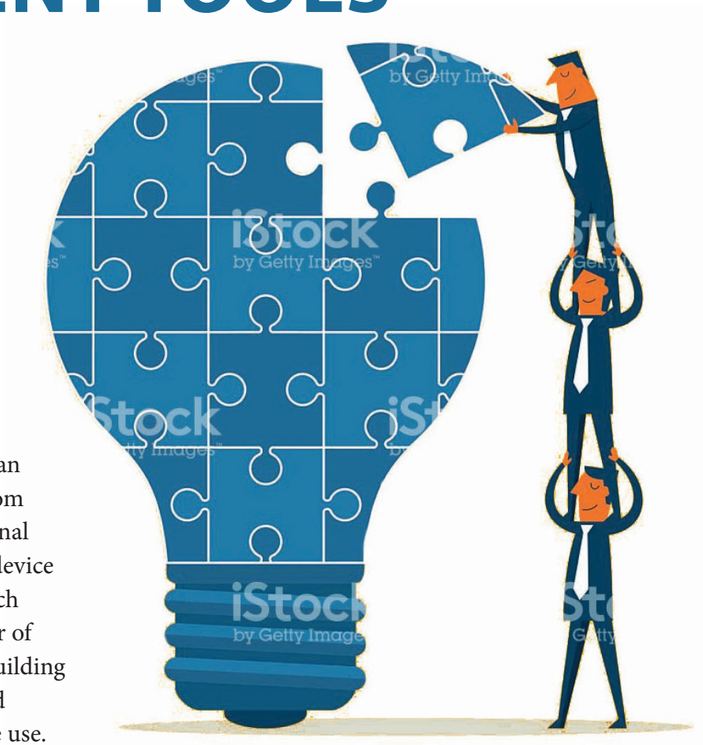
Rohit Ramesh and Prabal Dutta *University of Michigan*

Editors: Prabal Dutta and Iqbal Mohomed

EMBEDDED DEVELOPMENT TOOLS REVISITED:

Verification and Generation from the Top Down



Illustration, istockphoto.com

We envision a future where any programmer can automatically generate an embedded device from application code. A future in which a professional embedded engineer can "sketch" out a partial device design and automation can complete it, in which a hobbyist can write a program and, in a matter of days, receive a custom hardware in the mail. Building this future requires us to rethink the embedded development process and redesign the tools we use.

The transition towards co-design tools over the last few decades has brought embedded system development to a fork in the road. Our current path involves piecing together co-design features in an ad-hoc and unprincipled manner by making incremental changes to existing development tools. Instead, we should proceed down a second path, taking the opportunity to radically redesign the architecture of our embedded development tools. We propose *single model* development tools that support co-design and verification workflows, and also enable entirely new innovations and manufacturing processes. The first road is easier to travel, but the second road will take us to a whole new world.

Embedded devices make up a 140 billion dollar market that includes medical electronics, factory control systems, automotive computers, and the Internet of Things. The complexity of embedded applications continues to grow, making embedded developers rely progressively more on tools that can detect errors early in the design process. The longer an error goes undetected, the costlier it is to find and correct, so improved verification tools can dramatically decrease the cost and increase the speed of embedded development.

Despite the importance of verification, modern embedded tools have difficulty verifying systems holistically. This is because modern tools split the development process into three *design domains*: the source code, for the device's control logic; the schematic,

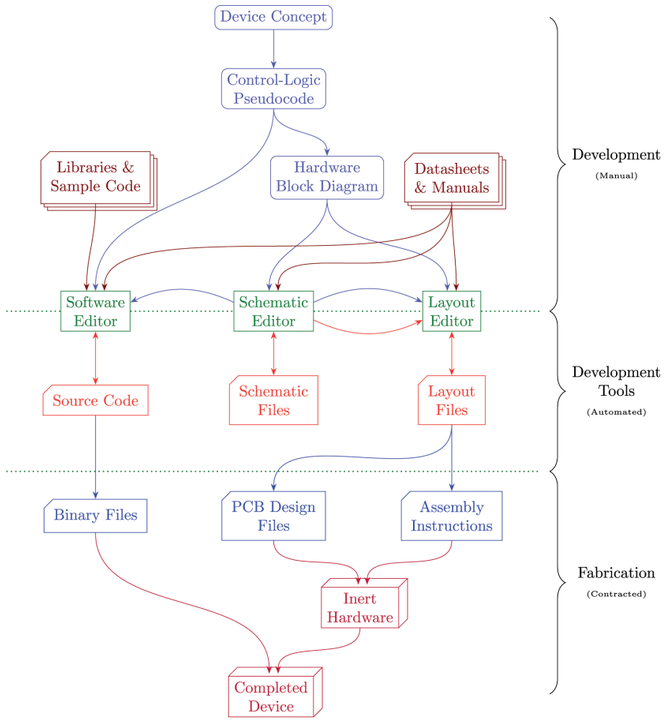
for the device's electrical connectivity; and the PCB layout, for the device's physical structure. This split keeps modern tools from providing strong guarantees to developers about the correctness of their designs, and forces developers to spend time manually collating metadata and checking for errors over long design-build-test cycles.

