Component-Based Real-Time Operating System for Embedded Applications

Frédéric Loiret[1], Juan Navas[2], Jean-Philippe Babau[3] and Olivier Lobry[2]

[1] INRIA-Lille, Nord Europe, Project ADAM
USTL-LIFL CNRS UMR 8022, France
frederic.loiret@inria.fr
[2] Orange Labs
{juanfernando.navasmantilla | olivier.lobry}@orange-ftgroup.com
[3] Université Européenne de Bretagne
jean-philippe.babau@univ-brest.fr

Abstract. As embedded systems must constantly integrate new functionalities, their developement cycles must be based on high-level abstractions, making the software design more flexible. CBSE provides an approach to these new requirements. However, low-level services provided by operating systems are an integral part of embedded applications, furthermore deployed on resource-limited devices. Therefore, the expected benefits of CBSE must not impact on the constraints imposed by the targetted domain, such as memory footprint, energy consumption, and execution time. In this paper, we present the componentization of a legacy industry-established Real-Time Operating System, and how component-based applications are built on top of it. We use the Think framework that allows to produce flexible systems while paying for flexibility only where desired. Performed experimentions show that the induced overhead is negligeable.

1 Introduction

Until recently, embedded systems were defined as resource constrained, dedicated and closed computing systems buried within an electro-mechanical structure they interact with. Much of the embedded systems are also real-time systems, i.e. systems in which *temporal predictability* is a key issue.

While limited resources constraint remains, paradigms like *Everyware* [13] boosted embedded systems development; growing demand in embedded devices market imposes new preoccupations such as Time to Market, industrial standards compliance, and adaptability to dynamic operation context. Consequently, embedded systems can no longer be closed and specific-task systems, since they must adapt themselves to the surrounding environment. System's design, development, deployment, and maintenance complexity has increased and traditional code-centric methodologies are no longer suitable, not only at application level, but also at operating system level since the latter is integral part of embedded applications.

Component-Based Software Engineering (CBSE) [27] addresses several aspects in today's embedded systems development. Systems are designed by assembling *software* and *system* components [15] and may evolve at execution time through dynamic reconfiguration [24]. CBSE is particularly useful in handling the multiple variants of a same product line, as components can be treated as independent, arbitrary fine-grained entities to be deployed in heterogeneus devices. Several approaches, some of them inspired by CBSE paradigm, have been applied to obtain similar design and run-time benefits in the resources-constrained systems domain:

- Virtual machines and bytecode/script interpreters [17, 22, 30] allow run-time flexibility in very resource-limited platforms such as sensor networks *motes*. However, peformance penalty increases as virtual machines are designed to be less application-specific, making this approach not scalable for more general real-time embedded systems product lines.

- Real-time embedded systems can be built by compiling component-based architecture descriptions [29, 18]. By this way, design-time CBSE benefits are preserved but the notion of *component* disappears at run-time, failing to take advantage of a global CBSE approach. Also, existing framework programming models make difficult the introduction of real-time specific concepts and prior developements.

- Component-based versions of executive parts can be integrated to real-time component-based applications that means to assemble OS-related services or to componentize existing RTOS source code. This approach globally preserves CBSE benefits, but may induce a significant overhead concerning critical metrics of embedded systems such as memory footprint, real-time responsiveness and execution time. However, recent studies [20] show that it is possible to control and to limit the possible overheads caused by flexibility support.

The contribution of this paper follows this last approach. It presents how an existing real-time operating system, μC/OS-II, is componentized, as well as the considerations that shall be respected in this re-engineering task. We use the THINK component framework [8, 4], an implementation of the FRACTAL component model [6] that fullfills the constraints of embedded-systems. This component model is used in a homogeneous way at both application and operating system levels. A THINK component is an entity that is preserved during the whole system life cycle with minimal performance overhead.

The paper is organized as follows: Section 2 briefly describes μC/OS-II RTOS, THINK framework and its underlying FRACTAL component model. Section 3 identifies challenges to be faced in a RTOS componentization task; Section 4 details the componentization scheme. Section 5 presents a use-case benchmark that demonstrates the benefits of our approach. Section 6 describes related work. Section 7 concludes the paper.

2 Background

This section presents the features of the μC/OS-II kernel and the THINK component framework. We stress in Section 2.3 the reasons that led us to conduct our work grounded on these two projects.

2.1 The μC/OS-II RTOS

μC/OS-II is a preemptive, real-time multitasking kernel for microprocessors and microcontrollers. It is implemented in ANSI C and certified by the FAA[4] for use in software intended to be deployed in avionics equipment. It has been massively used in many embedded and safety critical systems products worldwide.

The main services provided by μC as an RTOS are sketched out in Fig. 1, which gives the module structure of the kernel distribution. The main services are implemented within the `Core`, the `Task` and the `Port` modules. The latter implements the hardware-dependent services.

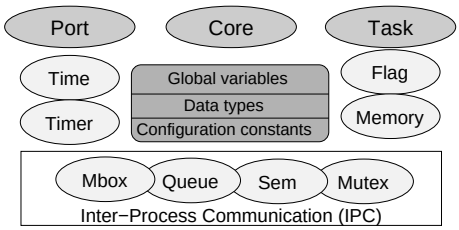


Fig. 1. μC Modules.

