Hypergraph-Based Type Theory for Specifications-Conformant Code and Generalized Lambda Calculus: A case study in security protocols for biomedical devices

Nathaniel Christen

August 14, 2019

Abstract

Most CyberPhysical Systems are connected to a software hub which takes responsibility for monitoring, validating, and documenting the state of the system's networked devices. Developing robust, user-friedly central software is an essential project in any CyberPhysical Systems deployment. In this chapter, I will refer to systems' central software as their "software hub". Implementing software hubs introduces technical challenges which are distinct from manufacturing CyberPhysical devices themselves — in particular, devices are usually narrowly focused on a particular kind of data and measurement, while software hubs are multi-purpose applications that need to understand and integrate data from a variety of different kinds of devices. CyberPhysical software hubs also present technical challenges that are different from other kinds of software applications, even if these hubs are one specialized domain in the larger class of user-focused software applications.

Any software application provides human users with tools to interactively and visually access data and computer files, either locally (data encoded on the "host" computer running the software) or remotely (data accessed over a network). Computer programs can be generally classified as *applications* (which are designed with a priority to User Experience) and *background processes* (which often start and maintain their state automatically and have little input or visibility to human users, except for special troubleshooting circumstances). Applications, in turn, can be generally classified as "web applications" (where users usually see one resource at a time, such as a web page displaying some collection of data, and where data is usually stored on remote servers) and "native applications" (which typically provide multiple windows and Graphical User Interface components, and which often work with data and files saved locally — i.e., saved on the filesystem of the host computer). Contemporary sofware design also recognizes "hybrid" applications which combine features of web and of native (desktop) software.

Within this taxonomy, the typical CyberPhysical software hub should be classified as a native, desktop-style application, representing the state of networked devices through special-purpose Graphical User Interface (GUI) components. Networked CyberPhysical devices are not necessarily connected to the Internet, or communicate via Internet protocols. In many cases, software hubs will access device data through securitized, closed-circuit mechanisms which (barring malicious intrusion) ensure that only the hub application can read or alter devices' state. Accordingly, an application reading device data is fundamentally different than a web application obtaining information from an Internet server.[1] CyberPhysical networks are designed to prioritize real-time connections between device and software points, and minimize network latency. Ideally, human monitors should be able (via the centralized software) to alter device state almost instantaneously. Moreover, in contrast to Internet communications with the

[1] It may be appropriate for some device data — either in real time or retroactively — to be shared with the public via Internet connections, but this is an additional feature complementing the monitoring software's primary oversight roles.

TCP protocol, data is canonically shared between devices and software hubs in complete units — rather than in packets which the software needs to reassemble. These properties of CyberPhysical networks imply that software design practices for monitoring CyberPhysical Systems are technically different than requirements for web-based components, such as HTTP servers.

At the same time, we can assume that an increasing quantity of CyberPhysical data *will* be shared via the World Wide Web. This reflects a confluence of societal and technological forces: public demand is increasing for access both to conventional medical information and to real-time health-related data (often via "wearable" sensors and other technologies that, when properly deployed, can promote health-conscious lifestyles). Similarly, the public demands greater transparency for civic and environmental data, and science is learning how to use CyberPhysical technology to track ecological conditions and urban infrastructure — analysis of traffic patterns, for instance, can help civic planners optimize public transit routes (which benefit both the public and the environment).

Meanwhile, parralel to the rise of accessible health or civic data, companies are bringing to market an increasing array of software products and "apps" which access and leverage this data. These applications do not necessarily fit the profile of "hub software". Nevertheless, it is still useful to focus attention on the design and securitazation of hub software, because hub-software methodology can provide a foundation for the design of other styles of application that access CyberPhysical data. Over time, we may realize that relatively "light-weight" portals like web sites and phone apps are suboptimal for interfacing with CyberPhysical networks — too vulnerable and/or too limited in User Interface features. In that scenario, software used by the general public may adopted many of the practices and implementations of mainframe hub applications.

As I argued, software hubs have different design principles than web or phone apps. Because they deal with raw device data (and not, for example, primarily with local filesystem files), software hubs also have different requirements than conventional desktop applications. As CyberPhysical Systems become an increasingly significant part of our Information Technology ecosystem, it will be necessary for engineers to developed rigorous models and design workflows modeled expressly around the unique challenges and niche specific to CyberPhysical software hubs.

