Adaptation and dependability and their key role in modern software engineering

Vincenzo De Florio[*] and Chris Blondia
University of Antwerp
Department of Mathematics and Computer Science
Performance Analysis of Telecommunication Systems group
Middelheimlaan 1, 2020 Antwerp, Belgium

Interdisciplinary institute for BroadBand Technology
Gaston Crommenlaan 8, 9050 Ghent-Ledeberg, Belgium
Phone: +32 3 2653905, Fax: +32 3 2653777
vincenzo.deflorio@ua.ac.be, chris.blondia@ua.ac.be

Abstract

Current software systems and the environments such systems are meant for require a precise characterization of the available resources and provisions to constantly re-optimize in the face of endogenous and exogenous changes and failures. This paper claims that it is simply not possible today to conceive software design without explicitly addressing adaptability and dependability. As an example we remark how mobile computing technologies call for effective software engineering techniques to design, develop and maintain services that are prepared to continue the distribution of a fixed, agreed-upon quality of service despite of the changes in the location of the client software, performance failures, and the characteristics of the environment. We conclude that novel paradigms are required for software engineering so as to provide effective system structures for adaptive-and-dependable services while keeping the design complexity under control. In this paper we discuss this problem and propose one such structure, also briefly surveying the major milestones in the state of the art in this domain.

Keywords

Software engineering, adaptive software, dependability, software fault-tolerance, application-layer software fault-tolerance.

---

[*] Corresponding author.

INTRODUCTION

Aim of this paper is providing a personal vision on some key prerequisites for current software design, namely dependability and adaptability, and introducing some ideas we are currently developing at the University of Antwerp and the Institute for Broadband Technology. The key message we want to convey is that, today, conceiving software design without the adaptability concern is simply not possible anymore, as this course of action would not match the systems and environments our software is meant for.

The Computer Era may well be thought of as a series of "revolutions": The 19th-century concept of computer was that of a mechanical engine, initially intended to compute, quickly and reliably, tables of polynomials. That which now sounds like a trivial achievement, in times past was but a dream whose fulfillment had puzzled mankind nearly since the beginning of its history. This achievement was to be reached only in 1855 through the design of Babbage and the craft of the Scheutzes. And this marked indeed an actual revolution, that of mechanical computing[1], which may be well summarized by Babbage's famous quote "*I wish calculations had been executed by steam*": For the first time in human history, a tool to perform calculations more quickly and reliably than a human being had been realized.

The 20th Century witnessed a variety of those revolutions, soon to provide a new meaning to the word "computer". One such revolution was brought about by the advent of vacuum tubes, in the Forties. The supercomputer of those times was the ENIAC, with a weight of about 30 tons and an avail-ability quite disappointing when considered out of the historical context the ENIAC had made its appearance in[2]. Clearly there was room for improvements and several further "revolutions" were to be expected[3].

Further revolutions sprang from new concepts, now concisely summarized by terms such as "compiler", "virtual machine", "object orientation" or "service orientation". Each of these concepts brought about a sheer revolution, for it modified the meaning and the use we made of computers thereafter.

Each of these revolutionary steps marks a fundamental leap in the history of computing and of the influence of computers in human society. Each step allowed new services to be conceived while, in turn, these services called for additional requirements and adjustments of our "view" to the concept of "computing."

As a consequence of this, each of those steps also marks the need for new models, both for the computer and for its system software.

We are currently in the middle of another important step in this progression of revolutions, namely the one marked by the spread of personal computing facilities that allow their services moving with us and our goods: It is the so-called "wireless revolution". In this paper we investigate on some of the key requirements for software components meant for wireless or other dynamically changing environments, and describe the main ideas of a prototypic system that we are currently designing as a tool to support the development and execution of mobile services.

The structure of this paper is as follows: In the second section we introduce our target problem. Adaptive systems are discussed in the third section. The fundamental services to be provided by any architecture for adaptability are conjectured in the fourth section. Two examples are given in the following one. A concise survey on adaptability is given in the sixth section, followed by our conclusions in the last section.

SERVICES AND PROGRAMS

In the following we consider a service as a set of manifestations of external events that, if compliant to what agreed upon in a formal specification, can be considered by a watcher as being "correct". Moreover we refer to a program as a physical entity, stored as voltage values in a set of memory cells, which is supposed to drive the production of a service. Goal of software engineering is being able to set up of a robust homomorphism between a service's high-level specification and a low-level computer design (the program).

More formally, we say that for some functions f and g,

$$\text{Service} = f\,(\text{program}),\ \text{program} = g\ (\text{specification}),$$

$$\text{Service} = g \cdot f\,(\text{specification}).$$

Building robust versions of f and g is commonly a very difficult job.

We now concentrate on the range of g (the software set) and for any two systems **a** and **b**, if **a** relies on **b** to provide its service, we say

$$\mathbf{a} \rightarrow \mathbf{b}.$$

We call this relation as the "dependence" between two systems. Clearly it is true that e.g. Service → program, program → CPU, and CPU → memory. Figure 1 provides a possible expansion of the dependence relation. The leaves of such trees provide a designer's view of the "atomic threats" he or she aims to tolerate.

As evident from that picture, dependences call for dependability – a fundamental property to achieve dependable services, which has been defined by Laprie as "*the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers*" (Laprie, 1985). A dependable service is then one that persists even when, for instance, its corresponding program experiences faults—to some agreed upon extent. From the above definitions we propose to consider ***F***-dependable services (resp. ***F***-dependable programs, systems, etc.) as those that persist despite the occurrence of faults as described in ***F, F*** being a set called the fault model.