μC/OS is implemented as a monolithic kernel, i.e. it is built from a number of *functions* that share common *global variables* and *data types* (such as *task control block*, *event control block*, etc).

It is a highly configurable kernel, whose configuration relies on more than 70 parameters. Since the kernel is provided in source form, configurability is done via conditional compilation at precompilation time, based on `#define` constants. μC allows to scale down, the main objective being to reduce the memory footprint of the final executable (up to several KBytes, depending on the processor). Hence, it is possible to avoid code generation of non required services, or to reduce the size of data structures used by the kernel. Several parameters allow developers to configure essential properties of the kernel, e.g. the *tick* frequency.

The execution time for most of these services is both constant and deterministic, which is a compulsory requirement for real-time systems in order to avoid unpredictable *kernel jitter* [3].

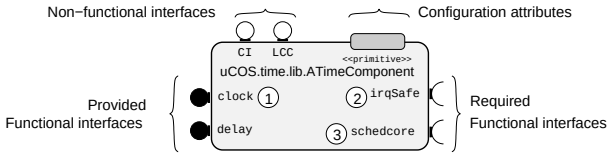
[4] Federal Aviation Administration.

For a full description of the features, the design and the internals of μC/OS-II, we refer the interested reader to the book [2][5].

2.2 The Think component framework

THINK is an implementation of the FRACTAL component model that aims to take into account the specific constraints of embedded systems development. The FRACTAL specifications [6] define a hierarchical, reflective and general-purpose component model. A *component* definition exports *functional interfaces* (provided or required), configuration *attributes*, and may also provide *non-functional interfaces* implementing introspection and architectural reconfiguration services at run-time.

The THINK framework allows developers to build embedded systems made out of FRACTAL components. A system architecture is described using an Architecture Description Language (ADL), interfaces are defined using an Interface Description Language (IDL). The code that implements the method of server interfaces is written in regular C (or wrapped assembler language) where ADL symbols are represented by convenient C symbols. The mapping between ADL symbols and C symbols can be specified using annotations in commentary section of the C files which facilitates the encapsulation of legacy code. An example of a THINK component definition is given in Figure 2[6].




```
component uCOS.time.lib.ATimeComponent {

 // Public interfaces provided
 provides uCOS.time.api.ClockCommon as clock
 provides uCOS.time.api.DelayCommon as delay

 // Internal interfaces required
 requires uCOS.task.api.SchedCoreInt
    as schedcore
 requires irq.api.IrqSafe as irqsafe
 // Attributes definition
 attribute INT16U OS_TICKS_PER_SEC = 100

 content uCOS.time.lib.os_time
}
```

(1) Interface clock

```
public interface uCOS.time.api.ClockCommon
  INT32U OSTimeGet (void);
  void OSTimeSet (INT32U ticks);
```

(2) Interface irqSafe

```
internal interface irq.api.IrqSafe
  void OS_ENTER_CRITICAL (void);
  void OS_EXIT_CRITICAL (void);
```

(3) Interface schedcore

```
internal interface uCOS.task.api.SchedCoreInt
  void OS_Sched (void);
```

Fig. 2. Definition of a primitive component: ADL (left part) and IDL (right part).

[5] In this paper, we refer to the version 2.86 of μC/OS.

[6] By convention, the *functional interfaces* are graphically placed on the left and right sides of the component, the *non-functional interfaces* on the top side.

The THINK compiler maps architectural elements to C variables in implementation code, transforms existing functional code and produces meta-data and implementation of non-functional interfaces. The meta-data typically allow to retrieve an attribute, the descriptor of a bound-to server interface of the component context of a bound-to component. The resulting C code is then passed to traditional C compilers and linkers to generate the final binary file.

Since applying the component paradigm can easily impacts on performances, the THINK compiler provides tools to produce, from a same architectural description, different binary images with different performance versus flexibilty tradeoffs. Architectural elements can be tagged with flexibility-oriented properties in an Aspect-Oriented-Programming manner. These properties will be interpreted by the compiler to generate an optimized binary image that only embeds flexibility where actually desired. For example:

- The address of the context of a *single* component is known at compile-time and need not be passed in calls;
- A *constant* attribute is implemented as a compile-time constant and no meta-data is generated;
- A *static* binding does not generate meta-data and calls to the corresponding client interface are implemented as direct function calls;
- The implementation code of the server interface can even be inlined in the caller;
- no meta-data is generated for a server interface to which all bindings are static.

2.3 μC/OS-II and Think are good canditates

Regarding our experiment, μC/OS is a good candidate for the following reasons:

- It is a mature real-time operating system used in many industrial projects.
- A fundamental particularity of μC/OS resides in its determinist nature, which is a basic property to consider, and to preserve in a reengineering of its internal structure.
- Its highly configurable capability is an interesting property that can help to compare with a component-based approach for which only the required services of the application are linked by composition within the final binary.
- μC/OS was designed to be portable, and a special attention has been paid to clearely distinguish between the generic code and the hardware dependent code. This eases the separation of concerns applied to an implementation designed with the component paradigm.

