Think: View-Based Support of Non-Functional Properties in Embedded Systems

Matthieu Anne, Ruan He, Tahar Jarboui, Marc Lacoste, Olivier Lobry, Guirec Lorant, Maxime Louvel, Juan Navas, Vincent Olive, Juraj Polakovic, Marc Poulhiès, Jacques Pulou, Stéphane Seyvoz, Julien Tous, and Thomas Watteyne
Orange Labs, France – Email: {firstname.lastname}@orange-ftgroup.com

Abstract

Component-Based Software Engineering (CBSE) does not yet fully address non-functional requirements of embedded systems. To reach this goal, we show how to extend a component model like FRACTAL *with relevant abstractions such as threads, protection rings, or security domains. The* FRACTAL *Architecture Description Language (ADL) is extended by means of properties that tag components, bindings, and interfaces of the system architectural definition with execution schemes, dynamic reconfiguration strategies, protection and isolation patterns, or QoS features. Each extension captures a property-specific "system view" offering a sound basis to address some non-functional requirement. These extensions were experimented in the* THINK *framework, a C-based implementation of* FRACTAL*. Results show that* THINK *provides a generic and efficient approach to fully support these extensions thanks to a customizable toolchain.*

1. Introduction

CBSE is now widely accepted for embedded software design, though few experiments have been carried out up to mature industrial levels [2]. CBSE tackles the productivity bottleneck in software engineering by promoting intensive code re-use. This leads to structured code with clear dependencies between code fragments. However, today's component technologies only consider functional requirements. Due to heavy resource limitations and continuous interactions with the environment, embedded systems must also fulfill non-functional requirements regarding resource usage, timely behavior, dependability, security, etc. Thus, finding a way to control non-functional concerns in produced systems is mandatory.

Several projects have successfully overcome this difficulty for real-time constraints [3], [4]. However, they remain limited to programming patterns that are not suitable to other non-functional requirements or application domains. The programming model must therefore be extended with high-level entities (e.g., threads, protection rings, security domains, etc.) to abstract from the source code a *system view* focused on a given non-functional concern, to help the programmer handle the corresponding properties. That view may also be turned into a formal model for analysis and verification. The problem is to select the right abstractions, examine how they fit into the component paradigm, and extend the programming model accordingly.

FRACTAL [5] is a hierarchical and reflective component model to design, implement, deploy, and manage software systems. A FRACTAL component is both a design-time and a run-time entity, acting as unit of encapsulation, composition, and configuration. Components provide server interfaces as access points to the services that they implement, while functional requirements are expressed by client interfaces. Components interact through bindings between client and server

This paper reports on recent developments in the THINK *project initiated in 1998 at France Télécom R&D [1]. Since then, the project has fostered many research results through collaborations with industrial and academic partners.* THINK *is now a reference architecture for open-source component-based embedded systems. The* THINK *team presents here a selection of key mature results. These are the outcome of efforts of many people, notably in the IST projects E2R I and E2R II, the ARTIST and ARTIST2 Networks of Excellence, and the MIND and Flex-eWare French-government funded projects, whose much appreciated contributions are hereby acknowledged.*

interfaces. Any component may have attributes that represent primitive properties. The FRACTAL model also defines standard interfaces to control the internal structure of a component at run-time. The software architecture of a system is then given by its hierarchy of bound components.

FRACTAL is agnostic regarding the size of components. A FRACTAL architecture can thus describe finely low-level resource access mechanisms (e.g., memory management, CPU schedulers, etc.) which directly impact the embedded system non-functional properties. A system view based on FRACTAL can thus also capture these concerns. However, FRACTAL itself does not define any precise semantics for bindings. Extensions to the model are thus needed.

This paper illustrates how a system software architecture can be modified using dedicated architectural patterns to enforce property-specific views on the target system. The essence of the needed mechanisms can be captured taking as example the distributed view of a system: binding remote components is captured by composite (component-based) bindings that may be implemented by interposing proxy and stub components. In that example, the FRACTAL programming model can be extended with the location abstraction that includes the components deployed on a given site. Realizing these extensions was experimented using THINK, a C implementation of the FRACTAL model, through a series of proof-of-concept embedded prototypes, where view-specific architectural patterns were defined and implemented. We used extensively the specific architecture of the THINK compiler to support the needed enhancements of the software system architecture.

