Software Complexity and Management for Real-Time Systems

Ankur Agarwal, A. S. Pandya, and Young-Uhg Lho, *Member, KIMICS*

Abstract—The discipline of software performance is very broad; it influences all aspects of the software development lifecycle, including architecture, design, deployment, integration, management, evolution and servicing. Thus, the complexity of software is an important aspect of development and maintenance activities. Much research has been dedicated to defining different software measures that capture what software complexity is. In most cases, the description of complexity is given to humans in forms of numbers. These quantitative measures reflect human-seen complexity with different levels of success. Software complexity growth has been recognized to be beyond human control. In this paper, we have focused our discussion on the increasing software complexity and the issue with the problems being faced in managing this complexity. This increasing complexity in turn affects the software productivity, which is declining with increase in its complexity.

Index Terms—Componentization, Real-Time System, Software Complexity, Software Management.

I. INTRODUCTION

The definition of software complexity is multi-dimensional and issues of concern are highly dependent on the domain. For example, the software complexity issues at the mathematics level [1] are different from the issues at the computer science level. Figure 1 depicts some issues related to the software complexity in different domain. The complexity of the embedded systems is increasing with the user requirements [3] and therefore the market demands contributing to the emergence of enhanced applications. These applications are increasingly more dependent on the computer architecture and hardware that have timing requirements. The complexity of these applications and therefore the system is further amplified by the non-functional considerations such as dynamically changing environment and their real-time requirements. Understanding and estimating the complexity in Real-time Systems can lead us to build more reliable and robust systems.

Manuscript received Feburary 20, 2006.

Ankur Agarwal and A. S. Pandya are with the Dept. of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL-33431, USA.

Young-Uhg Lho is with the Dept. of Computer Education, Silla University, Busan, 617-736, Korea (Tel: +82-51-309-5570, Email: yulho@silla.ac.kr)

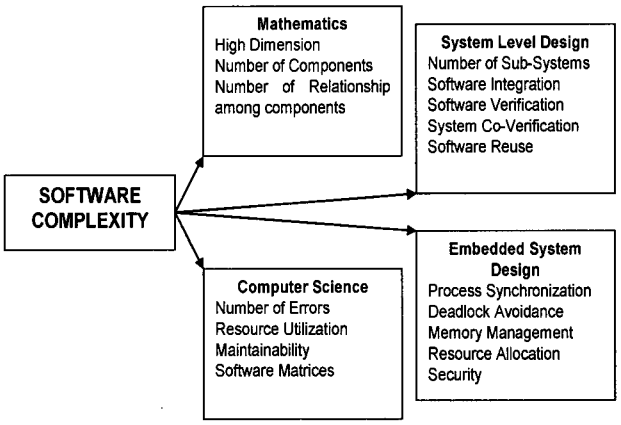

Fig. 1 Software Complexity Issues in Different Domain.

Some of them include but not limited to communications transportation, medicine, energy, finance, and defense; all increasingly involve processes and activities that require precise observance of timing constraints [5]. Coordination and synchronization of many different processes and activities are always the important issues for the Real-Time Software. As a result, real-time and embedded software that must observe timing constraints is experiencing explosive growth. On the other side, there is a conspicuous lack of effective methodology and tools for verifying timing properties of software, despite an increasingly pressing need for such methodology and tools. Examples of proposed formal methods for real-time systems, include, among others, timed and hybrid automata [4,5]; timed transition systems/temporal logic [6-8]; timed Petri-nets [9], theorem proving techniques using PVS to analyze real-time protocols and algorithms [10]. The most industrialized of the formal methods is model checking [3]. While the formal methods have been used successfully for verifying hardware designs, the use of formal methods to verify actual software code is rare [10], [11], [12], [13], and [14]. Further, there has been a negligible or a rare use of the formal methods for verifying timing properties of actual large scale real-time software implementations. The problem is the complexity of software, especially non-terminating concurrent software, and the complexity of such software's possible timing behaviors. The specific problems for software inspection and software verification are being posed due to the timing requirements and constraints in concurrent, real-time software. The basic premise of any software inspection or verification method is that it should be able to cover all the possible cases of the software's behavior. Taking into account timing parameters and constraints adds a whole new dimension to the solution space. The number of different possible interleaving

and/or concurrent execution sequences of multiple threads of software processes and activities that need to be considered when timing constraints are included may increase exponentially as a function of the size of the software and may result in an explosion of the number of different cases that needs to be examined. This may make it exceedingly difficult to use verification and inspection techniques that systematically examine all the possible cases of software behavior.

