Tsinghua Science and Technology

Volume 20 | Issue 2 Article 2

2015

A Survey of Language-Based Approaches to Cyber-Physical and Embedded System Development

Paul Soulier
University of Hawaii, Manoa, HI 96822, USA.

Depeng Li
University of Hawaii, Manoa, HI 96822, USA.

John R. Williams
Massachusetts Institute of Technology (MIT), Cambridge, MA 02139, USA.

Recommended Citation

Paul Soulier, Depeng Li, John R. Williams. A Survey of Language-Based Approaches to Cyber-Physical and Embedded System Development. Tsinghua Science and Technology 2015, 20(2): 130-141.

TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 02/11 pp130-141
Volume 20, Number 2, April 2015

A Survey of Language-Based Approaches to Cyber-Physical and Embedded System Development

Paul Soulier*, Depeng Li, and John R. Williams

Abstract: As computers continue to advance, they are becoming more capable of sensing, interacting, and communicating with the physical and cyber world. Medical devices, electronic braking systems in automotive applications, and industrial control systems are examples of the many Cyber-Physical Systems (CPS) that utilize these computing capabilities. Given the potential consequences of software related failures in such systems, a high degree of safety, security, and reliability is often required. Programming languages are important tools used by programmers to develop CPS. They provide a programmer with the ability to transform designs into machine code. Of equal importance is their ability to detect and avoid programming mistakes. The development of CPS has predominantly been accomplished using the C programming language. Although C is a powerful language, it lacks features present in other languages that facilitate the development of reliable systems. This has prompted research into language-based alternatives for improving program quality through the use of programming languages. This paper presents an overview of the characteristics of embedded and cyber-physical systems and the associated requirements imposed on programming languages. This is followed by a survey of relevant research into language-based methods for creating safe, reliable, and robust software for CPS.

Key words: cyber-physical systems; embedded systems; programming languages; type systems

1 Introduction

Cyber-Physical Systems (CPS) exist at the intersection of computation and the physical world. A CPS perceives the world through its sensors and affects change through connected actuators. In a form of feedback, sensors and external inputs influence computation that allows the system to interact with the physical world in a tangible way. CPS exist in various forms, sizes, and complexity including small, stand-alone devices (e.g., sensor nodes or implanted medical device) or embedded as a subcomponent in a large system (fly-by-wire systems in aircraft). (The terms *embedded system* and *cyber-physical system* are generally used interchangeably.)

CPS have become an intrinsic part of modern society. They can be found in appliances, medical devices, automotive applications, avionics, military weapons, industrial control systems, power grids, and countless other applications. The seemingly inexorable advances in hardware technology have enabled CPS to expand into new domains. Ubiquitous wireless connectivity has made possible the Internet of Things (IoT) where CPS will undoubtedly play a significant role. As new advancements are made in other disciplines (biotech, medicine, and robotics), it is easy to envision any number of possible applications where CPS will be an essential component.

Given current and potential future applications of CPS, the ability to create safe, reliable, and secure software for these systems is self-evident. Developing

• Paul Soulier and Depeng Li are with University of Hawaii, Manoa, HI 96822, USA. E-mail: psoulier@hawaii.edu; depengli@hawaii.edu.

• John R. Williams is with Massachusetts Institute of Technology (MIT), Cambridge, MA 02139, USA. E-mail: jrw@mit.edu.

* To whom correspondence should be addressed.

Manuscript received: 2015-01-26; revised: 2015-03-29; accepted: 2015-03-30

such systems has been a long-standing challenge in computer science and software engineering. Software engineering and design methodologies, formal verification, simulation, and various other techniques have been devised to aid in the production of error-free software. Programming languages are another such tool. In much the same way CPS exist at the boundaries of the computational and physical worlds, programming languages bridge the gap between human-created concepts and the corresponding machine code computers used to realize those concepts. Consequently, a language that can effectively enable the transformation of concepts into code will result in systems that operate as expected.

The primary goal of this paper is to survey research focused on improving software quality in CPS through language-based techniques. To contextualize the relevance of language-based techniques to CPS, the unique characteristics of CPS are described as well as their influence in the design of programming languages. The contributions of this paper are as follows:

- Describe the elements of CPS that differentiate them from other application domains and influence the design of programming languages.
- Detail languages currently available for CPS development and the aspects of languages that affect their suitability for use as a CPS development tool.
- Survey the works over the period 2000-2014 intended to improve software quality and reliability of CPS through language-based techniques.
- Enumerate current challenges and open problems that exist with language-based techniques.

The structure of the paper is as follows: Section 2 details the differentiating aspects of CPS from other application domains. Section 3 covers the importance of programming languages and deficiencies that exist with current tools. Section 4 is a survey of works related to language-based techniques as they relate to CPS. Section 5 discusses open issues and challenges in language-based approaches and the paper concludes in Section 6.