The need for holistic system verification has led to the increased importance of co-design development tools [3]. Co-design tools explicitly model the relationships between the different domains of an embedded system. The ability to capture the interactions and dependencies between different domains makes co-design tools well-suited for optimization, simulation, and verification of complex cyber-physical systems.

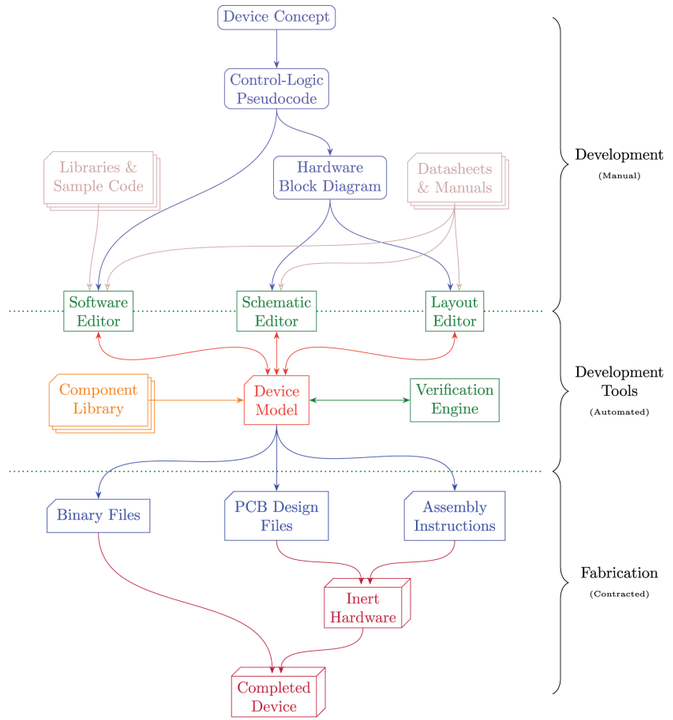
However, it is not clear how to best incorporate co-design into the embedded development tools that we already have. The most common current approach opportunistically implements individual co-design features by stitching together parts of domain-specific tools, and produces only incremental improvements. Our alternative approach takes a step back to fundamentally restructure embedded development tools to support holistic verification. We believe this approach is likely to pay large dividends.

We propose refactored architectures that follow a *single model* paradigm. Single model tools create a unified view of an embedded system by consolidating the design domains that modern tools hold separate. This architecture captures the inter-domain constraints and dependencies that are usually manually maintained. While this makes verification much easier, it comes with a high startup cost. The single model approach relies upon a powerful formal representation of the embedded design space that must be carefully crafted. In addition, single model tools require an extensive component library expressed within that formalism, compiled from the components' specifications and other metadata.

Though single model tools require more up-front effort to implement, the potential benefits are tremendous. Not only do single model tools support holistic verification of complete embedded systems; they can also combine with advances in programming language research to enable a revolutionary new embedded development process called *automatic device generation*.



(a) The architecture of modern tools
Each editor uses its own internal representation for its portion of the design. This stovepiped structure makes it difficult to create the global view of a device's design needed for strict verification. In addition, metadata needed during development is held outside the development tools, and forces the developer to manually incorporate that information into the design process. Other development chains, like those for FPGAs or PSoCs, would generally form parallel stovepipes that further fracture the development process.



(b) The single model architecture
Instead of a stovepiped structure, each editor now modifies the shared device model. The device model also serves as a global view of a device's design, which can be used by the verification engine. Much of metadata needed for embedded development has been pulled into a component library, reducing the need for external metadata. Despite these changes to the development tools, the broader embedded development process stays largely the same for developers and fabricators.

