Trading off complexity for expressiveness in programming languages for embedded devices: visions and experiences

Trading off Complexity for Expressiveness in Programming Languages for Embedded Devices: Visions and Experiences

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. When programming resource-scarce embedded smart devices the designer requires both the low-level system programming features of a language such as C and higher level capability typical of a language like Java. The choice of a particular language often implies trade offs between conflicting design goals such as performance, costs, time-to-market, and overheads. The large variety of languages, virtual machines, and translators provides the designer with a dense trade off space, ranging from minimalistic to rich full-fledged approaches, but once a choice is made it is often difficult and tricky for the designer to revise it. In this work we propose a different approach based on the principles of language-oriented programming. A system of light-weighted and modular extensions is proposed as a method to adaptively reshape the target programming language as needed, adding only those application layer features that match the current design goals. By doing so complexity is made transparent, but not hidden: While the programmer can benefit from higher level constructs, the designer and the deployer can deal with modular building blocks each characterized by a certain algorithmic complexity and therefore each accountable for a given share of the overhead. As a result the designer is provided with finer control on the amount of computing resources that are consumed by the run-time executive of the chosen programming language.

1 Introduction

The December 2010 Tiobe Programming Community index [32], ranking programming languages according to their matching rate in several search engines, sets C as the second most popular programming language, barely 2% less than Java. C's object-oriented counterpart C++ is third but quite further away (9.014% vs. 16.076%). Quite remarkably, C scored the top position in April 2010 and was even "programming language of the year" for Tiobe in 2008, exhibiting that is the highest rise in ratings in that year—a notable feat achieved by Java in 2005.

Both quite successful and wide-spread, C and Java represent two extremes of a spectrum of programming paradigms ranging from system-level to service-level development. Interestingly enough, in C complexity is mostly in the application layer, as its run-time is often very small [18]; in Java, on the other hand, non-negligible complexity comes also with a typically rich execution environment (EE). The latter comprises a virtual machine and advanced features e.g. autonomic garbage collection. The only way to trade off the EE complexity for specific services is then by adopting or designing a new EE.

Various EE's are available, developed by third parties to match specific classes of target platforms. Fine-tuning the EE is also possible, e.g. in Eclipse; and of course it is also possible to go for a custom implementation. In general, though, the amount and the nature of the EE complexity is hidden to the programmer and the designer: after all, it is the very same nature of Java as a portable programming language that forbids to exploit such knowledge.

Though transparent, such hidden complexity is known to have an impact on several aspects, including overhead, real-timeliness, deterministic behavior, and security [10]. In particular, when a computer system's non-functional behavior is well defined and part of that system's quality of service—as it is the case e.g. for real-time embedded systems—then any task with unknown algorithmic complexity or exhibiting non-deterministic behavior might simply be unacceptable. As an example, a run-time component autonomically recollecting unused memory, though very useful in itself, often results in asynchronous, unpredicted system activity affecting e.g. the processors and the memory system—including caches. Taking asynchronous tasks such as this into account would impact negatively on the analysis of worst-case execution times and consequently on costs as well. Moreover, the availability of different flavors of the Java EE is likely to bring about assumption failures—as explained in [5].

In what follows we propose an alternative—in a sense, an *opposite*—direction: Instead of stripping functionality from Java to best match a given target platform, we chose to add functionality to C to compensate for lack of expressiveness and linguistic support. More specifically, in our approach, C with its minimalistic run-time executive becomes a foundation on top of which the designer is made able to easily lay a system of modular linguistic extensions. By doing so the above mentioned partitioning of complexity is not statically defined and unchangeable, but rather revisable under the control of the designer. Depending on the desired linguistic features and the overhead permitted by the target platform as well as by mission and cost constraints, our approach allows the programming language to be flexibly reshaped. This is because our approach employs well-defined "complexity containers", each of which is granting a few specific functions and each of which characterized by well-defined complexity and overhead. Syntactic features and EE functions are weaved together under the control of the designer, resulting in bound and known complexity. A dynamic trade off between complexity and expressiveness can then be achieved and possibly revised in later development stages or when the code is reused on a different platform. In principle such

combination of transparent functionality and translucent complexity should also reduce the hazards of unwary reuse of software modules [22].

The structure of this paper is as follows: A comparison with other related approaches is given in Sect. 2. In Sect. 3 we introduce a number of "basic components" implementing respectively syntactic and semantic extensions for context awareness, for autonomic data integrity, and for event management. In Sect. 4 we discuss how we built such components and how they can be dynamically recombined so as to give raise to specific language extensions. Section 5 introduces a case study and its evaluation. Our conclusions are finally produced in Sect. 6.

