System Structure for Dependable Software Systems

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 (IBBT)
Gaston Crommenlaan 8, 9050 Ghent-Ledeberg, Belgium

Abstract. Truly dependable software systems should be built with structuring techniques able to decompose the software complexity without hiding important hypotheses and assumptions such as those regarding their target execution environment and the expected fault- and system models. A judicious assessment of what can be made transparent and what should be translucent is necessary. This paper discusses a practical example of a structuring technique built with these principles in mind: Reflective and refractive variables. We show that our technique offers an acceptable degree of separation of the design concerns, with limited code intrusion; at the same time, by construction, it separates but does not hide the complexity required for managing fault-tolerance. In particular, our technique offers access to collected system-wide information and the knowledge extracted from that information. This can be used to devise architectures that minimize the hazard of a mismatch between dependable software and the target execution environments.

1 Introduction

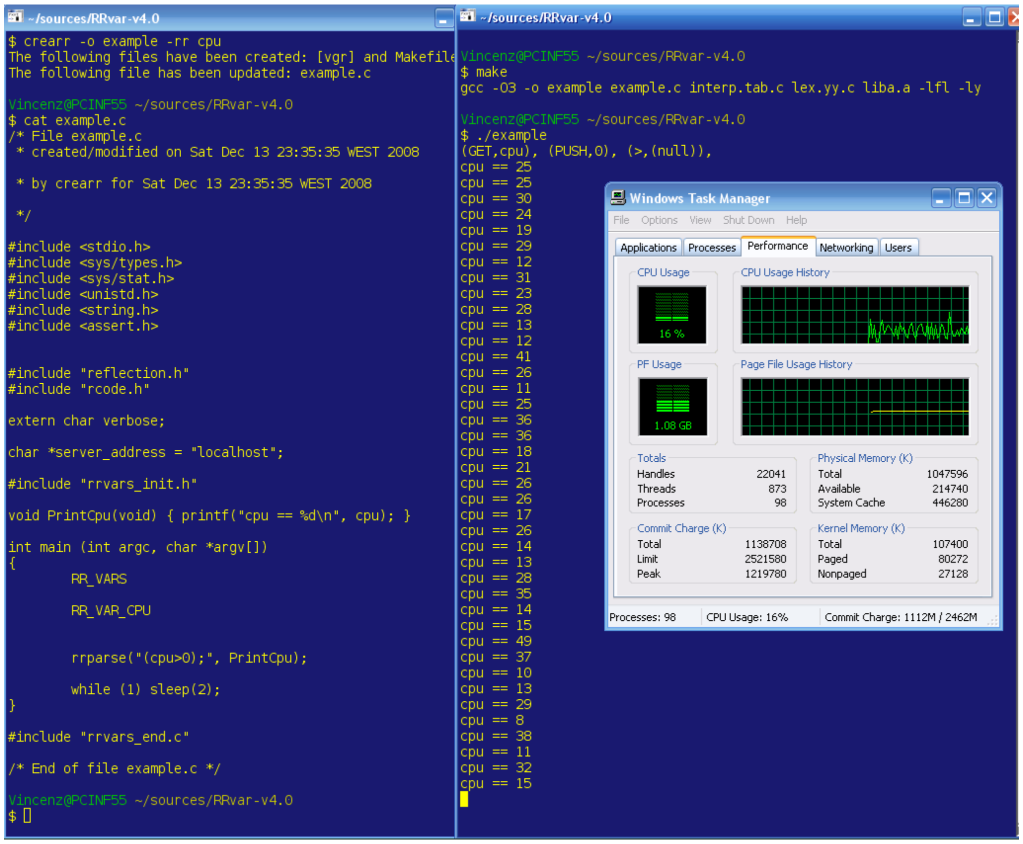
We are living in a society of software-predominant systems. Computer systems are everywhere around us—ranging from supercomputers to tiny embedded systems—and regrettably we are often reminded of the importance that services appointed to computers be reliable, safe, secure, and flexible. What is often overlooked by many is the fact that most of the logic behind those services, which support and sustain our societies, lies in the software layers. Software has become the point of accumulation of a large amount of complexity. Software is ubiquitous, mobile, and has pervaded all aspects of our lives. Even more than that, the role appointed to software has become crucial, as it is ever more often deployed so as to fulfill mission critical tasks. The key ingredient to achieve this change in complexity and role has been the conception of tools to manage the structuring of software so as to divide and conquer its complexity. Dividing complexity was achieved through specialization, by partitioning complexity into system layers; conquering that complexity was mainly reached by hiding it by means of clever organizations (e.g. through object orientation and, more recently, aspect and server orientation). Unfortunately, though made transparent, still this complexity is part of the overall system being developed. As a result, we have been given tools to compose complex software-intensive systems in a relatively short amount of time, but the resulting systems are often entities whose structure is unknown and that are likely to get inefficient and even be error-prone.

A particular case of this situation is given by the network software. In fact, the system and fault assumptions on which the original telecommunication services had been designed, which were considered as permanently valid and hence hidden and hardwired throughout the system layers, now turn out to be not valid anymore in the new contexts brought about by mobile and ubiquitous computing. Retrieving and exploiting this “hidden intelligence” [1] is very difficult.

Hiding intelligence is particularly risky when dealing with application-layer software fault-tolerance. Research in this domain was initiated by Brian Randell with his now classical article on the choice of which system structure to give our programs in order to make them tolerant to faults [2]. The key problem expressed in that paper was that of a cost-effective solution to embed fault-tolerance in the application software. Recovery blocks was the proposed solution. Regardless of the well-known merits of Randell’s solution, which throughout these decades have been the subject of many a research contribution, we observe here how recovery blocks do not attempt to hide the fault-tolerance management complexity. This fact can be read in two ways—one could say that recovery blocks are characterized by fault-tolerance code intrusion, as they provide no special container for the expression of the fault-tolerance provisions. On the other hand, this “open” approach brings the complexity to the foreground minimizing the risk of hiding “too much.”