FIGURE 1. The Architectures of Modern and Single Model Tools

Automatic device generation (ADG) compiles high-level application logic into embedded hardware. ADG transforms the problem of hardware development into a software development problem. This opens up hardware development to anyone who can code, dramatically expanding the labor market and lowering the cost of hardware innovation. ADG will allow even novice programmers to quickly prototype new embedded applications. This has implications for a broad range of people. Hobbyists could satisfy a wide variety of niche needs with small-batch manufacturing. Scientists could pursue research that needs embedded control systems or sensor networks without the need to learn an additional field of engineering. Professional embedded engineers could design the critical features of new devices, generate the rest of their systems, and move quickly to testing.

We believe that holistic verification and automatic device generation will have a major impact on embedded system innovation, and thus on society. However, both rely on fundamental changes to embedded development tools. If we take the time to revisit embedded development tools now, we can ensure a prosperous tomorrow.

MANAGING EMBEDDED SYSTEM COMPLEXITY THROUGH STRICT VERIFICATION SEMANTICS

As embedded systems grow more complex, developer error becomes more likely and more costly. Managing this complexity requires development tools that can quickly catch developer errors before the cost of correcting them grows. Like most forms of engineering, embedded development is a cycle of designing, building, and testing. The hardware fabrication stage of this cycle usually takes days or weeks, depending on how much one is willing to pay. On the other hand, reprogramming existing hardware takes minutes and is inexpensive. Hardware errors are costly, while software errors are cheap. This dynamic, where costs are concentrated in hardware fabrication, means developer tools that detect hardware errors can significantly lower the costs of embedded development.

However, to substantially reduce a developer's workload, tools need to do more than detect a subset of individual hardware errors – they need to provide reliable guarantees of correctness. Modern tools tend to be *permissive verifiers*, which only throw errors when they are confident that the design is broken. When a permissive verifier reports no hardware errors, it does not mean that the device's hardware will work. Lacking this guarantee, engineers need to check the design manually to find errors outside the scope of their tools. This is particularly problematic for inexperienced embedded developers, who are both the least likely to know what errors they must check for and the most likely to make those errors. Permissive verification does not serve developers or students well, at best detecting some errors sooner while leaving behind a large amount of manual labor.

Embedded development tools should instead provide *strict verification*, which guarantees that the device's hardware is correct if no errors are raised. Strict verification's strong guarantee means that if no errors are found, the developer's work is finished. The downside is that strict verifiers will show errors for some correct designs, because the verifier cannot understand the design. However, we believe that manually examining a finite list for false positives is easier than checking an entire design for an unknown error, especially as devices grow more complex. Strict verifiers can also act as guides for students and developers, pointing them towards potential issues and telling them when development can stop.

In order to support strict verification, development tools need to capture relationships between design domains, something that modern tools are ill-equipped to do. Consider an embedded device whose software reads data from a port connected to a thermometer, but accidentally treats that data as input from a microphone. A software-only verifier can see that the program expects the port to act as an interface to an audio stream. A hardware-only verifier can see that the port is capable of supplying temperature data because it is connected to a thermometer. The mismatch between interface and capability only becomes visible when both pieces of information are taken together. When modern embedded development tools split the design domains, they discard the information needed to connect a software interface to the corresponding hardware capability.

Strict verification also needs access to rich metadata that specifies what each component does and what resources it requires to function. For instance, checking that a thermometer will function requires

verification tools to know which pins are connected to power, what voltage the component requires, and the format in which it sends data. Modern development tools do not carry most of this information, so developers must trawl through datasheets, manuals, and numerous other sources to gather it. If verification tools had access to rich metadata, engineers could save the time they spend manually searching for metadata and verifying that components are correctly connected.

Modern development tools – whose architecture is shown in Figure 1(a) – cannot be simply augmented to support strict verification. Each design domain is given its own internal representation and file format, which are not directly connected. In addition, the metadata available to existing tools is poor, serving mainly as a set of common design elements with human-readable labels. The design of modern tools assumes that the design domains are independent and relies on the engineers to make sure the complete design is internally consistent. The ensuing variety of backend representations makes it difficult to incorporate a global design view, rich metadata, and strict verification into the architecture of modern design tools.

