Tolerant Software

A White Paper for the Workshop on New Visions for Software Design and Productivity

By

Robert Laddaga

MIT AI Lab
NE43-804
200 Technology Square
Cambridge, MA 02139

617 253 4150

rladdaga@ai.mit.edu

Among the most difficult problems in software development that we will face in the future, are those connected with producing software for embedded systems. Examples of such systems include cell phones, watches, cd players, cars, airplanes, and oil refineries. Many of these use several, hundreds or even thousands of processors. The real-world processes controlled by embedded processors introduce numerous difficult constraints that must be met in programming these embedded processors. Among these difficult constraints are real-time requirements (such as scheduling and prioritizing), signal processing requirements (mitigating noise, signal fusion), and actuator control problems (stability and predictability). These constraints are difficult precisely because they affect our programs globally, and because our programs contribute globally to these problems. In this white paper, we discuss why rational, distributed resource management and tolerant, negotiated interface technologies are needed for embedded systems, and how we might develop them. Resource management is required for all sufficiently complex software driven systems. Rational resource management considers diverse deterministic and probabilistic information about resource allocation issues, as well as explicit utilities for resource usage, and then computes or approximates maximization of expected utility. A distributed approach to resource management envisions individual agents solving local problems via negotiation, and building dynamic problem solving hierarchies. Tolerant interfaces in software components are designed to inquire or discover the requirements of another component, be able to communicate its own requirements, and then operate on itself to produce the required interface behaviors.

Embedded applications

Among the most difficult problems in software development that we will face in the future, are those connected with producing software for embedded systems. Embedded systems are systems in which physical, chemical, or biological processes or devices are controlled by computer processors. Examples of such systems include cell phones, watches, cd players, cars, airplanes, and oil refineries. Many of these use several, hundreds or even thousands of processors. Today, most such systems operate with limited memory and use eight bit processors. Each year, we produce more processors than there are people on the earth, and there is a significant rate of growth. Almost all of these processors are used in embedded applications, rather than the workstation, personal computer or mainframe systems with which we are most familiar.

The real-world processes controlled by embedded processors introduce numerous difficult constraints that must be met in programming these embedded processors. Among these difficult constraints are real-time requirements (such as scheduling and prioritizing), signal processing requirements (mitigating noise, signal fusion), and actuator control problems (stability and predictability). These constraints are difficult precisely because they affect our programs globally, and because our programs contribute globally to these problems.

The main simplifying tool we have in software development is functional abstraction in the form of functional decomposition. To the extent that we can modularize via functional decomposition, we can reduce the effective complexity of the software that we write. Complexity increases whenever such modularity is interfered with, in the sense of needing to take account of the internals of other modules while coding a different module. As such cross cutting concerns multiply, the job of programming can become impossible. Real-time requirements, and coordination with physical processes introduces a huge number of such cross cutting concerns, violating modularity with abandon.

In order to deal with these problems, we need software technology in several related areas, including rational and distributed resource management, multiple decomposition dimensions, self-adaptive software and tolerant and negotiated interfaces, among others. Multiple decomposition dimensions (or Aspect Oriented Programming) is a hotly discussed and important issue today [1]. Self adaptive software is also an important topic, that I have discussed elsewhere [2],[3],[4].

This white paper concentrates on resource management and tolerant interfaces. In the following sections, we discuss why rational, distributed resource management and tolerant, negotiated interface technologies are needed, and how we might develop them. In the process, we will see that negotiated interfaces play a central role in both technologies.

Rational, Distributed Resource Management

Resource management is required for all sufficiently complex software driven systems. Rational resource management considers diverse deterministic and probabilistic information about resource allocation issues, as well as explicit utilities for resource usage, and then computes or approximates maximization of expected utility. This approach is of significantly greater use if it is combined with a dynamic, rather than static, approach to resource allocation. Often when we speak of resources, we are talking about computational resources, including processors, time slots, memory, disk space, network bandwidth. In general, however, resources can include all

peripheral devices, such as sensors and effectors, the objects sensed and effected, functions, plans, organizational notations, and programming capabilities. It is this extended notion of resources that we will consider in this white paper.