In what follows, we first review related work (Section 2). We then present the THINK main architectural principles, including how system views can be supported (Section 3). We illustrate how this design enables to support easily and efficiently typical non-functional properties (Section 4). Some evaluation results are also described (Section 5). Finally, we provide some insight on future work.

2. Related Work

Previous research explored the balance between flexibility (understood as design-time and run-time configurability), performance, and other non-functional properties by studying either: the flexibility impact of embedded kernel design, mostly related to the abstraction level of the system vs. applications interface; or the properties guaranteed for a given system architecture.

Embedded OS architectures have considerably evolved from monolithic to micro- [6], exo-, or extensible kernels [7], [8], the OS offering greater flexibility as its structure gets more explicitly defined. In this process, the kernel itself tends to get smaller: system services are externalized from a core structure as modules, servers, libraries, or extensions, which may specialized for each concern. The frontier between system and applications is thus increasingly closer to the hardware. The component approach adopted in THINK and other kernels [9]–[11] totally removes that frontier, the kernel disappearing altogether. It does not impose a predefined OS structure, allowing the full spectrum of previous system designs by flexible arrangement of components.

In addition to design-time customizability [11], [12], those systems studied a fixed set of non-functional properties, such as run-time reconfigurability [6]–[8], [10], [13], [14], security [7], [9], [10], safety [7], [8], real-time/QoS constraints [11], [12], or architectural consistency [10], [13], [14]. Yet, the number of handled properties remained limited and the approach system-specific.

Aspect-Oriented Programming (AOP) [15] provides a more generic approach to separation of concerns by weaving together different aspects. Despite some first attempts like reflective OSes [16] with solutions based on composing meta-objects, the AOP approach was little applied to the OS context, generally being considered as too expensive for embedded systems [17] – or due to lack of tools. The extensible THINK compiler allowing selective and view-specific optimizations could be a solution to the performance problem, while providing the programmer with a complete toolchain.

3. Architectural Principles

3.1. THINK in Practice

THINK [18] (latest version: NUPTSE) is an open-source C implementation of the FRACTAL model. The THINK compiler takes among inputs an architecture description written in an ADL such as the FRACTAL ADL [19]. Figures 1 and 2 show a sample architecture and its ADL description [1].

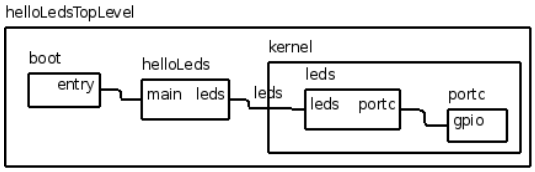

Figure 1. The HelloLeds Architecture.

```
component helloLedsTopLevel {
  contains boot = boot.lib.boot
  contains helloLeds = helloLeds
  contains kernel = em128Kernel
  binds boot.entry to helloLeds.main
  binds helloLeds.leds to kernel.leds
}
...
component em128Kernel {
  provides em128.api.Leds as leds
  contains leds = em128.lib.leds
  contains portc = avr.atm128.hw.lib.PORTC
  binds this.leds to leds.leds
  binds leds.portc to portc.gpio
}
...
component helloLeds{
  provides activity.api.Main as main
  requires em128.api.Leds as leds
  content helloLeds
}
```

Figure 2. ADL Description.

The `helloLedsTopLevel` component contains three sub-components (`boot`, `helloLeds` and `kernel`) which interfaces are bound together. The `kernel` component (of type `em128Kernel`) contains two sub-components. This description defines the server interface `leds` (of type `em128.api.Leds`) and two inner bindings. Components `boot`, `leds`, and `portc` are simply re-used from the KORTEX component library included in the THINK distribution. Figure 3 shows the implementation of the `helloLeds` primitive component.