2 Related Approaches

Modular extensions to programming languages have been the subject of many an investigation, both theoretical and empirical. The most significant and possibly closest *genus* here is given by the family of approaches, which includes e.g. Language Workbenches [17] and Intentional Programming [28], and is collectively known as Language-oriented Programming [33,16] (LoP). In LoP it is argued that current paradigms "force the programmer to think like the computer rather than having the computer think more like the programmer" [16], which makes programming both time consuming and error prone. A considerable distance exists between the conceptual solution of a problem and its representation in an existing computer language. Such distance is called the redundancy of the language in LoP and is also known as syntactical adequacy [7]. The LoP vision is that, the larger this distance, the more programming becomes difficult, time-consuming, unpleasant, and error prone. To remedy this, LoP advocates that programming should not be simply the process of encoding our concepts in some conventional programming language; it should be creating a collection of domain-specific languages (DSLs) each of which specializes in the optimal representation of one of our concepts. Programming becomes then (1) solving a number of such sub-problems of optimal expression and (2) creating a "workflow" that hooks together all the bits and pieces into a coherent execution flow. With the words of one of its pioneers, LoP advocates the ability "to use different languages for each specialized part of the program, all working together coherently" [16]. Each language ideally should offer the least redundant expression of a concept, in a form that is as close as possible to the means a person would use to communicate their concept to another person.

The approach suggested in this paper goes along the very same direction of LoP: We propose to create collections of "little languages" (tiny DSLs providing minimal-redundancy expressions of domain-specific problems) and to use a simple application layer to bind everything together. Next sections show how natural language concepts such as "tell me at anytime what is the current temperature" or "cycle this operation continuously" can be embedded in the programming language as modular, reusable extensions.

Another family of approaches related to ours is given by the quite popular "AspectC" projects, including among others AspectC [1], WeaveC [26], and

AspectC++ [25]. The latter project is particularly relevant in that it was used successfully to program small embedded systems such as the AVR ATmega microcontroller. In the cited paper the authors describe how to represent abstract sensors in AspectC++ in an elegant and cost-effective way. Indeed we witnessed ourselves how expressive and effective aspects can be by coding an adaptive strategy to tolerate performance failures in video rendering applications [29]. In the reported experience we also used an Eclipse plug-in as in [25], but with AspectJ as programming language. Also in this case aspects reify abstract sensors reporting the current value of relevant context figures such as available CPU and bandwidth.

It is worth remarking how aspect languages exhibit minimal redundancy of the language only in specific domains. When confronted with domain-specific problems such as exception handling, also aspects call for dense and cumbersome translations of natural language concepts [24,7].

In what follows we describe our method—a minimalistic implementation of LoP based on so-called "little languages" and simple run-time executives based on Posix threads. The specific difference between mainstream programming language approaches and ours is that, by using modular, reusable DSL extensions, we provide minimally language-redundant translations of natural language concepts. Such translations constitute a composable high-level language rather than a collection of low-level methods.

3 Basic Components

This section introduces three basic components of our approach: Linguistic support to context awareness (Sect. 3.1), adaptive redundancy management (Sect. 3.2), and application-level management of cyclic events (Sect. 3.3). In all three cases the syntactical extension instruments the memory access operations on certain variables.

3.1 Context Awareness Component

Our first component, called Reflective Switchboards (ReS), provides high level linguistic support for context awareness to a host language such as C or C++. Building on top of our reflective variables system [6], ReS provides transparent access to sensors and actuators by means of a source-to-source translator and a number of run-time components. A description of the ReS features and its components constitutes the rest of this section.

The idea behind ReS is best described by considering Fig. 1 and Fig. 2. In the former picture four components are displayed: the user application (UA), a Callback Interpreter (CI), a server thread (ST), and a sensor driver (SD). The CI is unique for each user application, while instances of ST and SD are executed for specific families of physical sensors—e.g., operating system-specific or network layer-specific. In Fig. 1 the UA makes use of a single sensor called **V**. UA executes a simple domain-specific task, which could be expressed in

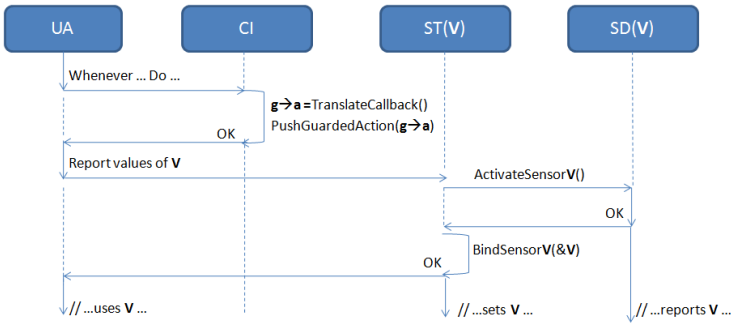

Fig. 1. Sequence diagram of the initialization phase of Reflective Switchboards.