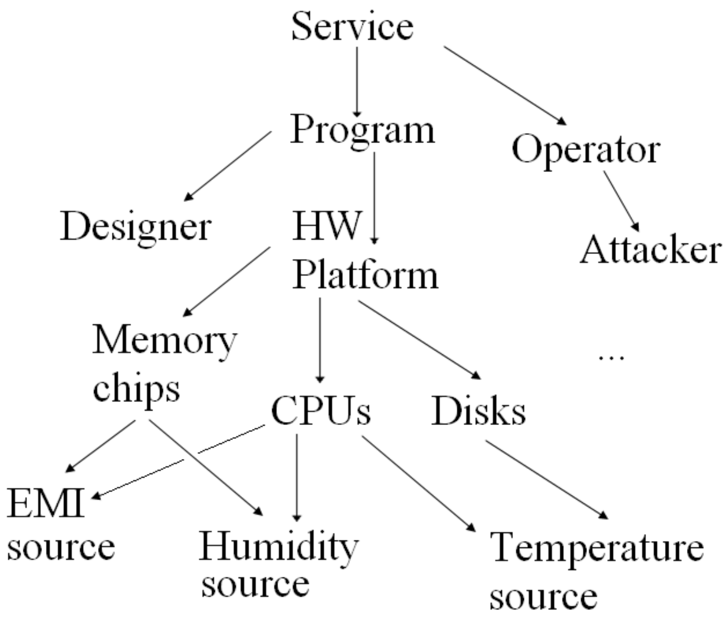


Figure 1: One possible expansion of the dependence relation.

What is ***F*** exactly? It is a set of events whose occurrence we consider as possible and likely to hinder the quality of the service distribution. An important property of ***F*** is that it is a model of an environment ***E*** where the service (or better, its corresponding program) is operating. Clearly an ***F***-dependable service may tolerate faults in ***E'*** and may not those in ***E'' :*** an airborne service may well experience different events than, e.g., one meant to operate in a primary substation (Unipede, 1995).

Obviously the choice of ***F*** is an important aspect to successfully achieving service dependability. Imagine for instance what may happen if our fault model ***F*** matches the wrong environment, or if the target environment changes its characteristics (e.g. a rising of temperature due to a firing). But the key point we remark herein is—what if the service *moves?* In all these cases, lacking provisions to compensate for the changes in the environment and in the corresponding fault model, *a failure may occur,* i.e., an interruption of the service may take place. We summarize the above reasoning with the famous quote by James Horning:

> "*What is the most often overlooked risk in software engineering?*
> *That the environment will do something*
> *the designer never anticipated*" (Horning, 1988).

In other words, if in the early days of modern computing it was – to some extent – acceptable to have lengthy service disruptions due to the occasional faults, being the main task of computers basically that of a fast solver of numerical problems, the criticality associated with many tasks nowadays appointed to computers demands high values for properties such as availability and data integrity.

At the same time and for similar reasons nowadays provisions are required to compensate for the changes occurring "around" a service, i.e., in all the components that the service is dependent on. We call this property **adaptability**.

ADAPTIVE SYSTEMS

As a consequence of the more complex and dynamic environments our software systems are meant for today, as it is the case e.g. for mobile computing, a system's environment has become a variable, which translates in a strong need for adaptability. In what follows we provide our personal vision to adaptive services and the current state in our quest for an effective solution to this problem.

Ideally, we would require our services to be structured as "$\boldsymbol{X}$-dependable services", where $\boldsymbol{X} = f(\boldsymbol{E})$ can change dynamically when e.g., the service is moved to another environment or the environment mutates. $\boldsymbol{X}$ should be indeed considered as an $\boldsymbol{X}(t)$, that is, a system evolving over time: A dynamic system. We consider also as an important prerequisite for an effective crafting of those services that the expression of adaptability and dependability concerns should not increase complexity "too much," so as to avoid possible "bottlenecks of system development" (Lyu, 1998). In other words, whatever solution we may come up with, it must keep complexity bounded and under control.

Our proposal is then to consider an adaptive system as a triple

$$\mathbf{AP} = (\mathbf{F},\mathbf{FT},\mathbf{E}),$$

where **F** expresses the functional concerns (the service), **FT** is some fault tolerance provision to withstand the faults in a fault model $F_{\mathbf{FT}}$ and **E** is a set of environments (to be described later on).

Let us suppose that program (**F**, **FT**) distributes a certain $F_{\mathbf{FT}}$-dependable service and that a family of fault tolerance strategies, $(\mathbf{FT}(k))_{k\square\in K}$ be available for some set of indices K. Furthermore, let us assume that program (**F**, $\mathbf{FT}(j)$) distributes a $F_{\mathbf{FT}(j)}$-dependable service, $j \in K$.

Then we can translate the problem of crafting an adaptive system into that of designing an architecture that senses the environment and, each time the environment changes, it changes program (**F**, $\mathbf{FT}(j)$) into a program (**F**, $\mathbf{FT}(k)$). If the resulting $F_{\mathbf{FT}(k)}$-dependable service matches the new environmental condition, and this is true for a set of environments **E**, then we have realized an adaptable service.

In the following section we briefly sketch the main components of an architecture for adaptable services.