We believe such risk to be particularly severe when dealing with fault-tolerance in software. History of computing is paved with examples that show how software designed to be dependable proves defenseless against wrong assumptions about their execution environment and unforeseen threats. This happens because any dependable software builds upon two fundamental sets of assumptions—the system and the fault model. The hidden intelligence syndrome manifests itself in software e.g. when the link with those two models is lost. When the fault-tolerant service is eventually deployed, often no verification is foreseen that the deployment environments actually match the system model and that the experienced threats actually comply with the fault model. More details on this may be found in [3].

Our thesis here is that dependable software systems should be built with architectures and/or structuring techniques able to decompose the software complexity without hiding in the process important hypotheses and assumptions such as those regarding their target execution environment and the expected fault- and system models. On the contrary, the run-time executive of those systems should continuously verify those hypotheses and assumptions by matching them with context knowledge derived from the processing subsystems and their environment.



```
~/sources/RRvar-v4.0
$ crearr -o example -rr cpu
The following files have been created: [vgr] and Makefile
The following file has been updated: example.c

Vincenz@PCINF55 ~/sources/RRvar-v4.0
$ cat example.c
/* File example.c
 * created/modified on Sat Dec 13 23:35:35 WEST 2008

 * by crearr for Sat Dec 13 23:35:35 WEST 2008

 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <assert.h>


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

extern char verbose;

char *server_address = "localhost";

#include "rrvars_init.h"

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

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

        RR_VAR_CPU


        rrparse("(cpu>0);", PrintCpu);

        while (1) sleep(2);
}
#include "rrvars_end.c"

/* End of file example.c */

Vincenz@PCINF55 ~/sources/RRvar-v4.0
$ 
```

```
~/sources/RRvar-v4.0

Vincenz@PCINF55 ~/sources/RRvar-v4.0
$ make
gcc -O3 -o example example.c interp.tab.c lex.yy.c liba.a -lfl -ly

Vincenz@PCINF55 ~/sources/RRvar-v4.0
$ ./example
(GET,cpu), (PUSH,0), (>,(null)),
cpu == 25
cpu == 25
cpu == 30
cpu == 24
cpu == 19
cpu == 29
cpu == 12
cpu == 31
cpu == 23
cpu == 28
cpu == 13
cpu == 12
cpu == 41
cpu == 26
cpu == 11
cpu == 25
cpu == 36
cpu == 36
cpu == 18
cpu == 21
cpu == 26
cpu == 26
cpu == 17
cpu == 26
cpu == 14
cpu == 13
cpu == 28
cpu == 35
cpu == 14
cpu == 15
cpu == 49
cpu == 37
cpu == 10
cpu == 13
cpu == 29
cpu == 8
cpu == 38
cpu == 11
cpu == 32
cpu == 15
```

```
Totals                      Physical Memory (K)
Handles        22041        Total          1047596
Threads          873        Available       214740
Processes         98        System Cache    446280

Commit Charge (K)           Kernel Memory (K)
Total        1138708        Total           107400
Limit        2521580        Paged            80272
Peak         1219780        Nonpaged         27128

Processes: 98   CPU Usage: 16%   Commit Charge: 1112M / 2462M
```

Fig. 1. An "hello world" RR vars program is produced, typed, compiled, and executed. The result is a program tracking the amount of CPU time used in the system.

The structure of this paper is as follows: in Sect. 2 we introduce an architecture and a structuring technique to craft dependable software systems complaint to the above vision. Section 3 discusses our approach. Section 4 describes an experiment to measure its performance and latency. A short summary of the state of the art in reflection and related approaches is the topic of Sect. 5. Finally, our conclusions and a glance to future contributions are given in Sect. 6.

2 Reflective and Refractive Variables

The idea behind reflective and refractive variables (RR vars) [4] is quite simple: As well known, variables in high-level programming languages can be formally described as a couple consisting of a fixed name (the identifier, chosen by the programmer) and a set of memory cells (chosen by the compiler and fixed throughout the execution of the code). Those cells have fixed sizes which are determined by so-called data types; data types also determine the interpretation of the bit patterns that can be stored into the memory cells as well as the semantics of the operations that can be carried out with the variables.

From the point of view of the programmer, RR vars are just variables with a special, predefined name and type; for instance "cpu", "mplayer", "bandwidth", and "watchdog" are all integer RR vars, while "alphacount[]" is an array of floating point RR vars. As plain variables, RR vars are associated with some memory cells; only, these memory cells have "volatile" contents: Their value can change unexpectedly, and provide for instance an estimation of the current amount of CPU being used on some processing node; or the current state of a media player; or an estimation of the end-to-end bandwidth between two processing nodes; or the current state of a task being guarded by a watchdog timer. We use to say that each variable *reflects* the value of a certain property. Furthermore, writing into one of these variables can trigger an adjustment of some parameter, e.g., resetting a watchdog timer or changing the frame dropping policy of a video player. If that is the case, we use to say that writes *refract* and trigger a request for adaptation of their corresponding properties.

The concealed side of RR vars is currently a set of processes (called RR vars servers) that are associated to sensors and detectors (for reflective variables) and to actuators and recoverers (for refractive variables). RR vars are defined in a shared memory segment where those processes have read/write access rights. In an initialization phase the addresses of the memory cells of the RR vars are passed to their associated processes. From then on, each external change is reflected in the memory cells, while each write access in the refractive variables is intercepted and refracts into a predefined procedure call. This latter functionality is carried out by parsing the source code for assignment operations. Management of concurrent accesses to the memory cells has been foreseen but is not present in the current prototypical implementation.

The programming model of RR vars is twofold:

1. The user may directly access the RR vars, verifying for instance that certain conditions have been met and certain thresholds have been overcome, performing corrective actions possibly involving refractive variables.
2. The user may define callback functions, i.e., functions that are executed asynchronously by the system whenever certain guards are verified as true.

