Theories of Everything

Pamela Zave
AT&T Labs—Research
Bedminster, New Jersey, USA
pamela@research.att.com

ABSTRACT

In 2025 semantic tools for software engineering will be mature, and their frequency of use in software development will still be disappointing. This proposal explains how research directed at building theories of everything (or, at least, important software domains) can consolidate progress and bring semantic tools into the mainstream of software practice.

1. INTRODUCTION

It is now 2025. In the last ten years the explosive improvement in "semantic" tools for software engineering, which was well underway in 2015, has continued. In addition to programming tools and environments, tools for analysis, verification, constraint satisfaction, optimization, and machine learning have all matured. These tools are powerful and easy to use. They solve problems that were infeasible in 2015.

At the same time, industry's heart has been won by DevOps, the successor to agile methods. In the DevOps framework, agile principles are extended to operations; development and operations engineers collaborate to create a smoothly running cycle of software change, deployment, virtualization, monitoring, performance tuning, and problem diagnosis. The prominent tools, used by developers and operations staff alike, do code configuration control, provisioning, load testing, measurement, and data visualization.

The divergence between these two worthwhile trends has become a major disappointment. The mathematical tools that reveal and optimize the semantic core of a software system are under-utilized in practice, despite the decades of research on which they are built, and the dramatic improvements in their quality. The engineering tools that work for all software systems, regardless of their purposes—on their code configuration, deployment, and resource usage—still bear almost all of the weight.

Our perspective on this situation is that the semantic tools are much easier to use than they were in 2015, but they are still difficult to use in the most important ways. Three points in particular highlight the situation:

Formal modeling: To use these tools on a software development problem, it is necessary to have a comprehensible, abstract formal model of important aspects of the problem. Producing good models has always been arduous and challenging, and that has not changed.

Tool knowledge: It may not be difficult to get a verifier or optimizer to run, but choosing the inputs properly and interpreting the outputs correctly is a subtle matter that often requires deep understanding of the underlying mathematics. Ideally all software engineers would have this understanding, but most of them do not. Although the sophistication of tools has grown tremendously, the time available to train software engineers has not grown at all, and this is a fundamental limitation.

Computational complexity: Although the sizes of problems that tools can handle are greatly increased, so are the sizes of problems to be solved. As in 2015, it is still not feasible to get good results from tools used in naive ways.

It should be clear why the research between 2015 and 2025 did not solve these problems. As in 2015, most papers on semantic tools were illustrated with simplistic examples showing little interest in real software domains. Yet the problems above are problems of applying semantic tools to particular software domains. As they require insight, relative judgments, and compromises, they cannot have one-size-fits-all solutions.

In this paper we argue that the only research that will solve these problems is research dedicated to understanding important software domains, to exploiting that understanding, and to packaging the results in re-usable ways. This is not a new idea, and unfortunately has proven insufficiently convincing in the past.

The new emphasis here is that the concept of a *mathematical theory* strengthens the old arguments in many ways. It shows that formal modeling of a domain is not an empty exercise, because there are strong relationships such as proof obligations among different artifacts (Section 2). These formal relationships in a domain theory reveal new opportunities for semantic tools to contribute directly to producing software (Section 3). The benefits of a domain theory are multiplied by generalization to families of domains (Section 4), and by theorems that reflect increasingly deep domain understanding (Section 5). The benefits can be multiplied further by composition of domain theories (Section 6).

In short, future research should be aimed at producing theories of everything (or at least important software domains). The work will be intellectually rigorous, and will

ICSE '16 Companion, May 14 - 22, 2016, Austin, TX, USA

ISBN 978-1-4503-4205-6/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2889160.2889213

demand the contributions of all available research methods. Most importantly, it will consolidate and amplify the impact of past research on semantic tools, as it eases their way into the mainstream of software development.[1]

2. SOFTWARE DEVELOPMENT AS THEORY-BUILDING

A mathematical theory has well-known parts, namely declarations, definitions, axioms, theorems, and proof obligations. This section explains how these parts relate to software system development and the use of semantic tools.

The example theory in this section supports development of a controller for a very specific simple network. The concept of a centralized network controller is a key part of today's major trend toward Software-Defined Networks (SDN). Our development problem is deliberately narrow and well-defined. In Section 4 we will discuss some ways in which such a theory can be generalized.