In addition to considering rational resource management, we also want to consider distributed approaches to resource allocation. Some systems seem to scale remarkably well, despite enormous growth in the number of components. Economic systems, and ecosystems come to mind as two examples. Ecosystems seem to mostly be limited by resource availability, which is quite different from excess complexity (the complexity of ecosystems is staggeringly high). Economic systems also seem to be enormously complex and function remarkably well, and in addition are human creations, rather than natural occurrences.

What these two system types have in common, is distributed control with individually controlled agents. Such systems appear to have inherent robustness and adaptability, and scale well in part because they are self organizing around local concerns. When we build static organizations, like teams, armies, and business units, we create a hierarchical structure that has certain capabilities and assigns certain roles and responsibilities to members. As problems are encountered, we attempt to map the problem against the organization, and apply the appropriate organizational elements to the problem. Naturally, the fit is never really very good.

An alternative approach solves local problems with locally grown organizations, temporarily assembled to solve the problem at hand. In this way the problem can be addressed with an organization tailored to the problem, rather than force fitting the organization to the problem. In this sense we see that organization is dynamically bound to the problem, and is a simple extension of ideas of dynamic object technology and self-adaptive software.

One approach to such decentralized resource management is resource management by negotiation protocols. The goal of Negotiation Software Agents (NSA) is to autonomously negotiate the assignment and customization of dynamic resources to dynamic tasks. Negotiation systems will scale to much larger problem sizes by making maximum use of localized, rather than global information, and by explicitly making decision theoretic trade-offs with explicit time-bounds on calculation of actions. This new technology will enable us to build systems that are designed to utilize, *at the application level*, all the distributed, networked computational resources (hardware, operating systems, and communication) that have been developed over the past two decades. These advantages will be especially effective in cooperative action, since the process of matching tasks and resources is naturally decentralized in the NSA approach [5], [6].

The first NSA task is development of the basic framework for bottom-up organization. This includes three subtasks, the first of which is discovery of peer NSAs, and their capabilities, tasks and roles. Second, NSAs must be able to determine how to get access to data, information and authorizations, and they must have secure authorization procedures. Finally, there must also be processes for coordination and information sharing at levels above the peer NSAs.

The second NSA task is reasoning based negotiation. Its subtasks are: handling alternatives, tradeoffs, and uncertainty, including noting when new circumstances conflict with current plans; enabling speedy, optimized and time-bounded reasoning processes to allow for real-time response; and procedures for assuring that goals are met [7],[8].

Software tolerance and active interfaces

We have all learned that an important and enabling component of the industrial revolution was the invention of interchangeable parts. The concept of interchangeable parts is the mechanical

equivalent of the ideal decomposition story that we told above. When items are made so that each part is designed and crafted to fit just in the assembly it will be added to, it is difficult to develop procedures and process that scale with the number of items to be produced. It is also more difficult to repair and replace parts. Interchangeable parts require the ability to manufacture parts to reasonably accurate tolerances, and the ability to design assemblies such that interfaces require no more severe tolerance than manufacturing can deliver. To support such notions careful, accurate design technology was needed, the ability to communicate designs and specifications, and the ability to set standards. A significant part of our current industrial capability is grounded on significant limitations in the number and kind of mechanical parts that are readily obtainable: a small finite number of screw sizes, pitches, lengths and materials of which they are made, for example.

A similar tale could be told for software. Develop technology for producing modules that have interfaces designed to less exact tolerances, design systems to support those tolerances in components, and develop standards for component interfaces. We could do this, *but it would be wrong!* Why this is so is somewhat subtle, but basically boils down to wasting an enormously valuable capacity of software to be flexible and active.

Let's go back to our mechanical motivation, in order to understand how we would be wasting software's special capabilities. We could in fact, design and build mechanical screws that have actively variable thread sizes. Imagine a mechanism inside the screw that can be made to push the threads, now made of separate components, apart and out. We control this feature with a small set screw in the head of the screw. Such a screw would be rather valuable. For example, as wood dries, and our screw got loose, we could simply turn the set screw in the head of the screw, growing the threads and causing the screw to tightly grip the wood again. However there are some disadvantages in such a screw. For one thing, it is much more difficult to design, but that effort and cost could be easily amortized over several million screws. But it would also be much more expensive to produce. Each screw would be an *assembly* instead of a single casting. It would be unrealistically expensive to replicate such screws.