Clearly the second option is characterized by a much higher degree of separation of the design concerns, as the callbacks may be defined in a separate code. Greater efficiency is also reached through callbacks, because they avoid the greedy resource consumption typical of polling.

Let us now describe this model in more detail by means of a practical example: Let us suppose we want to make use of reflective variable "cpu". To write an "hello world" program using that variable, we can use a utility called "crearr". For instance, the following command-line:

```
crearr -o example -rr cpu
```

produces the source code “example.c” plus some ancillary scripts to compile that code. Figure 1 shows the produced source code and the executable code running on a Windows/Cygwin PC. The program initializes the RR vars system through the “RR_VARS” macro and then defines reflective variable “cpu” through macro “RR_VAR_CPU”. Our utility produces a code that makes use of callbacks. In this case a single callback and its guard are defined through function “rrparse”. The default guard is quite trivial as it checks that the amount of CPU used is greater than 0. The callback is also very simple as it just prints out the current amount of CPU being used in the system. To prevent the code from terminating, an endless loop is also produced. A similar behavior would be observed by omitting the callback and polling the value of “cpu” within the endless loop, as in:

```
while (1) { if (cpu > 0) Callback(); }.
```

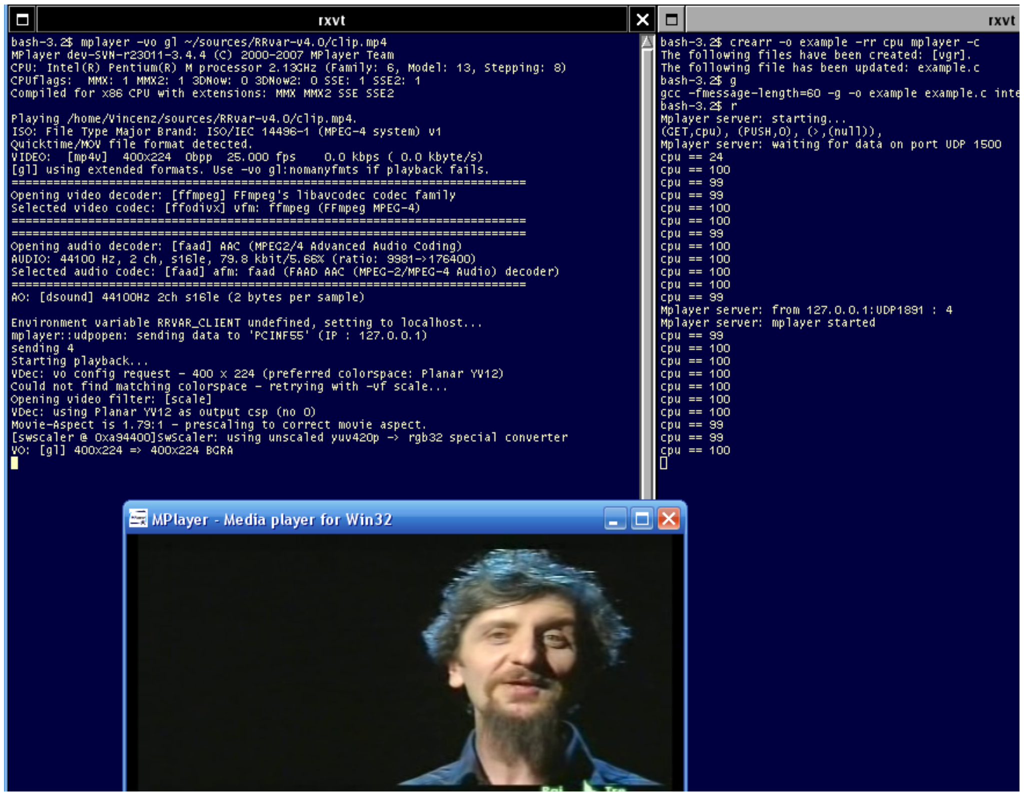
Worth noting is the fact that the guard is specified as a string. Such string is parsed against a grammar similar to that of C language expressions. Our parser produces a pseudo-code which is interpreted at run-time each time a change or a fault is detected. The Windows Task Manager is also displayed to show how the evolution of the contents of reflective variable “cpu” matches the one reported by that system utility.

Figure 2 produces a more complex example: The right-hand side windows redefines “example.c” so as to make use of two variables, “cpu” and “mplayer”. The latter is a reflective and refractive variable used to monitor and control an instance of the mplayer movie player [5]. The same callback as in Fig. 1 is used here—actually the only difference between the previous code and this one is the declaration of variable “mplayer”, which implicitly executes an “mplayer server”. The latter is a process that waits for and reacts on messages on UDP port 1500.

In the left-hand side window the actual mplayer program is launched so as to play an MPEG-4 clip. This is an instrumented version of mplayer which forwards messages to a service defined by the TCP address in environment variable “RRVAR_CLIENT” and port number 1500. Default value for that address is “localhost” (the node where the program is being executed.)