Hubs have at least three key responsibilities:

1. To present device and system data for human users, in graphical, interactive formats suitable for humans to oversee the system and intervene as needed.

2. To validate device and system data ensuring that the system is behaving correctly and predictably.

3. To log data (in whole or in part) for subsequent analysis and maintenance.

Prior to each of those capabilities is of course receiving data from devices and pooling disparate data profiles into a central runtime-picture of device and system state. It may be, however, that direct device connection is proper not to the software hub itself but to drivers and background processes that are computationally distinct from the main application. Therefore, a theoretical model of hub software design should assume that there is an intermediate layer of background processes separating the central application from the actual physical devices. Engineers can assume that these background processes communicate information about device state either by exposing certain functions which the central application can call (analogous to system kernel functions) or by sending signals to the central application when devices' state changes. I will discuss these architectural stipulations more rigorously later in this chapter.

Once software receives device data, it needs to marshal this information between different formats, exposing the data in the different contexts of GUI components, database storage, and analytic review. Consider the example of a temperature reading, with GPS device location and timestamp data (therefore a four-part structure giving temperature at one place and time). The software needs, in a typical scenario, to do several things with this information: it has to check the data to ensure it fits within expected ranges (because malformed data can indicate physical malfunction in the devices or the network). It may need to show the temperature reading to a human user via some visual or textual indicator. And it may need to store the reading in a database for future study or troubleshooting. In these tasks, the original four-part data structure is transformed into new structures which are suitable for verification-analytics, GUI programming, and database persistence, respectively.

The more rigorously that engineers understand and document the morphology of information across these different software roles, the more clearly we can define protocols

for software design and user expectations. Careful design requires answering many technical questions: how should the application respond if it encounters unexpected data? How, in the presence of erroneous data, can we distinguish device malfunction from coding error? How should application users and/or support staff be notified of errors? What is the optimal Interface Design for users to identify anomalies, or identify situations needing human intervention, and then be able to perform the necessary actions via the software? What kind of database should hold system data retroactively, and what kind of queries or analyses should engineers be able to perform so as to study system data, to access the system's past states and performance?

I believe that the software development community has neglected to consider general models of CyberPhysical sofware which could answer these kinds of questions in a rigorous, theoretically informed manner. There is of course a robust field of cybersecurity and code-safety, which establishes Best Practices for different kinds of computing projects. Certainly this established knowledge can and does influence the implementation of software connected to CyberPhysical systems no less than any other kind of software. But models of programming Best Practices are often associated with specific coding paradigms, and therefore reflect implementations' programming environment more than they reflect the empirical domain targeted by a particular software project.

For example, Object-Oriented Programming, Functional Programming, and Web-Based Programming present different capabilities and vulerabilities and therefore each have their own "Best Practices". As a result, our understanding of how to deploy robust, well-documented and well-tested sofware tends to be decentralized among distinct programming styles and development environments. External analysis of a code base — e.g., searching for security vulnerabilities (attack routes for malicious code) — are then separate disciplines with their own methods and paradigms. Such dissipated wisdom is unfortunate if we aspire to develop integrated, broadly-applicable models of CyberPhysical safety and optimal application design, models which transcend paradigmatic differences between coding styles and roles (treating implementation, testing, and code review as distinct technical roles, for instance).

It is also helpful to distinguish cyber *security* from *safety*. When these concepts are separted, *security* generally refers to preventing *deliberate*, *malicious* intrusion into CyberPhysical networks. Cyber *safety* refers to preventing unintended or dangerous system behavior due to innocent human error, physical malfunction, or incorrect programming. Malicious attacks — in particular the risks of "cyber warfare" — are prominent in the public imagination, but innocent coding errors or design flaws are equally dangerous. Incorrect data readings, for example, led to recent Boeing 737 MAX jet accidents causing over 200 fatalities (plus the worldwide grouding of that airplane model and billions of dollars in losses for the company). Software failures either in runtime maintenance or anticipatory risk-assessment have been identified as contributing factors to high-profile accidents like Chernobyl [**?**] and the Fukushima nuclear reactor meltdown [**?**]. A less tragic but noteworthy case was the 1999 crash of NASA's US $125 million Mars Climate Orbiter. This crash was caused by software malfunctions which in turn were caused by two different software components producing incompatible data — in particular, using incompatible scales of measurement (resulting in an unanticipated mixture of imperial and metric units). In general, it is reasonable to assume that coding errors are among the deadliest and costliest sources of man-made injury and property damage.