Considering the THINK framework and its underlying component model, we can highlight the following points:

- The component paradigm adopted by the THINK framework provides a high degree of flexibility, which combined with the genericity and configurability of the FRACTAL component model, allows the construction of dedicated and fully configurable operating systems.

- Architecture-oriented optimizations make THINK specially well suited for resource-limited embedded systems, as they allow to specify where flexibility capabilities should be added and which FRACTAL non-functional interfaces should be provided by components.
- Several case studies have been conducted with THINK to design minimal operating systems for various embedded platforms[7] (ARM, PowerPC, AVR, Xscale, etc). These experiments demonstrate the robustness of the tools constituing the THINK framework.

3 Challenges considering a μC componentization

Since a component is a basic first-class reuse entity, the separation of concerns is a key concept of CBSE. Improving this separation implies maximizing the decoupling between components in terms of encapsulated data and services. As a first challenge, considering that our experiment is based on an RTOS which was not initially designed as a component-based architecture, an important re-engineering effort of the μC/OS's implementation should be considered.

The component paradigm is based on abstractions which are well-suited to address flexibility requirements for complex systems. In our approach, we are interested in providing such a flexibility not only at application level but also at RTOS level.
From a design point of view, we want to provide a framework to build both an application-dedicated executive part and also reusable parts of executive off-the-shelf components. Since the component is a configurability unit for the developer, the granularity of components is a crucial point to consider according to these flexibility requirements. Furthermore, we have to provide the ability to choose and tune existing components. It should be possible within the design process to tailor the components according to various execution contexts. For example, to provide several implementations of a given component without changing its specifications, or to implement various parts of a complex communication protocol, etc. These aspects play a major role in the design and the developement of embedded systems. At the end, to mitigate the constraints of resource limited platforms, only the strictly required RTOS components should be embedded in the deployed system.

A third challenge appears when considering flexibility at runtime. Indeed, since the basic reconfiguration units are the components, their attributes and their assemblies are specified via bindings, and these concepts should thus be reified at runtime. Moreover, considering that our experiment tackles several very low-level RTOS services which will not be adapted at runtime, it is not compulsorily required to provide such reconfiguration capabilities for any components of the architecture. This requirement is a basic aspect to tackle considering the targetted application domain.

Last, but not least, the impact on execution time and memory footprint must be considered. An RTOS implementation designed following the compo-

[7] See `http://think.objectweb.org/`

nent paradigm and the aforementionned flexibility requirements should not involve unbounded overheads in term of performance. Moreover, the component framework should not introduce indeterminism in the execution of the RTOS services.

4 μC/OS Componentization

4.1 Motivations for a μC/OS's reengineering

The modular structure of μC is a good starting point for our componentization process since the core functionalities of the OS are clearly identified. However, we considered several parameters in order to improve the separation of concerns in the resulting component-based infrastructure proposed in the next section.

First, we considered the basic aspects usually defined within a real-time kernel [7, 26]. The two main aspects of an RTOS are the *Task Management*, for scheduling and the dispatching of tasks, and the *Event Management*, mainly for hardware events processing. Because these two aspects expose key features of an RTOS, we considered as a requirement the reification of them at architectural level. Moreover, since they are tangled over several μC modules, a reengineering of the kernel source code has been performed.

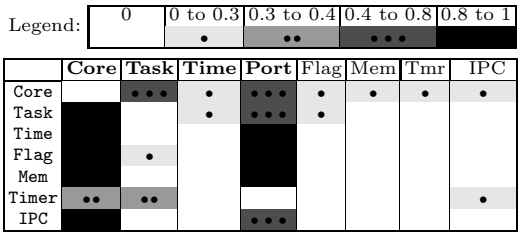


Fig. 3. Coupling between μC modules based on function dependencies.

Second, we have conducted several coupling analysis within the original μC/OS implementation. The Fig. 3 presents a coarse-grained coupling between μC modules[8] based on the function *dependencies*. To compute the coupling metric between modules `A` and `B`, the number of function calls between them is divided by the total number of function call between `A` and the other modules[9]. This analysis highlights for example a tightly coupling between the `Task` and the `Core` modules. Indeed, the latter implements several functions related to

[8] The `Mbox`, `Queue`, `Sem` and `Mutex` modules are gathered under the `IPC` appellation (for *Inter-Process Communication*).

[9] For example, the `Core` module makes 28 function calls to the `Task` module, and sums a total of 36 calls with all the modules. The result is a coupling metric of 0.77.

Task Management. The `Port` module provides the functions to enable/disable hardware interrupts necessary for global variables protection. This explains the thightly coupling between all the modules to these functions. We have also conducted this coupling analysis with the global variables access, the data types definition and the configuration constants. This analysis helped us to propose the fine-grained component-based design of μC presented in the next section.

4.2 Description of the Componentization Process

The componentization process of the operating system has been conducted in the following steps:

1. Interface Definitions. The μC function definitions are specified with the THINK IDL. For each type of service (e.g. *timer management*, *task management*, etc), a set of interfaces are defined according to their nature – creation or deletion of a resource, commonly or rarely used functions – and ordered within packages (e.g. see Fig. 2). Furthermore, we make a distinction between *internal interfaces* which are used only between μC modules, and *public interfaces* which provide all services visible at the application level following a set of system calls that may be invoked by application tasks.

