Toward a Unified Object Model for Cyber-Physical Systems

Yu David Liu
SUNY Binghamton
davidL@cs.binghamton.edu

ABSTRACT

Cyber-physical systems are a coordinated combination of computational and physical elements. This position paper calls for the design of a unified object model that blends the boundary of the two. The unified object model has the benefits of bringing classic software engineering technologies and tools – such as UML – to the new application domain of cyber-physical systems, and further equipping programs written for these systems with the traditional strengths of object-oriented languages, such as encapsulation, code reuse and customization, and strong guarantees for avoiding runtime errors.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Objects, cyber-physical systems

1. INTRODUCTION

The past decade has seen the significant rise of cyber-physical systems (CPS). One distinct trait of these systems is the incorporation of embedded-systems-controlled *physical objects* into computing. Often enabled by wireless sensor networks (WSNs), the collection of physical objects communicate, coordinate, and actuate to achieve a common goal. For example, light-powered smart tags can be associated with books [4]; together the ad hoc network formed by hundreds of books can help librarians to locate misplaced ones. For another example, a swarm of robotic bees [10] can collectively help with pollination when bee colonies in the real world collapse.

SESENA '11, May 22, 2011, Waikiki, Honolulu, HI, USA

Designing programming models for CPS – sensor networks in particular – is an active branch of research. On one end of the spectrum, systems programming languages such as C and its variants such as nesC remain dominant, where programming a physical object (a tagged book or a robotic bee) is tantamount to intricate embedded systems programming on low-level details of protocols and packet formats. On the other end, very high-level programming models exist – *e.g.* Regiment [8] and Eon [11]. These languages focus on the coordination of physical objects with clean and succinct code. To achieve these benefits, they usually require significant departures from today's programming paradigms.

This position paper calls for a middle-of-the-road approach. The central question to answer is, "how far is the standard object model [1] known to millions of Java/C++/C# programmers away from programming CPS?" Historically, one primary reason why OO languages gained popularity is their imitation to the physical world. In the good old days of Smalltalk programming on DOS-empowered PCs, a 1980s programmer writing a bee colony simulation game was encouraged to think about physical bees, encapsulating properties of bees as object fields and behaviors of bees as object methods. Bee objects in the game communicate through message passing. Now fast forward to 2011, the world of robotic bees. Doesn't this world of CPS strike a sense of *deja vu* of the 1980s game, except that the bees are now really flying around?

1.1 Opportunities

Aligning physical objects in CPS with computation objects *a la* Java/C++/C# opens up opportunities for principled software development for CPS.

First, classic *software lifecycle support* can now aid CPS software development. Software engineering research for CPS is a nascent area. If an object model unifying the computational elements and the physical elements is available, decomposing CPS software can be performed with little difference from decomposing a traditional system with no physical objects. A large number of software artifacts – such as UML, software architectures, design patterns – can be immediately made available to CPS software developers, together with numerous tools.

Second, the traditional object model provides natural *encapsulation for physical objects*. When physical objects are aligned with computation objects, a programmer writing a program for a robotic bee is encouraged to focus on the interface of a bee – behaviors that contribute to the coordination of bees, such as turn, land, fly, pollinate – and hide implementation details of embedded system control behind

the intuitive interface. As a justification of its usefulness, encapsulation as a benefit also motivated the design of Tiny web services [9].

Third, rich and time-honored *code reuse* mechanisms can be ported to CPS programming. As an increasing number of CPS systems are developed, a design goal high on the wish list of CPS developers is code reuse. Object models encourage programmers to divide code into smaller pieces and offer flexible code composition mechanisms, such as inheritance (*e.g.* Java) and traits (*e.g* Scala). In addition, powerful code customization mechanisms such as Java generics and C++ templates further facilitate code reuse.

Fourth, maximally reducing run-time errors is to the great interest of CPS systems. As many physical objects are deployed in uncontrolled environments, run-time errors may at best lead to expensive rebooting, and at worst result in permanent loss of physical objects. There is a long tradition of designing *type-safe* object models to avoid run-time errors.

Overall, research on object-oriented software design, languages, and systems is a more mature field than research on developing CPS systems. If a bridge can be built, unexpected opportunities may exist by tailoring known OO techniques to CPS. For instance, existing research on coordinations and first-class relationships for object-oriented software may offer insights on CPS macro-programming.