AN EMBEDDED DEVELOPMENT TOOL ARCHITECTURE FOR STRICTLY VERIFIABLE CO-DESIGN

In order to build a useful strict verifier, we need a new development tool architecture that unifies disparate design domains into a single model view. The architecture that we advocate – shown in Figure 1(b) – is divided into three major parts. The *device model* is the core of the single model model architecture and stores design information that is currently split between source code, schematics, and PCB layouts. The *component library* acts as a searchable store of software and hardware, all annotated with rich metadata and ready to be used within the device model as part of an embedded design. The *verification* system examines the device model to determine whether each design is viable before passing any errors to the appropriate editor. The editors, instead of acting in separate design domains, all act as windows into the device model showing the same slices of each design as modern editors.

There are several properties that device models must have in order to support efficient design practices, minimize developer work, and be used with strict verification tools. Device models need to be modular and composable so that developers can design embedded sub-systems once and reuse them in future designs. Device models should use existing embedded programming languages and support existing source code. Finally, device models need to allow all elements of each design to be annotated with metadata that can be used by the verifier. Work in cyber-physical systems [2] and data structures can be incorporated into the design of device models to provide these properties while accurately capturing the entire structure of an embedded device.

We envision single model verification as a system that layers constraints onto a device model. These constraints capture properties that an embedded device must have in order to function correctly. The verification system will be made of three parts. *Verification plugins* capture one aspect of embedded development, encoding limitations within its domain as a set of constraints attached to a device model. For instance, a bus-management plugin may add a constraint to the device model specifying that "No two devices on this bus may share an address." The *verification engine* executes plugins against a device model while collecting the generated constraints into a monolithic constraint satisfaction problem. The verification engine then solves the constraint satisfaction problem before forwarding any unsatisfiable constraints to the developer as errors. The *verification framework* defines a specification for plugins and define how the verification engine operates. The framework acts as the mathematical glue for verification, defining how plugins are written, how they are combined and evaluated, and proving that the process as a whole is sound.

Implementation work for single model tools can be minimized through reuse of existing systems and careful design. Existing compilers can use code within the device model after it is stripped of annotations and metadata. In order to provide a familiar interface for embedded professionals, the frontends on modern tools can be directly incorporated into single model tools. The component library can use templating and inheritance to allow large classes of common components to be added quickly. Verification can be built incrementally, with new plugins each making the verifier more accurate. Single model tools can be designed to minimize implementation effort, making them a feasible project for the academic community.

Elements of single model tools can also provide a foundation for other short-term improvements. The verification engine can search through the component library for parts that bring the device closer to completion, acting as auto-complete for embedded designs. The device model contains information that can be used by auto-placement and auto-routing tools to capture the same PCB design intuition that developers use. The device model can also act as a framework for simple co-simulation, automatically merging analog and digital simulations to create a holistic device simulation.

Open formats and specifications are key to the adoption of the single model architecture. Early implementations can be built on top of open source development tools like KiCAD and Fritzing, allowing their users to quickly take advantage of better verification without changing their workflows. Likewise, an open component library would allow hobbyists and professionals to contribute parts and create a robust ecosystem for less experienced users. This process can allow an single model tool ecosystem to grow large enough that parts manufacturers are incentivize to contribute. Open standards for the device model, component library, and verification framework can catalyze both commercial investment in single model tools and research efforts to improve them.

Single model tools are the next major step in embedded development. They fit directly into the modern development process, because they reuse user interfaces and output formats. Research and development work is needed to create these tools, but that work builds on existing knowledge and infrastructure. By providing a basis for strict verification, single model tools will make developers more efficient, reduce hardware errors, and provide a solid foundation for more advanced tools.

AUTOMATIC GENERATION OF EMBEDDED DEVICES

Single model tools provide numerous benefits, but also provide the building blocks for more powerful development processes. Cloud computing, commodity hardware, and better distributed systems are making access to large amounts of compute power cheaper than ever. These trends, when combined with verification tools and component libraries, will allow valuable engineer time to be replaced with cheap compute cycles. *Embedded device generation* will allow engineers to design the most important portions of an embedded device before filling in missing portions with automated searches.

We envision device generation iterating over a partial device model, adding components and removing errors. At each step, verification errors identify what keeps the design from functioning. The component library will have parts that can fix these errors, creating a set of new partial designs. The verifier can vet the new designs, removing those that cannot be fixed. This new set of designs will be a step closer to completion, and repeating the process will lead to a design that passes verification entirely. This device generation process pulls from the component library and thus cannot write any of the unique control logic for a device. However, device generation will automatically add supporting infrastructure like power management systems, ports for programming, or connections for data.