Given the risks of undetected data corruption, seemingly mundane questions about how CyberPhysical applications verify data — and respond to apparent anomalies — become essential aspects of planning and development. Consider even a simple data aggregate like blood pressure (combining systolic and diastolic measurements). Empirically, systolic pressure is always greater than diastolic. Software systems need to agree on a protocol for encoding the number to ensure that they are in the correct order, and that they represent biologically plausible measurements. How should a particular software component test that received blood pressure data is accurate? Should it always test that the systolic quantity is indeed greater than the diastolic, and that both numbers fall in medically possible ranges? How should the component report data which fails this test? If such data checking is not performed — on the premise that the data will be proofed elsewhere — then how can this assumption be justified? How can engineers identify, in a large and complex software system, all the points where data is subject to validation tests; and then by modeling the overall system in term of these check-points ensure that all needed verifications are performed at least one time? To take the blood-pressure example, how would a software procedure that *does* check the integrity of the systolic/diastolic pair indicate for the overall system model that it performs that particular verification? Conversely, how would a procedure which does *not* perform that verification indi-

cate that this verification must be performed elsewhere in the system to guarantee that the procedure's assumptions are satisfied?

These questions are important not only for objective, measurable assessments of software quality, but also for people's more subjective trust in the reliability of software systems. In the modern world we allow sofware to be a determining factor, in places where malfunction can be fatal — airplanes, hospitals, electricity grids, trains carrying toxic chemicals, highways and city streets, etc. Consider the model of "Ubiquitous Computing" pertinent to the book series to which this volume (and hence this chapter) belongs. As explained in the series introduction:

> U-healthcare systems ... will allow physicians to remotely diagnose, access, and monitor critical patient's symptoms and will enable real time communication with patients. [This] series will contain systems based on the four future ubiquitous sensing for healthcare (USH) principles, namely i) proactiveness, where healthcare data transmission to healthcare providers has to be done proactively to enable necessary interventions, ii) transparency, where the healthcare monitoring system design should transparent, iii) awareness, where monitors and devices should be tuned to the context of the wearer, and iv) trustworthiness, where the personal health data transmission over a wireless medium requires security, control and authorize access.[2]

Observe that in this scenario, patients will have to place a level of trust in Ubiquitous Health technology comparable to the trust that they place in human doctors and other health professionals.

All of this should cause software engineers and developers to take notice. Modern society places trust in doctors for well-rehearsed and legally scrutinized reasons: physicians need to rigorously prove their competence before being allowed to practice medicine, and this right can be revoked due to malpractice. Treatment and diagnostic clinics need to be licenced, and pharmaceuticals (as well as medical equipment) are subject to rigorous testing and scientific investigation before being marketable. Notwithstanding "free market" ideologies, governments are aggressively involved in regulating medical practices; commercial practices (like marketing) are constrained, and operational transparency (like reporting adverse outcomes) is mandated, more so than in most other sectors of the economy. This level of oversight *causes* the public to trust that clinicians' recommendations are usually correct, or that medicines are usually beneficial more than harmful.

The problem, as software becomes an increasingly central feature of the biomedical ecosystem, is that no commensurate oversight framework exists in the sofware world. Biomedical IT regulations tend to be ad-hoc and narrowly domain-focused. For example, code bases in the United States which manage HL-7 data (the current federal Electronic Medical Record format) must meet certain requirements, but there is no comparable framework for software targeting other kinds of health-care information. This is not only — or not primarily — an issue of lax government oversight. The deeper problem is that we do not have a clear picture, in the framework of computer programming and software development, of what a robust regulatory framework would look like: what kind of questions it would ask; what steps a company could follow to demonstrate regulatory compliance; what indicators the public should consult to check that any software that could affect their medical outcomes is properly vetted. And, outside the medical arena, similar comments could be made regarding software in CyberPhysical settings like transportation, energy (power generation and electrical grids), physical infrastructure, environmental protections, government and civic data, and so forth — settings where sofware errors threaten personal and/or property damages.