2 Characteristics of Cyber-Physical Systems

The development of software for CPS has many of the same expectations of a programming language as other application domains. Memory allocation, concurrency, and defining and manipulating data structures are all concerns. The differences found between domains become more distinct when the amount of control over these common aspects of programming are examined. Many applications designed for general-purpose computers are not generally concerned with how fields are organized within a structure, the size of data structure, where memory comes from when an object is allocated, or even when memory is released. Conversely, CPS are very attuned to these, and many other, aspects of a system. The manner in which data is represented, where it exists within a structure, and where it is stored can all have a dramatic influence on the ability for a CPS to function as needed. CPS also differ in functional requirements where reliability and real-time timing constraints can be significantly more important than other fields. This section provides an overview of the characteristics of CPS that differentiate it from other application domains.

2.1 Reliability

High reliability is a trait frequently attributed to CPS. Users of general purpose computing platforms are accustomed to their computer crashing or rebooting to install updates. While such events are unwanted, they seldom result in anything more than an inconvenience. Conversely, the failure of a system controlling a power grid or aircraft flight mechanics can have a significantly more profound impact. Failures of software in CPS can have catastrophic consequences with some examples including aerospace[1], military[2], medical devices[3], and avionics[4]. These cases underscore the importance of software-correctness in CPS and the potential consequences of software-related errors.

2.2 Security

For many CPS, where the device is physically separated from any source of unwanted, external influence, security is not typically a significant concern. As systems continue to grow in complexity, embedded systems not directly vulnerable to security threats are frequently connected to those that are. CPS can be vulnerable to attack even when not directly accessible. Such a case was demonstrated with the Stuxnet virus[5]. Wireless communication, Internet connectivity, and the IoT are further exposing CPS to new varieties of security threats. For example, Halperin et al.[6] have demonstrated that some Implantable Medical Devices (IMDs) are subject to a form of wireless attack.

Where software flaws were once the only significant mode of failure for CPS, they are now becoming vulnerable to potential modes of attack similar to those experienced by web services, personal computers, and other wireless or Internet connected device. As with reliability issues, it is primarily the result of a security breach that differentiates a CPS from other systems. A security breach in a web service or database is likely to compromise data whereas a breach in a CPS can also involve data, but may additionally have a detrimental impact to person or property. Security is quickly becoming a significant aspect of CPS design.

2.3 Real-time requirements

Cyber-physical systems frequently interact with physical systems. This often necessitates the need to react within some specific window of time. This differs considerably from other software applications. Consider a word processor, the difference of 1 ms vs. 10 ms would likely be unnoticeable to a user in most circumstances. This small timing difference in a CPS, however, can have a significant impact. Take, for example, an electronic breaking system in an automobile. A similar delay in response time could result in an increased breaking distance with obviously negative consequences. Timing and deadlines are critical in CPS. Software for a CPS can execute without error and properly perform whatever computation it was designed for and still fail if it can't complete the task within the proper amount of time. Certain language features, such as garbage collection, have the potential to add an element of nondeterminism that can complicate the task of developing a system capable of achieving necessary real-time constraints. Real-time requirements are an important distinction when defining program correctness in the domain of CPS.

2.4 Data representation

Data representation relates to the manner in which a program organizes and manipulates in-memory data structures. For many systems, managing the detailed nuances of how memory is allocated and the specific placement of data is a burden that is best managed by the run-time environment. CPS, on the other hand, care a great deal about these details.

A CPS routinely interfaces directly with hardware or communicates with other devices via well-defined protocols. To accomplish these tasks, a program must have control over the specific layout of data structures down to individual bits. In addition to functional necessity, data representation has a tremendous impact to performance. The organization of a data structure can be tuned to optimize data locality to take advantage of CPU cache memory or optimally "pack" fields to minimize memory requirements.

2.5 Constrained environment

CPS are known for operating in resource constrained environments. Memory is typically less plentiful and CPU clock-speeds are often slower than other hardware platforms. For some CPS, advances in hardware technology have enabled the use of fully-featured programming languages such as Java or Swift. However, many CPS still operate in highly constrained environments that do not allow the use of such languages.

Clearly, not all CPS have limited 8-bit processors and a few kilobytes of RAM. Some are equipped with large amounts of memory and powerful CPUs equivalent to those found in general-purpose computer but still operate within a constrained environment. Systems of this nature are typically built for a specific purpose. They have only enough computational ability to adequately perform a defined function. Additional hardware comes at the expense of additional cost, space, or power consumption. Adding more powerful hardware for the sole purpose of enabling the use of a feature-rich language is often not viable.

Another limited resource is energy. For data centers, high-performance clusters, or general-purpose computers, consuming less power is sometimes a goal and can equate to financial and environmental benefits, but a constant power source is typically available. Power consumption presents a very different challenge when the energy source is a battery — a common characteristic of mobile devices and many CPS. These systems attempt to conserve power whenever possible, but may still have the opportunity to recharge. For a class of CPS, such as IMDs or remote sensor networks, recharging a battery is either difficult or simply not possible; energy is a finite and consumable resource. For these systems, effective power management is crucial.

2.6 Software updates

Software updates pose yet another challenge to CPS not found in many other environments. For many CPS, the device may require specific tools and processes to update and may incur significant costs. A software flaw in an automotive application may require thousands of vehicles to be recalled at great expense to the manufacturer. Some systems can be difficult to update (for example, remotely located sensor networks) or the task may simply not be possible (consider distant unmanned spacecraft).

