of dimensional analysis, which invalidate quantities such as $3m + 4m/s$; and 3) signal processing algorithms must treat time domain and frequency domain signals differently even though they are both represented by arrays of floating point numbers.

Our work is grounded in the belief that a framework that augments scientific workflows by alternately inferring, injecting, and rewarding the inclusion of semantic information in code via the type system can help to mitigate these challenges in a way that is computationally efficient, verifiable, reproducible, and open to improvement through iterative feedback and expansion with respect to domains.

1.2 Significance

As computational models of complex world systems grow increasingly sophisticated, program analysis tools must understand and manipulate these models. We introduce a formalism to study the augmentation of scientific modeling code. These ideas are implemented in a software package for analyzing and manipulating models written in the Julia programming language [1]. The Julia language is ideal for this problem because it includes a capable type system programmers can use with multiple dispatch and is widely used in scientific computing. By encoding information about model semantics into the type system, the Julia compiler can understand, enforce, and manipulate those model semantics. These manipulations are studied in the context of epidemiological models, but are broadly applicable to both agent-based and differential equations-driven simulations.

1.3 Related Work

The semantic metamodeling system proposed here is informed by foundational concepts from several disciplines, including software engineering, programming language theory, natural language processing, and statistical meta-analysis. Software engineering emphasizes modular design, automation of repeated tasks, and incremental modifications [2]. Within this context, refactoring code refers to a process in which developers modify portions of an existing code to maintain or improve correctness while increasing maintainability. The correctness of a piece of software can be formally defined, and it is possible to design an automatic verification system; this is a well-established field of study within theoretical computer science [4].

Our system represents extends this notion of automatically verifiable program correctness to the semantic level by identifying, connecting, and verifying the unwritten invariants of scientific modeling code. In contrast to the explicit type and syntax rules employed by traditional verification approaches, the rules we seek to identify, extract, and use, are often informally specified or encoded in non-operational code and text, such as documentation, comments, and/or variable naming conventions. Our system can also be viewed as connecting existing work related to the use of knowledge graphs for the storage, retrieval, and emergent pattern mining of semantic information, and the development of ontologies, data flow graphs, and workflow management tools to integrate and automate common machine learning tasks and pipelines [8, 3, 7].

2 Methodological Approach

Our metamodeling approach requires a corpus of scientific code (containing model implementations) and domain-specific knowledge (currently in the form of scientific text) as input. Program analysis methods (both static and dynamic) and natural language processing techniques are used to build a knowledge graph representing the relationships between elements in the code (variables, values, functions, and expressions) and semantic elements of scientific understanding (i.e., concepts, terms, relations). This knowledge graph may then be used to reason over in support of model augmentation, synthesis, and validation. This paper focuses on model augmentation.

We begin by presenting a formal framework that can be used to represent and reason about these

different metamodeling use cases. We then provide examples from epidemiology to illustrate how this framework and associated semantic knowledge graph construction process, can be applied to augment real-world scientific workflows.

Our motivation for focusing on epidemiology is twofold: the associated literature demonstrates the use of a shared model structure with many variations. Furthermore, math represented therein spans both discrete and continuous systems of equations, solved by a diverse set of algorithms.

Scientific programmers represent models at 3 levels: 1) as a set of domain concepts understood by the developer, but not explicitly stated or encoded; 2) code implementations in a high-level language; and 3) an executable program compiled or interpreted on a specific computer architecture. We introduce the semantic level, derived from pertinent information from the code (such as that imparted through the type system) as well as from conceptual relations in the knowledge graph.

2.1 Theoretical Foundations

We formally define the components of our system using the language of category theory [11]:

A model $M = (D, R, f)$ is a tuple containing a set D, called the domain, and a set R, called the co-domain, with a function $f : D \mapsto R$. If D is the cross product of sets $D_1 \times D_2 \cdots D_k$, then $f = f(x_1 \ldots x_k)$, where x are the independent variables of M. If $R = R_1 \times R_2 \cdots R_d$, then R_i are the dependent variables of M.

The parsimonious formalization of what constitutes a valid transformation, or set of rules for modifying or combining models, requires us to assess not only mathematical and programmatic behavior of the system, but also the extent to which the resulting set of models are internally consistent and reflective of domain-specific scientific facts. We address this challenge through our approach to knowledge graph construction, as well as through ontology logs and the Julia type system.

