Preface

This volume emanated from a workshop on Self-Adaptive Software held at Lady Margaret Hall, Oxford in April, 2000. The aim of the workshop was to bring together researchers working on all aspects of self-adaptive software in order to assess the state of the field, map out the research areas encompassed by the field, develop a research agenda, and enumerate open issues.

The papers presented at the workshop were in some cases revised after the workshop.

The first paper in the collection: "Introduction: The First International Workshop on Self-Adaptive Software" provides a brief overview of self-adaptive software and a description of the layout of the volume.

November 2000   Paul Robertson

Organizers

Paul Robertson   (Oxford University)
Janos Sztipanovits   (DARPA, Vanderbilt University)

Program Committee

Paul Robertson   (Oxford University)
Janos Sztipanovits   (Vanderbilt University)
Howie Shrobe   (MIT)
Robert Laddaga   (MIT)

Introduction:
The First International Workshop
on Self-Adaptive Software

Paul Robertson, Robert Laddaga, and Howie Shrobe

MIT

1 Introduction

This collection of papers was presented at the international workshop on self adaptive software (IWSAS2000) held at Lady Margaret Hall at the University of Oxford between April 17th and April 19th 2000. They represent the state of the art in a new evolving field of research that attempts to build software systems that are more robust to unpredictable and sometimes hostile environments than we have managed to produce by conventional means.

As we try to solve increasingly difficult problems with computer software we develop new architectures and structures to help support our aggressive endeavors. It is in this light that the need for and the potential benefits of "Self-Adaptive Software" have become apparent.

2 What is Self Adaptive Software

The traditional view of computation is one in which a computational process executes within a fixed and well defined environment. Traditionally computers have existed in isolated environments with minimal contact to the outside world. On occasions the real world has leaked into the equation. Input and output is an example of such leakage. Programs that (say) have to read characters typed by a human operator are operating in an environment that is not well defined or predictable. Writing robust programs to perform input is notably hard. When the environment becomes even slightly unpredictable it becomes necessary for the programmer to carefully consider all possibilities and program them in to the code. Programs that are coded only for what is wanted as input rather than the full range of what might be presented are brittle and such program often fail badly in practice.

If writing programs that robustly perform input is hard it is only the tip of the iceberg. Increasingly we are taking computation away from the tightly controlled environments that have characterized the early development of computer science – away from the machine rooms and trained expert users – and putting them into situations where the environment is unconstrained. Computers are now often connected to a network which provides many additional forms of interaction with the outside world and the incredible reduction in both size and cost of computers

now allows us to embed processors throughout our homes, cars, and workplaces in ever widening and inventive roles.

Where we have done well with these embedded applications of computation have been precisely those areas where the environment can still be constrained. For example Digital Signal Processors (DSP's) are embedded into a wide array of devices (such as cell phones) where they can perform simple repetitive operations on an input signal that may be voice or image. Where the range of unpredictability is limited to the sequence of button presses on a cellular telephone keyboard or a microwave oven the computation can be adequately addressed using traditional techniques.

Gradually our ambitions are drawing us to use computation in situations where the unpredictable nature of the environment poses a formidable challenge. Obvious examples include robot navigation in unconstrained environments, the interpretation of human speech in noisy environments, the interpretation of visual scenes by mobile robots and so on. In the cited examples there is a significant domain specific component to the problem – such as visual feature extraction – that is unique to each domain but there is also a shared problem. The shared problem is that of writing programs that operate robustly in complex environments.

It is already the case that embedded systems represent the majority of our use of computers and it will never be less true than it is today. Most current embedded systems involve rather simple processors with highly limited memory and computational power. Until recently the limitations of what these embedded systems could do has constrained them to being rather simple and manageable. That picture however is changing rapidly. The new breed of embedded systems can expect to have enough computational power and memory for them to get themselves into some serious trouble. Over time these complex embedded systems will come to dominate.

While the model of computation as operating within a fixed and well defined environment has served us well during the early phases of development of computer science it now seems likely that in order to move forward we must develop a less restrictive model for computation and along with the new model, new methodologies and tools.