Domain knowledge: A mathematical theory is a theory of *something,* in this case a particular packet-switched network. Our purpose in building the theory is to produce software to control the network. Note that the network and the software controller are two different things; in common terminology, the network is the *domain* and the controller is the *system* that will control it. The theory is a theory *of the domain.*

The theory must have a formal representation of the static parts of the network, which are its nodes (networked machines) and links (one-way communication links between pairs of nodes). Nodes can be endpoints (sources and destinations of packets) or routers. Links have attributes such as length and bandwidth. Some of this information can be declared; where the expressive power of the formal declaration language fails, the remainder of the information must be expressed in a more powerful language as axioms.

The theory must also have a formal representation of the dynamic behaviors of the network. Because the set of possible behaviors is infinite and the actual behaviors are not known in advance, this representation can only constrain the possibilities.

A particular behavior of the network could be represented or modeled using three components: (1) for each endpoint, a timestamped stream of packets of which this node is the source, (2) for each endpoint, a timestamped stream of packets for which this node is the destination, and (3) for each router, a sequence of forwarding tables with timestamps indicating when each table is installed in the router. A forwarding table instructs a router what to do when it receives a packet. The entries of a forwarding table map packet headers onto actions such as "drop this packet" and "forward on outgoing link *k*."

In the theory, some axioms model operating assumptions about the network, such as the maximum rates at which endpoints can generate new packets. In the theory, other axioms model how all the network components behave to produce a particular behavior. When a node sends a packet on a link, the link delivers the packet at some later time to the node at its receiving end. When a router receives a packet, it looks up the header in the router's forwarding table, and performs the action mandated by the table. Although the axioms may be nondeterministic with respect to timing, for a given behavior of source streams (1) and forwarding tables (3), only some behaviors of destination streams (2) are possible.

Together all these declarations and axioms constitute a formal representation or model of *domain knowledge.* They record what we need to know about how the domain works.

Specification: The controller to be developed will provide and maintain the forwarding tables for the routers. So there must be an *interface* between the controller (system) and the network (domain) through which the controller can update the forwarding tables. This interface is a special part of the domain that is shared with the system. As with other parts of the domain, it is modeled formally with declarations and axioms.

The controller must have input as well as output. To give it the input it needs, the domain and interface must be augmented with various sensors such as packet logs and performance measurements. If a router receives a packet whose header has no match in its forwarding table, the router can ask the controller for a new table entry. Domain axioms relate network behavior to the controller inputs provided by these sensors.

The *specification* is a set of axioms, separate from domain knowledge, that prescribes how the controller should behave. More specifically, it prescribes how the controller inputs in the interface should relate to the controller outputs in the interface. The controller must be implemented to satisfy this specification.

Requirements: Network requirements come from its customers. Customers might wish certain destinations to be reachable from certain sources, and certain destinations to be *unreachable* from certain sources. They might give bandwidth and latency requirements for some traffic or all traffic.

All parts of the theory—including domain knowledge, specification, and requirements—can benefit from formal definitions that extend the vocabulary of the formal modeling. For example, we might define a *path* as a chain of links and routers that is traversed by packets with a particular header, from a source endpoint to a destination endpoint or router where they are dropped. Requirements might stipulate that paths have no loops. They might demand that two paths must be isolated, in the sense that they share no links or routers.

Requirements are conjectures that should be proved as theorems from the domain knowledge and specification. This is a crucial proof obligation of a software theory.

Use cases: Because the axioms of a theory usually take the form of constraints on the declared objects, there is a proof obligation to show that an instance of the theory exists, *i.e.,* that the constraints can be satisfied by something.

Software theories are complex, and this proof obligation must be taken very seriously. There should be a set of *use cases* or *test cases*, including both static structure and dynamic behavior, that illustrate how the domain is supposed to work when the specification is implemented and connected to the domain through the interface. Formally speaking, all the use cases should be instances of the theory.

Traditional software development: Domain knowledge, requirements, and specifications in a theory are all mathematical expressions. They can *describe behavior*, but they cannot *behave.*

[1]In this brief presentation, references are omitted. There is an expanded version with references at http://www2.research.att.com/~pamela/ToE.pdf.

In traditional software development, domain knowledge is unique among these three because it describes a real domain that does *behave*. The purpose of implementing the specification is to make a real thing that behaves as described by the specification. The basic way to do this is to create a program (another mathematical expression) in the instruction set of a computer, and install it on the computer, thus creating a real machine that behaves according to the program/specification. The final step of development is to connect the machine to the domain through the interface.

If the theory's proof obligations have been met, the result will be a domain that behaves according to the requirements and use cases.