A Category C is a set of objects and morphisms, which are structure preserving functions between the objects. Common examples of categories include the category of all groups, the set of all finite graphs, and the set of all finite preorders [11]. Ontology logs (ologs) are a diagrammatic approach to formalizing scientific knowledge used to precisely specify a conceptual model of a phenomenon or experiment [10]. An olog is composed of types (boxes) and aspects (edges). Figure 1 represents the susceptible-infected-recovered (SIR) model as an olog [9].

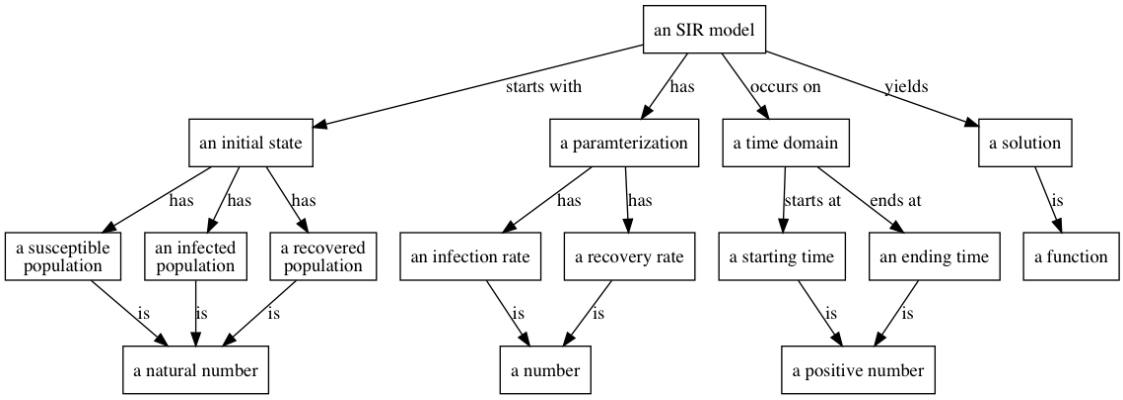


Figure 1: Ologs can be used to represent the structure of scientific models without the mathematics.

All programs in a strongly typed language have a set of types and functions that map values between those types. For example, the Julia program: `a = 5.0 b = 1; c = 2*a; d = b + c` has the types `Int, Float` and functions `*, +`, which are both binary functions. These types and functions can be represented as a category, where the objects are the types and the morphisms are the functions. We refer to the input type of a function as the domain and the output type as the codomain of the function. Multi-argument functions are represented with tuple types representing their arguments. For example `+(a::Int,b::Int)::Int` is a function $+ : Int \times Int \rightarrow Int$. These type categories are well studied in the field of Functional Programming. We apply these categories to the study of mathematical models.

Functional programming and category theory are intertwined and base the analysis of programs on the types and functions used in the program [12]. `SemanticModels.jl` implements a dynamic analysis tool using Cassette to instrument code to extract the run time type information for every function. That is to build a graph where the nodes are types and the edges are functions, where a function f connects types T, U if T is the type of f's arguments and U is the type of f's output values as expressed in `julia` syntax, `f(x::T)::U`[1]. The category theoretic approach enables reasoning over the semantics of programs.

The most salient consequence of programming language theory is that the more information that a programmer can encode in the type system, the more helpful the programming language can be for improving performance, quality, and correctness. Haskell programmers often use the type system to encode program semantics to improve software quality [5]. `SemanticModels.jl` uses the type system to encode model semantics to improve understanding, adaptability, and extensibility of the modeling code.

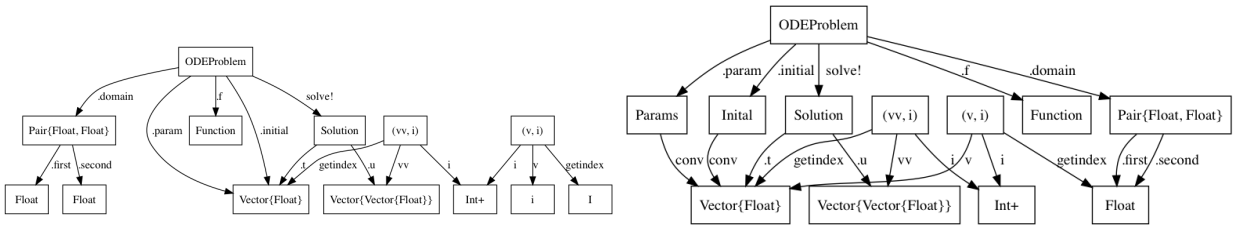
Model developers use conventions to encode semantic constraints into their code – for example, prefacing all variables that refer to time with a `t_`, such as `t_start, t_end`. This semantic constraint that all variables named `t_` are temporal variables is not encoded in the type system. Another example is that vectors of different lengths are incompatible. In a compartment model, the number of initial conditions must match the number of compartments. For example in an SIR model, there are 3 initial conditions, $[S, I, R]$, and there are 2 parameters $[\beta, \gamma]$. These vectors are incompatible. Computational systems employed by scientists will use a runtime check on dimensions to detect malformed linear algebra [2]. Scientists rely on this limited form of semantic integrity checking provided by the language, `SemanticModels.jl` is intended to rigorously apply such integrity checking across the modeling ecosystem.

