Meta-Debugging Pervasive Computers

Alexander Bernauer
Institute for Pervasive Computing
ETH Zürich, Switzerland
bernauer@inf.ethz.ch

Kay Römer
Institute of Computer Engineering
University of Lübeck, Germany
roemer@inf.ethz.ch

ABSTRACT

As computers get more complex, the task of programming them gets more complex as well. This is especially true for the "Pervasive Computer", which is a massively distributed system consisting of unreliable embedded devices that communicate with each other over lousy wireless links. A common approach to address the programming problem is to offer programming abstractions that hide certain aspects of the complexity from the programmer. While several such abstractions and mappings thereof to low-level target languages have been proposed, there is a glaring lack of debugging support. It is typically impossible to debug at the conceptual level offered by the programming abstractions, instead one has to resort to debugging the generated target code. In this position paper we argue that programming abstractions should be designed in a way that allows debugging at the same conceptual level as programming. We further present requirements for such debugging tools, a taxonomy of programming abstractions and discuss debugging challenges, existing solutions, and potential approaches in each class.

1. INTRODUCTION

Computer systems tend to get more complex over time. The *pervasive computer*, for example, consists of many resource-constrained embedded devices that form wireless ad hoc networks to cooperatively solve a task. Programming these systems is difficult.

One common way of addressing this difficulty is to raise the level of abstraction in order to relieve the programmer from lower-level details. Hereby the basic idea is to provide proper programming abstractions by means of a formal meta-language to a software developer in a given application domain.

The benefits of programming abstractions are frequently cited. More abstraction means less code which in turn means fewer possibilities for bugs, be it typos, copy-and-paste errors, inconsistencies, or logical errors. Furthermore, by explicitly representing the concepts of the problem domain in the programming language, a program is easier to create and understand for domain experts. On the long way from an idea in somebodies mind to an executable representation of the same, some steps have to be performed manually and some can be automated. The overall goal of programming abstractions is to reduce the manual part as much as possible.

For the scope of this article, we will adopt the term *programming abstraction* that is embodied in a meta-language. A program is an instance of a language and forms a solution for a problem in the domain of that language [3]. A programming abstraction is implemented by mapping it to lower-level abstractions which are themselves embodied as a program in a target language and so on until a program can be directly interpreted or executed by a virtual or physical machine. We focus on programming abstractions in the field of pervasive computing, in particular for applications of wireless sensor networks (WSN).

In this field, several programming abstractions do already exist. For most existing programming abstractions though, debugging cannot be performed at the conceptual level of the programming abstraction, but has to be performed at the level of the target language to which the programming abstraction is mapped [11]. This is a major difficulty, as the developer suddenly has to become aware of the target language, and has to understand its semantics and the semantics of the mapping to the target language.

In this position paper we discuss the problem of debugging abstract programs and we postulate that researches should address the question how such programs can be debugged at the conceptual level of the programming abstraction, thus hiding the very existence of the target language, its semantics, and the mapping – much like most C programmers are unaware of the syntax and semantics of the assembly target language.

Besides the C programming language and its tool chains, there are various different technologies which raise the level of abstraction in order to mitigate the growing complexity of computer systems. For example, the software engineering community has devised *Model Driven Development*. Others have proposed *Language Oriented Programming* [3] or *Intentional Programming* [10]. Additionally, internal and external *Domain Specific Languages* (DSL) are hot topics in the design of programming languages. While some debugging concepts from those technologies could potentially be applied to the pervasive computing context, pervasive computers differ from other systems in their distributiveness, scale, heterogeneity, resource and energy constraints, as well as their deep embedding into the real world. Hence, we believe that research is required to devise not only tools, but the fundamental concepts of how to debug pervasive computers.

In the remainder of this paper, we present requirements for debugging and a coarse taxonomy of programming abstractions. For each class in the taxonomy, we discuss the challenges involved in debugging, what kind of debugging support is required, and how severe the lack of the latter is.

2. REQUIREMENTS AND TAXONOMY

While it would be desirable to build systems that are correct by design or at least to perform a complete static verification of the program code, this is only partially possible for pervasive computers because it would require a complete formal model of the real world. Unlike other computer systems, pervasive computers closely interact with the real world, not only by means of sensing and controlling physical processes, but also through undesirable interactions due to environmental influences. For example, environmental temperature has a strong influence on the frequency and start-up time of crystal oscillators, on received radio signal strength, and on the (dis)charging characteristics of batteries and is thus sometimes causing unpredictable partial node and communication failures. Hence, debugging pervasive computers is a necessity. By debugging we mean finding and removing the causes of incorrect application behavior by examining its runtime.