In so doing the instrumented mplayer creates a stream of notifications directed to the already mentioned “mplayer server” running concurrently with the RR vars program in the right-hand side window. One such notification is “mplayer started”. Notifications also include *exceptions*, whose nature is identified, and performance failures (see Fig. 3), which may be due to several reasons, including an insufficient amount of available CPU. If that is the case a possible strategy to adjust the performance would be to change the frame dropping policy of the mplayer so as to gracefully degrade the user quality of experience. This is obtained through the following simple callback and guard:

```
void SystemIsSlow(void) {
 printf("Mplayer reports 'System too slow to play \
         clip' and CPU above threshold:\n");
 mplayer = HARDFRAMEDROP; // drop frames more easily
```



```
bash-3.2$ mplayer -vo gl ~/sources/RRvar-v4.0/clip.mp4
MPlayer dev-SVN-r23011-3.4.4 (C) 2000-2007 MPlayer Team
CPU: Intel(R) Pentium(R) M processor 2.13GHz (Family: 6, Model: 13, Stepping: 8)
CPUflags:  MMX: 1 MMX2: 1 3DNow: 0 3DNow2: 0 SSE: 1 SSE2: 1
Compiled for x86 CPU with extensions: MMX MMX2 SSE SSE2

Playing /home/Vincenz/sources/RRvar-v4.0/clip.mp4.
ISO: File Type Major Brand: ISO/IEC 14496-1 (MPEG-4 system) v1
Quicktime/MOV file format detected.
VIDEO:  [mp4v]  400x224  0bpp  25.000 fps    0.0 kbps ( 0.0 kbyte/s)
[gl] using extended formats. Use -vo gl:nomanyfmts if playback fails.
==========================================================================
Opening video decoder: [ffmpeg] FFmpeg's libavcodec codec family
Selected video codec: [ffodivx] vfm: ffmpeg (FFmpeg MPEG-4)
==========================================================================
==========================================================================
Opening audio decoder: [faad] AAC (MPEG2/4 Advanced Audio Coding)
AUDIO: 44100 Hz, 2 ch, s16le, 79.8 kbit/5.66% (ratio: 9981->176400)
Selected audio codec: [faad] afm: faad (FAAD AAC (MPEG-2/MPEG-4 Audio) decoder)
==========================================================================
AO: [dsound] 44100Hz 2ch s16le (2 bytes per sample)

Environment variable RRVAR_CLIENT undefined, setting to localhost...
mplayer::udpopen: sending data to 'PCINF55' (IP : 127.0.0.1)
sending 4
Starting playback...
VDec: vo config request - 400 x 224 (preferred colorspace: Planar YV12)
Could not find matching colorspace - retrying with -vf scale...
Opening video filter: [scale]
VDec: using Planar YV12 as output csp (no 0)
Movie-Aspect is 1.79:1 - prescaling to correct movie aspect.
[swscaler @ 0xa94400]SwScaler: using unscaled yuv420p -> rgb32 special converter
VO: [gl] 400x224 => 400x224 BGRA
```

```
bash-3.2$ crearr -o example -rr cpu mplayer -c
The following files have been created: [vgr].
The following file has been updated: example.c
bash-3.2$ g
gcc -fmessage-length=60 -g -o example example.c inte
bash-3.2$ r
Mplayer server: starting...
(GET,cpu), (PUSH,0), (>,(null)),
Mplayer server: waiting for data on port UDP 1500
cpu == 24
cpu == 100
cpu == 99
cpu == 99
cpu == 100
cpu == 100
cpu == 99
cpu == 100
cpu == 100
cpu == 100
cpu == 100
cpu == 99
Mplayer server: from 127.0.0.1:UDP1891 : 4
Mplayer server: mplayer started
cpu == 99
cpu == 100
cpu == 100
cpu == 100
cpu == 100
cpu == 100
cpu == 100
cpu == 99
cpu == 99
cpu == 100
```

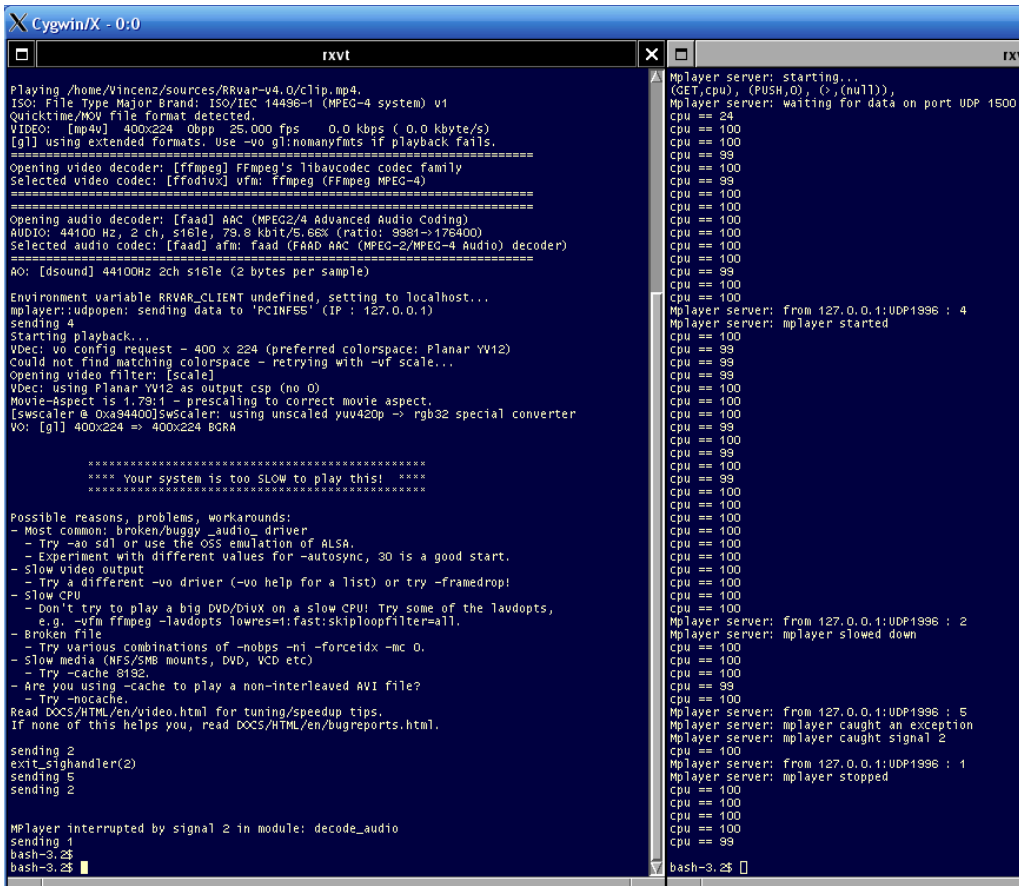
Fig. 2. An instrumented mplayer renders a video-clip and forwards notifications to a "hello world" RR vars program. Each notification is stored as an integer in RR var "mplayer".

```
}
...
 rrparse("(cpu>98)&&(mplayer==2);", SystemIsSlow);      // 2 == UDPMSG_SLOW
```

An important remark here is that the "SystemIsSlow" callback requests the adjustments just by setting variable "mplayer" with integer value represented by symbolic constant HARDFRAMEDROP. This is because "mplayer" is both reflective and refractive: Setting one such variable refracts, that is, it triggers a predefined action. This fine-grained remote control of mplayer has been possible by exploiting the so-called "slave mode protocol" designed by the authors of mplayer [6].