1. By convention, server interfaces are represented on the left side of a component and client interfaces on the right.

```
//@@ ServerMethod(main, main, mainentry) @@
//@@ ClientInterfacePrefix(leds, LEDS_) @@
voids mainentry(int argc, char** argv){
  LEDS_setLEDs(0);
  while(1){};
}
```

Figure 3. HelloLeds Functional Code.

NUPTSE provides a convenient programming language (`NuptC`) to write the functional code of a component. `NuptC` is regular C with *annotations* added as comments which specify the C symbols representing the architectural entities described in the ADL. In Figure 3, annotations specify that function symbol `mainentry` stands for method `main` in server interface `main`, and that all methods in the `leds` interface are prefixed by `LEDS_`: `LEDS_setLEDs` stands for the `setLEDs` method.

THINK does not make any assumption on the generated meta-data, e.g., depending on the desired flexibility, interface calls may be implemented as direct or indirect function calls, attributes may be implemented as variables or compile-time constants, etc. Annotations also simplify the burden of encapsulating legacy code into components, facilitating code evolution.

3.2. Architecture Overview

Following a separation of concerns principle, the THINK compiler separates tasks related to the target architecture from those related to the production of C code. Hence, compilation is a two-step process: the *load* stage and the *build* stage.

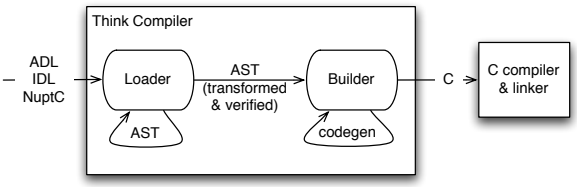

Figure 4. Think Compiler Architecture.

3.2.1. The Load Stage. The architectural description of the system is compiled into an intermediate Abstract Syntax Tree (AST). The loader is built as a stack of elementary loaders that perform *architecture checking* and *transformation*

operations on this AST. For example, a loader component may check bound interfaces for type compatibility. Another loader may transform the architecture by adding component stubs between client and server components – typically, a security loader will intercept calls on a server interface by injecting a stub to perform access control. Since the compiler can generate very optimized code (THINK may finally inline the stub component in the functional code), injecting code through components is as efficient as directly injecting code in the implementation of the server interface. The benefits of this approach are: (1) loaders can reason in terms of architectural transformations without needing to interpret the implementation code; and (2) code generation does not depend on architectural transformations made in the load stage.

3.2.2. The Build Stage. This stage is executed by builder components that together form the compiler *backend*. Each builder manages an instance of an architectural view of the system to compile. In Figure 1, server interface builders generate meta-data for server interfaces `gpio` (`portc` component) and `main` (`helloLeds` component). Similarly, a client interface builder generates meta-data for client interface `portc` (`leds` component). Builders do not directly generate meta-data but an abstract representation of the C code to produce. This operation is performed by CODEGEN, a Java package (part of the THINK distribution) for generation and transformation of C code. In CODEGEN, C entities (variables, types, expressions, etc.) are represented by Java objects. Code fragments are modelled as semantic graphs.

This approach enables the backend to be decomposed into loosely connected builder components [20]. Builders can make decisions, like optimizing or not and how to optimize, independently of other builders. A client interface builder could generate direct calls in place of client interface calls for client interfaces that are statically bound to a server interface. Independently, a server interface builder could add a *this* parameter to declarations of the C functions implementing methods of a server interface only when the corresponding code is shared by multiple component instances. These two decisions which do not depend on the same architectural system properties are taken by separate builders, and can be combined thanks to CODEGEN.

3.3. Supporting System Views

3.3.1. Compiler Plug-ins. The embedded systems programmer may define *system views* to control non-functional system properties. A view can be enforced in the software architecture with specific architectural patterns. Those patterns can be implemented directly. However, it is usually easier to delegate this task to the compiler, which will perform transformations on the system functional architecture.