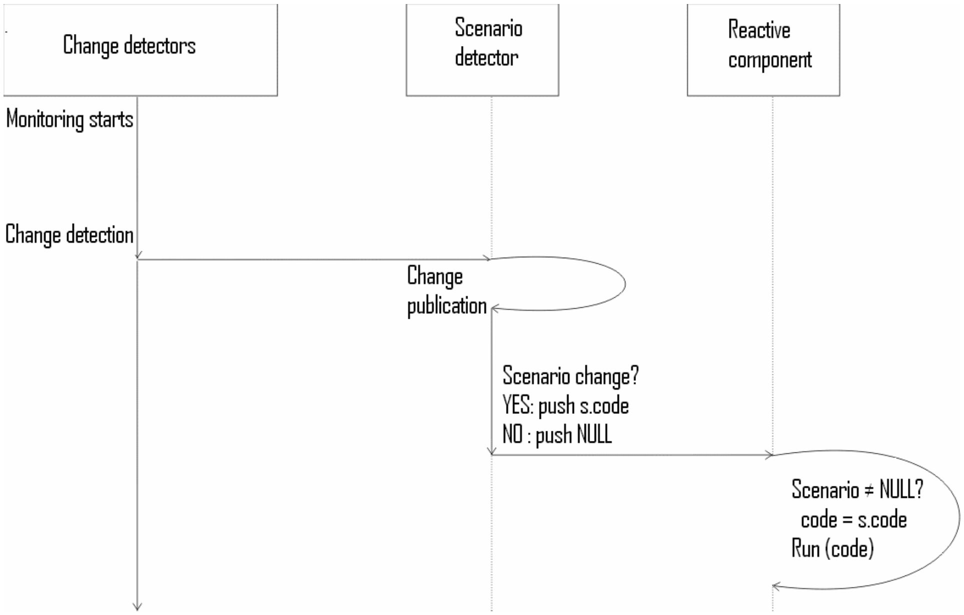


Figure 2: A possible sequence of actions to realize change detection and change reaction.

COMPONENTS OF AN ARCHITECTURE FOR ADAPTABILITY

Our conjecture is that any effective architecture for adaptable applications should be structured on top of two basic services, namely *change detection* and *change reaction*. Our approach is summarized in Figure 2. We assume to have detection components, ranging from simple sensors providing raw data like temperature or heartbeats rate to scenario detectors able to correlate raw data and provide higher level, more structured information about the state of the subjects or properties being monitored. As soon as a relevant change is detected, we publish the event in a reflective shared space and check whether the change brings in a new scenario (e.g., a patient has fallen and is in need). The next phase is executing the actions attached to the detected scenario: this is done in our prototypic architecture by the Reactive component, a virtual machine interpreting a programming language called Ariel (De Florio, 2000).

In practice, we envision the availability of a middleware component to update dynamically the $(\mathbf{FT}(k))_{k\square\in K}$ programs (we call them "recovery codes"), aided in this by a set of "change detectors" monitoring e.g., available energy, network load, local or overall CPU usage, or failures.

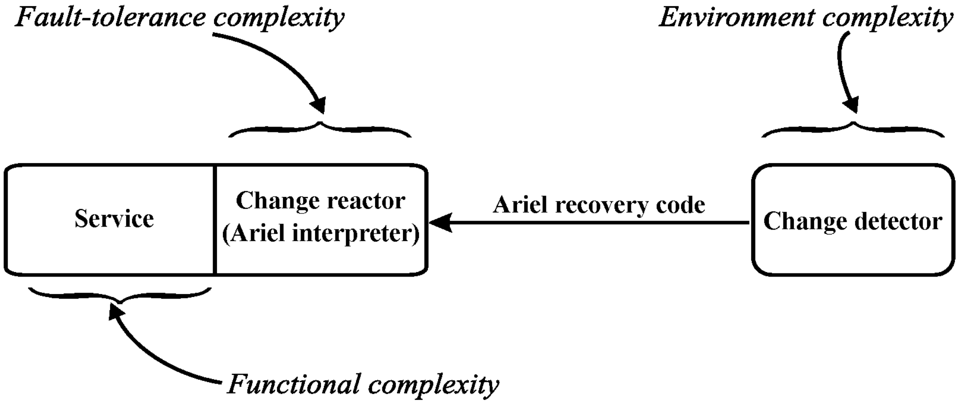


Figure 3: Main architectural components of a system to support adaptive services based on Ariel (De Florio & Blondia, 2005).

A prototype of a compliant architecture has been described in (De Florio & Blondia, 2005). Figure 3 provides the key components of that architecture: in that picture, BB is a middleware, called "backbone". Another approach, situated in the application-layer, reflects the environmental changes into the memory cells associated to so called "reflective variables" and achieves reactivity by assigning so called "refractive variables" (De Florio& Blondia, 2008a).

In the following we propose two examples to clarify how our vision of adaptive systems may provide a solution to seemingly contradicting quality of service requirements.

TWO EXAMPLES OF ADAPTIVE SERVICES

Adaptive Voting Sensors

Systems such as Body Area Networks (BANs) of wireless sensors are becoming more and more important for many reasons ranging from healthcare effectiveness to social security costs. In such systems patients are constantly monitored by mobile units that continuously transfer and publish the value of a set of vital parameters between a patient's location and the clinic or the doctor in charge (Ross, 2004). As remarked in (De Florio & Blondia, 2005), a true effective service like this is based on the contemporary fulfillment of two seemingly contradicting requirements:

R1: (Hard) guarantees are required so that, whenever the patient is in need, the system is to trigger a system alarm (e.g., dispatching medical care to the patient).

R2: (Soft) guarantees are required, such that no false alarm is triggered when the patient is not in real need—the latter to reduce the service costs.

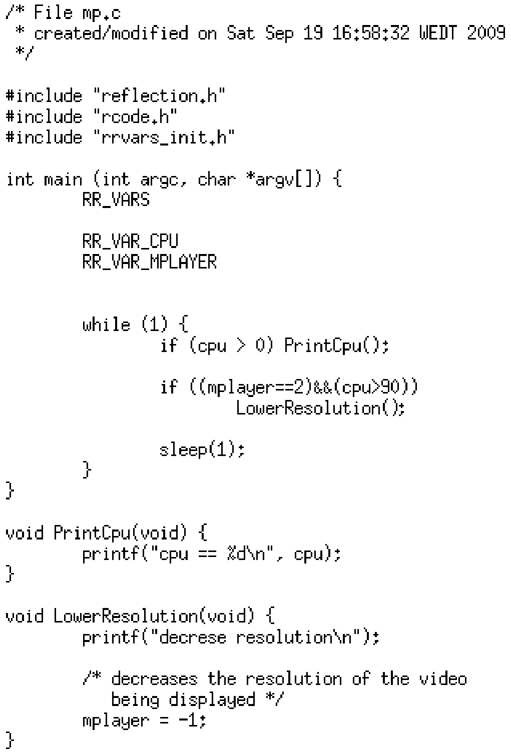
Clearly **R1** triggers the system alarm whenever any one of the sensors alerts or disconnects while **R2** does so only when the condition is confirmed by the occurrence of several sensor alerts. As explained in (De Florio & Blondia, 2005), a possible solution may be trading off between **R1** and **R2** through *m*-out-of-*n* majority voting, and triggering the system alarm after *m* out of the *n* sensor alerts. Whatever the choice of *m*, this approach is too inflexible, because the choice of *m* is fixed ahead of the run-time. An alternative solution would be to set up a series of strategies, $(\mathbf{FT}(k))_{k\square K}$, each of which may consider a different environment (for instance "heartbeat = 70, temperature = 38°C, arterial pressure = 120") and compute an $m(k)$-out-of-*n* majority voting. The consequence of this strategy would be that we would decompose the space of events into a set of blocks with known characteristics (we call them "scenarios" in the cited paper) and provide the best strategy matching each of these blocks, basically decomposing an unstable environment like our sensor networks into a set of quasi-stable environments.