A slightly more complex example is given by "watchdog", a reflective and refractive integer. This variable monitors and controls a watchdog timer. As usual, a "hello world" instance of an RR var program using a watchdog is produced by executing for instance

`crearr -o w -rr watchdog`,



```
Playing /home/Vincenz/sources/RRvar-v4.0/clip.mp4.
ISO: File Type Major Brand: ISO/IEC 14496-1 (MPEG-4 system) v1
Quicktime/MOV file format detected.
VIDEO:  [mp4v]  400x224  0bpp  25.000 fps    0.0 kbps ( 0.0 kbyte/s)
[gl] using extended formats. Use -vo gl:nomanyfmts if playback fails.
==========================================================================
Opening video decoder: [ffmpeg] FFmpeg's libavcodec codec family
Selected video codec: [ffodivx] vfm: ffmpeg (FFmpeg MPEG-4)
==========================================================================
==========================================================================
Opening audio decoder: [faad] AAC (MPEG2/4 Advanced Audio Coding)
AUDIO: 44100 Hz, 2 ch, s16le, 79.8 kbit/5.66% (ratio: 9981->176400)
Selected audio codec: [faad] afm: faad (FAAD AAC (MPEG-2/MPEG-4 Audio) decoder)
==========================================================================
AO: [dsound] 44100Hz 2ch s16le (2 bytes per sample)

Environment variable RRVAR_CLIENT undefined, setting to localhost...
mplayer::udpopen: sending data to 'PCINF55' (IP : 127.0.0.1)
sending 4
Starting playback...
VDec: vo config request - 400 x 224 (preferred colorspace: Planar YV12)
Could not find matching colorspace - retrying with -vf scale...
Opening video filter: [scale]
VDec: using Planar YV12 as output csp (no 0)
Movie-Aspect is 1.79:1 - prescaling to correct movie aspect.
[swscaler @ 0xa94400]SwScaler: using unscaled yuv420p -> rgb32 special converter
VO: [gl] 400x224 => 400x224 BGRA


           ***********************************************
           ****  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
exit_sighandler(2)
sending 5
sending 2


MPlayer interrupted by signal 2 in module: decode_audio
sending 1
bash-3.2$
bash-3.2$
```

```
Mplayer server: starting...
(GET,cpu), (PUSH,0), (>,(null)),
Mplayer server: waiting for data on port UDP 1500
cpu == 24
cpu == 100
cpu == 100
cpu == 99
cpu == 100
cpu == 99
cpu == 100
cpu == 100
cpu == 100
cpu == 100
cpu == 100
cpu == 100
cpu == 99
cpu == 100
cpu == 100
Mplayer server: from 127.0.0.1:UDP1996 : 4
Mplayer server: mplayer started
cpu == 100
cpu == 99
cpu == 99
cpu == 99
cpu == 100
cpu == 100
cpu == 100
cpu == 99
cpu == 100
cpu == 99
cpu == 100
cpu == 99
cpu == 100
cpu == 100
cpu == 100
cpu == 100
cpu == 100
cpu == 100
cpu == 100
cpu == 100
cpu == 100
cpu == 100
Mplayer server: from 127.0.0.1:UDP1996 : 2
Mplayer server: mplayer slowed down
cpu == 100
cpu == 100
cpu == 100
cpu == 99
cpu == 100
Mplayer server: from 127.0.0.1:UDP1996 : 5
Mplayer server: mplayer caught an exception
Mplayer server: mplayer caught signal 2
cpu == 100
Mplayer server: from 127.0.0.1:UDP1996 : 1
Mplayer server: mplayer stopped
cpu == 100
cpu == 100
cpu == 100
cpu == 100
cpu == 99

bash-3.2$
```

Fig. 3. Mplayer forwards notifications such as performance failures ("System is too slow to play this!") and exceptions (e.g. illegal operations or a termination request from the user.)

which produces a code an excerpt of which is depicted in Fig. 4.

Clearly making use of a watchdog requires a proper configuration of several key parameters. This is done via a custom language called Ariel, described in [7,8]. Without going into details unnecessary to the current discussion, we just recall that Ariel can translate texts such as

```
WATCHDOG TASK 1
 HEARTBEATS EVERY 3000 MS
 ALPHA-COUNT IS threshold = 3.0,  factor = 0.4
 END ALPHA-COUNT
END WATCHDOG
```

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

        while (1) {
                // when watchdog is less than zero,
                // it represents status messages
                //
                // such messages are available as
                // strings in wmsgs[-watchdog]
                if (watchdog < 0)
                     printf("watchdog == %d (%s)\n",
                        watchdog, wmsgs[-watchdog]);
                else {
                     // when watchdog is greater than zero,
                     // it counts the "kicks" received
                     // from the watched task
                     if (watchdog > 0)
                        printf("watchdog == %d (kick no.%d)\n",
                                watchdog, watchdog+1);
                }