At the end of the load stage, the AST mirrors the originally described functional architecture. The compiler can then transform this architecture by injecting the targeted patterns through the loading of a dedicated *plug-in*. Plug-ins can be defined to perform any operation on the AST: adding components, intercepting and redirecting bindings, etc. These plug-ins can have various impacts on the AST. Some can be passive, e.g., a plug-in that only dumps the AST. Others might concern specific AST nodes, e.g., to count invocations on a particular method, a plug-in could instanciate new components to intercept those invocations. Finally, a plug-in could be relevant only for a subset of a particular type of nodes, e.g., only bindings between components located in different hosts being implemented by a proxy/stub pair. We introduce below how *ADL properties* enable developers to specify non-functional properties on specific elements of the architecture.

Several plug-ins can be invoked on the same system architecture. For instance, a common sequence is to: dump the AST; invoke a plug-in that transforms the architecture; and finally dump the AST again to visualize the effectiveness of the transformation. In this case, specified plug-ins are invoked sequentially and independently. However, plug-ins that implement several AST transformations may give rise to conflicts, and generally will not be able to be scheduled sequentially. This point is not addressed in the paper but identified as a forthcoming crucial issue.

3.3.2. ADL Properties. The THINK ADL was extended so that any architectural entity (e.g., components, bindings, etc.) in the system could be tagged with *properties* to express non-functional concerns. Predefined properties are optimization-oriented and interpreted by default builders. For example, interface calls through a binding tagged with the `[static]` property will be implemented as direct function calls. Similarly, attributes tagged with `[const]` will be implemented as compile-time constants instead of C variables. Properties may also be interpreted during the load stage. The set of properties can be extended according to new plug-ins, enabling the developer to define targeted *system views*.

3.3.3. Global Extensions. FRACTAL clearly separates the role of architect (who defines the system architecture) from that of developer (who implements the functional code). THINK goes one step further with the notion of *global extension*. The ADL already allows defining a component type by extending an existing definition. A global extension is simply an ADL definition where names of components, interfaces, attributes, etc. are expressed using regular expressions. Each component of a given architecture is matched against the regular expression. When matching occurs, the original definition is automatically extended, the regular expression being replaced with matching names. This mechanism makes it possible to transform an architecture in a global way: non-structural properties are clearly isolated from the architecture description to achieve full separation of concerns.

```
a:  component em128Kernel {
      binds * to * [static]
    }

b:  <main.kernel.**.*> component * {
      binds * to * [static]
    }
```

Figure 5. Global Extensions.

For example, from a given ADL description, it is very simple to generate a system with all bindings being either static (i.e., unmodifiable, but offering a very effective RAM footprint) or dynamic (i.e., fully reconfigurable). Figure 5a declares static all bindings of the `em128Kernel` component. In Figure 5b only bindings of sub-components of the `main.kernel` component are made static.

4. View-Specific Extensions

We now present five system views covering typical non-functional properties of an embedded system using the THINK plug-in mechanism. For each view, we describe the main abstractions, corresponding properties, architectural patterns, and dynamic behavior.

4.1. Behavioral View

The THINK programming interface currently offers no way to express non-functional properties like timing constraints, concurrency or communications semantics. Using more abstract primitives instead of low-level C code should help the programmer to better understand the behavior of the system at run-time. The compilation toolchain should thus be extended to take as input these additional primitives.

For example, in a classical programming interface, a FIFO queue that is blocking and has a thread would be called *ThreadedBlockingFifo*. Aside from its name and its functionnal code, there is no way to know its dynamic behavior, which is not suitable for automatic processing. The BUZZ THINK extension enhances the programming interface with new high-level abstractions to express the dynamic behavior of the system. BUZZ introduces new classes of components and bindings, which have been chosen to support existing programming and execution models. Contrary to other approaches like TinyOS/nesC [12], [21] which force the developer to use a specific programming model, BUZZ does not impose a particular execution model, letting the developer decide what is best in his case.

BUZZ components are split in 3 classes: *active* components embed their own execution context in which incoming calls are treated in FIFO order and executed in run-to-completion; *passive* components do not embed an execution context; *interrupters* are a mix. They do not have a context and use the interruption context. Invocations from other components are similar to passive ones. Interrupters mainly forward interrupt calls to active components.