In the case of personal medical data, as one example, there is general aggreement that data should be accessed when it is medically necessary — say, in an emergency room — but that each patient should mostly control how and whether their data is used. When data is pooled for epidemiological or meta-analytic studies, we generally believe that such information should be anomalized so that socioeconomic or "cohort" data is considered, whereas unique "personal" data remains hidden. These seem like common-sense requirements. However, they rely on concepts which we may intuitively understand, but whose precise definitions are elusive or controversial. What exactly does it mean to distinguish uniquely *personal* data, that is indelibly fixed to one person and therefore particularly sensitive as a matter of due privacy, from *demographic* data which is also personal but which, tieing patients to a cohort of their peers, is of potential public interest insofar as race, gender, and other social qualities can sometimes be

[2] https://sites.google.com/view/series-title-ausah/home?authuser=0

statistically significant? How do privacy rights intersect with the legitimate desire to identify all scientific factors that can affect epidemiological trends or treatment outcomes? More deeply, how should we actually demarcate *demographic* from *personal* data? What details indicate that some part of some data structure is one or the other?

More fundamentally, what exactly is data sharing? What are the technical situations such that certain software operations are to be *sharing* data in a fashion that triggers concerns about privacy and patient oversight? Although again we may intuitively picture what "data sharing" entails, producing a rigorous definition is surprisingly difficult.

In short, the public has a relatively inchoate idea of issues related to cyber safety, security, and privacy: we (collectively) have an informal impression that current technology is failing to meet the public's desired standards, but there is no clear picture of what IT engineers can or should do to improve the technology going forward. Needless to say, software should prevent industrial catastrophes, and private financial data should not be stolen by crime syndicates. But, beyond these obvious points, it is not clearly defined how the public's expectations for safer and more secure technology translates to low-level programming practices. How should developers earn public trust, and when is that trust deserved? Maxims like "try to avoid catastrophic failure" are too self-evident to be useful. We need more technical structures to identify which coding practices are explicitly recommended, in the context of a dynamic where engineers need to earn the public trust, but also need to define the parameters for where this trust is warranted. Without software safety models rooted in low-level computer code, software safety can only be ex-post-facto engineered, imposing requirements relatively late in the development cycle and checking code externally, via code review and analysis methods that are sepated from the core development process. While such secondary checking is important, it cannot replace software built with an eye to safety from the ground up.

This chapter, then, is written from the viewpoint that cyber safety practices have not been clearly articulated at the level of software implementation itself, separate and apart from institutional or governmental oversight. Regulatory oversight is only effective in proportion to scientific clarity vis-à-vis desired outcomes and how technology promotes them. Drugs and treatment protocols, for instance, can be evaluated through "gold standard" double-blind clinical trials — alongside statistical models, like "five-sigma" criteria, which measure scientists' confidence that trial results are truly predictive, rather than results of random chance. This package of scientific methodology provides a framework which can then be adopted in legal or legislative contexts. Continuing the example, policy makers can stipulate that pharmaceuticals need to be tested in double-blind trials, with statistically verifiable positive results, before being approved for general-purpose clinical use. Such a well-defined policy approach *is only possible* because there are biomedical paradigms which define how treatments can be tested to maximize the chance that positive test results predict similar results for the general patient population.

Analogously, a general theory of cyber safety has to be a software-design issue before it becomes a policy or contractual issue. It is at the level of low-level software design — of actual source code in its local implementation and holistic integration — that engineers can develop technical "best practices" which then provide substance to regulative oversight. Stakeholders or governments can recommend (or require) that certain practices adopted, but only if engineers have identified practices which are believed, on firm theoretical ground, to effectuate safer, more robust software.

This chapter, then, considers code-safety from the perspective of computer code outward; it is grounded on code-writing practice and in the theoretical systems which have historically been linked to programming (like type theory and lambda calculus), yielding its scientific basis. I assume that formal safety models formulated in this low-level context can propagate to institutional and governmental stake-holders, but discussion of the legal or contractual norms that can guide software practice are outside the chapter's central scope.

In the CyberPhysical context, I assume here that the most relevant software projects are hub applications; and that the preeminent issues in cyber safety are validating data and responding safely and predictably to incorrect or malformed data. Here we run into gaps between proper safety protocols and common programming practice and programming language design. In particular, most mainstream languages have limited *language-level* support for foundational safety practices such as dimensional checking (ensuring that algorithms do not work with incommensurate measurement axes) or range checking (ensuring that inaccurate CyberPhysical data is properly identified as such — in the hopes of avoiding cases like the Boeing 737 crashes, where onboard software failed to recognize inaccurate data from angle-of-attack sensors). More robust safety models are often implicit in software

libraries, outside the core language; however, to the degree that such libraries are considered experimental, or tangential to core language features, they are not likely to "propagate" outside the narrow domain of software development proper. To put it differently, no safety model appears to have been developed in the context of any mainstream programming language far enough that the very existence of such a model provides a concrete foundation for stakeholders to define requirements that developers can then follow.