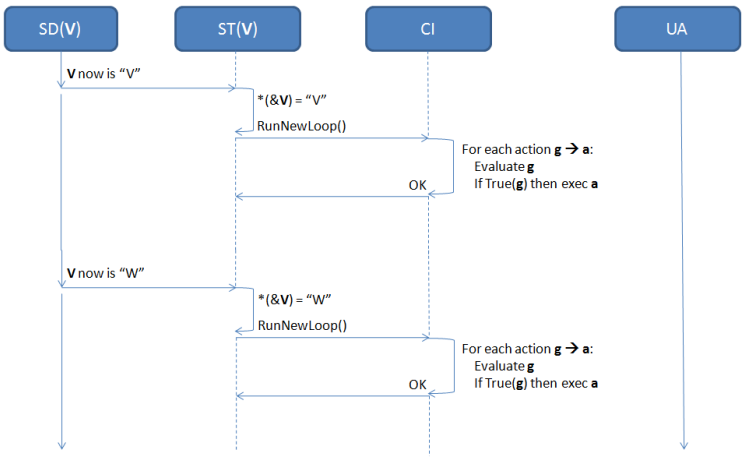

Fig. 2. Sequence diagram of the steady state of Reflective Switchboards.

natural language e.g. as follows: "Keep me posted about the running values of **V**, and execute this maintenance task whenever **V** reaches a certain value". A simple "little language" (the popular name for DSLs coded with traditional tools Lex and YACC [23]) makes it possible to code expressions such as the above with very limited redundancy of the language. As an example, Fig. 3 shows a ReS program that (1) continuously updates the content of variable int cpu with an integer approximating the current percentage of CPU being used, and (2) automatically calls function largerThan10 each and every time condition "(cpu>10);" becomes true.

"Whenever *condition* do *action*" is realized by methods RR_VARS and rrparse (char *guard, int (action*)(void)). The former launches the CI as a Posix thread and performs some initializations. Method rrparse then requests the CI to register a new *guarded action*—that is, an action whose execution is conditioned by the validity of an arithmetic expression. The first argument of rrparse is such

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

int     largerThan10 () {
        printf("CI:largerThan10: cpu == %d\n", cpu);
}

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

        RR_VAR_CPU

        rrparse("(cpu>10);", largerThan10);

        while (1) {
                printf("UA:cpu == %d\n", cpu);
                sleep(1);
        }
}
```

Fig. 3. Reflective Switchboards: A simple coding example.

expression—a string that is parsed by the CI and translated into a simple and compact pseudo-code. The second argument is the action associated with the guard. The pseudo-code for the guarded action is then pushed onto an internal stack. This is represented on top of Fig. 1.

The "keep me posted about **V**" part is realized by method RR_VAR_**V**. Such method initializes the run-time executive for reflective variable **V**. This executes a corresponding ST as a Posix thread. As described in Fig. 1, ST then activates and binds to its associated sensor driver SD. By doing so, ST requests to be notified by SD for each context change pertaining sensor(s) **V**.

After the above initializations, ReS enters its "steady state", depicted in Fig. 2: While the UA proceeds undisturbed, actions are triggered on the ReS components when the SD's reports a new value of their sensors. In this case the ST updates the memory cells of the corresponding "sensor variable" and then requests the CI to interprete the pseudo-code stored in its stack. For each guarded action $\mathbf{g} \rightarrow \mathbf{a}$, guard **g** is evaluated and, when found true, the CI executes method **a**.

ReS also uses variables as access points to actuators by intercepting all the assignments in which the left value is a legal "actuator variable". After the assignment, the translator simply adds a method to communicate the new value to the corresponding actuator.

name	type	class	short description
cpu	int	sensor	CPU usage (%)
bandwidth	int	sensor	bandwidth available between localhost and a TCP remote host (Mbps)
mplayer	int	sensor	status of instance of the mplayer video player
mplayer	int	actuator	sets properties of an mplayer instance
watchdog	int	sensor	status of an instance of a watchdog timer thread
watchdog	int	actuator	controls an instance of a watchdog timer thread
linkbeacons	lb_t []	sensors	MAC beacons received from a network peer in a MANET
linkrates	lr_t []	sensors	estimated IP bandwidth between localhost and a MANET peer.

Table 1. Currently available sensor and actuator variables and arrays.

Table 1 briefly lists the currently available sensor and actuator variables. It is worth discussing two special variables—linkbeacons and linkrates. In this case yet another "little language" was designed to allow the definition of dynami-