3. EXPLOITING SEMANTIC TOOLS

If software development is approached as theory-building, then the formal artifacts provide many opportunities to use semantic tools. Most fundamentally, static analysis can help make these artifacts syntactically consistent and complete. Verification can be used to satisfy the proof obligations of the theory and to prove that the implementation (described formally as a program) satisfies the specification. If the theory contains quantitative properties and the tool technology can handle them, it may be possible to verify timing and reliability properties as well as logical ones.

In some cases it may be possible to apply semantic tools in even more powerful ways. Consider a moment in the life of our network, when the controller's internal state (obtained from domain knowledge, dynamic sensor data, and its own history) shows a particular profile of incoming traffic. The controller's job is to ensure that the next state of all the forwarding tables in the network is such that the traffic will be handled by the network in compliance with the requirements. With the help of all the information in the domain model, this is a constraint-satisfaction problem. If we had a sufficiently capable constraint solver, the constraints from domain knowledge and requirements could *be* the specification and the solver could *be* the implementation. This skips two development steps: deriving a specification from domain knowledge and requirements, and implementing the specification. The idea of constraint-solver-as-controller is adaptable to a wide range of real-time process-control and Internet of Things applications, at least in principle.

Alternatively, engineers could define a cost function on the links and routers of the network, and there could be a requirement to handle the traffic at the lowest cost. This would change the problem from constraint satisfaction to optimization, so that an optimizer would be the appropriate implementation tool.

In cases where computing a satisfactory or optimal state is not so straightforward or feasible, machine learning may be applicable. Reinforcement learning could help a controller keep link utilizations lower during peak periods, thus substituting for a predetermined specification and implementation. Even unsupervised learning could allow the controller to detect traffic anomalies that might indicate denial-of-service attacks in the making, in effect simplifying specification and implementation by simulating a highly-desirable sensor that does not exist.

These development shortcuts are not possible without formalization of domain knowledge and requirements. When these shortcuts become available, the benefits of formalization will be far greater than they were in 2015.

4. THEORY GENERALIZATION

It is unlikely that anyone would build a theory as elaborate as the example in Section 2 to describe a single network (domain). A better theory would describe a family of similar networks (domains), so that it could be used to develop controllers for all of them. Just as applying semantic tools increases the benefits of formalization, generalization dramatically decreases the marginal costs of formalization.

For the remainder of the paper, unless specifically noted, "domain" will be used collectively, referring to a family rather than an individual.

Whenever there is a generalized domain theory, there is a need to describe particular information about individual domains. This is built into the theory with *domain-specific languages*, which can substitute for any of the parts of a software theory. In the network example, a domain-specific configuration language can allow engineers to describe the nodes and links of a particular network. A domain-specific requirements language can allow customers to express their desired traffic policies. A domain-specific test language can allow customers and engineers to express use cases.

All of these languages need tool support, if only to translate them to general-purpose languages as inputs to other tools. A test language might be supported by a combination of drivers, simulators, and real domain components to see how a specification or implementation works.

The most powerful domain-specific languages so far express specifications. They are powerful because they can eliminate the implementation step of software development. Some domain-specific specification languages are executable, which means that there exists a machine that executes them directly, just as an ordinary computer executes its general-purpose instruction set. Another variation is to define a domain-specific specification language from which distinct implementations of distinct specifications can be generated automatically. For example, Verilog and VHDL are specification languages from which semiconductor chips can be fabricated.

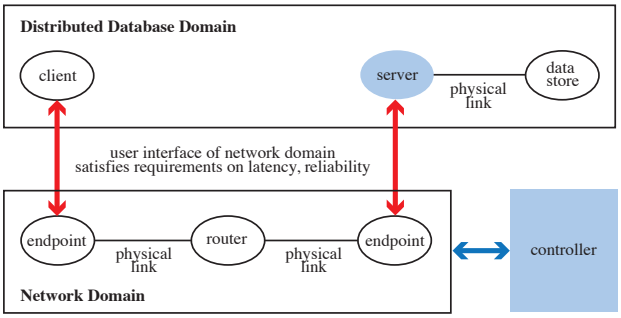
5. THEOREMS

Section 3 offered an enticing prospect: constraint solvers serve as all-purpose software systems! Realistically, however, general-purpose constraint solvers will never be powerful enough to replace all domain-specific software.

A better prospect is that domain theories will encourage the discovery of domain theorems—significant results that facilitate all aspects of software development for the domain. The best-understood aspect of networks is routing, and there are many theorems grounded in algebra, graph theory, and control theory that are exploited for network routing. These theorems relate routing algorithms (specifications) to network properties (requirements) such as latency, bandwidth, resource utilization, and fault-tolerance.

