

Blueprint for an Embedded Systems Programming Language

Paul Soulier

*Dept. of Information and Computer Sciences
University of Hawaii, Manoa
Honolulu, Hawaii 96822
Email: psoulier@hawaii.edu*

Depeng Li

*Dept. of Information and Computer Sciences
University of Hawaii, Manoa
Honolulu, Hawaii 96822
Email: depengli@hawaii.edu*

Abstract

Embedded systems have become ubiquitous and are found in numerous application domains such as sensor networks, medical devices, and smart appliances. Software flaws in such systems can range from minor nuisances to critical security failures and malfunctions. Additionally, the computational power found in these devices has seen tremendous growth and will likely continue to advance. With increasingly powerful hardware, the ability to express complex ideas and concepts in code becomes more important. Given the importance of developing safe and secure software for these applications, it is interesting to observe that the vast majority of software for these devices is written in the C programming language—an inherently unsafe language as compared to other modern languages. This paper examines the characteristics and requirements that uniquely differentiate embedded systems from other application domains. The result is a blueprint for a modern, high-level programming language specifically designed for embedded systems.

1. Introduction

Embedded systems exist in a multitude of applications and advances in hardware technology will continue to make them capable of greater degrees of sophistication and intelligent functionality. While often unnoticed and unseen, these systems are responsible for properly controlling medical devices, automobile braking systems, industrial control systems, and numerous other cyber-physical systems that interact with the world in profound ways. The growing fields of sensor networks and the Internet of Things combined with ubiquitous Internet connectivity will further expand the use of embedded systems.

Given the significant role embedded systems have, the implications of faulty software is clearly evident. However, software logic errors are not the only manner in which a system can malfunction. The Stuxnet virus [14] is an example of a failure caused by a malicious software attack that ultimately caused an industrial control system to destroy itself. As Internet connectivity becomes increasingly common in embedded systems, they too will be susceptible to software-based security exploits.

Many areas of software development have benefitted from the improvements made to programming languages. Modern languages are more capable of detecting errors at compile-time through their type system and many of the low-level and error prone aspects of programming have been abstracted away. This enables the development of more reliable and complex applications. Embedded systems are an exception to this. The vast majority of embedded systems are still developed using the decades-old C programming language—an inherently unsafe language with only the basic features of the imperative paradigm.

Despite its flaws, C has stubbornly remained the de facto standard for embedded system development. The reasons for this are difficult to identify (although a massive existing code base and the lack of a compelling replacement could be factors). Various language-based approaches have been created to address many of the shortcomings of C—all with varying degrees of success as they apply to low-level programming. These solutions tend to focus only on a subset of all the issues involved with low-level software development while not considering other critical aspects. What is missing is a cohesive and practical language that effectively incorporates all of these methods and techniques in a manner consistent with the needs of embedded systems.

On language design Landin [9] remarks in the paper

“The Next 700 Programming Languages” that
“...we must systematize their design so that
a new language is a point chosen from a
well-mapped space, rather than a laboriously
devised construction.”

To design a compelling language to replace C, we must first identify what the language needs to look like. With this in mind, the contribution of this paper is a description of the features and constructs necessary in a language designed to implement secure and reliable embedded systems software.

This paper is structured as follows. Section 2 describes the characteristics that differentiate embedded systems from other programming disciplines. Section 3 presents the blueprint of a programming language designed to produce secure and reliable embedded software. Section 4 highlights related research that has attempted to address the shortcomings of programming languages for embedded systems, and finally, section 5 concludes.

2. Characteristics of Embedded Systems

Embedded systems have a number of characteristics that differentiate them from other application domains. These particularities make most programming languages ill-suited for this type of software development. High-level languages generally attempt to provide helpful abstractions for tasks that can be automated by the compiler or those that are error-prone. While these abstractions can improve development efficiency and reduce errors, they have the unfortunate side effect of obscuring the low-level details that embedded systems must deal with. The necessity to interact directly with hardware, specify the organization of data within a structure, operate with limited resources, and performance requirements are all elements that bring a unique set of challenges to developing this type of software. This section examines the various aspects of embedded systems that necessitates a domain-specific language.

2.1. Safety and Reliability

As mentioned in the introduction, embedded systems are often found in devices that can have a major impact to the physical world. It is frequently required that these systems operate without error and with no down-time. Furthermore, if a problem is found and a software fix is identified, upgrades can be difficult and sometimes impossible. Consider faulty software in an automobile component —the implications are massive

with potentially thousands of vehicles being recalled. With these strict requirements, the need to successfully develop and release error-free software very important.