```
/* File switchboard.c
 * created/modified on Mon Jul 26 14:26:26 CEST 2010
 */

#include <stdio.h>
#include "reflection.h"
#include "rcode.h"
#include "rrvars_init.h"

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

        RR_VAR_LINKRATES    /* defines linkrates[MACaddress] */

        RR_VAR_LINKBEACONS  /* defines linkbeacons[MACaddress] */

        RR_VAR_IEEE802      /* defines char ieee802_11 (holds either 'a' or 'g') */

        while (1) {
                /* each new cycle, do: */
                sleep(LINK_QUERY_PERIOD);

                /* for each peer in proximity: retrieve its MAC address (a string) */
                while ( (mac_address = (char*)anext()) ) {

                        /* compute the routing metric as a function of
                           linkbeacons, linkrates, and the wireless LAN
                           protocol (either IEEE 802.11a or IEEE 802.11g)
                         */
                        SetRoutingMetric(mac_address, ieee802_11,
                                linkbeacons[mac_address], linkrates[mac_address]);
                }

                /* zeroes the set of peers in proximity */
                arewind();
        }
}
```

Fig. 4. ReS code for MAC-IP cross-layer optimizations.

cally growing arrays of C structures representing the properties of the nearest neighboring peers in a Mobile Ad-hoc Network (MANET). The first variable, linkbeacons, is an array of sensor variables reporting Medium Access Control layer properties of MANET peers. A new object comes to life dynamically each time a new peer comes in proximity. When a peer node falls out of range, the corresponding object becomes "stale" until its node becomes reachable again. The above mentioned "little language" provides syntactic sugar to allow the linkbeacons array to be addressed by strings representing the MAC address of peer nodes. Array linkbeacons reflects a number of properties, including the number of MAC beacons received by a peer node during the last "observation period" (defined in our experiments as sixty seconds) or the number of periods elapsed without receiving at least one beacon from a certain node.

Similarly, array linkrates returns Network layer properties of peers in proximity—specifically, it returns the estimated bandwidth between the current node and the one whose address is being used as an index.

The above mentioned arrays are currently being used in our research group to set up cross-layer optimizations such as MAC-aware IP routing in mobile ad-hoc networks.

The program used to steer this cross layer optimization is quite simple: Every new observation cycle, the program retrieves the MAC addresses of the peers in proximity via a simple function call (anext) and then requests to adjust the routing metric using the above mentioned arrays. The actual adjustments to the routing protocol are carried out through a Click [19] script.

As can be seen by the above examples, despite C being a relatively simple programming language, the modular addition of linguistic features covering domain-specific sub-problems does lead to a new and more powerful language characterized by a better linguistic redundancy.

```c
#include <stdio.h>

int main(void)
{
        int a;
        redundant int myProtectedInteger;


        while (1) {
                sleep(1);

                myProtectedInteger = 1;


                a = myProtectedInteger;


        }

}
```

```c
#include <stdio.h>
// bookkeeping...
int main(void)
{
        int a;
        /* redundant */  int myProtectedInteger;

        // ...bookkeeping

        while (1) {
                sleep(1);

                myProtectedInteger = 1;
RedundantAssign_int(&myProtectedInteger);  // multiplex

                a =
RedundantRead_int(&myProtectedInteger);    // de-multiplex via voting

        }

}

// bookkeeping...
```

Fig. 5. A simple example of use of redundant variables. An "extended C" source code that accesses a redundant variable (left-hand image) and an excerpt from the translation in plain C (right-hand picture) are displayed.

3.2 Adaptive Redundancy Component

Another important service that is typically missing in conventional programming languages such as C is transparent data replication. As embedded systems are typically streamlined platforms in which resources are kept to a minimum in order to contain e.g. costs and power consumption, hardware support to memory error detection is often missing. When such embedded systems are mission critical and subjected to unbound levels of electro-magnetic interference (EMI), it is not uncommon to suffer from transient failures. As an example, several Toyota models recently experienced unintended acceleration and brake problems. Despite Toyota's official communications stating otherwise, many researchers and consultants are suggesting this to be just another case of EMI-triggered failures [31,15,34]. More definitive evidence exists that EMI produced by personal electronic devices does affect electronic controls in modern aircrafts [27], as it is the case for control apparatuses operating in proximity of electrical energy stations as well [11].

Whenever EMI causes unchecked memory corruption, a common strategy is to use redundant data structures [30]: Mission-critical data structures are then "protected" by replication and voting and through redoing [11].

Our adaptive redundancy component is yet another "little language" that allows the user to tag certain C variables as being "redundant". A run-time executive then transparently replicates those variables according to some policy (for instance, in separate "banks") and then catches memory accesses to those variables. Write accesses are multiplexed and store their "rvalues" [18] in each replica, while read accesses are demultiplexed via a majority voting scheme. Figure 5 summarizes this via a simple example.

In some cases, for instance when the application is cyclic and constantly re-executed as in [12], the behavior of the voting scheme can be monitored and provides an estimation of the probability of failure: As an example, if the errors induced by EMI are affecting a larger and larger amount of replicas, then this