1.2 Challenges

The theme of this paper – adapting the familiar object model (and the coding habits) to CPS programming – should by no means be interpreted as using Java/C++/C# *as is* on CPS.

On the semantic level, CPS software is inherently concurrent: computations on multiple physical objects happen at the same time. With the resource constraints faced by many CPS systems, a much simpler and more lightweight model than *e.g.* Java locks is needed. We propose to view each method invocation as a run-to-completion thread, and forgo the standard synchronous semantics for object message roundtrips. Instead, each message roundtrip is semantically divided as two asynchronous messages – one for the method call, and one for the return value. This design is aligned with the split-phase operations of TinyOS/nesC semantically (but does not require the explicit syntax of breaking one message roundtrip into two). On the conceptual level, this design in essence models the objects representing physical objects as actors [2]. Blending actors into the standard object model without changing the coding habits of Java/C++/C# programmers is the focus of several recent research projects, *e.g.* [5, 6].

Standard reuse technologies we described in Sec. 1 should help construct CPS software more modularly, but a richer and potentially domain-specific model might be needed to cover the complex landscape of CPS software reusability. CPS code fragments may be platform-dependent, compatible only with specific OS versions, behaviorally sound only when calibration parameters satisfy certain constraints, or feasible only when a particular energy budget is met. We are interested in exploring whether these refined requirements can be expressed through advanced language techniques such as contracts [3] or reasoned as invariants (such as version compatibility [7]).

Language primitives capturing the recurring themes of CPS software should be designed. For instance, broadcasting is a fundamental operation in many CPS systems. Messaging in the standard object model however is more aligned with unicast (as a receiver object is required). A broadcast messaging construct should be useful for the CPS object model. Richer support such as first-class neighborhoods may be helpful as well [13].

Last but not least, resource constraints of many CPS systems demand careful compiler design. Languages such as Virgil [12] have proven objects can be compiled onto platforms with extreme resource constraints. Some decisions made in Virgil are very relevant for CPS software, such as in-constructor-only object instantiations, compact object layout, and aggressive compiler optimizations.

Acknowledgments

We would like to thank Scott Smith, Andreas Terzis, and the participants of 2011 NSF Workshop on Pervasive Computing at Scale (PeCS) for useful discussions.

2. REFERENCES

[1] Abadi, M., and Cardelli, L. *A Theory of Objects*. Springer, 1996.

[2] Agha, G. *ACTORS : A model of Concurrent computations in Distributed Systems*. MITP, Cambridge, Mass., 1990.

[3] Findler, R. B., and Felleisen, M. Contract soundness for object-oriented languages. In *OOPSLA'01* (2001), pp. 1–15.

[4] Gorlatova, M., Kinget, P., Kymissis, I., Rubenstein, D., Wang, X., and Zussman, G. Energy harvesting active networked tags (enhants) for ubiquitous object networking. *IEEE Wireless Communications 17*, 6 (2010), 18 –25.

[5] Haller, P., and Odersky, M. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci. 410* (February 2009), 202–220.

[6] Kulkarni, A., Liu, Y. D., and Smith, S. F. Task types for pervasive atomicity. In *OOPSLA '10* (Reno, NV, USA, 2010).

[7] Liu, Y. D., and Smith, S. F. A Formal Framework for Component Deployment. In *OOPSLA'06* (2006), pp. 325–344.

[8] Newton, R., Morrisett, G., and Welsh, M. The regiment macroprogramming system. In *IPSN '07* (2007), pp. 489–498.

[9] Priyantha, N. B., Kansal, A., Goraczko, M., and Zhao, F. Tiny web services: design and implementation of interoperable and evolvable sensor networks. In *SenSys '08* (2008), pp. 253–266.

[10] The robobees project, `http://robobees.seas.harvard.edu/`.

[11] Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M. D., and Berger, E. D. Eon: a language and runtime system for perpetual systems. In *SenSys '07* (2007), pp. 161–174.

[12] Titzer, B. L. Virgil: objects on the head of a pin. In *OOPSLA '06* (2006), pp. 191–208.

[13] Whitehouse, K., Sharp, C., Brewer, E., and Culler, D. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04* (2004), pp. 99–110.