In particular, we consider two common types of bugs. Firstly, logical errors in the application code and, secondly, incorrect assumptions in the application code about the semantics of the target language and system to which the application code is mapped. In existing systems, incorrect assumptions concerning the reliability and the timing of message delivery for example are a common cause for failures.

In principle it is possible - and this is what we are aiming at - to find the above two types of bugs at the conceptual level of the programming abstraction, i.e., without exposing to the user the target language, its implementation, and the mapping of application code to the target language.

Debugging techniques for pervasive computers have to meet a number of constraints. Firstly, it must be possible to debug large heterogeneous networks of pervasive computers. Secondly, resource and energy constraints of pervasive computers require that resource and energy consumption of debugging techniques is minimized - otherwise we run the risk of severe probe effects, where the debugger significantly alters the program execution, resulting in (dis)appearance of bug symptoms.

To better understand the problem of *debugging abstract programs*, we will discuss debugging characteristics of different classes of programming languages for pervasive computers. For this taxonomy, we see two major dimensions: imperative vs. declarative languages and languages for node-centric vs. distributed applications. Actually, those two orthogonal dimensions form a subset of current and common taxonomies for WSN programming models [11] and we consider them crucial for debugging.

With imperative languages "the intended application processing is expressed through statements that explicitly indicate how to change the program state" [11] while with declarative languages "the application goal is described without specifying how it is accomplished" [11]. This difference is important because with imperative languages the mapping between source code and target code tends to be one to one. But by raising the level of abstraction towards declarative languages the mapping becomes more complex and even ambiguous, thus making it hard to perform the back-mapping which is required for debugging.

The second dimension of our taxonomy addresses the problem of shared state which is changed concurrently. As opposed to node-centric programs, with distributed programs well-known concepts such as breakpoints need to be reconsidered to make them applicable for distributed applications.

In the following sections we discuss the four classes of programming abstractions which are formed by those two dimensions. While we focus on programming abstractions for pervasive computers, we also look at related application domains that raise similar problems as an inspiration of how one might tackle the problem for pervasive computing.

3. IMPERATIVE META-LANGUAGES

In the WSN community the most famous example of an imperative meta-language is probably TinyOS [6]. The programming language of TinyOS is nesC, which is a DSL for the domain of event-based and component-based applications. The nesC compiler translates nesC code into C code which is then compiled by a platform compiler for an embedded platform such as avr-gcc[1].

Imperative programming languages are a list of commands and command abstractions which tell the executing machine what it is supposed to do. So the most natural way of debugging is to observe the sequential execution of these commands by means of a source-level debugger. Programmers are familiar with this concept because most tool chains provide such as debugger. In fact, a C programmer would very likely refuse to use a tool chain which does not offer a debugger that allows to set breakpoints in the source code and to monitor variables during execution. This is because by the lack of a source-level debugger he is forced to trace execution on the assembler level, find the flaw there and infer the location of the bug in the C code. Not only must he know the complete target language but also the details of the transformation performed by the compiler in order to be able to undo it.

In the case of TinyOS this means that a developer must use the debugger of the C tool chain and step through the generated C code. As a consequence she must be familiar with

[1]http://gcc.gnu.org/

the implementation-specific naming scheme for the generated C symbols [4] and must know how the wiring of nesC components is mapped to C code in order to map back the location of a bug in the generated C code to the corresponding location in the nesC code.

To support source level debugging, C compilers add debugging information to the assembly output, indicating which line in the C source code has been mapped to which lines in the assembly code. A source-code debugger can use this information to map the execution state of a binary program to lines in the source code and to names of variables. As C itself is a programming abstraction, this technique can be adopted to support debugging of imperative meta-languages.

In fact this has been done for the YETI Eclipse plugin [1] which is the first source-level debugger for the nesC programming language. It uses the Eclipse C/C++ Development Tooling (CDT)[2] to communicate with an underlying instance of the GNU Debugger (GDB)[3] and automatically performs the inverse mapping of the nesC compiler. This is possible because the nesC compiler records which part of the generated C file originates from which part of the various nesC input files by exploiting C preprocessor instructions. To the best of our knowledge, the YETI project is gladly embraced by the TinyOS community because it enables the developer to completely ignore the fact that nesC programs are in fact mapped to C.