2. Componentization. As a first componentization level, we define an architecture structured as the original μC implementation: each module sketched in Fig. 1 is reified as a primitive component using the THINK ADL. The architectural artifacts of these ADL descriptions are expressed within source code using annotations to inform the THINK compiler of this mapping. These components are encapsulated into a top-level composite that exposes the public interfaces available for the application tasks.

From this componentization level, and following the reengeneering motivations exposed in Section 4.1, the architecture is refined. RTOS key-features scattered over the modules were reified as components, and resulting tighly-coupled functions are merged.

3. Global Variables Expressed at Architectural Level. Within our componentization process, each μC/OS' global variable is defined as private data of primitive component (i.e. "task-related variables" defined within the `TaskManager Component`). Getter/setter functions are specified according to the data type definition of each variable and mapped to internal interfaces. The access to these variables between components are then expressed at architectural level via basic bindings.

4. Resources Components. From our point of view, within a CBSE design process, the resources of the operating system used by the application (such as a semaphore, a mailbox or a timer) should be reified at the architectural level. Therefore, μC/OS resources are represented by primitive components, called *resource wrappers*. Application components access these resources using basic bindings. This approach allows the developer to configure these resources from their exported configuration attributes (e.g. the task priority, the initial value of a semaphore, etc). Examples of such *resource wrappers* are given in Fig. 6.

5. Attribute Definitions. From the configuration parameters defined by μC/OS, we isolate two kinds of *attributes*. First, those which let the architect configure essential properties of the RTOS, such as the tick frequency, the priority assigned to the task managing the timer. Second, those which specify the thresholds for the use of ressources managed by the RTOS, e.g. the maximum number of tasks or semaphores supported by the kernel, the size of strings for event names, etc. Attribute signatures are thus defined (with naming conventions according to their kind) and attached to component definitions.

6. Component Library. Finally, we provide a library of ready-to-use RTOS, as a set of composite components, according to the services that they provide to the application (via their public interfaces).

An example of such a componentized RTOS is presented Fig. 4. It corresponds to the minimal operating system required by the application presented in the next section. The non-functional interface `init` (e.g. for the TaskManager Component) is invoqued by a THINK-generated life-cycle-controller component that manages, among others, system initialization.

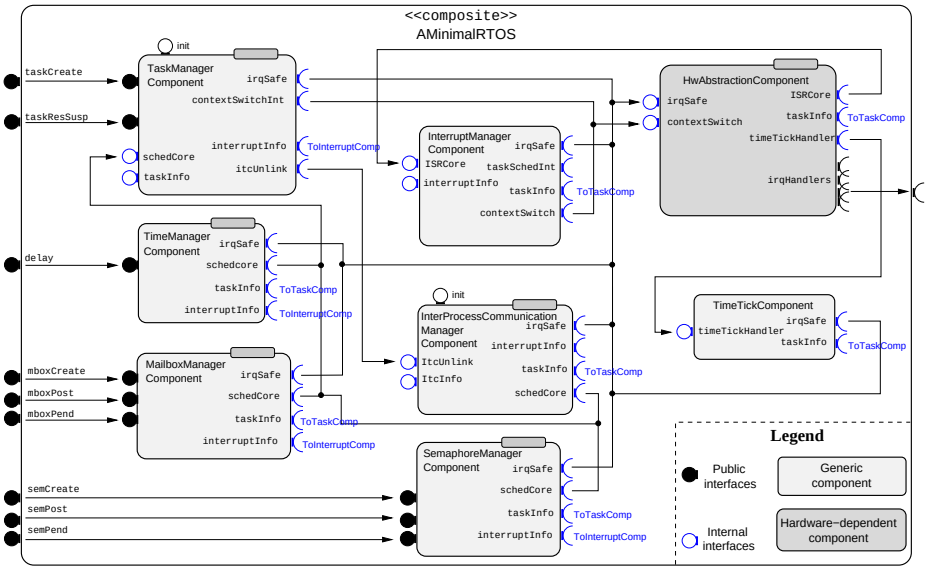


Fig. 4. Architecture's Example of a μC RTOS.

5 Evaluation

In this section, we evaluate our component-based design of μC/OS in comparison with its original implementation. As a qualitative evaluation in Section 5.2, we present how a classical multi-task application is designed using functional and resource components compared to a "pure C" implementation. We show how design time and runtime flexibility is addressed within our approach. Finally, we

conduct benchmark tests in Section 5.3 to measure performance and memory footprint overheads introduced by the component-based design.