Our goal is to extract and encode the maximum amount of information from scientific codes into the type system, where algorithms can analyze the integrity of programs in the language of categories. For example, if there are two types S, T and two functions $f, g : S \rightarrow T$ such that $Codom(f) = Codom(g)$ but $Range(f) \cap Range(g) = \emptyset$, then we say that the type system is ambiguous. In order to more fully encode program semantics into the type system, the programmer (or an automated system) should introduce new types into the program to represent these disjoint subsets. Category theory shows both why this is a problem for program analysis[3] and how to solve it with *union types*.

[1] The `a::A` operator in Julia asserts that the value of `a` is an instance of type `A`

[2] Julia, Scientific Python, and Matlab use run time checks, the C++ library Eigen supports both static and dynamic dimension verification

[3] If model transformations are represented as functors in this category, this form of ambiguity prevents the type system from enforcing semantic correctness of model transformations



(a) Type Diagram of the SIR Model

(b) Type Diagram of the SIR Model with Ambiguity Resolved

Figure 2: Program Types can express model structure

Returning to the SIR model example, Figure 2 shows how the `.param` and `.initial` functions both map `Problem` to `Vector{Float}` but with disjoint ranges. From our mathematical understanding of the model, we know that parameters and initial conditions are incompatible types of vectors because of the different lengths. Any program analysis of the model will be hampered by the ambiguity introduced by using the same type to represent two different concepts. The functions `.first` and `.second` which provide the beginning and end of the time domain of the system have overlapping ranges and are comparable as times. This is an example of how programming language ideas can improve the analysis of computational models.

2.2 Knowledge Graph Construction

To construct a epidemiological semantic knowledge graph, we use the Epicookbook[4], a repository implementing a variety of epidemiological compartmental models, along with explanatory natural language text. Here, we developed a set of grammatical rules to parse the existing code comments, from which we extract noun-predicate vertices to create an (edge) relationship of the type "definition". These extracted elements are inserted into the knowledge graph [6].

The Julia code associated with each epidemiological model, provides information through static analysis. We use the same parser as the `julia` program. This parser takes text representations of Julia code and returns an abstract syntax tree (AST). We then walk this AST looking for Julia program expressions that create information, including function definitions, variable assignments, and module imports. Function definitions are handled recursively to identify closures and local variables.

Static program analysis provides direct access to the function call graph; however, inferred types and runtime values require dynamic analysis. For this we use `Cassette.jl`[5], which is a library for context-dependent execution. `SemanticModels.jl` uses the overdub component of Cassette to build a dynamic analysis tool. We identify and insert the vertices and edges extracted from code into the knowledge graph in accordance with a fixed schema [6].

When two or more knowledge artifacts share provenance (e.g., the narrative text, programmer-provided comments, and source code that, when taken in tandem, represent a single recipe in the Epicookbook), we currently consider code text and markdown/comments text as strings, and use rule-based learning to associate text with code objects; these lexical matches are then parsed in an effort to extract edges of the type "representation" (abbreviated `repr`), which connect a (code)

[4] http://epirecip.es/epicookbook/
[5] https://www.github.com/jrevels/Cassette.jl
[6] https://aske.gtri.gatech.edu/v0.1/graph/#Schema-1

type source vertex to a (scientific) concept destination vertex.

3 Model Transformation Examples

Model augmentation refers to the set of metamodeling problems where a scientist gives the system a model, M and a transformation T, and uses the system to construct a new model, $T(M)$. Currently, when a scientist modifies an existing model, they must start with an implementation of the current model and directly modify the source code. This can be difficult in complex software implementations. We augment the scientist's capabilities by: (a) allowing scientists to manipulate aspects of the program's execution (e.g. norms, distributional assumptions) that are not directly accessible, including functions and models imported from libraries, and (b) automating the propagation of changes throughout the software system.

To implement this capability, `SemanticModels.jl`, provides the Dubstep module, which uses Cassette to modify programs by overdubbing their executions in a context[7]. Overdubbing allows you to define a context that allows a program to control the execution behavior of programs that are passed to it. Cassette is a novel approach to software development and integrates deeply with the Julia compiler to provide high performance, aspect-oriented programming.