Bindings are split in 3 main classes: calls using *asynchronous* bindings are not blocking and cannot have return values; calls using *synchronous* bindings are blocking and can have a return value; calls using a *delayed* binding are close to the asynchronous case, except that the callee will not receive the call until the caller method is finished.

Along with the system architecture, the developer can choose how the active components are scheduled, either by choosing a predefined policy or by writing its own. In particular, an ongoing work uses the Esterel tasking model [22] to specify the BUZZ scheduling policy.

For the implementation, we use a set of ADL properties (see Figure 6) attached to component instances: `active`, `passive`, `interrupter`; to bindings: `asynchronous`, `synchronous`, `delayed`. The scheduling policy is set by the property `scheduler` on the top level component definition.

```
component sample_kernel [scheduler=preemptive] {
 contains serial = avr.usart [interrupter]
 contains appli = my.lib.appli [active]
 contains tools = my.lib.tools [passive]
 binds serial.demux to appli.recv [asynchronous]
 binds appli.tools to tools.tools [synchronous]
}
```

Figure 6. ADL with BUZZ Properties.

A THINK compiler plug-in takes into account these properties. It modifies the architecture provided by the developer mainly by adding new components in the system. For each active component, the compiler creates a new component that intercepts incoming and outgoing calls. They are responsible for implementing the semantics for the different bindings. For server interfaces, the interceptor has FIFO queues to store the calls along with the caller information (the caller may have to be awakened when execution is finished). For client interfaces, the interceptor forwards the call to the callee, stores delayed calls in FIFO queues, and calls the scheduler for blocking in synchronous cases.

To implement the scheduling policy, a component is generated by the compiler. Its functional code is synthesized by using the value of the `scheduler` property and the specifics of the architecture being compiled.

Besides code generation, the BUZZ abstract primitives can also be used for system analysis purposes. Along with the executable image, we are able to generate a BIP [23] model to perform system verification using model checking.

4.2. Reconfiguration View

Dynamic reconfiguration consists in modifying a running system. It can be used to apply patches and updates, to implement adaptive systems, dynamic instrumentation, to support third-party modules... It cannot be ignored when one cannot stop a running system.

Dynamic reconfiguration can be realized in five steps: identify the part of the system to reconfigure; suspend its execution; modify the system (add, remove components); transfer the state to the newly-deployed components (only possible when the system is in a stable, so-called *quiescent* state); and resume execution. The main difficulty is to detect whether a component is in a quiescent state. Depending on the execution model, two mechanisms have been evaluated using THINK for quiescent state detection.

In a multithreaded system, a reference counting mechanism counts running method invocations on a component. When the reference counter reaches zero, the component is in a quiescent state. The system blocks all new invocations until the end of the reconfiguration. To count active calls, we use interceptors to maintain invocation counters on interfaces.

In an event-based system, quiescence is reached each time an event handler has completed. The reconfiguration can then take place between the handling of two events.

Different properties are assigned in the ADL to generate the reconfiguration mechanisms. The THINK compiler then automatically adds the thread counters, the event scheduling mechanisms, and the Fractal controllers necessary to modify the application [24]. For optimization purposes, components that should be reconfigurable can be explictly tagged as such in the ADL.

4.3. Isolation View

Some embedded components might be *critical* as they manipulate shared physical or logical

resources. The corresponding provided interfaces should not be accessed by any component, especially during run-time system reconfigurations. Isolation is a key property to guarantee non-circumventability of any accesss control mechanism. THINK supports the definition of *protection domains* to enforce multiple isolation policies: components in the same protection domain can freely bind their interfaces and access their critical data, as it is assumed that no security issues will occur during execution. Communications between protection domains are trapped by an automatically generated isolation mechanism which manages context switching between protected and non-protected hardware modes.

We implemented in THINK a Unix-like *system call* mechanism where two protection domains are defined: *user* and *kernel* operation modes. By tagging a component with the *kernel* property we can define its protection domain association.

