Towards Distributed Robustness in Embedded Systems

Václav Mikolášek and Michael Zolda
Institute of Computer Engineering
Vienna University of Technology
Vienna, Austria
{*mikolasek,michaelz*}*@vmars.tuwien.ac.at*

1. Degeneracy, Robustness, and Complexity

Recently, we witness a significant amount of effort being put into a research of robustness of complex computer systems. Computer scientists noticed that, despite decades of investigations into dependability, computer systems still lack the degree of resilience which can be seen in biological systems. In the biological context, one of the main aspects that stand behind a system's robustness is *degeneracy* [1] also known as *distributed robustness* [2].

We say that a system has a high degree of degeneracy if a high number of the system's parts *functionally overlap*. A system designer can harness degeneracy and provide fault-tolerance through *procedural* (in contrast to structural) redundancy. A system element is structurally redundant if another element in the system provides identical functionality. However, an element is procedurally redundant if various system parts can interact in such a way that it is possible to reproduce the element's functionality.

For illustration, consider a hypothetical calculator which was designed for degeneracy. Such a calculator does not have a central CPU, but its functionality is rather distributed among number of function blocks which provide basic arithmetic operations (plus, minus, multiplication, and division). If one of the function blocks is affected by a permanent error, there is no other functionally identical element which could take over the lost function. Yet, the blocks functionally overlap. If, for example, the multiplication operator failed, the calculator could transform any multiplication into a sequence of additions and use the plus operator. If the plus operator failed, the minus operator can be used in a similar fashion.

Dependable computer system of today are still clumsily "inorganic", with a nonexistent or very low degree of adaptability. Procedural redundancy due to degeneracy is a promising way to achieve highly robust and adaptable systems. But designing for degeneracy, and that is where we want to make our point, requires a paradigm shift in the way computing is done. We argue for a novel model of computation which enables a design for degeneracy.

2. Hierarchies

Alongside degeneracy, hierarchical structure, too, is essential for robustness – both in biological and artificial systems. System components on all levels of a hierarchy exhibit the same pattern: intra-component interactions are strong and frequent whereas the inter-component ones are relatively weak and sporadic. This pattern is responsible for *near-decomposability* of a system and this in turn provides for adaptability. A system component at a certain level of hierarchy can be replaced by another component at the same level (structural redundancy) or recreated through interactions of other components (procedural redundancy due to degeneracy).

With embedded systems – which lay at the borderline of hardware and software – we face a particular situation. There are two separate, unrelated hierarchies: hardware and software. Indeed, any software abstraction dissolves the moment a program is compiled into a machine code and executed, one instruction after another. The hardware layer is an execution platform for software and a software hierarchy is in no sense a continuation of the hardware hierarchy: *software components are expressed in programming languages which do not reflect the structure of hardware*.

The reason why current embedded systems do not exhibit degeneracy and do not make use of procedural redundancy is exactly the separation of the two hierarchies. Besides structural redundancy, present systems do not provide any other way to recover, for example, a failed PID controller. Indeed, it is because the PID controller is not expressed in terms of lower-level component interactions and therefore the system cannot make use of a procedural redundancy as in the calculator example. Even if the PID controller was expressed in terms of component interactions (and note that due to its simplicity, the lower-level components would need to be in fact basic arithmetic operations or similarly simple functions) it would not be of any help as long as the system itself is not able to separately invoke other lower-level components in a new context and thus recreate the PID functionality.

Edelman and Gally [1] pointed out that degeneracy can be observed only in sufficiently complex systems. In order to achieve a high degree of degeneracy in embedded systems, we need to change the way computation is done – we need to connect the software and hardware hierarchies.

3. A Design for Degeneracy

In order to create a computer system with a high degree of degeneracy (so that we can take advantage of procedural redundancy) we must meet the following requirements:

1) Each non-atomic component is expressed as a combination of lower-level components from a common component set.
2) Atomic components form fault-containment units and
3) they are also the basic building blocks of a programming language used to implement the system.
4) The system is built in a bottom-up fashion.
5) A good degree of a functional overlap is preserved on all levels of the system hierarchy.
6) The system has a sufficient degree of complexity.