Back to software. Making software flexible and actively changeable is not easy, but it is much easier than doing so for mechanical parts. For one thing, all software is already an assembly of component instructions (at whatever level they are expressed). Additionally, software can operate on itself, in ways that mechanical parts can rarely do because of physical constraints. But even more importantly, it is in principle no more expensive to replicate highly flexible software with active interfaces, than it is to replicate dumb, standardized, everything gets by with one size software. The cost is in figuring out how to interface diverse software, and if that burden can be placed on the software itself, there is no reason not to pursue enormous customization of functionality in software modules.

It is however, less easy to figure out how to build such active interfaces. In general, we need to design software components to inquire or discover the requirements of another component, be able to communicate its own requirements, and then operate on itself to produce the required interface behaviors. This certainly sounds difficult and complicated, but we already have a pretty good example of such a tolerant interface: modems.

Modems initiate communication using a negotiation protocol that attempts to find the best communication protocol that both modems have available, and that the current line conditions will support. If communication becomes difficult, they can renegotiate their protocol. This is a marvelous capability that is not as appreciated as it ought to be, and demonstrates that at least in some cases, tolerant interfaces aren't overly difficult.

Clearly, to have these active interface capabilities, code will need to contain and operate on explicit descriptions of intent, specifications, design, program structure and mapping of tasks to functions. Also, the code will need to contain a large number of alternative interface implementations, or have access to such alternatives over network connections.

Another important concept for tolerant interfaces is that of wrapped data (objects, agents or smart data). Consider, for example, self-extracting compressed archives of programs. The file contains the software needed to decompress, and install the program at your site. Even now such installation programs accept some parameters from the user, but more could be done in this regard. This type of approach is useful in situations where you have data exchange where provider and consumer share responsibility or capability to operate on the data.

Imagine a sensor-actuator application, in which image data is gathered by a sensor, and delivered to a consumer that controls actuators. The sensor module sends data wrapped/prefixed with code for interpretation of the data. The consumer needs to change some operators or filters, in the context of the sensor's interpretation code. The sensor and consumer code cooperatively arrange interfaces and a configuration suitable for the appropriate sharing of processing tasks and algorithms.

Tolerant software requires research on:

1. Automated analysis and implementation of interface requirements
2. data wrapping/prefixing
3. calculation and mitigation of risks and consequences of software breakdown
4. assurance/validity checking for tolerant combinations
5. performance optimization

References

[1] Kiczales G., Lamping J., Mendhekar A. et. Al. Aspect-Oriented Programming in proc. European Conference on Object Oriented Programming, Finland, 1997.
[2] ``Self adaptive software,'' 1998. DARPA, BAA 98-12, Proposer Information Pamphlet.
[3] G. Karsai and J. Sztipanovits. A model-based approch to self-adaptive software. IEEE Intelligent Systems, May/June 1999:46{53,1999.
[4] R. Laddaga. Creating robust software through self-adaptation. IEEE Intelligent Systems, May/June 1999:26{29,1999.
[5] Karsai G., Bloor G., Doyle J.: "Automating Human Based Negotiation Processes for Autonomic Logistics", Proceedings of the IEEE Aerospace 2000, Big Sky, MT, March, 2000.
[6] Sprinkle J., van Buskirk C., Karsai G.: "Analysis and Representation of Clauses in Satisfiability of Constraints", ISIS Technical Report, August 2001.
[7] Wellman, M. P., and Doyle, J. 1992. Modular Utility Representation for Decision-Theoretic Planning. In *Proceedings of the First International Conference on AI Planning Systems,* 236–242. San Francisco, Calif.: Morgan Kaufmann.
[8] Wellman, M. P., and Doyle, J. 1991. Preferential Semantics for Goals. In *Proceedings of the Ninth National Conference on Artificial Intelligence,* eds. T. Dean and K. McKeown, 698–703. Menlo Park, Calif.: AAAI Press.