The THINK compiler redirects all existing bindings to the kernel component server interfaces to automatically generated *Interceptor* and *Deliverer* components to perform parameter marshalling/unmarshalling (see Figure 7). These components are linked with a *Sink* component that traps invocations and changes hardware protection mode. It can also implement adaptable interception policies.

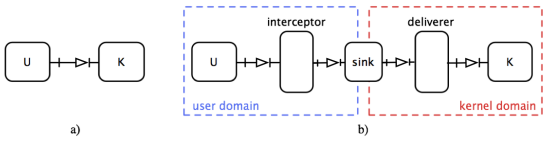

Figure 7. User (U) and Kernel (K) Component Binding: Without vs. With Protection Domains.

4.4. Protection View

Introducing flexibility in the OS kernel security services was also investigated to tune protection mechanisms of a pervasive device to the current security context. The objectives were: (1) to enforce different types of network-specific security policies; (2) to reconfigure policies dynamically according to shifting protection requirements when the device moves between networks. A THINK plug-in called CRACKER [25] was thus designed to enable policy-neutral access control enforcement. CRACKER supports a wide range of kernel-level authorization policies. It also allows policy reconfiguration between different security models, while maintaining acceptable system performance. The CRACKER component design prevents to be hindered by the rigidity of the OS architecture, since the component is both a reconfiguration and protection unit. Thus, different trade-offs can be found when combining these two views, trade-offs also weighed up against performance requirements.

In CRACKER, kernel resources are reified by software components, to which security controller can be added in the description of the kernel architecture to enforce access control. This operation attaches a *Reference Monitor (RM)* to perform security checks on the component interfaces. For a given authorization model, policy decision-making is encapsulated in a separate *Policy Manager (PM)* component.

Using the NUPTSE properties, the granularity of authorization can be adapted by selecting in the ADL description the components in which RMs should be inserted, RMs being possibly shared. Security stubs generated by NUPTSE then redirect method invocations to the RM. These properties also allow to select the applicable authorization model and policy to protect a component.

These policies may be changed dynamically. CRACKER supports a wide range of security models through a specific PM sub-component called *Compute Permissions* (CP) representing the format of permissions in the current authorization model. Other sub-components are security model-independent. Authorization policy reconfiguration then only amounts to changing the CP, eventually at run-time.

4.5. Quality of Service View

In embedded systems, run-time QoS management is critical, notably to reconcile reconfiguration with resource constraints. To capture the QoS view, the architecture called Qinna [26] has been developed. This THINK plug-in is introduced thanks to properties making components QoS-aware and describing the QoS provided through the component interfaces.

Based on the properties given by the developer in the ADL and on the assumption that every component managing a resource (CPU, memory, threads) also provides QoS, each component offering QoS is wrapped into a `QoSComponent`. Those `QoSComponents` are then integrated within the Qinna architecture which defines the notions of `QoSComponentBroker`, `QoSComponentManager`, `QoSObserver` and `QoSDomain`. Evaluation results for multimedia applications show that the Qinna architecture is suitable for QoS management in embedded systems with acceptable overheads.

5. Evaluation Results

5.1. WiFLY and Sense&Sensitivity

The WiFLY experiment [27][2] demonstrated the applicability of THINK in the context of Wireless Sensor Networks (WSN), where systems are extremely constrained (8bits μC ATM128L), with complex communication protocols [28]. The setup involved $\sim$ 20 nodes, a base station used to store data and to issue requests, and a mobile station embedded in a R/C plane that bridges the gap between the 2 previous ones.

The Sense & Sensitivity experiment extended the previous one to a large scale WSN used for urban monitoring. It used $\sim$ 100 MSP430-based nodes scattered on the premises of Orange Labs.

Both experiments were developped using THINK without plug-ins. From the collected raw data, results on the wireless propagation characteristics and their impact on the communication architecture were extracted[3].