In a sense the kind of computation that we developed in the last century was a peculiar and even unnatural one. Nature is full of examples of computation. Most of these examples like the orientation of leaves towards the sun and the opening and closing of flowers in response to changes in light and temperature are fixed computation but many involve what is more recognizable as a programmable capability – such as the ability to learn exhibited by many animals. Whatever degree of learning these computational devices may have virtually all of them are involved with direct interaction with the real world. These natural examples of computation work in the presence of complex and often hostile environments. Indeed they exist precisely because the environment is unpredictable and hostile.

At a fundamental level computation is not different in these unconstrained environment situations. We can still describe them in terms of Turing machines

and in principle we can program them using traditional techniques by carefully considering at every turn the full range of possibilities that the environment can present and programming for each case. On the other hand the traditional approach arguably missed the point. The notion of a computation halting is uninteresting in many of these applications and while it is theoretically possible to consider every possibility when programming a system it makes for unmanageably complex software that challenges our ability to build and test them.

When the environment is constrained we can develop an algorithm traditionally, whose complexity is a function of the problem being solved. If the environment must also be considered, again using traditional technology, the complexity of the program must mirror the complexity of the environment, because the program must consider every possibility... If a program's complexity is a linear or worse function of the complexity of the environment, then we are clearly in trouble because the world is a complex place.

Computation may be a much wider concept than we have historically believed it to be. In the future computation that operates in tightly constrained environments will represent a tiny fraction of computation as a whole. The Computer Science that we have developed to date may turn out to be a very special case of computation that occurs only when the environment is very tightly constrained.

For the time being those building systems involving sensors and effectors and those developing systems that must navigate robots, missiles, and land vehicles through complex and often hostile environments share a need for new approaches to building robust applications in these domains.

This need for a new approach was recognized by DARPA in 1997 when it published its Broad Agency Announcement (BAA) for proposals in the area of "Self-Adaptive Software". The BAA [14] describes Self Adaptive Software as follows:

"Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible."

2.1   Benefits of a Self Adaptive Approach

The principle idea behind the self adaptive approach to achieving robust performance is that it is easier and often more efficient to postpone certain decisions until their need is evident from the circumstances. It is often easier to make a program that monitors its performance and recovers from errors than it is to make a program that goes to great lengths to avoid making any errors. This notion is already familiar to us in our use of errors and exceptions in languages like Common Lisp [24] and Java. Exceptions allow simple code to be written that expects the environment to be well behaved. The code must also include checks that everything is as expected. If things are not as expected an exception is raised. Elsewhere an exception handler contains the code that decides how the exception should be handled. For example if the code was trying to read a number and a non numeric character was detected a 'non numeric character in number' exception may be raised. The handler could then clear the input buffer,

caution the user against using non numeric characters and retry the read. In this example the program has a 'model' of what it expects (a sequence of numeric digits). The program checks for differences between what it is getting as input and the model and when such differences occur the program tries to deal with them robustly–in this case by issuing a warning and trying again.

While the exception mechanism is invaluable for a certain class of problems and is suggestive of a general approach it does not easily extend to the full range of problems that can occur in complex systems. When a robot encounters difficulty in understanding a visual scene it could be for a variety of reasons: a sensor may be functioning incorrectly or have failed completely; the lighting may have changed so as to require a different set of algorithms; the contents of the scene may have changed requiring the robot to update its model of the environment; and so on. In general the environment cannot be given a warning and asked to try again! The program must deal with what it gets as best it can. In these situations when a sensor fails or an algorithm doesn't work or some other assumption about the environment turns out to be untrue the solution will often require reconstituting the program to deal with the actual environment rather than having enough complexity to deal with all possible environments hidden within exception handlers.

Self adaptive software therefore involves having suitable models of the environment and of the running program, the ability to detect differences between the models and the actual runtime environment, the ability to diagnose the nature of the disparity, and the ability to re-synthesize or modify the running program (or its models) so that the program can operate robustly.

The notion of programs that can manipulate their own representation is not new and has been a key characteristic of Lisp [24,12] since its inception although the full power of that capability has not often been used. The idea of programs that reason about their own behavior and modify their own semantics is well developed. Programming languages that support self reasoning and self modification are called 'reflective' a term coined by Brian Smith in his original development of the idea of reflective programs [4,5]. Reflection has been around long enough for efficient implementation methodologies to be developed [7,2,11, 8] and some level of reflective capability is now available in CLOS [12] and Java. Reflection makes it possible (and convenient) to write programs that can reason about the state of computation and make changes to the semantics of the running program. Self adaptive software can clearly benefit from such a capability but there is more to self adaptive software than reflection. Self adaptive software must have a model of what the computation is trying to achieve in order to have something against which to compare the current computation. Having determined some difference between the intended program state and the actual one a self adaptive program must be able to automatically adjust the semantics of the program so that the difference is reduced. Reflection provides the tools for writing such programs but it doesn't provide any guidance in how it should be done.

