This article presents a nice overview of the nesC programming language, which is specialized for embedded network-enabled systems. In particular, the article details the mind-set that the authors had when they set out to create nesC, including a discussion of the particular requirements they drew up for themselves. The key set of requirements they set out were reliability, being event-driven, having soft real-time requirements, and running well with limited resources. The authors believe that they have met all of these formal requirements, though they were not always entirely happy with the trade-offs they had to accept in order to accomplish that.

While the authors believe nesC to be a rather fine language in and of itself, they wisely chose to show its practical application via their implementation of TinyOS in nesC. They note the variety of architectural structures in TinyOS which make it a perfect candidate for translation into nesC, and post some rather impressive performance and space gains from doing so. For example, they achieved a 38% CPU cycle reduction in the timer event handler alone, and anywhere from 15% to 34% reductions in CPU cycles for several “representative” sample programs. Similarly, for the sample Surge application they post almost a 13% reduction in code size after optimization and inlining. Overall, they made a compelling case for at least considering nesC as a quality programming language custom-made for the design characteristics of embedded devices, particularly wireless sensor networks.

While the cycle reduction and code size improvements were notable, I was more interested in the issues of concurrency that the nesC team was considering. As they note, most of the applications that would likely be run on an embedded wireless device require strong support for concurrent activity. Previous languages used for such devices, particularly C, have insufficient or no concurrent data protection models. That is, they make few or no guarantees about data integrity in a
concurrent environment. This clearly is a major concern when designing an application for a system such as TinyOS, where concurrency is a reality of everyday life. Rather than leaving the programmer to hang himself with his own noose as C tends to do, nesC’s compiler runs some analysis routines in order to detect many possible race conditions caused by asynchronous code accessing variables outside of atomic blocks.

To me, these compiler tools seem to be a more important advancement than the speed gains, or especially the code-size reduction; while the authors believe Moore’s law will be used to shrink the embedded devices rather than make them faster, even in the current marketplace 2kB of ROM is cheap enough to be a negligible concern. On the other hand, no amount of shrinking the die or packing more transistors into a square millimeter is going to fix programming errors. Even excellent programmers are prone to making concurrency errors, as the article points out with their discussion of the errors they found while implementing TinyOS. In total, they found some 156 concurrency errors using their compiler tools, though only 103 of these were actual race conditions. Still, 103 race conditions worked out to be about 6 per 1,000 lines of code, which ought to give an impression of how easy it is to overlook these errors—I doubt the TinyOS designers missed them through incompetence or because they were rushing to meet a corporate deadline. This race-condition checker, while by no means fool-proof, should prove to be a valuable tool for those who utilize nesC.

Another part of the article which I found to be interesting was the discussion on split-phase operation, which curiously was not mentioned at all in the TinyOS article from Assignment 3 (I’m not sure if this is because split-phase operations are inherent only to nesC-TinyOS, with the original C-based TinyOS not utilizing them, or if it simply wasn’t deemed relevant to include in the previous article; all references I can find discuss the netC version of TinyOS. The closest I came was http://www.cs.berkeley.edu/~pal/pubs/patterns-tr.pdf, which notes that “TinyOS depends on split-phase operations, which are a significant departure from C programming with synchronous calls,” but this still doesn’t tell me if split-phase operations are merely uncommon in C, or actually impossible. I suspect “merely uncommon,” which leaves the question open as to whether C-based TinyOS included them or not). While I do not believe a detailed discussion on the semantics and cost/benefit analysis of split-phase operation is appropriate here, the presentation of it in the article makes it sound rather programmer-unfriendly. Instead, it seems that it was chosen more as an expedient and efficient way of getting the operation that they desired for nesC (in this case, high degrees of safe, event-driven concurrency), while sacrificing programmer simplicity; much the same way that well-written Assembly will always get you the fastest executing code, but many programmers would rather shoot themselves than program in it. As I have not actually used nesC myself, perhaps my concerns
here are unfounded, but it does seem to be a significant enough departure from conventional 
programming models to warrant a bit of worry.

The closing paragraphs of this article seem somewhat misplaced. The preceding 8 pages 
were all spent talking about how well nesC lent itself to TinyOS and how they achieved noticeable 
speed and efficiency improvements with only rather minor consequences in terms of expressibility, 
yet the final paragraphs discuss how they see nesC as potentially useful for “enterprise-class 
applications and internet services.” I’m not quite sure how this is. They note at several places the 
trade-offs they were required to make, which limit the expressibility of programmers and are tailor 
made to devices which need not do much computational “heavy lifting.” These trade-offs seem out 
of place with the general needs of an enterprise-class application or many internet services. Even 
the authors seem to realize this, stating that, among other things, nesC would need dynamic memory 
and component allocation as well as the ability for message buffer swapping and a better solution for 
multi-client services before it would be ready for anything other than embedded systems—further 
confusing me as to why they’d bother mentioning it. It would seem to me that they’d be better off 
leaving nesC as what it is—a programming language tailor made for embedded wireless sensor devices; 
If they feel the need to step into larger spheres, they should do so via their compile-time analysis 
tools, which would be of great use to a variety of existing languages that are better suited and intended 
for use in enterprise-class applications or internet services.