```
1.  /* declarations */
    TOM *tom; timeout_t t1, t2;
    int PeriodicMethod1(TOM*), PeriodicMethod2(TOM*);

2.  /* definitions */
    tom ← tom_init();
    tom_declare(&t1, TOM_CYCLIC, TOM_SET_ENABLE, TIMEOUT1, SUBID1, DEADLINE1);
    tom_set_action(&t1, PeriodicMethod1);
    tom_declare(&t2, TOM_CYCLIC, TOM_SET_ENABLE, TIMEOUT2, SUBID2, DEADLINE2);
    tom_set_action(&t2, PeriodicMethod2);

3.  /* insertion */
    tom_insert(tom, &t1), tom_insert(tom, &t2);

4.  /* control */
    tom_disable(tom, &t2);
    tom_set_deadline(&t2, NEW_DEADLINE2);
    tom_renew(tom, &t2);
    tom_delete(tom, &t1);
```

Table 2. Example of usage of the TOM time-out management class. In **1.** a time-out list pointer and two time-out objects are declared, together with two alarm functions. In **2.** the time-out list and the time-outs are initialized. Insertion is carried out in **3.** In **4.**, time-out t2 is disabled; its deadline is changed; t2 is restarted; and finally, time-out t1 is deleted.

can be interpreted as a risk that the voting scheme will fail in the near future due to the impossibility to achieve a majority. Detecting this and assessing the corresponding risk of voting failure allows the amount of replicas to be transparently and autonomically adjusted, e.g. as described in [4]. Such a run-time time scheme could also be complemented with compile-time explorations and optimizations as discussed e.g. in [20].

3.3 Cyclic Methods Component

As observed in [8], natural language expressions such as repeat periodically, at time t send heartbeat, at time t check whether message m has arrived, or upon receive, are often used by researchers to describe e.g. distributed protocols. The lack of those constructs in a language such as C led us in the past to implement another extension in the form of a library of alarm management methods. Such library allows user-specified function calls to be postponed by a given amount of time. In [8] we showed how this permits to implement the above natural language expressions by converting time-based events into message arrivals or signal invocations. In the cited paper we also proposed some preliminary "syntactic sugar" to ease up the use of our library. Table 2 is a simple example of how our time-out methods could be used e.g. to define and control two "cyclic methods," i.e., functions that are executed by the run-time system every new user-defined cycle.

In the experience reported in this paper we capitalized on our previous achievements and designed yet another "little language" to facilitate the definition of cyclic methods.

Table 3 shows the syntax of our extension. In a nutshell, the extension allows the user to specify a dummy member, Cycle, for those methods that have been

```
1.  /* declarations */
    cyclic_t int PeriodicMethod1(TOM*);
    cyclic_t int PeriodicMethod2(TOM*);

2.  /* definitions: unnecessary */

3.  /* insertion */
    PeriodicMethod1.Cycle = DEADLINE1;
    PeriodicMethod2.Cycle = DEADLINE2;

4.  /* control */
    PeriodicMethod2.Cycle = NEW_DEADLINE2;
    PeriodicMethod1.Cycle = 0;
```

Table 3. The new syntax for the example of Table 2. Two simple constructs are introduced—bold typeface is used to highlight their occurrences in this example.

tagged as cyclic_t. Every Cycle milliseconds the extension executes a new instance of the corresponding method—irrespective of the fact that previous instances are still running or otherwise.

4 Putting Things Together

In previous section we introduced several domain-specific languages each of which augments plain C with extra features. In the rest of this section we briefly report on some general design principles as well as on our current approach to combine together those domain-specific languages.

The key principle of our approach is the use of a set of independent and interchangeable linguistic extensions, each addressing a specific problem domain. Extensions augment a same base language (in the case at hand, C) and in the face of local syntax errors, assume that the current line being parsed will be treated by one of the following extensions. In other words, what would normally be regarded as severe errors is treated as a warning and flushed verbatim on the standard output stream. Obviously such strategy is far from ideal, as it shifts all possible syntax checks down to C compile time. A better strategy would be to let the system guess which extensions to apply based on the syntactic "signature" of each input fragment. A simpler alternative would be to use start conditions, as suggested for lexical analysis in the now classical article [21].

Our extensions are coded in C with Lex and YACC [23] and make use of some simple Bash shell scripts. Some extensions were originally developed on a Windows/Cygwin environment while more recent ones have been devised on Ubuntu Linux. All extensions run consistently on both environments.

Each of our extensions is uniquely identified by an extension identifier—a string in the form "cpm://e/v", where e and v are two strings representing respectively the extension and its version number.

Our current implementation makes use of a simplistic strategy to assemble components, requiring the user to manually insert or remove the translators corresponding to each extension. In particular the user is responsible for choosing the order of application of the various extensions. Figure 6 shows the script