Adaptively Redundant Data Structures

Another important requirement of computing services is that of enhancing data integrity: Protecting data against transient and permanent memory faults. The current practice for data integrity provisions (Taylor, Morgan, and Black, 1980) is to take the fault model as a static choice, which means that our data integrity provisions (DIP) will have a fixed range of admissible events to address and tolerate. This translates into two risks:

1. overshooting, i.e., over-dimensioning the DIP with respect to the actual threat being experienced, and
2. undershooting, namely underestimating the threat in view of an economy of resources.

Adaptively-redundant data structures constitute an effective strategy to avoid these two risks. In such system the amount of replicas of our data structures changes dynamically with respect to the observed disturbances. We assume that a monitoring tool is available to assess the probability of memory corruptions of the current environment (this corresponds to the *change detection* service described in the previous section. *Change reaction* is in this case tuning the employed redundancy after the observed disturbances. We have built a prototype for such a system, which is described in detail in (De Florio& Blondia, 2008a). In turn, such prototype realizes change detection and change reaction via a memory based metaphor that we call "reflective and refractive variables," briefly described in what follows.



```
/* File mp.c
 * created/modified on Sat Sep 19 16:58:32 WEDT 2009
 */

#include "reflection.h"
#include "rcode.h"
#include "rrvars_init.h"

int main (int argc, char *argv[]) {
        RR_VARS

        RR_VAR_CPU
        RR_VAR_MPLAYER


        while (1) {
                if (cpu > 0) PrintCpu();

                if ((mplayer==2)&&(cpu>90))
                        LowerResolution();

                sleep(1);
        }
}

void PrintCpu(void) {
        printf("cpu == %d\n", cpu);
}

void LowerResolution(void) {
        printf("decrese resolution\n");

        /* decreases the resolution of the video
           being displayed */
        mplayer = -1;
}
```

Figure 4: A simple example of the use of RR vars.

Reflective-and-refractive variables

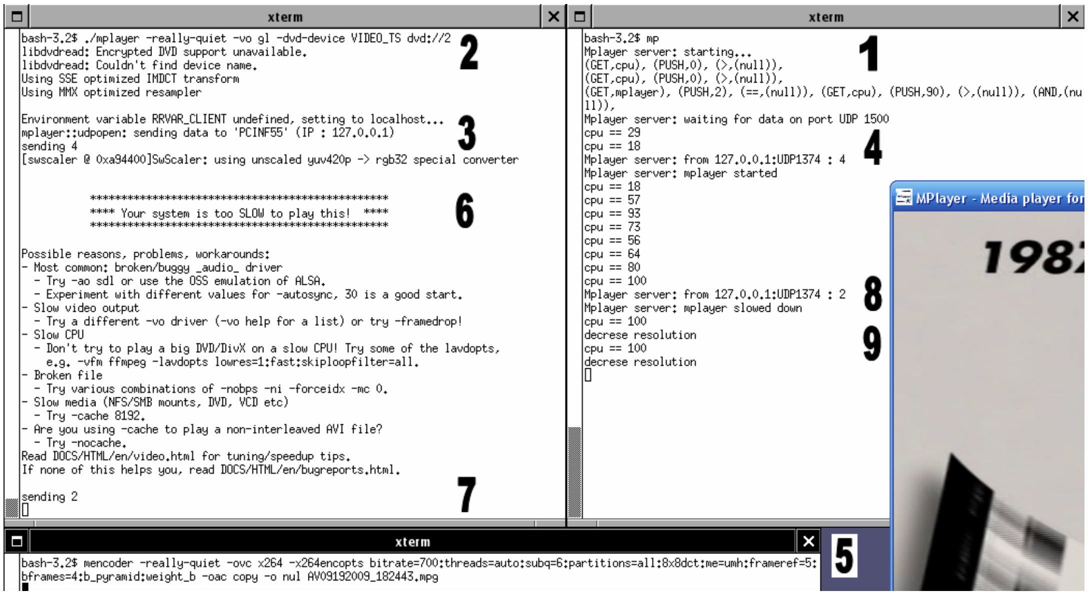
The idea behind reflective-and-refractive variables (RR vars) (De Florio & Blondia, 2007b) is to use memory accesses as an abstraction to perform concealed tasks. RR vars are volatile variables whose identifier links them to an external device, such as a sensor, or an RFID, or an actuator. In reflective variables, memory cells get asynchronously updated by service threads that interface those external devices. We use the well-known term "reflection" because those variables in a sense "reflect" the values measured by those devices. In refractive variables, on the contrary, write accesses trigger a request to update an external parameter, such as the data rate of the local TCP protocol entity or the amount of redundancy to be used in transmissions. We use to say that write accesses "refract" (that is, get redirected (Institute for Telecommunication Sciences, 2000) onto corresponding external devices.