2.2. Data Layout and Representation

In most applications, developers are really just concerned with what data needs to be stored in a structure; where the data goes and how much room it takes is generally unimportant. Embedded systems, on the other hand, care a lot about where data is located in a structure and how much space it consumes.

The ability to specify the size and location of data is necessary when defining language-based structures that must match hardware structures or standardized protocols. Data layout is also an important tool for tuning. Organizing a structure to improve memory locality based on knowledge of the CPU cache architecture or runtime access can have a significant performance impact. The ability to represent and manipulate data in arbitrary ways is a fundamental aspect of writing embedded systems code.

2.3. Hardware Interaction

Embedded systems interact directly with hardware through memory-mapped IO or low-level CPU instructions. In the case of memory-mapped IO, hardware registers appear as normal memory, but may behave in ways that are not entirely consistent with regular memory. Consider, for example, a hardware device that accepts a 32-bit value from the host through a 16-bit memory-mapped register. Listing 1 shows pseudocode that could be used to send this value to the device. The host writes the most significant 16-bit value to the register followed by the least significant value.

```
u16      *reg = device->input_reg;  
  
*reg = val >> 16;  
*reg = (val & 0xffff)
```

Listing 1. “Interfacing with Hardware”

The compiler, unaware that the memory location is different from others, may eliminate the first assignment upon seeing the same memory address is immediately overwritten by another value. Idiosyncrasies such as this are common in embedded systems where unique hardware properties do not always match the abstract machine of the programming language.

2.4. Transparent Expression

Transparency is a trait of language expression that is particularly important to embedded systems. One of the strengths of the C programming language is that it is easy for the programmer to conceptualize how source code will translate into machine instructions and data structures. This ability becomes very important when attempting to fit code or data into resource-limited hardware, gaining additional performance, or interface with hardware.

2.5. Constrained Environment

Embedded systems are almost always constrained in some fashion. The most common limitations encountered are computational power (memory and processor), time, and energy. Advances in hardware technology have come a long way in easing some of these constraints, but they are still a concern for many systems.

Time is an interesting constraint when considered in the context of cyber-physical systems. An occasional half second delay in a desktop application probably wouldn't be noticed. An equivalent delay in a real-time system such as an electronic braking system or avionics fly-by-wire system could have serious consequences. Many embedded systems have real-time deadlines that must always be met.

Energy is another constraint that can have a significant impact to embedded systems. Sensor networks and other devices that rely on battery power have a finite lifetime before they stop working. Power consumption must be managed to maximize operational time. Highly energy-efficient devices also tend to be very limited in memory and processing power.

Constraints are driven by a number of factors. Some, such as time and physical dimensions, are governed by the laws of physics. Other limitations are driven by business factors that require the use of less powerful hardware to save on manufacturing costs. Regardless of why limitations are present, they introduce unique challenges to embedded system development.

2.6. Performance

Some devices perform computationally intensive operations or transmit data at high speeds. In such cases, performance is a critical design goal where the difference of a few percentage points can determine the success or failure of a product. Consequently, the ability to save a few bytes in a data structure or eliminate a few microseconds from a section of code

can have a profound impact to a program's ability to achieve its goals. These small performance gains often come from specific knowledge of a system and the programmer's ability to generate appropriate code rather than a compiler's optimizer. Performance can be a major influence in the overall design of an embedded system.

3. Language Blueprint

Expressiveness can be described as the property of a language that allows a programmer to effectively translate concepts and ideas into code. The more expressive a language is, the easier it is for a programmer to realize a solution to a problem. This trait is domain specific; what constitutes an expressive language in one domain does not make it expressive in another. For example, assembly language is very expressive as compared to Javascript for executing a specific CPU instruction. Conversely, Javascript is far more capable of describing a web application than is assembly.

This section describes the features and constructs that make a language expressive when considering the characteristics of embedded systems. These features collectively form a blueprint that describes the features of a language ideally suited for embedded systems.

3.1. Paradigm

The functional language paradigm has many useful properties—particularly referential transparency (code has no side effects and there is no global state that can change). This property, among others, has many compelling benefits. However, the functional paradigm is somewhat at odds with embedded systems. Embedded systems are state-full by nature. They interact with hardware components that are themselves state machines. The advantages of the functional paradigm are unarguably valuable. However, applying it to systems that are defined by state would likely be ineffective. Conventional wisdom would suggest the imperative programming paradigm, which is based on state change, and is a natural choice for an embedded system program.