This is fundamentally a search process, which allows us to draw on the rich body of work on search quality and search optimization. Machine learning techniques can be used to examine existing designs and extract common patterns for reuse. Developers can choose heuristics to bias the search and modify the design priorities of device generation. Divide-and-conquer approaches can turn a monolithic search into multiple smaller problems. These techniques make device generation feasible, and allow it to automate large parts of the embedded development process.

GENERATING HARDWARE FROM SOFTWARE

Single model tools and device generation will allow the creation of new embedded device languages. These languages will look like modern embedded software, but act as self-contained descriptions for embedded devices, compiling into a finished device model. Device languages will reduce the problem of hardware development into one of software development and will allow anyone who can code to design hardware.

Modern embedded software is written as a description of what a processor does, rather than a description of what an embedded device will do. Instead of a developer specifying that the status LED should turn on, the developer has to describe what an output pin on the microcontroller does as well as ensuring that manipulating the output pin changes the state of the LED. This perspective forces embedded developers to design much of a device's hardware before writing the software and before device generation tools can complete their work.

We envision new high-level embedded design languages that change this, allowing software to act as a self-contained specification that device generation tools can use directly. Consider the example in Figure 2 for a simple temperature controller. This program has two major components: a declaration of the necessary components and the runtime logic needed to control them. While this is a simple example, the component declarations illustrate how a developer specifies the type of component needed and the properties that component must have. This means the developer does not need to choose a specific thermometer or heater, and device generation can automate that choice.

The control logic in Figure 2 looks like modern embedded software, if that software was built with well-designed libraries and used good variable names. The key difference between the two is semantic, with the new languages taking over what are now developer tasks. Writing modern embedded code requires the developer to find the libraries and drivers for the components they use as well as initialize them with information about how those components connect to the microcontroller. High-level embedded design languages will instead have common interfaces for major component types and allow device generation tools to choose the necessary libraries and generate their initialization code.

Figure 2 is how we envision the earliest versions of these languages, but they can grow to be much more powerful. Early device languages will likely use a hub-and-spoke model, with all the code running on

```
// Component Declarations
component thermometer = new Thermometer(
  immersion, min-temp <= 0c,
  max-temp >= 100c);
component heater = new Heater(immersion,
  power > 10w);
component cooler = new Cooler(immersion);
component status = new RGB-LED();

// Run-Time Control Logic
fn main(){
  while(true){
    if(thermometer.temp() < 16c){
      // Water temp too low
      cooler.off();
      heater.on();
      status.setColor(Blue);
    } else if(thermometer.temp() > 20c){
      // Water temp too high
      heater.off();
      cooler.on();
      status.setColor(Red);
    } else {
      // Water temp just right
      heater.off();
      cooler.off();
      status.setColor(Green);
    }
  }
}
```

FIGURE 2. This program acts as a self-contained specification and implementation for a temperature controller. The initial section acts as a declaration of the components the controller must contain, as well as the properties those components must have. The second section is a description of how the device should act, written as if the declared components had a well-defined generic software API. The program as a whole acts as a specification that embedded device generation tools attempt to satisfy when generating a device.

AUTOMATIC DEVICE GENERATION (ADG) WILL ALLOW EVEN NOVICE PROGRAMMERS TO QUICKLY PROTOTYPE NEW EMBEDDED APPLICATIONS.

a single microcontroller that is connected to all the peripherals. However, research in heterogeneous computing has already created tools that improve speed and efficiency by dividing code between CPUs, GPUs, and FPGAs. [1] Future device languages can incorporate these tools to explore a more complex space of device architectures in search of the fastest or cheapest implementation. The example we provide uses a run-loop and polling to control the temperature controller, but even early device languages will be able to support interrupt-driven applications. Future languages can expand on this and use tools from PL theory to create programming models that allow developers to express precise constraints on timing [6] or energy usage [5].

In order to be successful, device specification languages need to be easy for novices to learn and powerful enough for advanced users. We can accomplish this by pulling from the experience of the computer science community. Embedded microcontrollers like the Arduino have spurred the development of easy-to-use programming idioms and design patterns. These conventions allow those without formal training to learn embedded development skills quickly while continuing to make progress on the projects that impelled them to learn. Device specification languages can use tools from programming language design to create a robust framework for change over time [4] that can incorporate new technologies as they appear. Tools from type theory can allow expert users to express complex relationships between software and hardware while being invisible to novices. These languages can be constructed to provide a usable human-centric interface to the complexities of device generation and verification frameworks.