The first requirement ensures that a system has a hierarchical structure and particular system elements belong to a common component set, that is, they can be invoked independently in different contexts. Recovery, diagnosis, and procedural redundancy would be very difficult to achieve, if the system design did not meet the second point requiring a physical separation of atomic components. The argument behind the point 3 is that in a strictly hierarchical design, a programmer expresses each system part as a combination of lower-level system parts and a programming language must capture this feature by making the system's atomic components its own basic building blocks. Top-down design leads to a creation of specific purpose components which cannot be easily reused in different contexts, in contrast, a bottom-up design leads to a set of general purpose components which are repeatedly used across the whole system. Although the bottom-up approach is a help, only if a designer chooses the sets of components on different levels of the system hierarchy carefully, the resulting system will obtain a high degree of degeneracy; this requirement is captured by the point 5. Although the sixth requirement is probably the least specific one, it has a great importance. Even with an intuitive understanding of complexity, we can see that a component can be made procedurally redundant only if the system provides a number of functionally overlapping parts. Such complexity is lacking, for example, in a simple control system containing a few sensors, actuators, and a controller. However, in the next section, we shall show that by connecting the hardware and software hierarchies, even this simple system can make use of degeneracy.

4. Dense-network-based Computation

Although current software engineering practice favors a top-down system design, large-scale distributed systems of today (e.g., those seen on the Internet) can show degeneracy and make use of procedural redundancy. The complexity of such systems is arguably sufficient, computer nodes (i.e., servers) form fault-containment units, and there is no reason why the rest of the six requirements from Section 3 could not be met. However, in embedded systems we once again face a particular situation:

- either a computer node (or an IP core) is seen as an atomic, fault-containment unit thus meeting the second requirement, but failing to meet the first and fourth because the computer nodes implement application specific and high-level components such as a controller, actuator, etc.,
- or the high-level components are expressed in terms of lower-level components interactions, but then the later ones do not form fault-containment units due to the separation of hardware and software hierarchies.

For these reasons, we argue for a new computational paradigm based on *computation via dense-networks* [3] and present our vision of a *dense-network processor* (DNP).

A DNP moves away from the current instruction execution paradigm towards a network-based computing. The core of the processor comprises thousands of atomic, simple, and highly interconnected computational nodes. A program for such a processor expresses its functionality strictly in terms of creating connections and addressing among the atomic components. An operational memory is replaced by a component repository where the description of components is stored along with the component's state. A controller/scheduler oversees the processor's operation and authorizes components instantiations, serialization, addressing, and so forth. A DNP merges software and hardware hierarchies (each software component can be directly mapped to a dynamic hardware structure) and makes a design for degeneracy and procedural redundancy possible. With DNPs, we can meet all the six requirements from Section 3 and achieve a high degree of degeneracy even in embedded systems. On a system level, projects such as GENESYS [4] could provide the needed hierarchical continuation to the low-level degeneracy established by DNPs.

We believe that investigations of procedural redundancy via dense-network-based computing yields a very promising research program with a vast potential. With new nanoscale technologies emerging, a DNP need not remain only a vision.

Acknowledgement

Our thanks go to Hermann Kopetz and Sven Bünte for the important discussions we had. This work was in part supported by an Austrian FIT-IT project CLIC no.819482.

References

[1] G. M. Edelman and J. A. Gally, "Degeneracy and complexity in biological systems," in *Proceedings of the National Academy of Sciences of the United States*, 2001.

[2] A. Wagner, "Distributed robustness versus redundancy as causes of mutational robustness," *Bioessays*, vol. 27, no. 2, pp. 176–188, Feb 2005.

[3] N. Shanbhag, S. Mitra, G. de Veciana, M. Orshansky, R. Marculescu, J. Roychowdhury, D. Jones, and J. Rabaey, "The search for alternative computational paradigms," *Design & Test of Computers, IEEE*, vol. 25, no. 4, pp. 334–343, July-Aug. 2008.

[4] R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz, "Fundamental design principles for embedded systems: The architectural style of the cross-domain architecture GENESYS," in *Proceedings of ISORC conference*, 2009.