Along the lines of YETI, we suggest that programming abstractions should always be designed such that they are amenable to source-level debugging. Probably the most notorious imperative programming abstraction without debugging support are C++ templates. Compiler error messages caused by bugs in C++ template code are so verbose and hard to read that projects like stlfilt[4] try to provide a reverse mapping by means of compiler-specific heuristics. An additional problem with C++ templates is that they are also used to implement declarative languages as opposed to imperative ones, as discussed in the subsequent section.

4. DECLARATIVE META-LANGUAGES

Declarative languages specify a goal, but not how this goal can be achieved. Therefore, a program written in a declarative languages is often called a model. Parser generators are a well known example of tools which process declarative input in order to generate executable implementations of the parser. In the case of Boost.Spirit[5], the intended parser is declared by means of instantiations of C++ templates. Other projects like ANTLR[6] use a dedicated meta-language instead. Another common case of declarative meta-languages are persistence and database abstractions as in the case of the Hibernate Framework [7] or RubyOnRails[8].

In general, debugging becomes harder in the case of declarative languages. First, because they tend to be more abstract and second, because the transformation to an imperative target language is not a simple one-to-one mapping any more. As declarative languages are not based on the sequential execution of commands, debugging concepts such as single-stepping and breakpoints are not applicable.

Instead debugging can be supported by providing different views on the model so that the programmer can perform a visual inspection. Visual inspection is only applicable to small models, but fortunately raising the abstraction often leads to reducing the size of the code. Intentional Software[9] applies this approach and claims to have received positive feedback from their costumers[17], which in their case are usually domain experts without IT background.

An alternative general concept for debugging models is to simulate or execute example cases. KODOS[10], for example, offers such an approach for debugging regular expressions. Given a regular expression and an example string, the application highlights the parts of the string which match and displays the contents of the matching groups. So KODOS basically supports and automates trail-and-error cycles which otherwise would have to be performed manually.

In the worst case without debugging support, the programmer is forced to read or debug the generated code. As the gap between the programming abstraction and the target language is typically rather high, the disadvantages of this approach as described in Sect. 3 are even more severe.

5. IMPERATIVE AND DISTRIBUTED

The pervasive computer is a massively distributed system with unreliable components. As programming such distributed systems is difficult, imperative programming languages have been devised that abstract from certain distribution aspects. One example for such a language is Kairos[5], which allows computations involving different nodes without explicit network communication. This is very convenient but yet it is completely unclear how to debug Kairos programs. Traditional single-node source-level debuggers cannot be applied here, as single stepping or breakpointing one node may violate timing assumptions in the remaining network which can itself lead to failures. Clairvoyant [18] addresses this issues by adding WSN specific features to the GDB, i.e. global stop and continue commands. The downside still remains, though, which is that interactive debugging sessions on deployed networks deplete the batteries due to heavy usage of the energy consuming radios thus reducing the overall network lifetime.

A promising imperative meta-language for WSNs is MacroLab [7] which is a vector-based meta language and framework inspired by Matlab. MacroLab programs are composed of computations on vectors containing data on nodes such as sensor values, internal state, and parameters for actuators. Hereby the communication of the vectors among the nodes is hidden from the programmer. Though, the most important fact is that there is a debugger (MDB) [16] for MacroLab. MDB is a post-mortem debugger which synchronizes

[2]`http://www.eclipse.org/cdt/`
[3]`http://www.gnu.org/software/gdb/`
[4]`http://www.bdsoft.com/tools/stlfilt.html`
[5]`http://boostspirit.com/home/`
[6]`http://www.antlr.org/`
[7]`https://www.hibernate.org/`
[8]`http://rubyonrails.org/`
[9]`http://intentsoft.com/`
[10]`http://kodos.sourceforge.net/`

and analyzes log files written by the individual nodes. It supports common debugging concepts such as breakpoints and stepping for both temporally and logically synchronous views. Because MDB does not operate at runtime, it can and does support time travel and rendering effects of hypothetical changes. Furthermore there is a deployment mode in which no logging is done but the timing of the debugging mode is preserved thus avoiding probe effects.