                // when status message is "paused", we
                // set reset the watchdog variable.
                // Being refractive, this resets
                // the watchdog timer task.
                if (watchdog == -1) {
                        watchdog = 0;
                }

                sleep(2);
        }
}
```

Fig. 4. An excerpt from the "hello world" code to control watchdogs via RR vars, produced by `crearr`.

into C code (available here as file "watchdog1.c"). In this case watchdog 1 expects a "heartbeat" at most every 3 seconds and sets some parameters of a so-called alpha-counter (whose meaning is described in Sect. 3).

We modified the Ariel translator so as to produce a valid RR var server. In other words, our watchdog updates the memory cells associated with RR var "watchdog" so as to reflect its state. That is an integer that represents watchdog states if negative, and the amount of received "heartbeats" otherwise. Heartbeats are sent as UDP messages addressing a certain port.

It is worth noting here that the Mplayer movie player [5] allows a heartbeat command to be specified via its option "heartbeat-cmd *command*". For instance, the following command:

```
mplayer -heartbeat-cmd "sendHeartbeat localhost" v.avi
```

executes command "sendHeartbeat localhost" every 30 seconds while rendering video "v.avi". Such command sends a heartbeat to the first instance of our watchdog timer.

In summary, our system currently supports the above mentioned reflective (resp. refractive) variables, which

– can be used straightforwardly to report (resp. react after) exceptions in external services—such as mplayer,
– can be used to report performance failures (resp. to express error recovery and resilience engineering strategies in terms of callbacks)
– can monitor (resp. control) compliant FT tools, e.g. watchdogs,
– can be used straightforwardly to implement adaptively redundant data structures, as described in [9]

– can be used as a way to express the monitoring (resp. reactive) components of autonomic systems.

Based on the UDP protocol, which does not require connections to be established and maintained, our approach could hook up to any compliant "private side". This fact may be used to maintain that private side without having to shut down and restart the whole system. For the same reasons, a mobile RR vars service would hook up with the nearest private side, thus providing a foundation for ambient computing services.

Next section discusses the potential of RR vars as a means for the expression of effective and maintainable resilience engineering strategies in the application software.

3 RR vars and Their Potential

We just discussed the public view of RR vars and provided the reader with a few simple examples of their use. In what follows we focus on the use that we are making of our model and what we consider as its true potential.

As we have shown, the memory-based metaphor of RR vars lends itself well to represent, in standard programming languages such as C or C++, complex fault-tolerance mechanisms, with minimal burden for the user. Reflection into standard memory is used to define a homogeneous framework where heterogeneous system properties can be monitored and reasoned upon.

In particular RR vars currently manage

– exception handling (as shown in previous section),
– adaptively redundant data structures (with the approach shown in [9]),
– watchdog timers. For the time being, variable "watchdog" just reflects three states: "STARTED", meaning a watchdog has started and is waiting for a first activation heartbeat; "ACTIVE", that is, the activation message has been received and the watchdog is waiting for the next heartbeat; and "FIRED", meaning that no heartbeat was received during the current period. Refraction just resets the watchdog to the "STARTED" state.

Moreover, RR vars wrap standard tools such as "top" (reporting system summary information) and "iperf" [10] (reporting the end-to-end bandwidth available between two Internet nodes) respectively in reflective variable "cpu" and "bandwidth".

We are currently wrapping a number of fault-tolerance and resilience engineering provisions in our framework of RR vars, including a tool for building restoring organs [11].

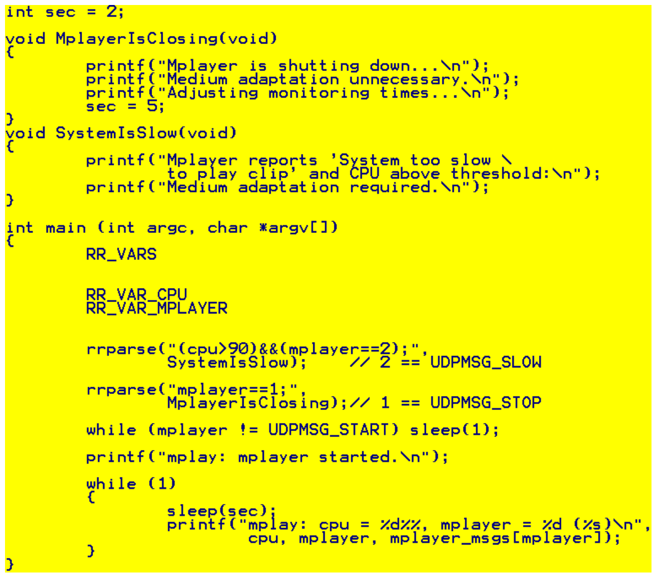
What we consider as the highest potential of our model is that it has a public side, where the functional service is specified by the user in a familiar form, and a private side, separated but not hidden, where the adaptation and error recovery logics are defined. The logic in the private side can be indeed monitored and controlled by means of "meta RR vars", i.e., variables reflecting and refracting on the state of the RR var system. Again, the idea is simple: As mentioned already, the private side of our system gathers information from sensors and detectors and "reifies" this information into corresponding RR vars. Obviously such information is a knowledge treasure to be profited from: Instead of just discarding it, such information is reflected into meta RR vars. Currently we only support one class of meta RR vars—an array of floating point numbers called "alphacount[]". The principle is quite simple: Information produced by error detectors is not discarded but fed into a a fault identification mechanism based on α-count [12]. The current value of this mechanism is available to the user in the form of meta RR var "alphacount[i]", i being an index that identifies the source error detector. This allows *assertions on the validity of the fault model* to be defined, as for instance in the following excerpt:

```
void AssumptionMismatch(void) {
 printf("Wrong fault model assumption caught\n");
}
...
 rrparse("(alphacount[1]>3.0);",
   AssumptionMismatch); // 3.0 = Alpha-count threshold
```

This is a first, still rather raw example of the direction we are currently moving towards with our RR vars: Our future goal is to set up more mature mechanisms such as this so as to allow the hypotheses that have been drawn at design time about the system and its environment to be easily expressed and then asserted during the run-time. Such mechanisms could involve e.g. distributed consensus, or cooperative knowledge extraction among the RR vars servers, or soft computing approaches. Detecting violations of the system and fault models would permit an intelligent management of the dependability strategies and make it possible "to create processes that are robust yet flexible, to monitor and revise risk models, and to use resources proactively in the face of disruptions or ongoing production and economic pressures" [13]—that is, the vision proposed by the pioneers of resilience engineering. For the time being this can be used to issue warnings and, in case of mission-critical or safety-first services, to prohibit the transition from deployment- to run-time in systems that do not pass the test. This could be useful to help prevent failures such as in the Ariane 5 flight 501 [14]. More information on this may be found in [3].

4 Performance and Reification Latency Assessment

We carried out some experiments to evaluate the performance penalty of using RR vars in our development environment (a Windows/XP Pentium M laptop at 2.13GHz with 1GB of RAM running the Cygwin 1.5.25 DLL). Aim of our experiments was also to measure the "reification latency" of our system, that we define here as the time between the detection of an error or a change and the update of the corresponding reflective var. Fault latency (the length of time between the occurrence of a fault and its removal) is clearly influenced by reification latency. In order to reach our objectives, we instrumented both the mplayer and the RR vars "private side" so as to record on a file the system time (the number of milliseconds elapsed since the machine started), as returned by function "timeGetTime"[1]. The same laptop was used to run both programs, which allowed to have a global notion of time. All functions producing messages were disabled. Times were recorded by using a buffered write function, flushing results from main memory to the actual destination on disk only at the end of each run of the experiment. The experiment consisted in launching the two involved codes in two terminals. First, a program defining RR vars "cpu" and "mplayer" is executed. The program also defines two callbacks with two simple guards with arithmetical and Boolean expressions requiring 3 accesses to RR vars (see Fig. 5). After a few seconds, the mplayer is launched so as to render an MPEG-4 clip. After another few seconds, the execution of the mplayer is aborted by typing "control-C" at the keyboard. This triggers an exception that is caught by mplayer. In the exception handling routine, a notification of this event is sent to the RR vars program. The latter catches the event, stores a corresponding state in the memory cells of variable "mplayer", and executes function "rrint", which evaluates all callbacks according to the order of definition.



```
int sec = 2;

void MplayerIsClosing(void)
{
        printf("Mplayer is shutting down...\n");
        printf("Medium adaptation unnecessary.\n");
        printf("Adjusting monitoring times...\n");
        sec = 5;
}
void SystemIsSlow(void)
{
        printf("Mplayer reports 'System too slow \
                to play clip' and CPU above threshold:\n");
        printf("Medium adaptation required.\n");
}

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


        RR_VAR_CPU
        RR_VAR_MPLAYER


        rrparse("(cpu>90)&&(mplayer==2);",
                SystemIsSlow);     // 2 == UDPMSG_SLOW

        rrparse("mplayer==1;",
                MplayerIsClosing);// 1 == UDPMSG_STOP

        while (mplayer != UDPMSG_START) sleep(1);

        printf("mplay: mplayer started.\n");