In a third experiment, we re-engineered the Sense & Sensitivity code to make use of BUZZ (see section 4.1). The activities related to the *application*, *network* and *link* layers were explicited by *active* components. Some communications between layers used *synchronous* message boxes – the caller being blocked until the callee finishes the work associated with the call, and the callee being awakened regularly to check if new requests are available. The use of the BUZZ *synchronous* bindings between active components made the architecture and the code easier to write and to maintain. The handling of these mechanisms by the compiler helped the programmer focus on its application and avoided bugs on code that could be automatically generated. Moreover, using active components instead of a hand-made polling mechanism allowed better power saving by waking up components only when needed.

2. http://think.objectweb.org/related_projects/wifly.html
3. See http://senseandsensitivity.rd.francetelecom.com/

5.2. Adaptable Security

The THINK support for reconfiguration and protection views was validated through a framework and prototype illustrating self-protection capabilities of pervasive networks [29]. An autonomic security loop was instantiated through several building blocks called *Configuration Management Modules* or *CMMs* for: monitoring the state of the external environment such as scanning available WLANs; deriving the ambient security context such as a previously unknown WLAN becoming available; decision-making to adapt or not protection to the new context; and safe reconfiguration of security mechanisms by changing the needed components, e.g., to select a new authentication mechanism, authorization policy, or cryptographic library.

On the device side, a sandbox called THINKBOX was defined using THINK to host resources to be protected and security mechanisms to be adapted. The CRACKER access control framework and THINK reconfiguration mechanisms were implemented on a Nokia 770 Internet tablet to realize the last link of the autonomic security loop. This experiment illustrated how the THINK component-based architecture allowed to realize context-aware security on a device in a simple manner, by combining security and reconfiguration views to achieve trade-offs between security and performance. Finer trade-offs such QoS vs. security [30] according to dimensions describing the quality of the context (QoC) information [31] are also possible by richer combinations of the THINK views.

6. Conclusion

We have shown how component-based design and an efficient compiler implementation enable easy support of non-functional properties in embedded systems through the definition of specialized plug-ins. The effectiveness of several of these plug-ins addressing optimization, behavior, reconfigurability, isolation, protection, and QoS has been demonstrated on several example kernels deployed on available boards.

How far is it possible to go towards an entirely view-based system, where each view can be modified independently? We advocate that the FRACTAL component model extended with dedicated properties can offer such a view-based programming model. This issue depends heavily on the effectiveness of weaving different views together through the composition of plug-ins. As already pointed out, the problem is more complex than just selecting the corresponding plug-ins processing order as this naive plug-in composition is only applicable in specific cases (e.g., isolation and security).

Many embedded systems especially in the home environment are open systems and thus present a continuously changing architecture. As a consequence, a well-chosen flexibility strategy at design-time may no longer be suited at run-time. Fractal controllers as well as the reconfiguration THINK mechanisms allow to define when, where, and how can flexibility be introduced in a system. For each view, trade-offs between static (design-time) and dynamic (run-time) properties of components remain an open issue.

Moreover, in these environments many applications are composed and deployed following Service-Oriented Architectures, where bindings are managed by a centralized or distributed broker component on top of different protocols stacks such as ZigBee, TR-069, UPnP, 6LoWPAN, DPWS, etc. [32]. In this setting, identifying and implementing the required systems views to master those very changing application architectures is also a forthcomming challenge.

References

[1] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller, "Think: A Software Framework for Component-Based Operating System Kernels," in *USENIX Annual Technical Conference*, 2002.

[2] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software," *Computer*, vol. 33, no. 3, pp. 78–85, March 2000.

[3] P. Feiler, B. Lewis, and S. Vestal, "The SAE Architecture Analysis and Design Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering," in *RTAS Workshop on Model-Driven Embedded Systems*, 2003.

[4] J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis, "VEST: An Aspect-Based Composition Tool for Real-Time Systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2003.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The Fractal Component Model and its Support in Java," *Software – Practice and Experience (SP&E)*, vol. 36, no. 11-12, pp. 1257–1284, 2006, special issue on "Experiences with Auto-adaptive and Reconfigurable Systems".

[6] O. Krieger, M. Auslander, B. Rosenburg, R. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowsli, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig, "K42: Building a Complete Operating System," *Operating Systems Review*, vol. 40, no. 4, pp. 133–146, 2006.