5.1 Motivation example

To better illustrate several aspects of the following evaluation, we introduce in Fig. 5 a typical real-time and embedded application scenario that will be revisited throughout the course of this section. This example is composed of

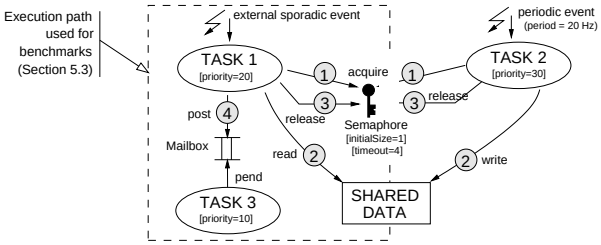


Fig. 5. A real-time application example.

three tasks. The tasks `Task1` and `Task2` read and write a shared data which is protected by a binary semaphore (execution steps ① -③). A task pending on the semaphore (via the `acquire` service) can not be blocked more than 4 time units (which corresponds to a timeout specified by the semaphore). `Task2` is activated periodically by a timer, while Task1 is activated in response to an external interrupt event. At the end of its execution cycle, `Task1` sends the content of the read data to the `Task3` using a mailbox (execution step ④). This example illustrates the basic concepts used to design a real-time application.

5.2 Qualitative evaluation

Design space provided to the application developer. Within our approach, the application is designed as interconnected components. The high-level design space provided to the application developer is based on the same architectural concepts used at RTOS level. Indeed, we use the THINK component model in a homogeneous way at these two abstraction levels (application and OS). As an illustration, the architecture of the application presented above is sketched out in Fig. 6. It shows a set of functional and resource components that require services provided by the operating system configuration in Fig. 4.

Each application task given in Fig. 5 is represented by a composition between a *Task resource component* and a *functional component* which implements the task's entry point (via the `runner` interface). Resource components are instances of already existing components found in previously built components library; developers simply configure their attributes according to desired behavior (e.g. the

priority for the tasks, the *timeout* for the semaphore, etc). The bindings between resource or functional components and the RTOS are generated automatically by our framework from interface signatures.

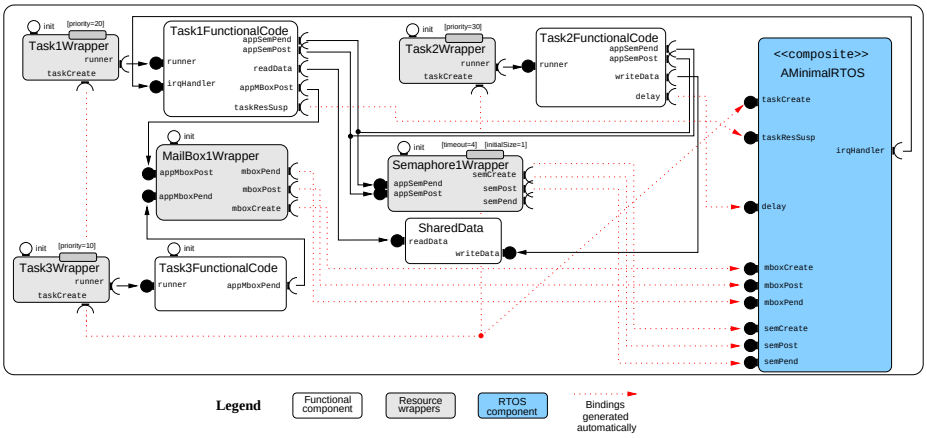


Fig. 6. A componentized example.

Bottom line, we provide the application developer a higher-level design procedure, compared to commonly used plain C language mechanisms. Besides, the operating system resources used by the application are reified as basic components. First, this feature offers a clear view, at architectural level, of the resources used by the system, and it reifies via bindings how they are shared by the functional components. Second, it makes easier their configuration since each instance exports its configuration attributes at the architectural level, simply accessible by the developer.

Flexibility at Design Time. Within the original implementation of μC/OS-II , compilation options allow to configure essential properties of the RTOS and to scale down the final executive using precompilation directives. Within our approach, these configuration capabilities are managed at architectural level:

- The configuration of the RTOS' essential properties is addressed using components configuration attributes. These attributes are defined at architectural level, as for instance the `OS_TICKS_PER_SEC` attribute defined for `ATimeComponent` definition given in Fig. 2. Moreover, we implemented a simple tool which returns, from a given architecture description, the whole set of component attributes configurable by the developer.
- The capability to scale down the final executive is addressed using the RTOS components stored in the library provided to the developer. For a given primitive component, several ADL definitions are stored in the library, differing by the number of interfaces – and thus the kind of services – they provide.

A set of RTOS composite component definitions are then built, exporting their public interfaces according to the interfaces provided by different combinations of subcomponents[10].

Finally, the RTOS componentization does not restrict the configuration capabilities supported by the original μC/OS-II implementation. Moreover, we can emphasize a feature arising from our approach. Since the application components export the services that they require from the RTOS, it is possible to automatically retrieve from the component library the corresponding composite definition which fits strictly the needs of the application. This feature is well-suited to address the constraints of resource-limited platforms, and represents a significant benefit of our approach compared to the manual configuration of original μC constants.

Flexibility at Runtime. As it has been presented in Section 2.2, the THINK compiler generates meta-data that describes the FRACTAL component model architectural concepts at the C source-code level and consequently in final binary file.

This meta-data provides the necessary infrastructure to enable flexibility from *inside* components. The standard FRACTAL API [6] that enables introspection and reconfiguration from *outside* components can be generated over these meta-data. Note however that THINK only injects these control interfaces (and their respective implementations) only where it has been specifically specified, none being injected by default.