Downtime is another component to software updates that can have a more significant impact when a CPS is involved. To perform an update, it is not unusual that a system will be taken off-line to complete the process. For general-purpose computing, this can be a bit of a nuisance, but nothing more. For a CPS in an industrial control application or an IMD, down-time may have a significantly larger impact.

3 Programming Languages and Cyber-Physical Systems

A programming language is the primary tool used by programmers to transform requirements and designs into code a computer can execute. A language that can enable a programmer to effectively and efficiently describe a concept and detect errors early in the development process will result in more reliable software. Boehm and Basili[7] have proposed that the cost of fixing a software bug increases with each phase of development — a bug detected in the test phase is more expensive than one found during the design process. A language that can assist a developer in correctly realizing designs and detecting errors can have a significant impact to overall software quality. This section examines some of the most common languages presently used for CPS development as well as important language characteristics.

3.1 Current state-of-the-art

There exists a large variety of programming languages offering support for different paradigms, specific domains, dynamically or statically typed, etc. Even with numerous languages, when put in context with the characteristics described in Section 2, there are only a handful that are suitable for CPS development. The following is a list of the most common languages used for developing CPS.

- C — The C language[8] is general purpose programming language that is statically-typed, type-unsafe, and memory-unsafe. It is, by an extremely large margin, the most common language used to develop CPS. C is a powerful language that can be used for virtually any programming task.
- C++ — As the successor to C, C++[9] is a superset of the C language that adds language constructs for object-oriented programming and various other language features. Like its predecessor, C++ is statically typed and is neither type or memory safe.
- Assembly — Assembly language is still used in CPS, often to access specific CPU instructions that are otherwise inaccessible in a high-level language. Assembly is untyped and unsafe.
- D — D[10] is a dialect of C and C++ that attempts to address various shortcomings of those languages. D is a statically typed language and type-safe language.
- Ada — The Ada programming language[11] was originally developed for the U.S. Department of Defense for high-reliability systems. It is a type-safe and statically typed language. The use of Ada is commonly found in military applications, avionics, and industrial systems that require a high degree of reliability.

3.2 Expression

A language's expressive ability relates to how well it allows a programmer to express relevant concepts necessary to implement an application. Expressive power also differs from one field to the next. For example, Javascript is better suited to developing a web application than assembly. Conversely, for a programmer that needs to utilize specific CPU instructions, assembly is far more expressive than Python. Languages well-suited for developing CPS will allow a programmer greater control over how data is represented and managed, how to control data representation, where data is located, and so forth.

Another valued trait of languages used for developing CPS applications is *transparency of expression*. The term relates to the ability for a programmer to read source code and generate a reasonably accurate mental model of the structure of assembly code produced by the compiler. This characteristic is important for programmers to tune performance, understand the runtime costs associated with code, as well as managing code space for resource constrained CPS.

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 specifications defined by the types in a language are adhered to. The primary role of a type system is to help promote program correctness and reduce bugs. This section describes the basic properties of a type system as well as addressing some issues that deserve special consideration in a language designed for CPS.

Memory and type safety are critical components of the type system. Ideally, the type system should reject any code that can undermine the underlying assumptions and rules of the language. By enforcing the rules of a language, a program can guarantee the absence of certain types of programming errors. Type safety ensures that an object created in memory can only be referenced as the type it was created as. Memory safety protects the system from erroneously accessing memory (e.g., enforcing array boundaries).

Another aspect of the type system is the time at which the rules of the language are enforced. Dynamic type systems offer flexibility and relieves the programmer from a degree of additional specification within a program by automatically checking and enforcing type correctness at runtime. Conversely, static type systems attempt to enforce the type rules at compile time. Dynamic type systems are undesirable in embedded systems where latent type errors are detected at runtime and are often unrecoverable resulting in program failure. 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 in the system, this often results in systems with fewer runtime 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, languages for CPS are generally statically typed.

4 Survey of Language-Based Approaches to CPS Development

This section presents a survey of language-based research with the goal of improving overall software quality and programmer productivity. The vast majority of the works found have focused on amending C through language extensions or syntactically similar dialects. The primary areas of research found addressed the following general topics: type and memory safety, concurrency, and memory management. Figure 1 provides an overview of the areas surveyed and the associated works.

4.1 Languages

There is a plentiful and varied selection of programming languages available for virtually every application domain. Language theory has continued to provide new type systems and abstractions to make programming more efficient and reliable. While not every language created gains widespread usage, most application domains periodically adopt new languages to reap the benefits of current technology. As mentioned previously, CPS are somewhat of an exception to this. Only a few research-based languages have been developed to address program safety and low-level programming in the context of CPS.

Cyclone[12], a dialect of C, addresses many of the shortcomings of C while maintaining many of the programming idioms commonly used in C programming. Cyclone, unlike C, provides type and memory safety through the use of additional pointer type specifications and annotations. A region-based memory management scheme is employed for memory management and guarantees all memory access is safe and unused memory is released. The language attempts to retain the expressive power and performance found in C that is necessary for low-level programming while simultaneously providing language-based safety features.