Object oriented programming, while not strictly necessary, can be useful for embedded systems. Object-orientation has proven to be a valuable method of reasoning about complex systems. Additionally, OO techniques can be effective at eliminating some of the unsafe idioms used in C. For example, C programmers sometimes use typeless “void” pointers, type casts, and unions to achieve unsafe versions of polymorphism.

Inheritance and sub-typing provided by OOP is a type-safe alternative.

The object-oriented features supported by the language must include the basics of the paradigm: data encapsulation/abstraction, dynamic binding of function calls, and inheritance/derivation. Each of these features is relatively transparent to the programmer in terms of overhead and the underlying code that is generated. Dynamic binding can impact runtime performance and memory overhead, but these concerns are generally insignificant. Achieving equivalent functionality using standard imperative techniques will generally incur the same costs. The concern for embedded systems is that the constructs generated by the compiler are hidden from the developer and reduce transparency.

Multiple inheritance can pose significant challenges and is worth mentioning. It is a frequently debated feature with the primary point of conflict being that of its utility compared to its drawbacks. When examining the implementation of multiple inheritance in C++ [17], [16], the effects on performance and memory overhead are not trivial. Multiple inheritance affects the organization of code and data structures; which can have a negative impact to both performance and data layout. Due to this, M.I. is not a good choice for embedded systems. There are alternatives (such as “interfaces”) that achieve functionally similar results, but without the same overhead.

3.2. Syntax and Semantics

A somewhat interesting trend of languages designed to replace C is the goal of retaining the same syntax and programming idioms. This is interesting because C and its associated idioms are often unsafe. Consider the C code in Listing 2. Both `if` statements are syntactically correct but are semantically very different. The logical “AND” operator `&&` and the bitwise “AND” operator `&`, while visually similar, have different runtime behavior. Accidentally adding or omitting a `&` character is an easy mistake to make and the compiler has no way to know which is correct.

Pointer arithmetic is another example of syntax that is unsafe and also largely unnecessary. A common C idiom is to use pointer arithmetic as a way to iterate over a subset of an array. It’s also used as an optimization technique to eliminate array references. This idiom is also responsible for numerous bugs. There are plenty of examples of safe syntax in other languages for expressing ranges of arrays and modern compilers can usually optimize array access more effectively than humans can.

```
int x=foo(), y=bar();

if (x & y) {
    // do stuff...
}

// vs.

if (x && y) {
    // do stuff...
}
```

Listing 2. “Syntax and Semantics”

Syntax and semantics must be clear and unambiguous. Although more convenient, preserving the error-prone syntax and idioms of C to avoid learning a new language isn’t justified. In terms of language design, there is no reason for the types of ambiguities shown in Listing 2. Language syntax and semantics must be designed to prevent these types of programming errors.

3.3. Type System

A *type* in a programming language is a form of specification that defines various characteristics of the constructs within a language. A *type system* is the mechanism used to enforce that all type specifications are correctly adhered to. The primary role of a type system is to help promote program correctness and reduce bugs. This section describes the basic properties a type system as well as addressing some specific types that deserve special consideration.

Static Type System Static type systems are essential for embedded systems. Dynamic type systems are undesirable as they can leave latent type errors undetected until runtime. These errors are often unrecoverable and result in program failure. Conversely, static type systems attempt to enforce the type rules at compile time. Static type systems allow type correctness to be verified earlier in the development process. While potentially requiring more effort on behalf of the programmer to properly define the type specifications, this results in systems with fewer bugs. Due to the nature of embedded systems, namely the difficulty of updating software and the implications of software failures, it is more important to identify errors early. Consequently, a language for embedded systems should be statically typed.

Type and Memory Safety A type and memory safe language is critical to minimizing program flaws. Type and memory violations (e.g.: out-of-bounds array

indexing, buffer overflows, invalid pointer casts, etc.) are responsible for numerous software related flaws in the C language. A type safe language guarantees that a memory region cannot be aliased by pointers of differing types and memory safety ensures a program cannot incorrectly access memory.