The RR var model does not require any special language: Figure 4 is an example in the C language. The portrayed program declares two variables: "cpu", a reflective integer, which reports the current level of usage of the local CPU as an integer number between 0 and 100, and "mplayer", a reflective *and refractive* integer, which reports *and sets* properties of a video player (mplayer). The code transparently launches two threads that interface a CPU probe and a controller of the video player. The code periodically queries and prints the CPU usage. Moreover, it checks that

CPU usage is larger than 90(%), and at the same time the mplayer reports internal state 2 (mplayer experiences performance failures). When this happens, we assume that the video player is not able to render the current video clip because of insufficient CPU, so we request to change the quality of the clip. This is done simply by setting variable mplayer with a code representing that request: Being a refractive variable, this implies that the request is handed transparently to the mplayer controller and, from there, to the instrumented mplayer.

Figure 5 shows this simple code in action on our development platform—a Pentium-M laptop running Windows XP and the Cygwin tools. The figure depicts portions from 3 xterm windows as well as the window with the mplayer videoclip. To fully explain the figure we have labeled portions of the portrayed windows as follows:

1. The code from Figure 4 ("mp") is run. In particular, the user side of the mplayer controller is spawned and enters a waiting state on UDP port 1500.
2. The instrumented mplayer is launched.
3. The mplayer controller sends message "4" ("*mplayer started*").
4. "mp" starts printing CPU usage, and while doing so it receives message "4" from the mplayer. Variable "mplayer" is transparently set to 4.
5. On another window we initiate a CPU intensive service (encoding an MPEG2 stream into a high quality H.264 MPEG4 clip); this has the effect of requesting to the full service of the CPU and brings CPU usage to about 100%.
6. The instrumented mplayer detects this change and reports "*Your system is too SLOW to play this!*" Mplayer has no clue about the reason that produced this change, and prints out several hypotheses. At this point the video is rendered very badly – many frames are lost, video output is delayed, and audio/video synchronization is lost.
7. The mplayer controller sends message "2" ("*mplayer slowed down*")
8. "mp" receives message "2". Variable "mplayer" is transparently set to 2.
9. At this point conditions "mplayer==2" and "cpu>90" are both true and this triggers a request to set variable "mplayer" to -1. In turn, this requests the mplayer controller to adjust the resolution of the videoclip. An alternative change reaction would be to keep the current resolution and to change the frame dropping policy of the mplayer.



```
bash-3.2$ ./mplayer -really-quiet -vo gl -dvd-device VIDEO_TS dvd://2
libdvdread: Encrypted DVD support unavailable.
libdvdread: Couldn't find device name.
Using SSE optimized IMDCT transform
Using MMX optimized resampler

Environment variable RRVAR_CLIENT undefined, setting to localhost...
mplayer::udpopen: sending data to 'PCINF55' (IP : 127.0.0.1)
sending 4
[swscaler @ 0xa94400]SwScaler: using unscaled yuv420p -> rgb32 special converter


          ***********************************************
          **** Your system is too SLOW to play this!  ****
          ***********************************************

Possible reasons, problems, workarounds:
- Most common: broken/buggy _audio_ driver
  - Try -ao sdl or use the OSS emulation of ALSA.
  - Experiment with different values for -autosync, 30 is a good start.
- Slow video output
  - Try a different -vo driver (-vo help for a list) or try -framedrop!
- Slow CPU
  - Don't try to play a big DVD/DivX on a slow CPU! Try some of the lavdopts,
    e.g. -vfm ffmpeg -lavdopts lowres=1:fast:skiploopfilter=all.
- Broken file
  - Try various combinations of -nobps -ni -forceidx -mc 0.
- Slow media (NFS/SMB mounts, DVD, VCD etc)
  - Try -cache 8192.
- Are you using -cache to play a non-interleaved AVI file?
  - Try -nocache.
Read DOCS/HTML/en/video.html for tuning/speedup tips.
If none of this helps you, read DOCS/HTML/en/bugreports.html.

sending 2
```

```
bash-3.2$ mp
Mplayer server: starting...
(GET,cpu), (PUSH,0), (>,(null)),
(GET,cpu), (PUSH,0), (>,(null)),
(GET,mplayer), (PUSH,2), (==,(null)), (GET,cpu), (PUSH,90), (>,(null)), (AND,(nu
ll)),
Mplayer server: waiting for data on port UDP 1500
cpu == 29
cpu == 18
Mplayer server: from 127.0.0.1:UDP1374 : 4
Mplayer server: mplayer started
cpu == 18
cpu == 57
cpu == 93
cpu == 73
cpu == 56
cpu == 64
cpu == 80
cpu == 100
Mplayer server: from 127.0.0.1:UDP1374 : 2
Mplayer server: mplayer slowed down
cpu == 100
decrese resolution
cpu == 100
decrese resolution
```

```
bash-3.2$ mencoder -really-quiet -ovc x264 -x264encopts bitrate=700:threads=auto:subq=6:partitions=all:8x8dct:me=umh:frameref=5:
bframes=4:b_pyramid:weight_b -oac copy -o nul AV09192009_182443.mpg
```

Figure 5: An excerpt from the execution of the code in Fig. 4.

We observe that through the RR var model the design complexity is partitioned into two well defined and separated components: The code to interface external probes and controller is specified in a separate architectural component, while the functional code is produced in a familiar way, in this case as a C code reading and writing integer variables.

The result is a structured model to express tasks such as cross-layered optimization, adaptive or fault-tolerant computing in an elegant, non intrusive, and cost-effective way. Such model is characterized by strong separation of design concerns, for the functional strategies are not to be specified aside with the layer functions; only instrumentation is required, and this can be done once and for all. This prevents spaghetti-like coding for both the functional and the non-functional aspects, and translates in enhanced maintainability and efficiency.