The nesC language[13], also a C dialect, has been specifically designed for Wireless Sensor Networks (WSN) and resource constrained platforms. The language has been designed to complement the TinyOS operating system — a commonly used OS for embedded systems. The nesC language is still type and memory unsafe, but has added various features to enable a more structured approach to software development. The language provides syntax and semantics that allow programs to be defined with components. Components contain internal implementations and external interfaces for interacting with other components. Additional safety is provided through static program analysis and can detect some run-time errors such as data races.

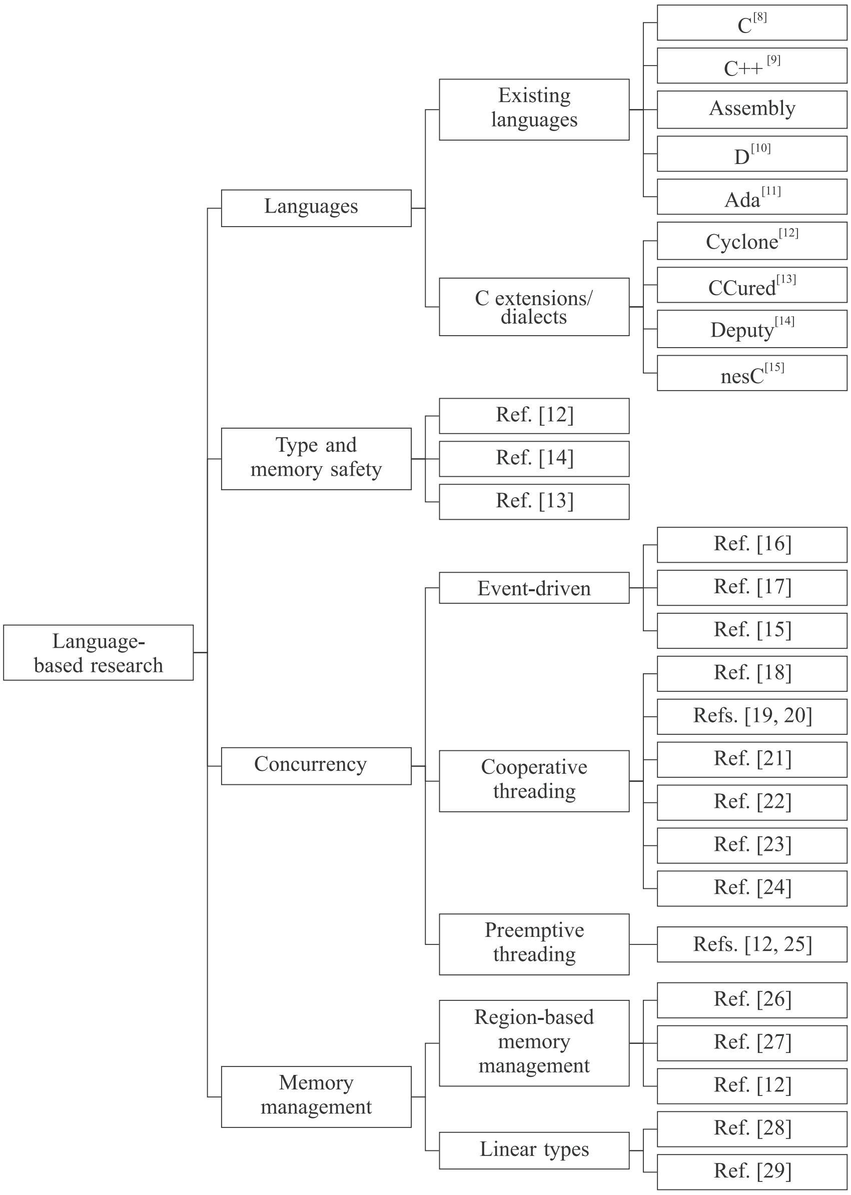

Fig. 1 Overview of language-oriented research for developing cyber-physical systems.

4.2 Type and memory safety

Type and memory safety are critical components of a programming language that helps ensure correctness. Although type and memory safe languages are plentiful, few are suitable for CPS. Because C is the predominant language used for CPS, a significant amount of work has focused on amending the shortcomings of the C/C++ type system either through language transformations, extensions, or new dialects.

Listing 1 is a trivial memory copy example that illustrates some of the type and memory safety issues that arise in a typical C program. In this example, the C compiler has no method to verify the source and destination memory locations are compatible with the range specified by the caller where an incorrect size may result in memory corruption or program fault. Furthermore, this routine uses “void” pointers to avoid the need for a duplicate function to be created

```
void memcpy (void *dst, void * src, int bytes)
{
   char *d = (char*) dest, *s = (char*) src;

   for (int i = 0; i < bytes; i++) {
      d[i] = s[i];
   }
}
```

Listing 1 Memory safety code.

for every combination of possible types. This, however, prevents the compiler from checking if the source and destination are compatible types. The works presented in this section attempt to resolve these type issues.

Necula et al.[13] developed the CCured type system for C to enhance memory safety of pointer operations through the use of annotations. The type system adds pointer type qualifiers that facilitate programming idioms common to C while enhancing the safety of the language. These aid in the compiler's ability to statically verify many uses of pointers at compile time. For instances that cannot be checked statically, runtime checks are added to the code. The underlying representation of pointers is determined by the compiler and may vary in size. This presents challenges when interfacing with C libraries built with a standard compiler. In addition, use of garbage collection potentially limits the use of CCured in certain CPS applications.