II. SOFTWARE CHANGES

The growth in complexity of software has tracked the growth in hardware processing power, which in turn has been motivated by the Moore's law. The complexity of applications has increased dramatically over the last five years, making it harder to be more productive. For example, in the telecommunications industry, a software developer we work with focused on newer technology, and their software complexity increased by a factor of 10. The new and innovative steps in the programming models, starting with assembly languages, to compiled high level languages, on to structured programming, and to object oriented programming, to component-oriented programming such as in Java 2 Enterprise Edition (J2EE) [12], to integration and co-ordination software [13], [14], and now to service oriented architectures [15] is the result of the evolution of faster and more capable processors. But without the application of performance engineering focus and technologies, these capabilities would still not have been considered ready for commercial use. For example, the first 3–4 years of the Java [16] programming language and runtime were completely dominated by a focus on performance and performance improvement; new technologies such as just-in-time compilers and high performance garbage collection were identified as essential to make the technology a satisfactory base for business computing [17].

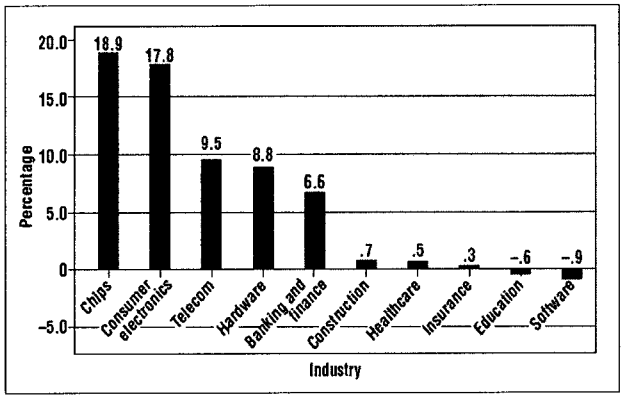

Fig. 2 The Annual Productivity Growth Rates for Industry from 1998 to 2003 (Data Taken from economy.com).

Nothing has changed in this regard. The performance discipline remains critical to the effective enablement of new architectural ideas. However, we are reaching a point where some profound new ideas are needed. As a direct consequence of the development of advanced programming models, the complexity of our software stack is experiencing extraordinary growth.

Large software systems are critical assets, which provide a competitive advantage to their owners. Software systems need to evolve gracefully to fulfill customers' changing needs and requirements otherwise, they will fail [18]. To ensure such graceful evolution of software systems, developers need to reduce and control the complexity associated with software systems. Figure 2 shows the annual productivity growth rates for the industry from year 1998 to 2003 depicting that there is a drop in the productivity associated with the software primarily due to increasing complexity for the same.

III. SOFTWARE ISSUES

As we have discussed so far that, the difficulty of managing today's computing systems goes well beyond the administration of individual software environments. The need to integrate several heterogeneous environments into corporate-wide computing systems, and to extend that beyond company boundaries into the Internet, introduces new levels of complexity. Computing systems' complexity appears to be approaching the limits of human capability, yet the march toward increased interconnectivity and integration rushes ahead unabated. Programming language innovations have extended the size and complexity of systems that architects can design, but relying solely on further innovations in programming methods will not get us through the present complexity crisis. Thus the first step to resolve this problem is to recognize the areas, which contributes to the increasing complexity of the software. Some of these particular events are discussed in this section that includes: Non-deterministic behavior due to External events in the environment and internal events in a computer system may happen non-deterministically at any time. This effect is found to be even worse in case of the mushrooming real-time systems. Secondly, due to insufficient knowledge about the runtime resource usage pattern before runtime leads to insufficient accuracy and unpredictable timing properties that can be verified. Thirdly due to the higher precision of the timing properties depends upon the more detailed the hardware/software characteristics that are taken into account. If the software has a high complexity, then this may increase the size of the state space that needs to be examined to the extent that it is impossible to do so within practical time/memory limitations. Forth in the list includes the fixed priorities are used to deal with every kind of requirement. The fixed priority assignments often conflict with other application requirements. In practice, task priorities are rarely application requirements, but are used instead as the primary means for meeting the timing constraints. These priorities frequently change, which greatly complicates the timing analysis. Lastly, due to the task blocking this is used to handle concurrent resource contention, which, in addition to making the timing unpredictable, may result in deadlocks.

IV. COMPLEXITY MANAGEMENT TECHNIQUES

Software development is recognized as a people-intensive and continually changing field. As systems become more interconnected and diverse, architects are less able to anticipate and design interactions among components, leaving such issues to be dealt with at runtime. Soon systems will become too massive and complex for even the most skilled system integrators to install, configure, optimize, maintain, and merge. In addition, there will be no way to make timely, decisive responses to the rapid stream of changing and conflicting demands. Programmers will need some methodology and new tools that help them acquire and represent policies—high-level specifications of goals and constraints typically represented as rules or utility functions—and map them onto lower-level actions. This methodology will also be responsible for building elements that can establish, monitor, and enforce agreements. Thus, the researchers have already established objective and quantitative data, relationships, and predictive models that help software developers in avoiding predictable pitfalls and improving their ability to predict and controlling efficient software projects. In this section we will discuss some techniques which are helpful for the Software Defect Reduction and complexity management.