Apart from MacroLab and MDB, few approaches exist to assist in debugging networks of pervasive computers at the conceptual level of the programming language. One such approach is to annotate the program with distributed assertions [14] that formulate hypothesis about distributed variables which are checked by a a runtime system. When placed carefully in the code, failed assertions can help debug. However, this is largely decoupled from the programming abstractions offered by the meta language. As in this example, today one typically has to resort to distributed debugging techniques that are independent of the actual programming abstractions. Dustminer [8], for example, applies data mining techniques to event traces obtained from nodes to find bug symptoms. Other tools [12, 2, 13, 15] can detect certain predefined problems such as node reboots, routing loops, or network partitions. But there is no easy way to relate these symptoms to possible causes in the program.

6. DECLARATIVE AND DISTRIBUTED

Languages in this class offer declarative programming of a distributed system as a whole. One well-known example in the context of WSNs is TinyDB [9] which provides a database abstraction. The outputs of distributed sensors form a virtual database table, over which the user can issue SQL queries. Hereby the user is largely unaware of the fact that he is actually programming a distributed system.

With respect to debugging, the difficulties of declarative and distributed languages multiply in this class of our taxonomy. It is already unclear which debugging concepts should be applied to declarative queries, let alone the adoption of such concepts to a distributed setting. To the best of our knowledge no solution exists to support debugging in this class.

7. CONCLUSION

We postulate that programming abstractions should be designed in a way that allows debugging at the same conceptual level as programming. The goal should be that developers can work on the abstract level without having to think about lower-level implementation issues. In the case of debugging pervasive applications, though, this goal is hardly achieved as there are only a few good examples such as YETI or MDB.

We believe that the reason for this is that augmenting programming abstractions with debugging capabilities is not merely an engineering issue. Research is required to devise fundamental debugging concepts for declarative and distributed languages. Known source-level debugging concepts such as single-stepping or breakpoints are neither applicable to declarative languages nor to distributed settings. Once we have devised such concepts, we can investigate the implications on the design of debuggable programming abstractions.

8. ACKNOWLEDGEMENT

The work presented in this paper was partially supported by the Swiss National Science Foundation under grant number 5005-67322 (NCCR MICS), and by the European Commission under contract number FP7-2007-2-224053 (CONET).

9. REFERENCES

[1] N. Burri, R. Flury, S. Nellen, B. Sigg, P. Sommer, and R. Wattenhofer. Yeti: an eclipse plug-in for tinyos 2.1. In *SenSys '09*.

[2] B.-R. Chen, G. Peterson, G. Mainland, and M. Welsh. Livenet: Using passive monitoring to reconstruct sensor network dynamics. In *DCOSS '08*.

[3] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains OnBoard Online Magazine*, 2004.

[4] D. Gay, P. Levis, D. Culler, and E. Brewer. *nesC 1.2 Language Reference Manual*, 2005.

[5] R. Gummadi, N. Kothari, R. Govindan, and T. Millstein. Kairos: a macro-programming system for wireless sensor networks. In *SOSP '05*.

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.

[7] T. W. Hnat, T. I. Sookoor, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrolab: a vector-based macroprogramming framework for cyber-physical systems. In *SenSys '08*.

[8] M. Khan, H. Le, H. Ahmadi, T. Abdelzaher, and J. Han. Dustminer: troubleshooting interactive complexity bugs in sensor networks. In *SenSys '08*.

[9] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[10] H. K. C. Magnus Christerson (Intentional Software). Intentional software. In *QCon London*, 2008.

[11] L. Motolla and P. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys*, 2010. To be published.

[12] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *SenSys '05*.

[13] M. Ringwald, K. Römer, and A. Vitaletti. Snif: Sensor network inspection framework. Technical Report 535, Dept. of Computer Science, ETH Zurich, 2006.

[14] K. Römer and M. Ringwald. Increasing the visibility of sensor networks with passive distributed assertions. In *REALWSN '08*.

[15] S. Rost. Memento: A health monitoring system for wireless sensor networks. In *In SECON 2006*, 2006.

[16] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrodebugging: global views of distributed program execution. In *SenSys '09*, 2009.

[17] M. Völter. Intentional Software with Shane Clifford. *software engineering radio*, 151. available at: http://www.se-radio.net/.

[18] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *SenSys '07*.