Deputy, by Condit et al.[14], provides an extension to the C language in the form of dependent types. Using annotations in C code, the programmer specifies constraints, such as ranges and boundaries, for various types. This enables the compiler to ensure program correctness by performing static compile time analysis and inserting runtime checks where necessary. By using this metadata, Deputy is able to avoid changing program data representation.

Cyclone[12] is a dialect of C that enhances the type system to avoid memory and type errors common in C code. By using additional syntax and type inference, Cyclone is capable of performing static analysis and inserting runtime checks when necessary to ensure memory violations do not occur. The language uses type inference and parametric polymorphism to provide a type-safe alternative to the idiomatic use "void" shown in Listing 1. Cyclone was developed with the explicit intent to preserve the expressive power of C in developing low-level software.

4.3 Concurrency

Cyber-physical systems routinely interact with physical processes that occur in a non-deterministic fashion. As a result, CPS must manage a number of asynchronous events and use either thread or event-based mechanisms to accomplish this. While some debate exists[30–32] as to the better method, CPS have traditionally used events-driven mechanisms when resource constraints are a concern. This is primarily due to the fact that, in practice, thread-based implementations have substantially higher operating overheads in terms of code and data requirements.

Events are an efficient mechanism. They do, however, place additional burdens on the programmer. Operations that span multiple events require the programmer to manually manage state transitions and data. For processes that involve a large number of states, event-based mechanisms can also become excessively complicated. The pseudo-code in Listing 2 illustrates a simple event-drive process that receives a long data stream from a wireless radio in smaller, 64-byte blocks. The code has the following properties:

- The code implements two states: The first waits for a buffer to become available. Once available, the buffer is acquired and then transitions to the next state. The second state repeats until all data has been received.
- State data must be explicitly managed. The programmer is required to manage where the information is stored as well as updating it.
- State transitions are also explicitly managed in the form of function pointer call-backs.
- The use of common language constructs, such as loops, is not possible when asynchronous events are present. In this example, loops must be translated manually into state transitions using function pointers and call-backs.
- Reusing code requires the integration of one state machine into another.

Threads offer, from a programming perspective, a simplified way of managing asynchronous events. The

```
void  long_receive(recv_context_t*ctx)
{
   // allocate buffer
   if  ( buffer_avail(ctx->byte_count) == false )
      buffer_wait(long_receive, ctx->byte_count, ctx
          );
   else {
      ctx->buff = buffer_alloc(ctx->byte_count);
      do_receive(ctx);
   }
}

void  do_receive(recv_context_t*ctx)
{
   if (ctx->bytes_recvd < ctx->byte_count) {
      radio_receive(do_receive, &ctx->buff[ctx->
          bytes_recvd], MIN(64, ctx->byte_count));
      ctx->bytes_recvd += MIN(64, ctx->byte_count);
   }

   ctx->complete(ctx);
}
```

Listing 2 Event-based code.

pseudo-code in Listing 3 implements the same functionality as Listing 2. By most standards, the thread-based code is intuitively obvious and needs little explanation beyond the code itself. The thread-based implementation contrasts the event-driven mechanism in several important ways:

- Common language constructs, specifically loops, are usable.
- Code reuse is simplified and amounts to a simple function call.
- All states are implicitly managed; the programmer is not required to manually save state between asynchronous operations and memory associated with state is automatically allocated and released.

Clearly, thread-based mechanisms appear to simplify programming. Code need not be broken into separate routines for each state, and loops are usable, reusing the code amounts to a simple function call, etc. However, threads are not without drawbacks. With traditional thread implementations, there is a significant cost both in memory and runtime execution overhead. Not surprisingly, the general trend of research seeks to provide thread-based semantics while reducing the typical overhead associated with traditional threading implementations. The majority of the research tends to be centered on sensor networks; this is not unexpected due to the resource constraints encountered in such systems. Although the focus may be on sensor networks, the work is equally applicable to any embedded or cyber-physical system, in essence.

One of the primary issues that arise from event-based implementations is complexity. Complex systems often have numerous distinct events associated with a single action. Event-based methods are frequently used due to their efficiency. Within the context of event-driven programming, several approaches have been taken to minimize the limitations associated complexity. The nesC language, developed by Gay et al.[15], is an extension to the C language designed specifically for the highly constrained environment found in sensor network applications. In conjunction with TinyOS[33], the language provides a structured approach to event handling to enhance developer productivity. One drawback of nesC is the focus on resource constrained systems. The compiler utilizes whole program compilation to enable effective optimization of type checking; as such it is not well-suited for large-scale projects.

```
void  long_receive(intbyte_count)
{
   char *buff;
   intbytes_recvd = 0;

   buff = buffer_alloc(byte_count);
   while (bytes_recvd < byte_count) {

      radio_receive(buff[bytes_recvd], MIN(64,
          byte_count));
      bytes_recvd += MIN(64, byte_count);
   }
}
```

Listing 3 Thread-based code.