There are times when type rules need to be bent. Memory allocation and data serialization (also referred to as “marshaling”) are two such examples. In the case of serialization, a program is converting an object of a specific type into a stream of bytes to be stored or transmitted. The opposite process turns a stream of bytes into an object. There must be some provision in the language to circumvent normal type rules. The trick is to allow this type of behavior to occur without undermining the fundamental rules guaranteed by type system.

Designing a language that provides type and memory safety while still providing the expressiveness to accommodate the characteristics of embedded systems is a challenging task and has been the subject of a lot of research.

Arrays It’s common sense that array access should be bounds-checked to ensure memory safety is maintained. In many languages this is trivially accomplished by storing the length with the array structure. It is not quite so simple with embedded systems. Arrays can be stored within a data structure defined by a protocol where the size is either implicit or is located at a non-deterministic position in the data structure. Parametric polymorphism and boxing are two approaches that can be used to ensuring memory safety of arrays without affecting their structure. Regardless of the method, memory safety of arrays must be guaranteed.

Reference Types References (or pointers) are a common source of errors - particularly in the case of the C language due to the unsafe type system. The type system, both at compile time and runtime, can play a critical role in ensuring pointer usage is safe by providing the following characteristics:

- The language should only allow only type safe casts. Invalid casts that are permitted in the C language through cast operators and unions are a constant source of headaches when attempting to develop reliable systems.
- All pointers should be guaranteed to be valid or checked for validity prior to dereferencing. The language should ensure, through static or runtime checking, that all references are valid before allowing a dereferencing operation. Numerous approaches have been devised to accomplish this such as region-based memory [12], [13] and garbage collection.

- Pointer aliasing occurs when an object or data structure is referenced by multiple pointers. As an object becomes more aliased it is more difficult to reason about the various interactions of those objects. A language that minimizes the need for aliasing will be more readily understood and less prone to errors.

Immutable Type Qualifier The ability to specify the immutability of data is a powerful mechanism to help ensure data isn’t unexpectedly modified. It offers some of the benefits of referential transparency found in pure functional languages. Unfortunately, immutability is not all that common. In languages where it does exist, the property is not strictly enforced by the type system and it can be undermined by aliasing and concurrency (or completely ignored in the case of C/C++).

Consider the pseudo-code found in Listing 3 where the procedure `foo` accepts a reference to an integer that is qualified as immutable. The expectation by the `bar` routine is that `aliased_y` will be the same before and after a call to `foo`. Additionally, since `x` is declared as an immutable reference in `foo`, it should be able to consider `x` an invariant for the duration of the procedure. While these semantics will be true for a sequential environment, they may not be in a parallel one; a concurrent thread may modify `aliased_y` during the execution of `foo`. The use of immutable qualifiers can be a helpful tool, so long as the language guarantees the qualifier is enforced in all circumstances.

```
foo(immutable int ptr) {  
    /* do something */  
}  
  
bar() {  
    if aliased_y < arr.length {  
        foo(aliased_y)  
        arr[aliased_y] := 5  
    }  
}
```

Listing 3. “Immutable Qualifier”

3.4. Error Handling

Errors are commonplace —especially for systems that interact with the unpredictabilities of the physical world. In the context of programming languages there are two primary methods of communicating errors: returns codes and exceptions. Returns codes are simply values returned by a procedure that the caller must

inspect to determine success or failure. Exceptions operate by changing the flow of execution when an erroneous or “exceptional” condition occurs. The language builds code into the application to find a suitable handlers —typically by searching through the call stack.

As is typically the case, each of these mechanisms have relative merits and flaws that prevent either from being perfect in all circumstances. Exceptions provide robust facilities to ensure all errors are detected and expressive mechanisms to convey information about to the error. However, this comes at the cost of overhead. Additional code is necessary to support the feature increasing the size of a program. If program size is a constraint, exceptions are a poor choice.

Conversely, return codes are very efficient and pose little additional overhead. However, they are more error prone and less expressive. Return codes can be unintentionally (or deliberately) ignored leading to abnormal program behavior. Failures that originate in a deeply nested call tree require propagating this error back to the initial caller. This can produce less comprehensible code.

Neither solution on its own is ideal and it would seem both have their place. Given the advantages of exceptions, they certainly belong in an embedded systems language.