Managing complexity is a key goal of self adaptive software. If a program must match the complexity of the environment in its own structure it will be very complex indeed! Somehow we need to be able to write software that is less complex than the environment in which it is operating yet operate robustly. By keeping the program code separate from the code the diagnosis and synthesis code the core code can remain simple and the synthesis can be performed by a purpose built synthesis engine. Such an architecture makes writing the core code much less of a tangled web. In essence the total runtime program may be a sequence of different programs each one individually simple. The complexity of the program as a whole is moved into the architectural mechanism that maintains models, checks models against performance, and re-synthesizes new programs.

There are many advantages to this approach beyond making it possible to write programs that operate in complex environments. Correctness of our programs may be easier to establish whether by testing, proof, or other means because at any point the operating program should be simpler than a program that considered every possibility would be. To some extent correctness testing may be guaranteed by the synthesis engine or may be checked automatically because there is a model of the program against which it may be tested. Greater efficiency may also be achieved through the self adaptive software approach. Greater efficiency occurs when a crude but fast program that may not always work is substituted for a complex but slow program that tries to guarantee that it always works. By making decisions at run time when much more is known about the actual environment than could have been known when the program was originally designed it is possible to select more suitable procedures for achieving the intended goal. this can result in a more direct solution with less failure and less backtracking. Having better information about the environment allows better decisions to be made which can result in more efficient and successful execution.

There are also concerns and difficulties evident in the approach. A program that changes itself at runtime may in some sense be a less well understood entity than a program that remains unchanging during its execution. The notion that it may be more robust than its fixed counterpart should make us feel more comfortable about its behavior but the notion that the program changes at runtime is unsettling to people responsible for building complex missile delivery systems (for example). How can we trust a smart missile that may reprogram itself in mid flight? There are (at least) two answers to this. The first is that if the program fails in mid flight because it was unable to deal with its complex environment it may do a very bad thing in its failure mode so the self adaptive version that is less brittle should be less of a concern. The second is that we must develop a better understanding of self adaptive software and new approaches to software testing, debugging, and semantics that allow us to be confident about the systems that we deploy either through a proof theoretic framework or through a testing regime.

Today self adaptive software draws upon a great many technologies that may be used as building blocks. Reflection and exception handling mechanisms have

already been mentioned but there are a great many other candidate contributing technologies including model based computing, theorem provers, models for reasoning about uncertainty, and agent based systems to name a few. The papers in this collection begin to address many of the issues described above as well and explore how many of these technologies can be harnessed to realize the goals of self adaptive software.

3   About the Papers

Self adaptive software is a relative new idea and the approach involves taking a significant departure from established software development methodologies and potentially requiring a rethinking of some basic concepts in computer science. Self adaptive software will be achieved by taking small steps from what we currently know. Some people are attempting to build self adaptive software solutions to solve existing problems, some are experimenting with existing technologies to see how they can be used to achieve self adaptive software, some are working with systems that interpret the world through sensors, and some are directly studying aspects of self adaptive software ideas in order to extend our understanding of their semantics. The papers in this collection provide examples of all of these areas of activity.

3.1   Understanding Self Adaptive Software

Robert Laddaga's paper on 'Active Software' [15] provides a good definition of Self adaptive software and a survey of related concepts and technologies.

One of the more ambitious self adaptive software projects that brings together a wide range of technologies is being being conducted at the University of Massachusetts at Amherst. Two papers from the University of Amherst project appear in this collection [19,6]. Osterweil et. al. describe a process of perpetual testing in 'Continuous Self Evaluation for the Self-Improvement of Software' as a means of achieving self improving software.

The environment in natural systems as well as artificial ones is not only complex but in many instances it can be hostile. Information survivability attempts to deal with systems that come under attack from outside agents. Shrobe and Doyle [23] consider how self adaptive software can provide software services in an environment that is actively hostile. Rather than considering issues of information survivability as a binary affair in which the system is either compromised (and useless) or not compromised (and useable) they consider how survivability can be achieved by self-testing to detect that the system has been compromised in various ways and then adapting so as to be able to provide the best level of service possible given the compromised state if the system.