```
#!/bin/bash

SERVERDEFS="-DLINUX -DTIMING"
SERVERDEFS="-DLINUX"

WINTIME=/usr/lib/w32api/libwinmm.a
WINTIME=

OUT=`basename $1 .c`
RAN="/tmp/${RANDOM}"

cat $1                      |   \
                                \
    redundancy              |   \
                                \
    refractive              |   \
                                \
    array                   |   \
                                \
cat > ${RAN}.c


gcc -I. ${SERVERDEFS} -g -c server.c
gcc -I. -DREFLECTIVE -O -c watchdog1.c
gcc -I. -DREFLECTIVE -O -c watchdoglib.c

gcc -I. -O -o ${OUT} ${RAN}.c interp.tab.c lex.yy.c \
   server.o watchdog1.o watchdoglib.o liba.a ${WINTIME} \
   -lfl -ly -lpthread -lrt

echo "Compilation ended. Temporary file is ${RAN}.c"
```

Fig. 6. A Bash script is used to selectively augment C with our modular extensions.

that we use for this. A Unix pipeline is used to represent the assembling process. Components of this pipeline are in this case redundancy, which manages the extension described in Sect. 3.2, followed by refractive, which adds operator overloading capabilities to context variables. The last stage of the pipeline is in this case array, which produces the dynamic array extension described in Sect. 3.1.

It is worth pointing out that each extension publishes its extension identifier by appending it to a context variable, a string called extensions_pipeline, e.g. "cmp://redundancy/1.1; cmp://refractive/0.5;cmp://array/0.5". By inspecting this variable the program is granted access to knowledge representing the algorithmic complexity and the features of its current execution environment.

As described in revious section, extensions make use of Posix threads defined in libraries and ancillary programs. Such ancillary code (and the ensuing complexity) is then selectively loaded on demand during the linking phase of the final compilation.

5 Evaluation

In order to analyze the performance of our method we shall focus on a particular case study: The design of a simple software watchdog timer (WDT). This particular choice stemmed from a number of reasons:

– First of all, WDT provides a well known and widespread "dependable design pattern" that is often used in either hardware or software in mission-critical

embedded systems, as it provides a cost-effective method to detect performance failures [2].
- Secondly, a WDT is a real-time software. This means that it requires context awareness of time. This makes it suitable for being developed with the extension described in Sect. 3.1.
- Moreover, a WDT is a cyclic application. Linguistic constructs such as the one described in Sect. 3.3 allow a concise and lean implementation of cyclic behaviors.
- Furthermore, WDT is a mission-critical tool: A faulty or hacked WDT may cause a healthy watched component to be stopped; this in turn may severely impact on availability. Protecting a WDT's control variables could help preventing faults or detecting security leaks. The extension described in Sect. 3.2 may provide to some extent such protection.
- Finally, the choice of focusing on a WDT allows us to leverage on our past research: In [9] we introduced a domain-specific language that permits to define WDTs in a few lines of code. This allows an easy comparison of the amount of the expressiveness of the two approaches.

As briefly mentioned in Table 1, a sensor/actuator variable called watchdog reflects the state of a WDT. States are reified as integers greater than -4. Negative values represent conditions, i.e. either of:

WD_STARTED, meaning that a WDT task is running and waiting for an activation message.
WD_ACTIVE, stating that WDT has been activated and now expects periodical heartbeats from a watched task.
WD_FIRED, that is, no heartbeat was received during the last cycle—the WDT "fired".
WD_END, meaning that the WDT task has ended.

Positive values represent how many times the WDT reset its timer without "firing."

That same variable, watchdog, is also an actuator, as it controls the operation of the WDT: Writing a value into it restarts a fired WDT.

Being so crucial to the performance of the WDT, we decided to protect watchdog by making it redundant. To do so we declared it as extern redundant_t int watchdog. Using the extern keyword was necessary in order to change the *definition* of watchdog into a *declaration* [18], as the context aware component defines watchdog already. In other words this is a practical example of two non-orthogonal extensions.

Figure 7 describes our prototypic implementation. The code uses all three extensions reported in Sect. 3. It executes as follows: A WDT thread is transparently spawned. Such thread is monitored and controlled via variable watchdog. Redundant copies of this variable are used to mitigate the effect of transient faults or security leaks affecting memory. The code then uses our cyclic methods extension to call periodically a management function. Such function in turn

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

        /* this transparently launches the watchdog timer;
         * from now on, it can be controlled and monitored
         * via context variable 'watchdog'
         */
        RR_VAR_WATCHDOG

        /* this declares a cyclic method for the
         * periodic management of the watchdog
         */
        cyclic_t int wdt_management(TOM*);

        /* this requests to protect variable 'watchdog'
         */
        extern redundant_t int watchdog;

        /* every 2 seconds, do execute wdt_management :
         */
        wdt_management.Cycle = 2000;

        /* wait until both watchdog some time, then end */
        while (watchdog != END || wdt_management.Cycle != 0)
                sleep(1);
        wdt_management.Cycle = 0;
        return 0;
}

int wdt_management(TOM *tom) {
        if (watchdog == WD_FIRED)
                /* restart WDT by setting 'watchdog' */
                watchdog = 0;
        else if (watchdog == WD_END)
                /* disable cyclic behavior */
                wdt_management.Cycle = 0;
}
```