Kasten and Römer[17] identified the static nature event-driven software and management of state information as two limitations of event-based programming. They proposed a language that utilizes finite state machines to enable more flexibility in the construction of software that handles asynchronous events by improving modularity and reducing overall complexity. State data is managed with *state variables* that behave as a traditional local variable, but automatic memory management is provided by the language. This enables efficient sharing of data between states.

Bernauer et al.[16, 34] seeked to combine the most favorable characteristics of event- and thread-based paradigms by extending the nesC language to allow a programmer write code using the semantics of threads. The compiler then transforms this code into equivalent event-based code. The compiler statically allocates memory to store local variables used to maintain state information. Due to the static nature of memory allocation, recursive function calls are not possible and this language assumes a cooperative multitasking model.

Protothreads (Dunkles et al.[19, 20]) provides a mechanism that permits a programming style similar to that of the sequential method used with threads. Protothreads are implemented using only standard C language constructs and are designed to be extremely low overhead and used in conjunction with an event-driven system. Through the use of C macros, this system interleaves code within a C “switch” statement. All threads of execution share the same stack. This has the advantage of not requiring a unique stack for each distinct thread, but requires the programmer to manually manage state when a blocking operation is performed. As a result of using a standard C compiler, the rules associated with Protothreads are not enforced by the compiler and the burden of adhering to these rules is incumbent on the programmer.

Many CPS do not use true parallelism. It is often unnecessary or the hardware is uniprocessor. For such systems, the need for costly synchronization mechanisms can be avoided by using cooperative multi-threading. To avoid the additional overhead typically required by threads, various approaches have been devised for stack sharing[21, 22, 24, 35]. These techniques provide the behavior expected from threads without the need for manual state management while reducing memory overhead. This comes at the cost of reduced runtime performance that results stack swapping and other operating overhead.

The works discussed thus far have focused on systems where parallel execution is not used or does not have synchronization concerns between parallel executing threads. However, multi-core hardware is becoming more common place. Cyclone[12] provides many desirable traits for programming CPS but does specifically address concurrency. The work by Grossman[25] proposes an approach to concurrency in Cyclone that reamins type-safe and provides race-free access to shared data.

4.4 Memory management

Managing memory allocation has always posed a challenge to programmers. For modern high-level languages, the need for manual memory management has largely been obviated by the use of garbage collecting systems. For CPS, however, manual memory management is still necessary in many circumstances. Garbage collectors impose significant runtime overhead and non-deterministic timing effects that are often unacceptable. In practice, manual memory management is sometimes unavoidable in CPS. This section examines some alternatives that attempt to combine efficient automatic memory management while maintaining a sufficient level of runtime performance.

Originally proposed by Tofte et al.[36, 37], region-based memory management provides a compelling mechanism for memory management in CPS. In region-based memory management, each object or structure is allocated in a specific region. The region may be defined automatically by the compiler or manually by the programmer. In either case, the memory associated to a region is not released until all objects allocated to the region have been freed. In essence, region-based memory techniques attempt to minimize the cost associated to automatic memory management over a collection of related objects. The language can statically check that programs are correct at compile time while the compiler inserts code to manage dynamic management at runtime. In addition to avoiding common pitfalls of memory management, related data can be co-located to produce good locality that can lead to better cache and overall system performance.

Gay and Aiken[27] described region-based memory management for dynamic memory. Their approach offers both explicit freeing of regions as well as reference counted regions and is dynamically checked at runtime. Grossman et al.[26] detailed region-based memory management used in Cyclone[12]. Their system uses additional annotations in code to allow compile-time, static checking of memory regions. The system used in Cyclone also applies regions to stack-allocated memory to prevent invalid references from occurring. In C, it is possible to bind an external reference to a local variable. When the function which the local variable was declared in goes out of scope, the memory is released and any reference to that data is no longer valid. The type system in Cyclone prevents this through the use of regions.

Linear types are another interesting method of potential memory management in CPS. With linear types, an object can be referenced by only a single entity. Once that reference ceases to exist, there can be no other references and the object can be released. Linear types require little runtime overhead making them ideally suited for CPS. Although linear types provide guaranteed memory management, they come at the expense of sharing data through aliases. Walker and Watkins[28] examined combining linear type and region-based memory management. Event-driven systems, common in CPS, often communicate through messages. Fähndrich et al.[29] discussed efficient and safe message-based communication using linear types.

5 Open Challenges

Many issues pertaining to language-based techniques for improving the software quality of CPS have well-understood solutions. Others are still open challenges that have yet to be addressed. Furthermore, of the issues that have been addressed, there are no languages that incorporate all of the potential techniques. This section reviews important areas of research in language-based approaches to improving software quality in CPS that do not have adequate solutions.

5.1 Combining safety, expression, and performance

Software designs often require trade-offs to achieve specific goals. Additional memory may be needed to obtain performance requirements or a useful abstraction that makes a task easier degrade performance. In language design, similar issues exist. Abstractions can reduce performance or limit expressiveness. Safety enforced through run-time checking can degrade performance. It remains to be seen if a language can be designed such that it simultaneously offers acceptable levels of safety, expressiveness, and performance.