A. Componentization

A key driver in software, with intriguing relevance to performance, is componentization–the ability to configure, deploy and operate software in smaller units that fit the needs of a particular environment. One of the most effective ways might to achieve this might be to get out of the dilemma of social dependency on uncontrollable complexity is to reorganize software into abstraction hierarchies so that we can manage software at higher levels of abstraction to cut down the complexity and then generate the lower levels automatically.

B. Test First

Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase. Ideally, we would like to monitor and control complexity as early as possible instead of detecting it in the code after it occurred. For example, it would be more beneficial to determine that the customer's requirements are excessively complex or that the development process is excessively complex, as early detection will help in better planning and preparation. This reveals that good architectural practices and spending more time during the specification and requirement phase can significantly reduce the cost-escalation factor even for large critical systems. Such practices reduce the cost of most fixes by confining them to small, well-encapsulated modules.

C. Pareto Rule

About 80 percent of avoidable rework comes from 20 percent of the defects. About 80 percent of the defects come from 20 percent of the modules, and about half the modules are defect free. Two major sources of avoidable rework involve hastily specified requirements and nominal-case design and development, in which late accommodation of off-nominal requirements causes major architecture, design, and code breakage. Data mining might prove to be a good option in detecting these modules, which are more prone to the faults [19], [20]. Several tool and techniques have been introduced for software verification with data mining but only few of them are often employed in the industrial practices.

D. Separation of Concern

For the real-time systems with large scale, complex, non-terminating concurrent real-time software, overly complex interactions between system components that execute in parallel is a main factor in creating an exponential blowup in complexity. The pre-runtime scheduling approach may be employed in such cases, which can aim at effectively reducing the complexity by structuring real-time software as a set of cooperating sequential processes and imposing strong restrictions on the interactions between the processes. The concept of cooperating sequential processes provides a powerful and well-understood abstraction for structuring concurrent software. With the number of different timing behaviors significantly reduced through pre-runtime scheduling, there should be much less of a need to use other less well-understood forms of approximation or abstraction.

E. Use Case Based Verification

Perspective-based reviews catch 35percent more defects than non-directed reviews. A scenario-based reading technique offers a set of formal procedures for defect detection based on varying perspectives. The union of several perspectives into a single inspection offers broad yet focused coverage of the document being reviewed. This approach seeks to generate focused techniques aimed at specific defect-detection goals by taking advantage of an organization's existing defect history. Scenario-based reading techniques have been applied in requirements, object-oriented design, and user interface inspections. Improvements in fault detection rates vary from 15 to 50 percent. Further, focused reading techniques facilitate training of inexperienced personnel, improve communication about the process, and foster continuous improvement.

F. Autonomic Intelligent Systems

The use of autonomic systems is emerging area for the software complexity management [21]. The idea is to use the power of the computer itself to help with management, tuning and repair. The goal is a system that is self-correcting, self-healing, self-optimizing and self-provisioning [22], [23]. In a sense, this is an acknowledgment that the software stack is too complicated; and that the resource management is too hard for users or even for programmers to handle. Fully autonomous systems represent an incredibly ambitious goal, and the path ahead offers extraordinary opportunity for the performance discipline.

V. CONCLUSION

We have reviewed several trends in software and identified opportunities for focus in performance work. Indeed, from the perspective of added flexibilities, these trends will contribute to software complexity growth. Therefore, we must find better ways to deal with it. The business week article suggests that due to the lack of industry wide standard definition for software productivity, software applications' increasing complexity, and the need for more formalized processes in the industry as a whole is resulting in the declining software productivity. Thus, some worldwide work should be dedicated in the area defining the software complexity and its productivity. Industry also needs to adopt the formal ways for the software verification along with the new emerging techniques in the data mining. The problem of software management and productivity will play a key role in the future technology.

REFERENCES

[1] E. Coskun, M. Grabowski, "Complexity in embedded intelligent real time systems", *ACM Proceeding of ACM*, the 20th international conference on Information Systems, 1999, Pages: 434-439.

[2] S elic, B., Motus, L., "Using Models in Real-Time Software Design" *IEEE Magazine of Control Systems*, Volume: 23, Issue: 3, June 2003 Pages: 31 – 42.

[3] D el Bianco, V., Lavazza, L., Mauri, M., "Model checking UML specifications of real time software" *Proceedings of IEEE International Conference on Engineering of Complex Computer Systems*, 2002. Eighth, 2-4 Dec. 2002 Pages: 203 – 212.

[4] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho, "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems," *Hybrid Systems*, R.L. Grossman, A. Nerode, A. Ravn, and H. Rischel, eds., 1993.