As run-time flexibility may not be necessary for all system components, the THINK framework provides mechanisms to specify whether a single component or a subset of system components is not likely to evolve at execution time. Using flexibility-oriented *properties* the architect can instruct the THINK compiler whether to generate the meta-data that make these components flexible. This feature allows to apply flexibility only where needed, and the way it is effectively needed, producing substantial reductions in memory footprint size and execution times, as it is presented in the following quantitative evaluation.

5.3 Quantitative Evaluation

As it has been described in Section 4.2, the componentization process led us to a reengineering of the μC's internal structure. Besides, we propose to reify the kernel resource instances as components, introducing a level of indirection between the functional and the operating system components. Finally, the tools provided by the THINK framework allow to introduce several level of flexibility at runtime. In the following sections, we measure how this impacts the resulting executable in term of memory footprint and execution time. The evaluation is based on the RTOS and the application example presented respectively in Fig. 4 and Fig. 6.

[10] The THINK ADL supports inheritance relationships between architectural definitions which facilitates that procedure.

Memory Footprint. The Fig. 7 presents the memory footprints for a monolithic implementation used as a reference, compared to its component-based design[11]. We measure the overhead in code (i.e. `.text` section) and data, including initialized (i.e. `.data` section) and uninitialized (i.e. `.bss` section) data. We make this distinction as code is usually placed in ROM, whereas data are generally placed in RAM. For the component-based design, we propose three different configuration scenarios for the RTOS and the application presented in the preceding sections: the *highly flexible*, where components' meta-data allowing run-time flexibility are conserved at execution time, the *not flexible* scenario for which a completely static system is generated, and the *partially flexible*, for which only two functional components (`Task3FunctionalCode` and `SharedData` given in Fig. 2) are made flexible.

The systems generated in these scenarios do not include the non-functional FRACTAL interfaces that allow components to export FRACTAL's introspection and reconfiguration capabilities. This means that flexibility can only be exploited from inside components. In many cases, this basic flexibility level is enough to modify attributes or outgoing bindings according to internal component events.

		Reference	**Component-Based Design**		
		(a)	**(b)** highly flexible	**(c)** not flexible	**(d)** partially flexible
(1) RTOS	**Code**	13508	+16.8 %	+0 %	–
	Data	14072	+3.26 %	+0 %	–
(2) Complete System	**Code**	14003	+20.6 %	+1.8 %	+2.3 %
	Data	20252	+4.59 %	+0 %	+0.5 %

Fig. 7. Memory footprint sizes (in Bytes) and overheads, **(1)** for the RTOS level from Fig. 4, and **(2)** for the complete system from Fig. 6.

From these results, we highlight the significative overhead implied by the *highly flexible* configuration (Fig. 7 (1)b and (2)b). This reflects the widespread use of interfaces and attributes within the component-based design that lead to the generation of the meta-data that enable flexibility, and confirms that optimization is an important issue. The *not flexible* configuration shows that such optimizations are feasible and they actually lead to an expected close to null overhead[12] (Fig. 7 (1)c). These results hence demonstrate the capability of the THINK approach to benefit from CBSE at design time, without having to pay any overhead if no flexibility is required. The *partially flexible* configuration's overheads (Fig. 7 (2)d) show how runtime flexibility could be configured for only a dedicated subset of components at a reasonable cost.

[11] These experiments have been conducted using GCC with the `-Os` option that optimizes the binary image size.

[12] The light overhead observed in Code section (Fig. 7 (2)c) corresponds to the encapsulation of the *resource components* presented in Sec.5.2, overhead that may be eliminated in a near release of the THINK compiler.

The Fig. 8 shows the memory footprint overheads for the *highly flexible* and *partially flexible* scenarios when FRACTAL introspection and reconfiguration APIs implementations are injected into the systems. The *highly flexible* scenario (Fig. 8 (3)b and (4)b) shows the significative overheads resulting in both Code and Data sections. This is explained by the meta-data required to control the many bindings introduced by the encapsulation of the whole system into small components. Again, the *partially flexible* scenario (Fig. 8 (4)c) shows that this overhead can be considerably reduced if these control interfaces are only injected where actually desired, as permitted by the THINK compiler.

		Reference	**Component-Based Design**	
		(a)	**(b)** highly flexible with FRACTAL APIs	**(c)** partially flexible with FRACTAL APIs
(3) RTOS	**Code**	13508	+32.2 %	–
	Data	14072	+16.8 %	–
(4) Complete System	**Code**	14003	+47.8 %	+4.1 %
	Data	20252	+20.9 %	+1.1 %

Fig. 8. Memory footprint sizes (in Bytes) and overheads, with the FRACTAL APIs, **(3)** for the RTOS level from Fig. 4, and **(4)** for the complete system from Fig. 6.

Measures at RunTime. Considering the performance, the Fig. 9 presents the comparison between the monolithic and the component-based design which has been conducted considering the execution path given in Fig. 5. It traverses more than ten components of the architecture, from the application level as well as the RTOS level (since semaphore and mailbox services provided by the RTOS are involved), and includes a context switch between `Task1` and `Task2` implemented by the hardware-dependent component. The testing environment consists of a Pentium 4 monoprocessor at 2.0 GHz. The scenario was simulated under a Linux 2.6 kernel (using a Linux port of μC) patched by Rt-Preempt. The latter converts the kernel into a fully preemptible one with high resolution clock support, allowing precise performance measures. The results show that even for the *highly-flexible* configuration, the impact on performance is small, and becomes negligible when considering the optimized configuration. The figure also shows that the maximal amount of memory which is dynamically allocated at runtime remains unchanged within our component-based design compared to the monolithic one.