Fig. 7. Excerpt from the code of the WDT.

makes use of two of our extensions—for instance, the WDT is restarted simply by writing a certain value in watchdog.

Our evaluation is based on a qualitative estimation of the redundancy of the resulting programming language (see Sect. 2). In other words, we are interested here in the amount of expressiveness of our language—how adequate and concise the language proved to be with respect to other existing languages. A rough estimation of this syntactical adequacy [7] may be done by measuring the required amount of lines of code (LoC).

If we restrict ourselves to the above discussed WDT we can observe how in this special case the programmer is required to produce an amount of lines of code notably smaller than what normally expected for a comparable C program.

Such amount is slightly greater than in the case treated in [9], where a C implementation of a WDT is produced from the high level domain-specific language Ariel [3,13]. It must be remarked though that the WDT produced by Ariel is much simpler than the one presented here—e.g. it is non-redundant and context agnostic.

6 Conclusions

We have introduced an approach inspired by LoP that linearly augments the features of a programming language by injecting a set of light-weighted extensions. Depending on the desired features and the overhead and behaviors permitted by the target platform and cost constraints, our approach allows the programming

language to be flexibly reshaped. This is because it employs well-defined "complexity containers", each of which grants limited domain-specific functions and is characterized by well-defined complexity and overheads. By doing so, complexity is made transparent but it is not hidden: While the programmer can benefit of high level constructs, the designer and the deployer can deal with modular building blocks each characterized by a certain algorithmic complexity and therefore each accountable for a certain overhead. A mechanism allows each building block to be identified, thus avoiding mismatches between expected and provided features. At the same time, this provides the designer with finer control over the amount of resources required by the run-time executive of the resulting language, as well as over its resulting algorithmic complexity.

We observe how our approach allows the designer to deal with a number of separated, limited problems instead of a single, larger problem. From the divide-and-conquer design principle we then conjecture a lesser complexity for our approach. Moreover, in our case the designer is aware and in full control of the amount and the nature of the complexity he/she is adding to C.

A full-fledged comparison between a library-based approach such as [14] and ours will be the subject of future research.

References

1. Y. Coady et al. Using aspectc to improve the modularity of path-specific customization in operating system code. In *Proc. of FSE-9*, pages 88–98, 2001. 2
2. F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991. 5
3. V. De Florio. *A Fault-Tolerance Linguistic Structure for Distributed Applications*. PhD thesis, Dept. of Elec. Eng., Univ. of Leuven, Belgium, Oct. 2000. 5
4. V. De Florio. Cost-effective software reliability through autonomic tuning of system resources. In *Proc. of the Applied Reliability Symposium, Europe*, April 2010. 3.2
5. V. De Florio. Software assumptions failure tolerance: Role, strategies, and visions. In *Architecting Dependable Systems VII*, vol. 6420 of *LNCS*, pages 249–272. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-17245-8_11. 1
6. V. De Florio and C. Blondia. Reflective and refractive variables: A model for effective and maintainable adaptive-and-dependable software. In *Proc. of SEAA 2007*, Lübeck, Germany, August 2007. 3.1
7. V. De Florio and C. Blondia. A survey of linguistic structures for application-level fault-tolerance. *ACM Computing Surveys*, 40(2), April 2008. 2, 5
8. V. De Florio and C. Blondia. Design tool to express failure detection protocols. *IET Software*, 4(2):119–133, 2010. 3.3
9. V. De Florio, S. Donatelli, and G. Dondossola. Flexible development of dependability services: An experience derived from energy automation systems. In *Proc. of ECBS 2002*, Lund, Sweden, April 2002. IEEE Comp. Soc. Press. 5, 5
10. B. De Win, T. Goovaerts, W. Joosen, P. Philippaerts, F. Piessens, and Y. Younan. *Middleware for Network Eccentric and Mobile Applications*, chapter Security Middleware for Mobile Applications, pages 265–284. Springer, 2009. 1
11. G. Deconinck et al. Stable memory in substation automation: a case study. In *Proc. of FTCS-28*, pages 452–457, Munich, Germany, June 1998. 3.2