There is a bone fide need for formal methods in a number of places in building self adaptive software. Formal methods may be involved in the automatic synthesis of the program when adaptation is required. Formal methods may also be useful in understanding the range of behaviors that a self adaptive program

may exhibit. Dusko Pavlovic's paper 'Towards semantics of self adaptive software' [20] provides a first attempt at building a theoretical basis for self adaptive software.

3.2   Self-Adaptive Software Compared to Control Systems

The straightforward characterization of self adaptive software as continual runtime comparison of performance against a preset goal and subsequent adjustment of the program so that performance is brought back towards the goal is very similar in structure to a control system. The task of building self adaptive systems may be much more like building a control system than it is to conventional software development. We are trying to develop new methodologies for building software that is robust to changes in the environment within certain limits and we have observed that systems characterized as self adaptive are more control like in structure than conventional program like. Since control systems and their design are well understood it is hoped that some ideas can be borrowed from control system design and adapted to self adaptive software. Two papers in this collection deal with attempting to understand self adaptive software in terms of control systems and the subsequent borrowing of ideas from control systems. Alex Meng's paper 'On Evaluating Self Adaptive Software' [16] makes explicit the relationship between control systems and self adaptive software and discusses terminology and issues surrounding viewing software as a control system. The Kokar et. al. paper entitled 'Mapping an Application to a Control Architecture: Specification of the Problem' takes a radar-based target tracking problem and shows how it can be mapped onto a control system architecture [17].

3.3   Technologies for Self Adaptive Software

When a system is reconfigured at runtime such as by the replacement of one set of sensors with another set undesirable transient effects may occur at the point of switching. Gyula et. al. [9] investigate the management of such effects in 'Transient Management in Reconfigurable Systems'.

A key notion of self adaptive software is that the software contains some embedded account (or model) of its intention and of its environment. In 'Model-Integrated Embedded Systems' Bakey et. al. [1] describe a model based generative technology they call 'Model-Integrated Computing' and show how the technology is applicable to self adaptive software.

In 'Coordination of View Maintenance Policy Adaptation Decisions: A Negotiation-Based Reasoning Approach' Bose and Matthews [3] develop a negotiation based reasoning model for adapting view maintenance policies to meet changes in quality of service (QoS) needs.

While much attention has been given to robotic, sensor based, and embedded applications for self adaptive software there is a role to be played in the network computing environment. Network computing provides all of the attributes of a naturally complex environment within a man made network. In 'Dynamic Self

Adaptation in Distributed Systems' Ben-Shaul [10] describes two implemented architectures that provide support for self adaptive software in Java.

Since the need for self adaptive software derives from running applications in an unpredictable environment many of the applications of self adaptive software involve real-time issues. One paper in the collection specifically addresses the real-time issue. Musliner [18] describes in 'Imposing Real-Time Constraints on Self Adaptive Controller Synthesis' a system that can adapt by dynamically re-synthesizing new controllers. Musliner shows how the synthesis process can be managed so that reconfiguration can meet real-time deadlines.

3.4   Systems that Interpret Their Environment through Sensors

The remaining papers describe implemented self adaptive software systems that involve sensors.

In 'Software Mode Changes for Continuous Motion Tracking' Grupen et. al. [6] describe a self adaptive motion tracking system that has redundant sensors and the ability to adjust which sensors are best suited to tracking.

In 'Port-Based Adaptable Agent Architecture' Khosla et. al. [13] describe an architecture that supports self adaptive software through an agent architecture. They go on to demonstrate the architecture on a multi-robot mapping problem.

In 'An architecture for self adaptation and its application to aerial image understanding' Robertson [22] describes a self adaptive architecture that is tailored to image understanding. The systems learns models from expert annotations of images from a corpus and then uses those models to monitor performance of the system on other images and to re-synthesize the program code when necessary in order to produce good interpretations of the images.

In 'Self Adaptive Multi-Sensor Systems' Reece [21] describes a system that utilizes redundant sensors for image interpretation. Sensor performance is learned from test sets and qualitative reasoning (QR) models of the imaged environment's physics allows a theorem prover to synthesize interpretation code.