A special reflective variable is “int _Redundance”. The device attached to _Redundance is in this case responsible for providing a trustworthy and timely estimation of the degree of redundancy matching the current disturbances. This device should be in most cases mission-dependent—for instance, for electrical substation automation systems, that could be a sensor able to assess the current electromagnetic interference. What we consider as a significant property of our approach is that such dependence is not hidden, but isolated in a custom architectural component. In so doing, complexity is partitioned but it is also explicitly available to the system designers for inspection and maintenance. This can facilitate porting *the service* which, as we pointed out in (De Florio & Deconinck, 2002), is something different from porting *the code* of a service. Failing to do so can bring to awful consequences, as can be seen in well-known accidents such as the failure of the Ariane 5 flight 501 or, even worse, in the failures of the Therac-25 linear accelerator [12].

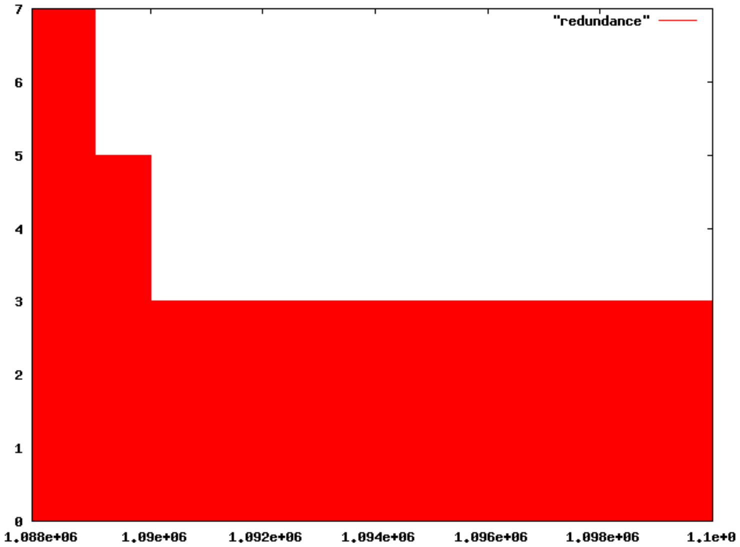


Figure 6: The *x* axis shows discrete time steps, while the *y* axis shows the amount of redundancy employed by the system. In this part of the experiment redundancy drops from 7 to 5 and then to 3.

In the case described in this section, the device connected to _Redundance is just a task that receives the return values of the majority voting algorithm executed when reading a redundant variable. Such return value is the maximum number of agreeing replicas in the current voting, which is used to compute the risk of failure: the less the agreeing replicas, the closer we are to the point of failure for the voting algorithm. Such "distance" is used to trigger the adaptation of redundancy. More information about the logics behind this mechanism can be found in (De Florio& Blondia, 2008b).

Performance Analysis

Our aim in what follows is to show how adaptively redundant data structures (ARDS) are an example of ***X***-dependable service, where ***X*** is not predefined and immutable, but rather it tracks effectively the environment it depends on.

Our experiment subjects a set of ARDS to disturbances whose effect is to zero the contents of a series of memory cells. Memory cells are first written and then continuously and sequentially read out in round-robin fashion. After some time t_1, disturbances corresponding to a certain environment ***E*** are injected; then at time t_2 disturbances pause; and then at time t_3 they start again but following another rule corresponding to a second environment ***E′***. The latter corresponds to more severe disturbances (longer bursts affecting larger series of cells). Figure 6 describes how the ARDS set reacts from the change at t_2 by decreasing the amount of redundancy from 7 to 5 and then to its minimum (in our system, this is 3). Figure 7 shows the behavior of the system after t_3 : more frequent and sudden changes bring the system rapidly to redundancy 7 and then let it fluctuate between 5 and 3 until the next pause.

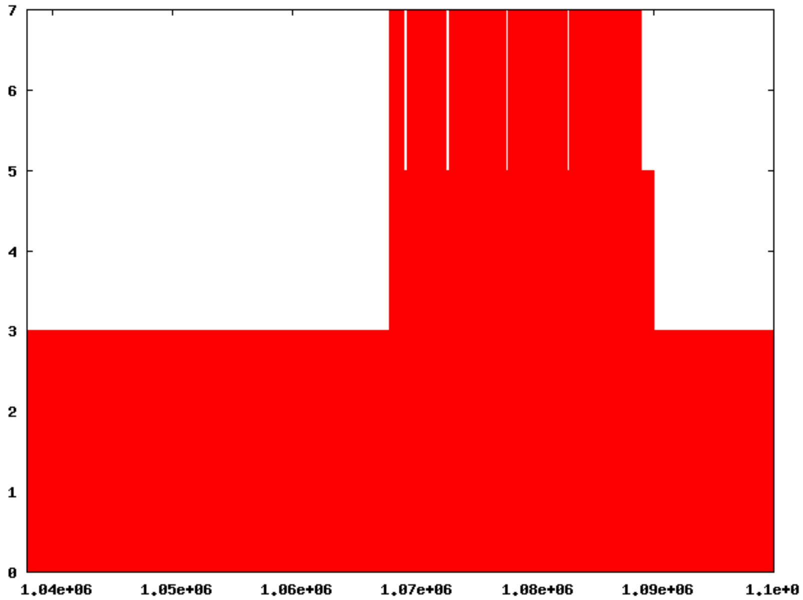


Figure 7: In this part of the experiment redundancy rapidly reaches 7 as a reaction to severe disturbances and then fluctuates between 7 and 5. Then the experiment restarts with a new pause and accordingly redundancy drops to 3.