        while (1)
        {
                sleep(sec);
                printf("mplay: cpu = %d%%, mplayer = %d (%s)\n",
                        cpu, mplayer, mplayer_msgs[mplayer]);
        }
}
```

Fig. 5. The core of the RR vars program used in the experiment reported in Sect. 4.

[1] Reason for using this function, available in a library from Microsoft, was the insufficient resolution of the standard "clock" function in Cygwin.

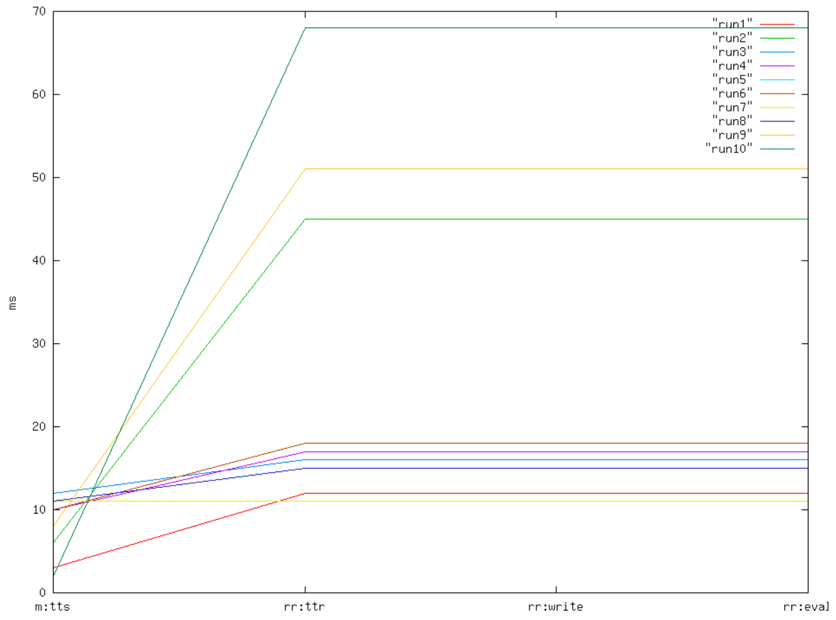


Fig. 6. This picture reports about our experiment to assess performance penalty and reification latency. "m:tts" stands for "time spent by mplayer to notify an exception". After "rr:ttr" milliseconds the RR var system receives that notification, which is stored in the corresponding RR var after time "rr:write". The callback is finally executed, and "rr-eval" is the total amount of milliseconds spent in this phase.

We run this experiment multiple times and recorded the results. The results are summarized in Figure 6, which shows how the bulk of the delay is given by the UDP transfer while the management of RR vars including the execution of the callback takes less than 1ms—the lower bound limit for the times reported by function "timeGetTime".

5 Reflection and Reflective-based Approaches: A Concise Survey

Computational reflection has been defined as the causal connection between a system and a meta-level description representing structural and computational aspects of that system [15]. In other words, reflection is the ability to mirror the feature of a system by creating a causal connection between sub-systems and internal objects. Events experienced by a reflected sub-system trigger events on the object representing that sub-system, and vice-versa. This concept has been adopted in several approaches. Among them it is worth recalling here a few noteworthy cases:

Meta-object protocols (MOPs) [16]. The idea behind MOPs is to "open" the implementation of the run-time executive of an object-oriented language like C++ or Java so that the developer can adopt and program different, custom semantics, adjusting the language to the needs of the user and to the requirements of the environment. By using MOPs, the programmer can modify the behavior of a programming language's features such as methods invocation, object creation and destruction, and member access. The protocols offer the meta-level programmer a representation of a system as a set of *meta-objects*, i.e., objects that represent and reflect properties of "real" objects, that is, those objects that constitute the functional part of the user application. Meta-objects can for instance represent the structure of a class, or object interaction, or the code of an operation.

Introspection. As mentioned in the introduction, data hiding and encapsulation may lead to hidden intelligence. The idea of introspection is to gain access into this hidden complexity, to inspect the black-box structure of programs, and to interpret their meaning through semantic processing, the same way the Semantic Web promises to accomplishes with the data scattered in the Internet. Quoting its author, "introspection is a means that, when applied correctly, can help crack the code of a software and intercept the hidden and encapsulated meaning of the internals of a program". The way to achieve introspection is by instrumenting the software with data collectors producing information available in a form allowing semantic processing. This idea is being used in the Introspector project, which aims at instrumenting the GNU programming tool-chain so as to create a sort of semantic web of all software derived from those tools. The ultimate goal is very ambitious: "To create a super large and extremely dense web of information about the outside world extracted automatically from computer language programs" [17]. Should this become possible, we would be given a tool to reason (and to let *computers* reason on our behalf!) about the dependability characteristics of the application software.

The idea to reflect system properties is also not new: Even the BASIC programming language of the Commodore C64 and the ZX Spectrum used a similar approach through their "peek" and "poke" functions [18]. Other examples include the `/procfs` file system of Linux. Our approach is different in that it provides the user with an extensible framework to collect and interact with heterogeneous resources and services originating within and outside the system boundaries; such resources and services are then reified in the form of common variables or objects.

6 Conclusions

The current degree of complexity backing the ubiquitous software-intensive systems that maintain our societies call for novel structuring techniques able to master that complexity. Achieving truly effective service portability, with smooth and seamless adaptation of both business *and* dependability layers, is a challenge we need to confront with—lest we face the consequences of the fragility of our design assumptions. We believe that one method to meet this challenge is by structuring software so as to allow a transparent "tuning" to environmental conditions and technological dependencies. The approach described in this article proposes one way to achieve this: A separated, open layer manages monitoring and adaptation and makes use of a simple mechanism to export to the application layer the knowledge required to verify, maintain, and possibly adapt, the software services with respect to the current characteristics of the system, the network, and the environment.

Our current research activity is to open and reify the internals of the RR vars "private side" in the form of meta RR vars. We believe that such private side could be an ideal "location" to specify fault model tracking, failure semantics adaptation, resource (re-)optimization, and other important non-functional processes. This could permit to conceive services that analyze the current events and the past history so as to anticipate the detection of potential threats. We have provided evidence to this claim by showing how a meta RR var exporting the value of α-counters [12] could be used to set up assertions on the validity of a system's fault model.

References

1. Intelligent content in FP7 3rd ITC Call. Available online: cordis.europa.eu/ist/kct/eventcall3-in-motion.htm.
2. Randell, B.: System structure for software fault tolerance. IEEE Trans. Software Eng. **1** (June 1975) 220–232
3. De Florio, V.: Software Assumptions Failure Tolerance: Role, Strategies, and Visions, In: Architecting Dependable Systems VII, LNCS 6420, Springer, (2010) 249–272
4. De Florio, V., Blondia, C.: Reflective and refractive variables: A model for effective and maintainable adaptive-and-dependable software. In: Proc. of the 33rd EUROMICRO SEAA Conference, Lübeck, Germany (August 2007)
5. Mplayer — the movie player (2008) Available online: www.mplayerhq.hu/design7/info.html.
6. Mplayer slave mode protocol (2008) Available online: mediacoder.sourceforge.net/wiki/index.php/MPlayer_Slave_Mode_Protocol.
7. De Florio, V. et al.: $\mathcal{RεL}$: A fault tolerance linguistic structure for distributed applications. In: Proc. of ECBS-2002, Lund, Sweden (April 2002)
8. De Florio, V.: A Fault-Tolerance Linguistic Structure for Distributed Applications, Doctoral dissertation, Dept. of Electrical Engineering, University of Leuven, Belgium, ISBN 90-5682-266-7 (October 2000)
9. De Florio, V., Blondia, C.: On the requirements of new software development. International Journal of Business Intelligence and Data Mining **3**(3) (2008)
10. Tirumala, A. et al.: Measuring end-to-end bandwidth with iperf using web100. In: Proc. of the Passive and Active Measurement Workshop. (2003)
11. De Florio, V. et al.: Software tool combining fault masking with user-defined recovery strategies. IEE Proc. Software **145**(6) (Dec. 1998) 203–211

12. Bondavalli, A. et al.: Threshold-based mechanisms to discriminate transient from intermittent faults. IEEE Trans. on Computers **49**(3) (March 2000) 230–245
13. Hollnagel, E., Woods, D.D., Leveson, N.G.: Resilience engineering: Concepts and precepts. Aldershot, UK, Ashgate (2006)
14. Leveson, N.G.: Safeware: Systems Safety and Computers. Addison (1995)
15. Maes, P.: Concepts and experiments in computational reflection. In: Proc. of OOPSLA-87, Orlando, FL (1987) 147–155
16. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. The MIT Press, Cambridge, MA (1991)
17. DuPont, J.M.: Introspector. Available online: introspector.sourceforge.net.
18. Peek and poke (2010) Available online: en.wikipedia.org/wiki/PEEK_and_POKE.