More recently, there has been exploration of the layered nature of networks, because many virtual networks are in fact built on top of physical networks. There is ongoing work toward theorems that relate mechanisms in networks and compositions of networks (specifications) to functional requirements for mobility and security.

To give a better-known example, in the domain of distributed databases, there are well-known trade-offs among consistency, availability, partition-tolerance, and latency. In



struggling to understand these trade-offs better, researchers are working toward a comprehensive theory of this important domain. The Holy Grail of distributed databases would be a theorem that tells us which points in the trade-off space are feasible. The theorem could lead to a tool that, for each feasible point, generates a distributed database system performing at the feasible point.

6. COMPOSITION OF DOMAIN THEORIES

Just as it is artificial to build a theory for an individual domain rather than a family of domains, it is also artificial to think of domains in isolation.

In software engineering, many domains of interest are *computational* domains, which means that they are made up of computational resources rather than "real-world" objects such as people, robots, institutions, documents, and farms. For our purposes, the main difference between computational and real-world domains is that computational domains always have interfaces through which they can be used by other domains. There is much to learn about how computational domains relate to each other. As an example, consider how a distributed database as mentioned in Section 5 is composed with a network (see figure).

In both domains, the objects of the domain are shown in black. Database clients and network endpoints both consist of software running as modules, threads, processes, or virtual machines on the shared resources of a computer. Note that when objects are aligned vertically, it means that they are running on the same computer.

In both domains, the software system to be developed and attached to the domain is shown in blue. Note that the domain/system boundary is simply defined by the boundaries of a particular software-development project. Once the software system is finished and deployed, it can be viewed as part of the computational domain.

The user interface of the network domain (in red, omitting dynamic policy changes) is the capability of the endpoints to accept messages from users in other domains, transmit them, receive them, and deliver them to users in the domain of origin. The user interface of a particular endpoint is available only to other domain objects on the same computer.[2] Some of the requirements on the network domain govern behavior at the user interface, including latency and reliability properties.

The distributed database domain without the system to be developed consists only of distributed clients and persistent data stores. Its requirements include the ability of the clients to access stored data with specific properties of consistency, availability, and latency. The requirements will be met by a distributed database system consisting of a collection of cooperating servers.

Domain knowledge in the database domain includes properties of the network imported from its user interface and requirements. The database domain uses the network domain for communication among its clients and servers. (Each data store has a local server, and communicates with it on a separate local link.) The ability of the servers to meet domain requirements on consistency, availability, and latency depends on the reliability and latency of the network service.

Doubtless there are many other relationships between software domains. If the relationships are sufficiently clear, software-development steps for them can be done in any order. The purpose of understanding domain relationships is to exploit these relationships in building and composing their theories.

Research questions concerning composition of domains overlap with many questions of software architecture that have been addressed in the past. The new slant is that each domain theory should be substantial in its own right, and tasks within each theory should be supported by effective semantic tools. This constrains the answers to questions in a constructive way, because the value provided by the tools must not be lost or undermined by architectural composition.

7. CONCLUSION

It should be obvious that building worthwhile theories for important domains (if not "everything," as in the title) involves solving hard research problems and embodying those solutions in an integrated, re-usable way. In doing so, it addresses the obstacles to tool use listed in the introduction.

Formal modeling: The formal model is supplied by the researchers who have created the generalized theory. Domain-specific languages make it easy for practitioners to customize the theory with specific details. Researchers have solved the problem of finding the important aspects of the domain and the right abstractions of them. Choices among general-purpose formal languages are not arbitrary because the chosen language must be able to express the necessary properties and serve as input to the semantic tools.

Tool knowledge: In choosing semantic tools for the theory, and defining the parts of the model that are their inputs and outputs, researchers must ensure that the interpretation of tools outputs is mathematically valid. Equally important, they must ensure that all the relevant operating assumptions are recorded in the domain model and marked for empirical validation by practitioners.

Computational complexity: Researchers must ensure that tools are capable of the jobs they are expected to do, at the scale of the real domain. This problem may be solved by incremental improvements in several areas. There might be empirical comparisons between competing tools. There might be carefully crafted limitations on the automated parts of the theory. And there might be domain-specific optimizations of general-purpose tools.

Of course, many such activities are going on in 2015, and may be well advanced by 2025. The more clearly researchers see their individual projects as unified by the goals of theory building, the more likely they are to make contributions that amplify the contributions of others.

[2]The interface is implemented by the operating system of the computer, which is another computational domain!