RELATED WORK

The embedding of adaptability and dependability aspects in software system has been discussed by several authors – a thorough survey of this topic may be found for instance in (De Florio, 2009) and (De Florio & Blondia, 2008b) – though in our opinion it is worth highlighting among them the classical work of Kenneth Boulding on so-called General Systems Theory (Boulding, 1956) – a paper whose genial ideas still apply today and reverberate in modern concepts such as adaptive, resilient, and autonomic systems. In the cited paper a classification of systems[4] is proposed according to their "openness" and to their ability to self-manage. It is interesting to realize how most current software architectures still comply to Boulding's categories of "*Clockworks*" ("simple dynamic system with predetermined, necessary motions") and "*Thermostats*" ("control mechanisms in which the equilibrium position is not merely determined by the equations of the system, but the system will move to the maintenance of any given equilibrium, within limits"). The introduction of adaptability in software means being able to design and craft actual open (software) systems with a self-maintaining structure ("*Cells*" and "*Plants*", following Boulding's terminology), which paves the way to the design of truly autonomic systems (or "*Animals*," i.e. systems characterized by "mobility, teleological behavior and self-awareness"). In such systems, "behavior is response not to a specific stimulus but to an "image" or knowledge structure or view of the environment as a whole."

Other important milestones in this domain must necessarily include Norbert Wiener's classical work (Wiener, 1948) where the study of the structure and design of control systems – that is, cybernetics – is first introduced. The role of feedback loops and concepts such as adaptation were systematically investigated for the first time in the cited paper.

Throughout the following decades and the advent and evolution of software systems, briefly recalled in the Introduction to this paper, solutions to embed adaptive concerns in all layers of computer systems, and software systems in particular, have been sought. It would make little sense to try and summarize each and every episode in this imposing series of achievements, though a fundamental milestone of this "history" is with no doubts the introduction of Autonomic Computing (Kephart and Chess, 2003), some of the ideas of which can be found also in earlier works such as (De Florio, 2000). Self-managing computing systems are nothing but adaptive systems constructed around an architectural design pattern that makes it possible to realize Boulding's just mentioned higher level systems[5]. Service orientation and component orientation are two particularly fitting infrastructures for realizing autonomic computing systems.

What the embedding of adaptability *in the application layer* is concerned, we remark the role of Computational Reflection, defined in (Maes, 1987) as "the causal connection between a system and a meta-level description representing structural and computational aspects of that system". Computational Reflection is available to the programmer in the form of so-called Meta-object Protocols, which provide the programmer with a representation of a system as a set of objects representing and reflecting properties of "real" objects. Such "meta-objects" can for instance represent the structure of a class, or object interaction, or the code of an operation. This mapping process is called *reification*. As remarked e.g. in (De Florio, 2009), the causality relation of meta-object protocols could also be extended so as to allow for a dynamical reorganization of the structure and the operation of a software system to perform adaptation in the form of, e.g. reconfiguration or error recovery.

Composition Filters (Bergmans & Aksit, 2001) provide the programmer with input and output filters, that is, components that intercept and manipulate message transmission and reception. An input filter set filters incoming messages and an output filter set filters outgoing messages. Filters in a filter set orderly evaluate and possibly manipulate the messages before passing them to the target object. This could be used to encode complex feedback loops based on hierarchies of sensors and actuators.

Aspect Orientation (Kiczales & Mezini, 2005) systematically automatizes and supports the process of adapting an existing code so that it fulfils specific aspects. Aspect-oriented programming may be defined as a software engineering methodology supporting those adaptations in such a way as to guarantee that they do not destroy the original design and do not overly increase complexity. A particularly interesting of recent aspect oriented architectures is so-called dynamic weaving, which allows the adaptation code to vary dynamically over time, e.g., as a response to environmental changes. Aspect Orientation has been used e.g. in (Sun et al., 2009) as a structure to express adaptability and dependability in the software application layer.

Recovery Languages (De Florio, 2000) is a hybrid approach – it offers both a software architecture and an application layer, both of which can be used to express adaptive-and-dependable software systems. Recovery Languages supports two application layers, namely: (1) the classical functional application layer, devoted to the fulfillment of the functional requirements, and (2) an ancillary application layer, specifically devoted to error recovery, which gets activated when errors are detected in the system. This second layer is supported by architectural components that implement a network of error detectors. The error recovery application layer provides feedback loops through nested guarded actions.

Finally, the programming language Oz (Van Roy & Haridi, 2004) provides at the same time transparency of unnecessary aspects[6], and translucency of events reflecting relevant changes in the system and its environment. The occurrence of such events is reflected in the language by means of so-called failure listeners: all Oz entities can produce streams of events that reflect the sequential occurrence of their notifications. Any Oz task can become a failure listener, that is, it can hook to such streams and be informed of all the faults experienced by any other tasks. In other words, fault detection is intrinsically managed by Oz. Error recovery can then be managed by guarded actions, as it is the case for Recovery Languages. Oz has been used to express adaptive control loops (Van Roy, 2006) for self-management.

CONCLUSIONS

We have introduced the problem of adaptability and dependability as key requirements for a system where the environment and hence the fault model varies over time. A model focusing on the anticipation of changes in the environment has been proposed. We have shown with two examples the potential of adaptive services as means to achieve a dynamic trade-off between seemingly contradicting requirements. Particular emphasis has been given to a prototypic system adopting the principles introduced in this paper and realizing adaptively redundant data structures.

ACKNOWLEDGMENTS

We express our gratitude to the reviewers for their useful and valuable remarks. We also thank the Editors who invited us to submit this paper that extends our ADAMUS 2007 contribution (De Florio & Blondia, 2007a).

REFERENCES

Anonymous (1870). Calculating by machinery. The Manufacturer and Builder, 2:8, (pp.225–227).

Bergmans, Lodewijk and Aksit, Mehmet (2001). Composing crosscutting concerns using composition filters. Communications of the ACM 44:10, (pp.51–57).

Boulding, Kenneth (1956). General Systems Theory, the Skeleton of Science. Management Science, 2:3, (pp.197–208).

de Brisse, Baron Léon (1875). Album de l'Exposition universelle de Paris en 1855.

De Florio, Vincenzo (2000, October). A Fault-Tolerance Linguistic Structure for Distributed Applications. PhD thesis, Dept. of Electrical Engineering, University of Leuven. ISBN 90-5682-266-7.