12. G. Deconinck et al. Integrating recovery strategies into a primary substation automation system. In *Proc. of DSN-2003*. 3.2
13. G. Deconinck et al. A software library, a control backbone and user-specified recovery strategies to enhance the dependability of embedded systems. In *Proc. of Euromicro '99*, vol. 2, pages 98–104, Milan, Italy, Sept. 1999. 5
14. G. Deconinck et al. Industrial embedded HPC applications. *Supercomputer*, 13(3–4):23–44, 1997. 6
15. I. Dividend. Toyota's Electromagnetic Interference Troubles: Just the Tip of the Iceberg, 2 2010. Available online: http://seekingalpha.com/article/187021-toyota-s-electromagnetic-interference-troubles-just-the-tip-of-the-iceberg. 3.2
16. S. Dmitriev. Language oriented programming: The next programming paradigm. *onBoard*, November 2004. 2
17. M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005. Available online: http://www.martinfowler.com/articles/languageWorkbench.html. 2
18. B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 1988. 1, 3.2, 5
19. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000. 3.1
20. M. Leeman et al. Automated dynamic memory data type implementation exploration and optimization. In *Proc. of ISVLSI'03*, Washington, DC, 2003. 3.2
21. M. E. Lesk and E. Schmidt. Lex – a Lexical Analyzer Generator. Technical report, Bell Laboratories, 1975. CS Technical Report No. 39. 4
22. N. G. Leveson. *Safeware: Systems Safety and Computers*. Addison, 1995. 1
23. J. Levine et al. *Lex & YACC*. O'Reilly, 2nd ed., 1992. 3.1, 4
24. M. Lippert and C. Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proc. of ICSE'2000*, Limmerick, Ireland, June 2000. 2
25. D. Lohmann and O. Spinczyk. Developing embedded software product lines with aspectc++. In *OOPSLA '06*, pages 740–742, New York, NY, 2006. 2
26. I. A. Nagy, R. E. van, and D. P. van der. An overview of mirjam and weavec. In *Ideals: evolvability of software-intensive high-tech systems*, pages 69–86. Embedded Systems Institute, Eindhoven, 2007. 2
27. T. S. Perry and L. Geppert. Do portable electronics endanger flight? The evidence mounts. *IEEE Spectrum*, 33(9):26–33, 1996. 3.2
28. C. Simonyi. Is programming a form of encryption?, 2005. Available online: http://blog.intentsoft.com/ intentional_software/2005/04/dummy_post_1.html. 2
29. H. Sun, V. De Florio, N. Gui, and C. Blondia: Adaptation strategies for performance failure avoidance. In *Proc. of SSIRI 2009*, Shanghai, July 2009. 2
30. D. J. Taylor at al. Redundancy in data structures: Improving software fault tolerance. *IEEE Trans. on Soft. Eng.*, 6(6):585–594, Nov. 1980. 3.2
31. P. Tekla. Toyota's Troubles Put EMI Back Into The Spotlight, 2 2010. Available online: http://spectrum.ieee.org/tech-talk/green-tech/advanced-cars/toyotas-troubles-put-emi-back-into-the-spotlight. 3.2
32. Tiobe. TIOBE Programming Community Index for July 2010, 7 2010. Available online: http://www.tiobe.com/index.php/content/paperinfo/ tpci/index.html. 1
33. M. P. Ward. Language-oriented programming. *Software—Concepts and Tools*, 15(4):147–161, 1994. 2
34. C. Weiss. Consultants Point to Electromagnetic Interference In Toyota Problems, 3 2010. Available online: http://motorcrave.com/consultants-point-to-electromagnetic-interference-in-toyota-problems/5927. 3.2