5.2 Unsafe code

In various circumstances, the rules of a type system interfere with the ability to accomplish a task. Memory management or access to a raw address that contains a memory-mapped register are common examples in CPS. For general-purpose computing, the need for such facilities is rare. Using an unsafe secondary language or less efficient mechanisms for isolated portions of code is a reasonable solution. These situations arise more frequently in CPS necessitating the need for a more comprehensive solution.

Most languages well-suited for CPS provide the ability to subvert the type system in some manner. Allowing such operations opens the door to various safety and security issues. An adequate solution that provides raw memory access while still providing strong guarantees regarding the integrity of the type system is not present in any language.

5.3 Timing semantics

As noted by Lee[38], systems with timing deadlines may execute code correctly but still fail to function as designed by missing a timing constraint. Currently, timing is verified through testing, simulation, or other mechanisms. To date, languages have no mechanism to specify timing requirements in code. With numerous hardware platforms, each with unique timing and performance characteristics, specifying timing requirements in code is difficult.

5.4 Concurrency

This survey has shown works that provide highly efficient concurrency mechanisms and the Cyclone language provides compiler support for type-safe, preemptive systems using a more traditional “heavy-weight” thread model. However, there does not exist a system that simultaneously addresses both of these aspects of parallel programming. Multi-core hardware is now common and developing error free software that exploits this potential parallelism is difficult. Thread-based systems with semaphores or other synchronization primitives still have a high degree of overhead while event-based systems face a significant increase in code complexity. In the domain of cyber-physical and embedded systems, no adequate solution exists.

5.5 Acceptance

Possibly one of the most significant issues in designing a new language is achieving even a moderate degree of acceptance from the programming community. From an organizational perspective, adopting a new programming language is difficult for a variety of reasons.

- Cost — Selecting a new language can have significant cost overhead, especially to smaller organizations, in both time and financial resources.
- Standardization — CPS are developed for a wide variety of hardware platforms that differ in architecture (RISC vs. CISC, 8-bit vs. 32-bit processors, etc.). A project may use different platforms from one generation to the next. This may also require a change in vendors that supply the compiler tool chain. Without standardization, switching from one vendor to another may result in costly porting efforts due to incompatibilities in compiler implementation.
- Existing Code Base — For any organization that has a substantial code base, using a new language can pose difficult logistical issues. Software engineers must be familiar with multiple languages or must be involved in non-trivial porting efforts.
- Inertia — Learning a new language takes considerable effort for both organizations and individual software developers. It is often easier to simply continue to use existing tools despite known flaws.

6 Conclusions

Cyber-physical systems exist in the Internet of Things, implantable medical devices, smart appliances, and a multitude of other technologies. Advances in computing technology and other fields that rely on computers will likely continue to fuel the growth of CPS. The ability to develop such systems with quality, safety, and security is clearly important. To the best of our knowledge, this is the first survey of language-based techniques for improving software designed for CPS.

As with many application domains, CPS posses characteristics that make domain-specific languages a necessity. In the first part of this paper, background was provided to illustrate the unique programming challenges often encountered in CPS. Elements of CPS that differentiate them from other application domains and the associated requirements imposed on programming languages used to build them were elaborated on. The second part of the paper presented a survey of the language-based techniques aimed at improving program correctness for CPS in addition to open challenges pertaining to the practical adoption and use of these techniques.

The C language, although powerful, is inherently unsafe and lacks many of the features found in modern programming languages. Despite these limitations and the availability of languages with better safety, C is still the most widely used language for building embedded and cyber-physical systems. Given the potential use and impact of these systems, there is a necessity to developing safe, reliable, and secure software. It is interesting to note that research in the areas discussed in this paper has dwindled in recent years; one can only speculate as to the reasons for this. As CPS become increasingly complicated and pervasive in society, new languages and language-based methodologies will be crucial to ensuring these systems function as expected.

References

[1] J. Lions, Report by the inquiry board on the ariane 5 flight 501 failure, Joint Communication ESA-CNES, 1996.

[2] E. Marshall, Fatal error: How patriot overlooked a scud, *Science*, vol. 255, no. 5050, pp. 1347–1347, 1992.

[3] N. G. Leveson and C. S. Turner, An investigation of the therac-25 accidents, *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[4] C. Bolkcom, V-22 osprey tilt-rotor aircraft, DTIC Document, 2004.

[5] R. Langner, Stuxnet: Dissecting a cyberwarfare weapon, *Security & Privacy, IEEE*, vol. 9, no. 3, pp. 49–51, 2011.

[6] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel, Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses, in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, 2008, pp. 129–142.

[7] B. Boehm and V. R. Basili, Software defect reduction top 10 list, *Computer*, vol. 34, no. 1, pp. 135–137, 2005.

[8] International Organization for Standardization, Programming languages c, Geneva, Switzerland, ISO 9899:TC2, 1999.

[9] International Organization for Standardization, Programming language c++, Geneva, Switzerland, ISO 14882:2011, 2011.

[10] A. Alexandrescu, *The D Programming Language*. Addison-Wesley Professional, 2010.

[11] S. T. Taft, *Ada 2005 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652/1995 (E) with Technical Corrigendum 1 and Amendment 1*. Springer, 2006, vol. 4348.

[12] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, Cyclone: A safe dialect of c, in *USENIX Annual Technical Conference, General Track*, 2002, pp. 275–288.