3.5. Memory Management

Numerous methods of memory management have been devised to avoid the problems inherent to dynamic memory allocation. Given the broad application space of embedded systems, dynamic memory management depends almost entirely on the needs of the system. Some small resource limited devices require no dynamic memory while others can accommodate a completely garbage collected mechanism. Furthermore, systems may require memory to have certain attributes such as cache alignment restrictions imposed by hardware or performance requirements. These varied requirements prevent any one mechanism from being acceptable to all systems. A language for embedded systems must be capable of allowing the memory management system to be defined by the programmer. Since memory allocation typically manipulates memory in a raw, typeless fashion, the language and type system must provide memory-safe infrastructure to support the definition of dynamic memory allocators.

3.6. Concurrency

Embedded systems almost always contain some degree of concurrency. In addition to the proliferation of multi-processor devices, most embedded systems interact with some type of peripheral hardware that operates asynchronously from the processor running the system code. As mentioned by Boehm [5], making concurrency a part of the language design allows the type system to enforce rules that ensure correct behavior.

In many cases, threads carry too much overhead to be used in certain systems. Events are a common method of managing concurrency in resource constrained systems. However, this efficiency comes at the cost of additional complexity and reduced expressiveness [21]. As an alternative to events (and in addition to threads), light-weight threads are important for embedded systems. Light-weight threads share much in common with continuations and co-routines. They offer semantics similar to threads, but with overhead closer to that of events. Protothreads [19] are a good examples how light-weight threads could be implemented.

3.7. Compile-Time Meta-Programming

Meta-programming is the writing of programs that generate or manipulate other programs and is present in many languages. Compile-time meta-programming provides a mechanism to generate code during the compilation process. This can improve expression as well as minimizing runtime overhead in certain circumstances.

- Conditional compilation is particularly useful for embedded systems (a compiler preprocessor provides this functionality in C). Minor variations in platforms or hardware must be accounted for in software. Conditional compilation may be used to generate code for a specific platform without the need to include all variations the final application resulting in reduced overhead.
- Code construction, such as templates in C++ or generics in Java, allows the programmer to reuse boilerplate code in a type-safe manner where the compiler generates code based on a template and input parameters. Data values or tables that would otherwise need to be generated by hand or at runtime can be created at compile time reducing the risk of error or runtime overhead.
- Reflection allows the meta-program to examine the program structure as it exists at compile time. This enables to the programmer to perform

tasks such as adjust structure padding to maintain certain a consistent size or compute resources defined at compile time based on internal structure in a programmatic manner.

3.8. Interoperability with C

While it would be pleasant to entertain the notion of ignoring the enormous amount of C code currently in existence and simply “start over” with a new language, this is unrealistic; the effort and cost involved would be prohibitive. For any new language to be a viable alternative, the ability to utilize existing software is crucial. Given the goal of creating a successor to C, efficiently interfacing with C code is a necessary component of such a language.

4. Related Works

Brewer et al. [1] observe the limitations and liabilities imposed by C in modern systems programs. This work further highlight some of the key elements that should be present in a new programming language and proposes Ivy as an alternative to C. Shapiro [2] echoes the same sentiment on the shortcomings of C and further illustrates the need for a replacement. Additionally, this work presents various aspects of systems programming that require special consideration by any language designer and briefly introduces the BitC language.

A variety of research has been done that involves extensions or dialects of C that have been enhanced with various mechanisms to improve safety. Cyclone [25] is a dialect of C that attempts to eliminate many of the memory safety issues while maintaining the most of the syntax and semantics of C. CCured [23] is also a type-safe extension of the C language, however this approach has severe performance penalties. Deputy [26] uses dependent typing to provide type safety in C. nesC [27] is a version of C specifically tailored to sensor networks and the TinyOS operating system.

5. Conclusion

The C programming language is a simple and powerful tool that has been used for decades to create some of the most critical computing infrastructure in use today. With all of its flaws, it has remained the predominant tool for embedded systems developers owing to its flexibility and power. Any language designed to replace C has the onerous task of retaining that same power and flexibility while offering compelling language features that will improve software reliability,

security, and expressiveness. As systems continue to increase in complexity and software security becomes an increasingly critical component of embedded system design, there is a clear need for advancement in programming language design that addresses these requirements.

This paper has described a blueprint of a modern programming language for embedded systems. This blueprint outlines language features and paradigms that are best suited to enable developers to create secure and reliable software. As embedded systems increase in use and sophistication, so will the need for the ability to produce reliable software and reason about complex systems through code. It is the hope that this work will motivate further discussions on language features and design that can contribute to an eventual successor to the venerable C programming language.