De Florio, Vincenzo (2002). Course notes of Advanced Computer Architectures. Katholieke Universiteit Leuven.

De Florio, Vincenzo (2009). Application-layer Fault-Tolerance Protocols. IGI Global, Hershey, PA. ISBN 1-60566-182-1.

De Florio V. and Blondia C. (2005, June). A system structure for adaptive mobile applications. Proceedings of the Sixth IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2005), Taormina - Giardini Naxos, Italy, (pp. 270–275).

De Florio V. and Blondia C. (2007a). Adaptation as a New Requirement for Software Engineering. Proceedings of the First IEEE WoWMoM Workshop on Adaptive and DependAble Mission- and bUsiness-critical mobile Systems (ADAMUS), Helsinki, Finland.

De Florio V. and Blondia C. (2007b). Reflective and refractive variables: A model for effective and maintainable adaptive-and-dependable software. Proc. of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Luebeck, Germany.

De Florio, V. and Blondia, Chris (2008a). A Survey of Linguistic Structures for Application-level Fault-Tolerance. ACM Computing Surveys", 40(2).

De Florio V. and Blondia C. (2008b). On the requirements of new software development. International Journal of Business Intelligence and Data Mining, 3(3). Inderscience.

De Florio V. and Degoninck, C. (2002, April). On some key requirements of mobile application software. Proc. of the 9th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), Lund, Sweden. IEEE Comp. Soc. Press.

Horning, James J. (1988, July). ACM Fellow Profile — James Jay (Jim) Horning. ACM

Software Engineering Notes, 23(4).

Institute for Telecommunication Sciences (2000). ANS T1.523-2001, Telecom Glossary 2000. Retrieved on 2009-09-09 from http://www.its.bldrdoc.gov/ projects/telecomglossary2000.

Kephart, J.O. and Chess, D.M. (2003). The Vision of Autonomic Computing. Computer 36:1, (pp.41–50).

Kiczales, G., & Mezini, M. (2005). Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer Lncs Series.

Laprie, Jean-Claude (1985). Dependable computing and fault tolerance: Concepts and terminology. Proc. of the 15th Int. Symposium on Fault-Tolerant Computing (FTCS-15), Ann Arbor, Mich., IEEE Comp. Soc. Press, (pp. 2–11).

Leveson, Nancy G. (1995). Safeware: Systems Safety and Computers. Addison-Wesley.

Maes, P. (1987). Concepts and Experiments in Computational Reflection. Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA-87), (pp. 147–155).

Lyu, Michael R. (1998). Reliability-oriented software engineering: Design, testing and evaluation techniques. IEE Proceedings – Software, 145(6), Special Issue on Dependable Computing Systems, (pp. 191–197).

Ross, Philip E. (2004). Managing care through the air. IEEE Spectrum international edition, 41(12), (pp.14–19).

Sun, Hong, De Florio, Vincenzo, Gui, Ning, and Blondia, Chris (2009). Adaptation Strategies for Performance Failure Avoidance. Proceedings of the 3rd IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI).

Taylor, David J., Morgan, David E., and Black, James P. (1980, November). Redundancy in data structures: Improving software fault tolerance. IEEE Trans. on Software Engineering, 6(6), (pp. 585–594)

Unipede (1995, January). Automation and control apparatus for generating stations and sub-stations

– electromagnetic compatibility – immunity requirements. Technical Report UNIPEDE Norm (SPEC) 13.

Van Roy, Peter and Haridi, Seif (2004). Concepts, Techniques, and Models of Computer Programming. MIT Press.

Weik, Martin H. (1961, January-February). The ENIAC story. ORDNANCE — The Journal of the American Ordnance Association. (Available at URL http://ftp.arl.mil/~mike/comphist/eniac-story.html).

Wiener, Norbert (1948). Cybernetics, or Control and Communication in the Animal and the Machine. MIT Press, Cambridge, MA.

ENDNOTES

[1] To fully appreciate the extent of this revolution, let us reprint here an excerpt from an article by Brisse (Baron de Brisse, 1875) celebrating the 1855 Paris Universal Exhibition, where the Scheutzes' machine was shown to the public for the first time: "*This machine solves equations of 4th and even greater degree; operates in any numerical system* [. . .] *The scientists, boasting their computation capabilities as a miracle of natural law, will be soon taken over by a simple machine that, under the nearly blind guidance of a common man and by means of custom movement, is going to dig the infinite outer space with a security and depth way greater than that of scientists. Any man capable to formulate a problem and having at his disposal Mr. Scheutz's machine will have no need for Archimedes, Newtons, or Laplaces* [. . .] *This quasi-intelligent machine not only computes in a few seconds what normally would require hours; it also prints the obtained results, adding the advantages of a neat calligraphy to those of computations with no chance for errors.*" See also (De Florio, 2002) and (Anonymous, 1870).

[2] This excerpt from a report on the ENIAC activity [17] gives an idea of how dependable computers were in 1947: "*power line fluctuations and power failures made continuous operation directly off transformer mains an impossibility* [...] *down times were long; error-free running periods were short* [...]". After many considerable improvements, still "*trouble-free operating time remained at about 100 hours a week during the last 6 years of the ENIAC's use*", i.e., an availability of about 60%!

[3] See for instance the following statement from Popular Mechanics, 1949: "*In the future computers will weigh at most 1.5 tons*"!

[4] An "arrangement of theoretical systems and constructs in a hierarchy of complexity, roughly corresponding to the complexity of the "individuals" of the various empirical fields, [. . .] leading towards a "system of systems."

[5] In (Kephart and Chess, 2003), the authors describe how in autonomic systems "New components integrate as effortlessly as a new cell establishes itself in the human body" – the terms used by the authors is indeed the same as Boulding's, "Cell."

[6] In Oz, it is simply not possible to tell whether a method or an entity is local or distributed.