[13] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, Ccured: Type-safe retrofitting of legacy software, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[14] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, Dependent types for low-level programming, in *Programming Languages and Systems*. Springer, 2007, pp. 520–535.

[15] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, The nesc language: A holistic approach to networked embedded systems, *ACM Sigplan Notices*, vol. 38, no. 5, pp. 1–11, 2003.

[16] A. Bernauer, K. Römer, S. Santini, and J. Ma, Threads2events: An automatic code generation approach, in *Proceedings of the 6th Workshop on Hot Topics in Embedded Networked Sensors*, ACM, 2010, p. 8.

[17] O. Kasten and K. Römer, Beyond event handlers: Programming wireless sensors with attributed state machines, in *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, 2005, p. 7.

[18] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, Cooperative task management without manual stack management, in *USENIX Annual Technical Conference, General Track*, 2002, pp. 289–302.

[19] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, Protothreads: Simplifying event-driven programming of memory-constrained embedded systems, in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, 2006, pp. 29–42.

[20] A. Dunkels, O. Schmidt, and T. Voigt, Using protothreads for sensor node programming, in *Proceedings of the REALWSN*, 2005.

[21] S. Rossetto and N. d. L. R. Rodriguez, A cooperative multitasking model for networked sensors. in *ICDCS Workshops*, Citeseer, 2006, p. 91.

[22] W. P. McCartney and N. Sridhar, Stackless preemptive multithreading for tinyos, in *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on*, 2011, pp. 1–8.

[23] J. Sallai, M. Maróti, and Á. Lédeczi, A concurrency abstraction for reliable sensor network applications, in *Reliable Systems on Unreliable Networked Platforms*. Springer, 2007, pp. 143–160.

[24] C. Nitta, R. Pandey, and Y. Ramin, Y-threads: Supporting concurrency in wireless sensor networks, in *Distributed Computing in Sensor Systems*. Springer, 2006, pp. 169–184.

[25] D. Grossman, Type-safe multithreading in cyclone, *ACM Sigplan Notices*, vol. 38, no. 3, pp. 13–25, 2003.

[26] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, Region-based memory management in cyclone, *ACM Sigplan Notices*, vol. 37, no. 5, pp. 282–293, 2002.

[27] D. Gay and A. Aiken, Language support for regions, *ACM Sigplan Notices*, vol. 36, no. 5, pp. 70–80, 2001.

[28] D. Walker and K. Watkins, On regions and linear types, *ACM Sigplan Notices*, vol. 36, no. 10, pp. 181–192, 2001.

[29] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi, Language support for fast and reliable message-based communication in singularity os, *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 177–190, 2006.

[30] H.-J. Boehm, Threads cannot be implemented as a library, *ACM Sigplan Notices*, vol. 40, no. 6, pp. 261–268, 2005.

[31] J. Ousterhout, Why threads are a bad idea (for most purposes), presentation at the 1996 Usenix Annual Technical Conference, San Diego, CA, USA, 1996.

[32] J. R. von Behren, J. Condit, and E. A. Brewer, Why events are a bad idea (for high-concurrency servers), in *HotQS*, 2003, pp. 19–24.

[33] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al., Tinyos: An operating system for sensor networks, in *Ambient Intelligence*. Springer, 2005, pp. 115–148.

[34] A. Bernauer and K. Römer, A comprehensive compiler-assisted thread abstraction for resource-constrained systems, in *Information Processing in Sensor Networks (IPSN), 2013 ACM/IEEE International Conference on. IEEE*, 2013, pp. 167–177.

[35] B. Gu, Y. Kim, J. Heo, and Y. Cho, Shared-stack cooperative threads, in *Proceedings of the 2007 ACM Symposium on Applied Computing*, 2007, pp. 1181–1186.

[36] M. Tofte and J.-P. Talpin, Region-based memory management, *Information and Computation*, vol. 132, no. 2, pp. 109–176, 1997.

[37] M. Tofte and L. Birkedal, A region inference algorithm, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 4, pp. 724–767, 1998.

[38] E. A. Lee, Cyber physical systems: Design challenges, in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on. IEEE*, 2008, pp. 363–369.

Paul Soulier received his BS degree in electrical engineering from the University of Colorado, Colorado Springs in 1999. He has worked in the private sector for 15 years developing real-time, embedded systems. Currently, he is a graduate student at the University of Hawaii, Manoa, specializing in security, cyber-physical systems, and programming languages.

Depeng Li obtained his PhD degree in computer science from Dalhousie University, Canada in 2010. He is currently an assistant professor in Department of Information and Computer Sciences (ICS) at University of Hawaii at Manoa (UHM). His research interests are in security, privacy, and applied cryptography. His research projects span across areas such as Internet of Things, smart grids, mobile Health-tech and physical-human-cyber triad.

John R. Williams (PhD, Swansea University, UK) is a professor of information engineering, civil and environmental engineering, and engineering systems at Massachusetts Institute of Technology, and he is also the director of Auto-ID Laboratory at MIT. His research area of specialty is large scale computer analysis applied to both physical systems and to information. He has been named, alongside Bill Gates and Larry Ellison, as one of the 50 most powerful people in Computer Networks.