This work will also enable a number of new tools and technologies. Device languages act as an implicit description of an embedded device and, before compilation even occurs, these descriptions could be used by simulation tools. While these simulations will not capture the exact functioning of each component as it will have not yet been chosen, they can be part of a high-speed virtual prototyping process. Designers could don their VR goggles and play with dozens of wildly-different versions of a device before moving to a physical prototype which will save time and lead to better products. Device fabricators also benefit, as they could work more efficiently and reduce costs for makers by working directly with device language code. When a maker creates a design, they could submit that code to a number of fabricators. Each fabricator can compile it using only the parts they have in stock, in order to provide the fastest service and lowest price possible.

Device specification languages represent a fundamentally new way to design embedded devices. By allowing any programmer to design embedded hardware, they empower students, professionals, hobbyists, the creative, and the curious. These new embedded developers will be able to make electronic art, automate their homes, experiment with robotics, and gather information about their world, all through the devices they create. Device specification languages also allow the creation of powerful new tools, enabling designers to iterate faster and allowing fabricators to work more effectively.

CONCLUSION

Embedded development can be put on a path that leads to powerful new tools. This process starts with the development of single model tools, which enable a powerful new verification system, and ways to automate much of the design process. The automation then sets the stage for new programming languages and compilers that allow pure software to be turned into hardware designs.

Each of these steps brings its own benefits, but together they radically improve the embedded development process. Hobbyists will have greater access to embedded development with tools that reduce the skill barrier and shrink cost. Embedded developers can use the new tools to catch mistakes, remove the most tedious aspects of their job, and automate their work. Companies can expect quicker development cycles and lower costs, enabling them to make better products faster.

Single model tools, strict verification, and embedded device generation are within our grasp. However, creating them requires a willingness to acknowledge the problems with modern embedded development and to redesign our tools to correct those issues. Our current ad-hoc, jumbled, and incremental approach to embedded development tools keeps us from a world in which designing embedded hardware is fast, easy, and cheap. We can make the correct choices now, and bring powerful new tools to everyone who wants to build embedded hardware. ■

Rohit Ramesh is a second-year PhD student at the University of Michigan working with Prabal Dutta. He received a BS in Computer Science at the University of Maryland. His research focuses on improving development tools and interfaces for embedded systems with insights from Formal Systems, Programming Language Theory, Machine Learning, and Human Computer Interfaces.

Prabal Dutta is an associate professor of Electrical Engineering and Computer Science at the University of Michigan. He holds a BS in Electrical and Computer Engineering and an MS in Electrical Engineering from The Ohio State University and a PhD in Computer Science from the University of California, Berkeley. His research interests are in wireless, embedded and networked systems, and their applications.

REFERENCES

[1] Joshua Auerbach et al. "A Compiler and Runtime for Heterogeneous Computing". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. San Francisco, California: ACM, 2012, pp. 271–276. isbn: 978-1-4503-1199-1. doi: 10.1145/2228360.2228411[1]. url: http://doi.acm.org/10.1145/2228360.2228411.

[2] P. Derler, E. A. Lee, and A. Sangiovanni Vincentelli. "Modeling Cyber-Physical Systems". In: *Proceedings of the IEEE* 100.1 (2012), pp. 13–28. issn: 0018-9219. doi: 10.1109/JPROC.2011.2160929[2].

[3] R. Ernst. "Codesign of embedded systems: status and trends". In: *IEEE Design Test of Computers* 15.2 (1998), pp. 45–54. issn: 0740-7475. doi: 10.1109/54.679207[3].

[4] Nathaniel Nystrom, Michael R Clarkson, et al. "Polyglot: An extensible compiler framework for Java". In: *Compiler Construction*. Springer. 2003, pp. 138–152.

[5] Adrian Sampson et al. "EnerJ: Approximate data types for safe and general low-power computation". In: *ACM SIGPLAN Notices*. Vol. 46. 6. ACM. 2011, pp. 164–174.

[6] Victor Fay Wolfe, Susan Davidson, and Insup Lee. "RTC: Language support for real-time concurrency". In: *Real-Time Systems* 5.1 (1993), pp. 63–87.

[1] http://dx.doi.org/10.1145/2228360.2228411
[2] http://dx.doi.org/10.1109/JPROC.2011.2160929
[3] http://dx.doi.org/10.1109/54.679207