	Monolithic Design	**Component-Based Design**	
	reference	highly flexible	not flexible
Execution Path (μs)	45.97	**+2.8 %**	**+1.3 %**
Memory usage (*Bytes*)	49200	49200	49200

Fig. 9. Performance overheads and memory allocated dynamically.

6 Related Work

Several approaches have been proposed to satisfy actual needs of fast development cycles and dynamic adaptation to changing work contexts. In the very resource-constrained domain of wireless sensor networks, Virtual Machines (VM) or similar approaches such as Maté [17], Tapper [30] and DAViM [22] provide support for run-time flexibility and low energy consumption at communication tasks. VM's implementations are strongly bound to the application hence hardly scalable to more general real-time embedded systems. DAViM provides support for more complex dynamic reconfiguration by adding or removing operation libraries and so modify a running VM.

Applying modularity benefits into OS executives development process, OSKit [10] and eCos [21] provide a set of operating system components that can be used as building blocks to configure an operating system. Component definition and binding languages such as Knit [25] can be used to assist the assemblage of components. While producing efficient applications, these frameworks do not support dynamic reconfiguration and flexibility control. OSKit is also more oriented to non-embedded hardware platforms.

Following a CBSE approach, TinyOS [18] and Koala [29] define component models and allow to build a whole system, i.e. an application bound to a particular executive, compiling component architectures.

TinyOS is a component-based operating system and programming framework focused to *motes* in sensor networks domain. It reduces the penalty of fine-grained components by imposing a component-programming language, nesC [11], a dialect of C that facilitates functional code analysis and inlining. The use of this language introduces some significant limitations: besides making difficult the use of legacy code, a TinyOS application must be statically declared, as dynamic memory allocation is not supported. Dynamic reconfiguration at applicative and executive level is hence not supported, although.

Koala is a component-based development framework and a component model oriented to consumer electronic (CE) devices. The component paradigm is exploited at design time, bindings and other model abstractions are translated to traditional C programming language structures and compiled. Consequently Koala-based systems do not suffer from components-related performance overheads, but lack dynamic reconfiguration support.

Robocop project [1] enriched Koala component model offering, among others, component discovery and instantiation services and dynamic bindings features achieving CE devices requirements. Resources consumption is exhibited by *IResource* interfaces and is considered constant per operation [23]. An evaluation mechanism to measure static memory consumption had been developed for Koala [9].

Robocop and Koala propose a similar approach to Think and Qinna, a QoS framework for Think [28, 12]. It has been tested in many industrial experiments and accepted as an ISO standard under the name of M3W. However, we have not found enough information about how the component model is compiled, neither comparison with legacy code performance.

CAmkES [16] is a general embedded-systems oriented component model. Component-based applications run on top on a L4 micro-kernel [19] and a set of basic operating-system services provided by Iguana [14]. Produced glue code is agressivily optimized from compile-time available information. If flexibility at run-time is required, extension layers can be added to the application. This approach is flexible as it allows to minimize memory size overhead in a basic application and to add additional functionnalities if required. Nevertheless, the extension layers strongly rely on services provided by L4 micro-kernel and Iguana, creating a tight dependence between the component model, the component-based application and the operating system.

Hartex [5] is a component-based framework dedicated to real-time kernels. The RTOS is designed as a composite component encapsulating the kernel primitives which are structured as interconnected sub-components. From a library of basic kernel components implemented from scratch, the framework allows to derive kernel configurations depending on the functionalities needed by the application. However, as far as we know, Hartex is not based on an ADL, providing a higher-level representation of the component dependencies and kernel configurations. It is unclear how composition is managed by their compilation process and the framework do not support dynamic reconfiguration as well.

7 Conclusions

Component-based Software Engineering has emerged as a technology for the rapid assembly of flexible software systems, where the main benefits are reuse and separation of concerns. However, applying CBSE in the context of embedded systems implies to master the flexibility cost in term of performance, since they rely on low-level operating system services and are deployed on resource-limited devices.

This paper brings the following contributions to tackle these issues. First, we propose a componentization from an existing and mature Real-Time Operating System, μC/OS-II, based on the Think component framework. From the RTOS resources reified as components, we present how component-based applications are designed on top of it. At design time, within our component-based approach, we fully address the configuration capabilities proposed by the original μC implementation. Furthermore, we can highlight an improvement arising from our approach, which fulfill a constraint imposed by resource-limited platforms: from the application architecture which express its dependencies towards the RTOS services, we can retrieve automatically from the component library the operating system component which fits stricly the needs of that application.

Second, the Think component framework allows to introduce runtime flexibility, for RTOS components, as well as for application components, since we use its component model in a homogeneous way at these two abstraction levels. From a quantitative evaluation, and based on the Think features allowing to apply flexibility only when desired, we show that overheads involved by our

component-based design in term of performance are reasonables, and negligeables when considering a completely static version of the executable.