[7] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System," in *ACM Symposium on Operating Sytems Principles (SOSP)*, 1995.

[8] M. Seltzer, Y. Endo, C. Small, and K. Smith, "Dealing with Disaster : Surviving Misbehaved Kernel Extensions," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[9] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "CAmkES: A Component Model for Secure Microkernel-Based Embedded Systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 687–699, 2006.

[10] M. Clarke and G. Coulson, "An Architecture for Dynamically Extensible Operating Systems," in *International Conference on Configurable Distributed Systems (ICCDS)*, 1998.

[11] "eCos," `http://ecos.sourceware.org/`.

[12] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System Architecture Directions for Networked Sensors," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[13] J. Helander and A. Forin, "MMLite: A Highly Componentized System Architecture," in *ACM SIGOPS European Workshop: Support for Composing Distributed Applications*, 1998.

[14] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors," in *IEEE International Conference on Local Computer Networks (LCN)*, 2004.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[16] Y. Yokote, "The Apertos Reflective Operating System: the Concept and its Implementation," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1992.

[17] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat, "A Quantitative Analysis of Aspects in the eCos Kernel," in *ACM EuroSys Conference*, 2006.

[18] "The Think Framework," `http://think.objectweb.org/`.

[19] "The Fractal ADL Toolchain," `http://fractal.objectweb.org/fractaladl/index.html`.

[20] O. Lobry and J. Polakovic, "Controlling the Performance Overhead of Component-Based Systems," in *Software Composition (SC)*, 2008.

[21] 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.

[22] G. Berry and the Esterel Team, *The Esterel v5.91 System Manual*, 2000, `http://www.inria.fr/meije/esterel`.

[23] A. Basu, M. Bozga, and J. Sifakis, "Modeling Heterogeneous Real-Time Components in BIP," in *4th IEEE International Conference International Conference on Software Engineering and Formal Methods (SEFM)*, 2006.

[24] J. Polakovic and J.-B. Stefani, "Architecting Reconfigurable Component-Based Operating Systems," *Journal of System Architecture*, vol. 54, no. 6, pp. 562–575, 2008.

[25] M. Lacoste, T. Jarboui, and R. He, "A Component-Based Policy-Neutral Architecture for Kernel-Level Access Control," *Annals of Telecommunications*, vol. 64, no. 1–2, pp. 121–146, 2008, special issue "Software Components: The Fractal Initiative".

[26] J.-C. Tournier, J.-P. Babau, and V. Olive, "Qinna, a Component-Based QoS Architecture," in *International Symposium on Component-Based Software Engineering (CBSE)*, 2005.

[27] T. Watteyne, D. Barthel, M. Dohler, and I. Augé-Blum, "WiFly: Experimenting with Wireless Sensor Networks and Virtual Coordinates," INRIA, Research Report RR-6471, 2008.

[28] T. Watteyne, A. Bachir, M. Dohler, D. Barthel, and I. Augé-Blum, "1-hopMAC: An Energy-Efficient MAC Protocol for Avoiding 1-Hop Neighborhood Knowledge," in *International Workshop on Wireless Ad-hoc and Sensor Networks (IWWAN)*, 2006.

[29] A. Saxena, M. Lacoste, T. Jarboui, U. Lücking, and B. Steinke, "A Software Framework for Autonomic Security in Pervasive Environments," in *International Conference on Information Systems Security (ICISS)*, 2007.

[30] M. Alia and M. Lacoste, "A QoS and Security Adaptation Model for Autonomic Pervasive Systems," in *International COMPSAC Workshop on Secure Software Engineering (IWSSE)*, 2008.

[31] M. Lacoste, G. Privat, and F. Ramparany, "Evaluating Confidence in Context for Context-Aware Security," in *European Conference on Ambient Intelligence (AmI)*, 2007.

[32] A. Bottaro and A. Gérodolle, "Home SOA - Facing Protocol Heterogeneity in Pervasive Applications," in *IEEE International Conference on Pervasive Services (ICPS)*, 2008.