

Towards an Architecture for Refactoring Embedded Software for Ubiquitous Environments

Jens H. Jahnke
University of Victoria, Dept. of Computer Science
Engineering Office Wing 321
Victoria, V8W3P6, Canada B.C.
+1 (250) 472 4542
jens@acm.org

1. Ubiquitous Embedded Nets

Today, small-scale computers programmed with dedicated control software (embedded systems) have replaced conventional electronics in almost every application domain. The cost effectiveness of mass-produced, multi-purpose micro controllers and the flexibility provided by the embedded control software has created a great spectrum of new applications. Embedded systems have become ubiquitous. They can be found in a vast variety of products ranging from cars over cellular phones and cameras up to household appliances. Ubiquitous networking of embedded systems is now considered to be one of the computer industry's fastest growing markets. The omnipresence of the Internet and the World Wide Web (Web) via phone lines, cable-TV, power lines, and wireless RF devices has created an inexpensive media for connecting and remotely controlling distributed electronic appliances. Embedded networks are extremely interesting for industrial purposes (e.g., for telemonitoring remote pump stations on a pipeline) as well as for private use (e.g., integration of home alarm system with the local police station).

Several companies have proposed architectures and protocols for integrating embedded components, e.g., *Jini* (general connectivity standard), *Bluetooth* (proximity-based wireless networking), *JetSend* (intelligent service negotiation), *HAVi* (Home Audio-Video interoperability) [12]. These technologies are sufficient for networking particular types of embedded devices in local communities. Still, they are insufficient for building ubiquitous networks of embedded services because of the following reasons.

- (1) Ubiquitous architectures are highly **heterogeneous**. Federated devices use different physical connectivity technology (RF, infrared, LAN, FireWire, Internet etc.). The use of different communication protocols and interface specifications will be inherent in ubiquitous embedded networking. Analogously to so-called *middleware* in federated networks of traditional information systems, infrastructure will be needed to federate heterogeneous embedded devices [1].
- (2) Ubiquitous architectures are **global** in principal. The power they promise stems from the fact that they are

not limited to a particular location or purpose. This feature has significant implications on the infrastructure required for ubiquitous embedded networks. For example, effective access security mechanisms are a major concern for any global network. Such mechanisms gain even more importance for embedded systems because of the critical functionality that they increasingly assume in our society (e.g., public transportation, security systems, power plants, telecommunications, etc.). On the other hand, computational resources on embedded platforms are very limited. Consequently, the access control infrastructure should be externalized as much as possible to centralized, more powerful computing systems.

- (3) Ubiquitous embedded architectures are highly **dynamic** and constantly in flux. Embedded services enter the network, connect to other devices, and drop out again. This characteristic raises several research questions. How can new embedded devices discover embedded services existing in the network? How can network evolution be constrained to “valid” architectures only? How can important quality aspects (e.g., the absence of deadlocks) be verified and validated at run-time?

2. Research challenge: connection-based refactoring of embedded components

Recently, the *connection-based programming* (CBP) paradigm has been introduced in the software engineering domain as an alternative to the traditional “procedure-call model” [2]. Central for the CBP paradigm is the concept of *components* as reusable, independent deployable software units of instantiation. Such components have contractually defined in- and outgoing interfaces of *properties* and *events*. Following the CBP paradigm, software applications are built by “wiring” components, i.e., by instantiating connections that define event and data flow between components. While this technique has not yet had a global breakthrough in *general* software construction, it has become fairly successful and popular in particular software engineering domains, such as GUI development and embedded systems. The reason for this adoption is that the software in these domains can most adequately be

modeled in terms of event and data flows. The key benefits of employing CBP are often summarized as increased *productivity* through reuse of “proven” solutions, and *maintainability* due to low coupling and strict encapsulation. In addition, CBP is ideally suited for developing network-centric systems, since connections are typically considered first-class objects that encapsulate (abstract from) the actual nature of the link (local or remote).

An increasing number of companies and organizations have realized the benefits of CBP, and, thus, strive to migrate existing software to this paradigm. Several integrated development environments (IDEs) and component libraries are offered for CBP. Among them being VisualAge Java (IBM) and mVisual (Intec) [3]. These environments have proven to be excellent tools for the construction of new applications or the replacement of well-defined existing software units (e.g., a GUI). However, in contrast to conventional software systems, many custom-built software assets in the area of *embedded* systems cannot simply be replaced by shrink-wrapped library components [4]. This is mainly because of two reasons:

- embedded software is often highly-optimized and fulfills very specific non-functional requirements;
- embedded software typically is platform-dependent, e.g., specific to a certain micro-controller port configuration.

It is because of these (and other) difficulties that most current vendors of net-centric embedded component libraries like e.g., *emWare* (www.emware.com) do only deliver shrink-wrapped GUI components for the integration with “native” embedded code – rather than offering components that include the embedded code. Even though it is unlikely that any component vendor will ever be able to create an exhaustive library of embedded components, the aforementioned benefits make it still highly recommendable for organizations to *refactor* existing embedded software to a component-based paradigm. Moreover, the migration towards CBP is a key prerequisite for integrating embedded systems in ubiquitous networks.

The envisioned component refactoring process will consist of the four following steps:

1. Component identification (Select to software fragments to be componentized.)
2. Reverse engineering of component interfaces (Recreate precise contractual interfaces for the new component.)
3. Transform of component interfaces (Transform the interfaces of the identified software fragment into the target component interfaces.)
4. Slice the component out of the legacy system.

In addition to these four steps, the refactoring process might involve further steps, depending on the actual nature of the migration:

- Integration of net-centric GUI to remotely control the embedded component
- Migration from proprietary language (assembler, C, etc.) to component-based language (e.g., embedded JavaBeans)

3. Future work: RESCUE

We believe that the sketched refactoring process can and should be automated as much as possible in order to increase the effort and cost involved in migrating to CBP. Although, full automation is highly unlikely, a human-centered refactoring tool could significantly reduce the effort and risk of migration. Integrated with a component-based IDE and a “wiring” tool for net-centric technology (e.g., Jini, BlueTooth, JetSend, etc.), such a refactoring tool would facilitate rapid migration of existing embedded software assets to ubiquitous environments.

In a future collaborative research project called RESCUE (*Refactoring Embedded Software Components for Ubiquitous Environments*), we will focus on the development of methods, processes, techniques, and tools to realize the described functionality. In particular, we will build on our previous experiences in reengineering and migrating legacy software to net-centric environments. The proposed research will be carried out in tight collaboration with companies in the area of embedded systems.

We intend to evaluate our research with a real-world case study provided by the Herzberg Institute for Astrophysics (Victoria). This case study will consider the development of software for a distributed network of embedded micro controllers for a new super radio telescope array. Software for these embedded controllers partially exists already in form of software fragments that were developed for previous telescopes. We apply our research results to this case study and evaluate the results against detailed success factors defined in a long version of this proposal.

References

1. Emmerich, W., *Engineering Distributed Objects*. 1999: Addison Wesley.
2. Szyperki, C., *Component Software, Beyond Object-Oriented Programming*. 1997: Addison-Wesley.
3. d'Entremont, M. and J.H. Jahnke. *microSynergy - Generative Tool Support for Networking Embedded Controllers*. NCC'01, 2001. Toronto: ACM Press.
4. Jahnke, J.H. *Engineering Component-based Net-Centric Systems for Embedded Applications*. ESEC/FSE '01. Vienna, Austria: ACM Press.