Users can define their own contexts, which can be used to: 1) modify the APIs associated with user-defined or imported methods without making lexical changes to the source code; and 2) mix compile-time and runtime computations, to allow the user to leverage the scope information in the program trace to control execution.

Consider the motivating example in which a scientist has: 1) defined a Cassette context for solving models (`SolverCtx`); 2) defined a function which contains a differential equation representing the SIR compartmental model and computes the $\frac{du}{dt}$ array for the SIR system, `du/dt = sir_ode(du, u, p, t)`; and 3) wishes to explore hypothetical scenarios that do not match baseline assumptions or observed empirical conditions, such as, "What if the infection was stronger by a factor of α?". Mathematically, this perturbation is represented as: $\frac{dS}{dt} = \alpha(\beta SI - \gamma I)$. This model augmentation can be performed with minimal runtime overhead using the Cassette-based system. Users of the `SemanticModels.jl` systems can add parameters such as α described above and then sweep over these parameters to analyze trends or sensitivities within the new model.

A more complex model transformation is to take two models and transfer components from one to the other. For example, an SEIR model without vital dynamics has 4 states and 3 parameters, while a SIS model with vital dynamics has 2 states along with terms for net population growth of the susceptible population. An epidemiologist would want to combine these models by transferring the vital dynamics component of the SIS model to the SEIR model to create a new model with 4 parameters. `SemanticModels.jl` supports this capability[8].

4 Conclusions

Semantic modeling aims to leverage scientific knowledge currently trapped in the body of scientific code. While open source code is a prerequisite for reproducible research, it is not sufficient for reuse and understanding of complex modeling code. We present an automated system for extracting information from, reasoning about, and augmenting computational models. This framework

[7] Please excuse the audio puns. Cassette was developed to support automatic differentiation which involves computing a program "tape", and the language of tape manipulation has stuck.

[8] http://aske.gtri.gatech.edu/v0.2/example/

does not require the scientists to program in a restricted framework, but instead directly consumes standard Julia code. This framework enables extending models with new parameters and components. This framework is based on knowledge representations with category theory and aims for both theoretical soundness and practical usability.

5 Acknowledgments

The authors thank the Julia programming language team, and the developers of `Cassette.jl` and `DifferentialEquations.jl`, without their software contributions, this work would be infeasible. We also thank Clayton Morrison, Adarsh Pyarelal and Rebecca Sharp for their advice on this manuscript. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00111990008.

6 Resources

Our documentation is hosted at `aske.gtri.gatech.edu/docs/latest`. Source code `github.com/jpfairbanks/SemanticModels.jl` where you will find instructions for getting started. Data sets can be found at `aske.gtri.gatech.edu/data`.

References

[1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *CoRR*, abs/1411.1607, 2014.

[2] Barry W. Boehm. Seven basic principles of software engineering. *Journal of Systems and Software*, 3(1):3 – 24, 1983.

[3] H2O.ai. H2o.ai - open source leader in ai and ml. `https://www.h2o.ai/`. (Accessed on 02/15/2019).

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[5] Cecilia Manzino and Alberto Pardo. A security types preserving compiler in Haskell. In Fernando Magno Quintão Pereira, editor, *Programming Languages*, pages 16–30, Cham, 2014. Springer International Publishing.

[6] Clayton Morrison. Automates: Automated model assembly from text, equations, and software. `https://ml4ai.github.io/automates/`, 2019. (Accessed on 02/15/2019).

[7] Evan Patterson, Ioana Baldini, Aleksandra Mojsilovic, and Kush R. Varshney. Teaching machines to understand data science code by semantic enrichment of dataflow graphs. *CoRR*, abs/1807.05691, 2018.

[8] Evan Patterson, Robert McBurney, Hollie Schmidt, Ioana Baldini, Aleksandra Mojsilovi, and Kush R. Varshney. Dataflow representation of data analyses: Toward a platform for collaborative data science. *IBM Journal of Research and Development*, 61(6):9:1–9:13, 2017.

[9] Christopher Rackauckas. Epicookbook: A cookbook of epidemiological models. `http://epirecip.es/epicookbook/chapters/simple`. (Accessed on 02/14/2019).

[10] David I. Spivak. Ologs: a categorical framework for knowledge representation. *CoRR*, abs/1102.1889, 2011.

[11] David I. Spivak. *Category Theory for the Sciences*. The MIT Press, 2014.

[12] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.