This chapter's discussion will be oriented toward the C++ programming language, which is arguably the most central point from which to consider the integration of concerns — GUI, device networking, analytics — characteristic of CyberPhysical hub software. In practice, low-level code that interfaces with devices (or their drivers) might be written in C rather than C++; likewise, there is often a role for functional programming languages — even theorem-proving systems — in mission-critical data checking and system design validation. But C++ is unique in having extensive resources traversing various programming domains, like native GUI components alongside low-level networking and logically rigorous data verification. For this reason C++ is a reasonable default language for examining how these various concerns interoperate.

In that spirit, then, the C++ core language is a good case-study in language-level cyber-safety support (and the lack thereof). There are numerous C++ libraries, mostly from scientific computing, which provide features that would be essential to a robust cyber safety model (such as bounded number types and unit-of-measurement types). If some version of these libraries were adopted into a future C++ standard (analogous to the "concepts" library, a kind of metaprogramming validator, which has been included in C++20 after many years of preparation), then C++ coders would have a canonical framework for safety-oriented programming — a specific set of data types and core libraries that could become an essential part of critical CyberPhysical components. That specific circle of libraries, along with their scientific and computational principles, would then become a "cyber safety model" available to CyberPhysical applications. Moreover, the existence of such a model might then serve as a concrete foundation for defining coding and project requirements. Stakeholders should stipulate that developers use those specific libraries intrinsic to the cyber safety model, or if this is infeasible, alternate libraries offering similar features.

Of course, the last paragraph was counterfactual — *without* such a canonical "cyber safety model", there is no firm foundation for identifying stakeholder priorities. We may have generic guidelines — try to protect against physical error; try to restrict access to private data — but we do not have a canonical model, integrated with a core language, against which compliant code can be designed. I believe this is a reasonable claim to make in the context of C++, and most or all other mainstream programming languages as well.

But this situation also implies that language designers and library developers can play a lead role in establishing a safety-oriented CyberPhysical foundation. Insofar as this foundation lies in programming languages and software engineering — in data types, procedural implementations, and code analytics — then the responsibility for developing a safety-oriented theory and practice lies with the software community, not with CyberPhysical device makers or with civic or institutional stakeholders. The core principles of a next-generation CyberPhysical architecture would then be worked out in the context of software language design and software-based data modeling. My goal in this chapter is accordingly to define what I believe are fundamental and canonical structures for theorizing data structures and the computer code which operates on them, with an eye toward cyber safety and Software Quality Assurance.

In general, software requirements can be studied either from the perspective of computer code, or from the persepctive of data models. Consider again the requirement that systolic blood pressure must always be a greater quantity than diastolic: we can define this as a precondition for any code which displays, records, or performs computations on blood pressure (e.g. comparing a patient's pressure at different times). Such code is only operationally well-defined if it is provided data conforming to the systolic-over-diastolic mandate. The code *should not* execute if this mandate fails. Design and testing should therefore guarantee that the code *will not* execute inappropriately. Conversely, these same requirements can be expressed within a data model: a structure representing blood pressure is only well-formed if its component part (or "field") representing systolic pressure measures greater than its field representing diastolic pressure.

These perspectives are complementary: a database which tracks blood pressure should be screened to ensure that all of its data is well-formed (including systolic-over-diastolic). At the same time, an application which works with medical data should double-check data when relevant procedures are called (e.g., those working with blood pressure),

particularly if the data is from uncertain provance. Data could certainly come from multiple databases, or perhaps directly from CyberPhysical devices, and developers cannot be sure that all sources check their data with sufficient rigor (moreover, in the case of CyberPhysical sensors, validation in the device itself may be impossible).

Conceptually, however, validation through data models and code requirements represent distinct methodologies with distinct theoretical backgrounds. This chapter will therefore consider both perspectives, as practically alligned but conceptually *sui generis*. I will also, however, argue that certain theoretical foundations — particularly hypergraph-based data representation, and type systems derived from that basis — serve as a unifying element. I will therefore trace a construction of *hypergraph-based* type theory across both data- and code-modeling methodologies.