3.5   Conclusion

The workshop brought together a wide range of researchers involved in the projects described above and provided an opportunity for us to collect our thoughts and experiences and access how far we have come with self-adaptive software, what problems remain to be solved, and how to characterize self-adaptive systems in terms of a set of capabilities.

The final paper summarizes some of these discussions and provides a taxonomy of capabilities and technologies and suggests areas where more research is needed.

References

1. Arpad Bakay Akos Ledeczi and Miklos Maroti. Model-integrated embedded systems. In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, *Self-Adaptive Software*. Spriger-Verlag, 2000.
2. Alan Bawden. Reification without evaluation. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 342–351, 1988.
3. Prasanta Bose and Mark G. Matthews. Coordination of view maintenance policy adaptation decisions: A negotiation-based reasoning approach. In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, *Self-Adaptive Software*. Spriger-Verlag, 2000.
4. C. Smith Brian. Reflection and semantics in a procedural language. Technical Report 272, MIT Laboratory for Computer Science, January 1982.
5. C. Smith Brian. Reflection and semantics in lisp. In *Proceedings 11th Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah*, pages 23–35, January 1984.
6. Elizeth Araujo Yunlei Yang Gary Holness Zhigang Zhu Barbara Lerner Roderic Grupen Deepak Karuppiah, Patrick Deegan and Edward Riseman. Software mode changes for continuous motion tracking. In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, *Self-Adaptive Software*. Spriger-Verlag, 2000.
7. Jim des Rivieres and C. Smith Brian. The implementation of procedurally reflection languages. In *Proceedings 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas*, pages 331–347, August 1984.
8. G. Daniel G. Kiczales, J. des Rivieres. *The art of the Metaobject Protocol*. MIT Press, 1993.
9. Tam s Kov csh zy Gyula Simon and G bor P celi. Transient management in reconfigurable systems. In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, *Self-Adaptive Software*. Spriger-Verlag, 2000.
10. Ophir Holder Israel Ben-Shaul, Hovav Gazit and Boris Lavva. Transient management in reconfigurable systems. In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, *Self-Adaptive Software*. Spriger-Verlag, 2000.
11. P. Cointe J. Malenfant and C. Dony. Reflection in prototype-based object-oriented programming languages. In *Proceedings of the OOPSLA Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, 1991.
12. S. E. Keene. *Object-Oriented Programming in Common Lisp: A programmer's Guide to CLOS*. Addison-Wesley, 1989.
13. Theodore Q. Pham Kevin R. Dixon and Pradeep K. Khosla. Port-based adaptable agent architecture. In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, *Self-Adaptive Software*. Spriger-Verlag, 2000.
14. Robert Laddaga. Self-adaptive software sol baa 98-12. *http://www.darpa.mil/ito/Solicitations/CBD_9812.html*, 1998.
15. Robert Laddaga. Active software. In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, *Self-Adaptive Software*. Spriger-Verlag, 2000.
16. Alex C. Meng. On evaluating self-adaptive software. In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, *Self-Adaptive Software*. Spriger-Verlag, 2000.
17. Kenneth Baclawski Mieczyslaw M. Kokar, Kevin M. Passino and Jeffrey E. Smith. Mapping an application to a control architecture: Specification of the problem. In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, *Self-Adaptive Software*. Spriger-Verlag, 2000.

18. David J. Musliner. Imposing real-time constraints on self-adaptive controller synthesis. In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, *Self-Adaptive Software*. Spriger-Verlag, 2000.
19. Leon J. Osterweil and Lori A. Clarke. Continuous self-evaluation for the self-improvement of software. In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, *Self-Adaptive Software*. Spriger-Verlag, 2000.
20. Dusko Pavlovic. Towards semantics of self-adaptive software. In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, *Self-Adaptive Software*. Spriger-Verlag, 2000.
21. Steven Reece. Self-adaptive multi-sensor systems. In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, *Self-Adaptive Software*. Spriger-Verlag, 2000.
22. Paul Robertson. An architecture for self-adaptation and its application to aerial image understanding. In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, *Self-Adaptive Software*. Spriger-Verlag, 2000.
23. Howard Shrobe and Jon Doyle. Active trust management for autonomous adaptive survivable systems (atm' for ass's). In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, *Self-Adaptive Software*. Spriger-Verlag, 2000.
24. G. Steele. *Common Lisp: The Language*. Digital Press, 1984.