References

- [1] E. Brewer et al, *Thirty Years is Long Enough: Getting Beyond C.*, Proceedings of the 10th conference on Hot Topics in Operating Systems-Volume 10. USENIX Association, 2005.
- [2] J. Shapiro, *Programming Language Challenges in Systems Codes.*, Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on. IEEE, 2008.
- [3] L. Cardelli, *Typeful programming.*, Springer-Verlag, 1991.
- [4] H. Sndergaard and P. Sestoft, *Referential transparency, definiteness and unfoldability.*, Acta Informatica 27.6: 505-517, 1990.
- [5] H. J. Boehm. *Threads cannot be implemented as a library.*, Technical Report HPL-2004-209, Hewlett Packard, 2004.
- [6] S. Macrakis. *Safety and Power.*, ACM SIGSOFT Software Engineering Notes, Volume 7 Issue 2, April 1982.
- [7] J. Hughes. *Why functional programming matters.*, The computer journal 32.2 (1989): 98-107.
- [8] T. Sheard, S. P Jones. *Template meta-programming for Haskell.*, Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. ACM, 2002.
- [9] P. J. Landin. *The next 700 programming languages.*, Communications of the ACM 9.3 (1966): 157-166.
- [10] C. de Dinechin. *C++ exception handling.*, IEEE Concurrency 8.4 (2000): 72-79.

- [11] A. Romanovsky, J. Xu, B. Randell *Exception Handling in Object-Oriented Real-Time Distributed Systems.*, Object-Oriented Real-time Distributed Computing, 1998.(ISORC 98) Proceedings. 1998 First International Symposium on. IEEE, 1998.
- [12] M. Tofte, J. Talpin. *Region-based memory management.*, Information and Computation 132.2 (1997): 109-176.
- [13] D. Grossman, et al. *Region-based memory management in Cyclone.*, ACM SIGPLAN Notices. Vol. 37. No. 5. ACM, 2002.
- [14] R. Langner. *Stuxnet: Dissecting a cyberwarfare weapon.*, Security and Privacy, IEEE 9.3 (2011): 49-51.
- [15] J. Hogg. *Islands: Aliasing protection in object-oriented languages.*, ACM SIGPLAN Notices. Vol. 26. No. 11. ACM, 1991.
- [16] S. Lippman. *Inside the C++ Object Model.*, Addison-Wesley, 1996.
- [17] B. Stroustrup. *Multiple Inheritance for C++.*, Computing Systems 2.4 1989.
- [18] R. Shahriyar, S. M. Blackburn, and D. Frampton. *Down for the count? Getting reference counting back in the ring.*, ACM SIGPLAN Notices. Vol. 47. No. 11. ACM, 2012.
- [19] A. Dunkels, et al. *Protothreads: simplifying event-driven programming of memory-constrained embedded systems.*, Proceedings of the 4th international conference on Embedded networked sensor systems. ACM, 2006.
- [20] M. Cohen, et al. *Using coroutines for RPC in sensor networks.*, Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. IEEE, 2007.
- [21] J.R. von Behren, J. Condit, E.A. Brewer. *Why Events Are a Bad Idea (for High-Concurrency Servers).*, HotOS. 2003.
- [22] J.S. Foster, M. Fhndrich, and A. Aiken. *A theory of type qualifiers.*, ACM SIGPLAN Notices 34.5 (1999): 192-203.
- [23] G.C. Necula, S. McPeak, and W. Weimer. *CCured: Type-safe retrofitting of legacy code.*, ACM SIGPLAN Notices 37.1 (2002): 128-139.
- [24] I. Haller, et al. *Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations.*, USENIX Security. 2013.
- [25] J.T. Morrisett, et al. *Cyclone: A Safe Dialect of C.*, USENIX Annual Technical Conference, General Track. 2002.
- [26] J. P. Condit, et al. *Dependent Types for Low-Level Programming.*, Programming Languages and Systems. Springer Berlin Heidelberg, 2007. 520-535.
- [27] D. Gay, et al. *The nesC language: A holistic approach to networked embedded systems.*, Acm Sigplan Notices. Vol. 38. No. 5. ACM, 2003.
- [28] W.P. McCartney, and S. Nigamanth. *Stackless preemptive multi-threading for TinyOS.*, Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on. IEEE, 2011.
- [29] D. Frampton, et al. *Demystifying magic: high-level low-level programming.*, Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. ACM, 2009.