Relevant future work concerns improving configurability features of the RTOS at design time. Firstly, we are investigating a predicate-based ADL support within the THINK framework to provide a precompilation feature at architectural level. Secondly, the extensibility capabilities of our component-based design could be improved by exporting basic data type definitions used by the RTOS (such as *task control block*, *event control block*) at architectural level.

Acknowledgment

This work was supported by the french Minalogic MIND project.

References

1. Robocop ITEA Project. Robocop: Robust Open component-based software architecture for configurable devices project. http://www.hitech-projects.com/euprojects/robocop/.
2. *MicroC/OS-II: The Real-Time Kernel*. CMP Media, Inc., USA, 2002.
3. C. Angelov and J. Berthing. A Jitter-Free Kernel for Hard Real-Time Systems. In *ICESS*, pages 388–394, 2004.
4. M. Anne, R. He, T. Jarboui, M. Lacoste, O. Lobry, G. Lorant, M. Louvel, J. Navas, V. Olive, J. Polajovic, J. Pulou, M. Poulhies, S. Seyvoz, J. Tous, and T. Watteyne. Think: View-Based Support of Non-Functional Properties in Embedded Systems. In *Proceedings of the 6th International Conference on Embedded Software and Systems (ICESS-09)*, 2009.
5. J. Berthing and C. Angelov. Component-Based Design of Safe Real-Time Kernels for Embedded Systems. *EUROMICRO Conference*, 0:129–136, 2007.
6. E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal Component Model, 2004. Version 2.0-3.
7. G. C. Buttazzo. *Hard Real-Time Computing Systems*. Springer, Second Edition, 2005.
8. J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. Think: A software framework for component-based operating system kernels. In *Proceedings of the USENIX Annual Technical Conference*, pages 73–86, jun 2002.
9. A. Fioukov, E. Eskenazi, D. Hammer, and M. Chaudron. Evaluation of static properties for component-based architectures. pages 33–39, 2002.
10. B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit A Substrate for Kernel and Language Research. *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51, 1997.
11. D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D.Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *ACM Conference on Programming language design and implementation (PLDI)*, 2003.
12. L. Gonnord and J.-P. Babau. Quantity of Resource Properties Expression and Runtime Assurance for Embedded Systems. In *to be published in ACS/IEEE International Conference on Computer Systems and Applications, AICCSA'09*, Rabbat, Morocco, May 2009.

13. A. Greenfield. *Everyware: The Dawning Age of Ubiquitous Computing*. Peachpit Press, Berkeley, CA, USA, 2006.
14. G. Heiser. Secure Embedded Systems need microkernels. *USENIX*, 30(6):9–13, 2005.
15. H. Kopetz and N. Suri. Compositional Design of RT Systems: A Conceptual Basis for Specification of Linking Interfaces. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, page 51, Washington, DC, USA, 2003. IEEE Computer Society.
16. I. Kuz, Y. Liu, I. Gorton, and G. Heiser. CAmkES: A Component Model for Secure Microkernel-based Embedded Systems. *J. Syst. Softw.*, 80(5):687–699, 2007.
17. P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM.
18. P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An Operating System for Sensor Networks. *Ambient Intelligence*, pages 115–148, 2005.
19. J. Liedtke. On Micro-Kernel Construction. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 237–250, New York, NY, USA, 1995. ACM.
20. O. Lobry and J. Polakovic. Controlling the Performance Overhead of Component-Based Systems. In *Software Composition*, pages 149–156. Springer, 2008.
21. A. Massa. *Embedded Software Development with eCos*. Prentice Hall, 2002.
22. S. Michiels, W. Horré, W. Joosen, and P. Verbaeten. DAViM: a Dynamically Adaptable Virtual Machine for Sensor Networks. In *MidSens '06: Proceedings of the international workshop on Middleware for sensor networks*, pages 7–12, New York, NY, USA, 2006. ACM.
23. J. Muskens and M. R. V. Chaudron. Prediction of Run-Time Resource Consumption in Multi-task Component-Based Software Systems. In *CBSE*, pages 162–177, 2004.
24. J. Polakovic, A. E. Ozcan, and J.-B. Stefani. Building Reconfigurable Component-Based OS with THINK. In *EUROMICRO Conference on Software Engineering and Advanced Applications*, 2006.
25. A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *In Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 347–360, 2000.
26. J. A. Stankovic and R. Rajkumar. Real-Time Operating Systems. In *Real-Time Systems*, volume 28, Numbers 2-3, pages 237–253. Springer, 2004.
27. C. Szypersky, D. Gruntz, and S. Murer. *Component Software. Beyong Object-Oriented Programming*. ACM Press, 2002. 2nd edition.
28. J.-C. Tournier, V. Olive, and J.-P. Babau. Qinna, an Component-Based QoS Architecture. In *8th SIGSOFT symposium on CBSE*, pages 107–122, Saint-Louis, USA, June 2005.
29. R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, March 2000.
30. Q. Xie, J. Liu, and P. H. Chou. Tapper: a Lightweight Scripting Engine for Highly Constrained Wireless Sensor Nodes. In *IPSN '06*, pages 342–349, New York, NY, USA, 2006. ACM.