[5] R. Alur and D.L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science*, vol. 126, pp. 183-235, 1994.

[6] P. Bellini, R. Mattolini, and P. Nesi, "Temporal Logics for Real-Time System Specification," *ACM Computing Surveys*, vol. 32, no. 1, pp. 12-42, 2000.

[7] Barreto, R., Maciel, P., Cavalcante, S., "A Modeling Methodology and Pre-Run Time Scheduling for embedded real-time software" *IEEE Proceedings of 15th Symposium on Computer Architecture and High Performance Computing*, 2003. 10-12 Nov. 2003, Pages: 72-79.

[8] T. Henzinger, Z. Manna, and A. Pneuli, "Temporal Proof Methodologies for Real-Time Systems," *Proc. 18th ACM Symp. Principles of Programming Languages*, pp. 353-366, 1991.

[9] F. Jahanian and A. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Trans. Software Eng.*, vol. 12, pp. 890-904, 1986.

[10] B. Dutertre, "Formal Analysis of the Priority Ceiling Protocol," Proc. *IEEE Real-Time Systems Symp.*, pp. 151-160, Nov. 2000.

[11] E.M. Clarke and J.M. Wing, "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, Dec. 1996.

[12] D. Dill and J. Rushby, "Acceptance of Formal Methods: Lessons from Hardware Design," *Computer*, vol. 29, pp. 23-24, 1996.

[13] C. Heitmeyer, "On the Need for Practical Formal Methods," *Proc Fifth Int'l Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems*, 1998.

[14] P. Cousot and R. Cousot, "Verification of Embedded Software: Problems and Perspectives," *Proc. Int'l Workshop Embedded Software (EMSOFT 2001)*, 2001.

[15] *Software Fundamentals*: Collected Papers by David L. Parnas. D.M. Hoffman and D.M. Weiss, eds., Addison-Wesley, 2001.

[16] WebSphere Business Integration Server. www.ibm.com/software/integration/wbiserver/

[17] Business Process Execution Language for Web Services, Version 1.1: BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems, May 2003. www-106.ibm.com/developerworks/library/ws-bpel/

[18] Simple Object Access Protocol (SOAP) 1.1: W3C, May 2000. www.w3.org /TR/SOAP/

[19] M. M. Lehman, J. F. Ramil, P. D.Wernick, D. E. Perry, and W. M. Turski. Metrics and Laws of Software Evolution The Nineties View. *In Fourth International Software Metrics Symposium (Metrics97)*, Albuquerque, NM, 1997.

[20] Gorodetsky, V.; Karsaeyv, O.; Samoilov, V, "Software tool for agent-based distributed data mining" *Integration of Knowledge Intensive Multi-Agent Systems*, 2003. IEEE International Conference on Data Mining, 30 Sept.-4 Oct. 2003, Pages:710 – 715

[21] J. M. Bieman, A. A. Andrews, and H. J. Yang. Understanding change-proneness in OO software through visualization. *In Proc. 11th International Workshop on Program Comprehension*, pages 44–53, Portland, Oregon, May 2003.

[22] Autonomic computing, IBM's perspective on the state of information technology. October 2001. www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf

[23] www.ibm.com/servers/autonomic

[24] IBM Corporation: 'Autonomic computing architecture: a blueprint for managing complex computing environments'. October 2002. www.ibm.com/autonomic/pdfs/ACwhitepaper1022.pdf

Ankur Agarwal
He is a Visiting Instructor and a Ph.D. student at the Computer Science and Engineering Department, Florida Atlantic University. He pursued his MS in computer engineering from Florida Atlantic University in year 2003. He also holds two post graduate diplomas in VLSI design and real-time embedded system design. He has earned his bachelor of engineering from Pune University, India in year 2000. His research areas are concurrency modeling, system level design, network-on-chip, real-time-operating system and VLSI design.

A. S. Pandya
He is a professor at the Computer Science and Engineering Department, Florida Atlantic University. He received his undergraduate education at the Indian Institute of Technology, Bombay. He earned his M.S. and Ph.D. in Computer Science from the Syracuse University, New York. He has worked as a visiting Professor in various countries including Japan, Korea, India, etc. His research areas are VLSI implementable algorithms, Applications of AI and Image analysis in Medicine, Financial Forecasting using Neural Networks.

YoungUhg Lho
(Corresponding author)
He received the B.S., M.S. and Ph.D degrees in Dept. of Computer Science from Busan National University, Busan, Korea, in 1985, 1989, and 1998, respectively. From 1989~1996, he was with the Electronics and Telecommunications Research Institute(ETRI), Daejeon, Korea. Since 1996, he has been with the Dept. of Computer Education, Silla University, where he is now Associate Professor. His research interests include real-time system, ubiquitous computing, embedded system, multimedia system, parallel and distributed system, intelligent system and computer education.