SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2011)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1143

Flex-eWare: a flexible model driven solution for designing and implementing embedded distributed systems

Mathieu Jan[5], Christophe Jouvray[2], Fabrice Kordon[1], Antonio Kung[2,*,†], Jimmy Lalande[6], Frédéric Loiret[8], Juan Navas[4], Laurent Pautet[3], Jacques Pulou[4], Ansgar Radermacher[5] and Lionel Seinturier[7]

[1]*Univ. P. & M. Curie, LIP6, CNRS UMR-7606, 4 place Jussieu, 75005 Paris, France*
[2]*TRIALOG, 25 rue du Général Foy, 75008 Paris, France*
[3]*Telecom ParisTech, 46 rue Barrault, 75013 Paris, France*
[4]*Orange Labs, 28 chemin du vieux Chêne, 38243 Meylan, France*
[5]*CEA-LIST, Centre de Saclay, 91191 Gif-sur-Yvette, France*
[6]*Schneider Electric Industries, Strategy & Innovation, 38050 Grenoble, France*
[7]*Univ. Lille 1 & INRIA, LIFL, 59655 Villeneuve d'Ascq, France*
[8]*Royal Institute of Technology (KTH), Embedded Control Systems, Stockholm, Sweden*

SUMMARY

The complexity of modern embedded systems increases as they incorporate new concerns such as distribution and mobility. These new features need to be considered as early as possible in the software development life cycle. Model driven engineering promotes an intensive use of models and is now widely seen as a solution to master the development of complex systems such as embedded ones. Component-based software engineering is another major trend that gains acceptance in the embedded world because of its properties such as reuse, modularity, and flexibility.

This article proposes the Flex-eWare component model (FCM) for designing and implementing modern embedded systems. The FCM unifies model driven engineering and component-based software engineering and has been evaluated in several application domains with different requirements: wireless sensor networks, distributed client/server applications, and control systems for electrical devices. This approach highlights a new concept: flexibility points that arise at several stages of the development process, that is, in the model (design phase), in the execution platform, and during the execution itself. This flexibility points are captured with model libraries that can extend the FCM. Copyright © 2011 John Wiley & Sons, Ltd.

Received 8 September 2010; Revised 30 September 2011; Accepted 1 October 2011

KEY WORDS: embedded system; software component; flexibility; model driven engineering

1. INTRODUCTION

Embedded systems tend to be more and more complex and incorporate many different concerns such as distribution and mobility. This raises a need for new features to be considered during their development such as architecture description, deployment strategies, and extensibility or to consider run-time adaptation in such systems.

From a software engineering point of view, model driven engineering (MDE) is now widely seen as a solution to master the development of complex systems such as embedded systems. In such approaches, development relies on models that are able to support code generation to ease and secure implementation on the one hand and to enable reasoning and to check properties such as schedulability on the another hand.

*Correspondence to: Antonio Kung, TRIALOG, 25 rue du Général Foy, 75008 Paris, France.
†E-mail: antonio.kung@trialog.com

However, current notations to support the design of embedded systems do not consider yet the new required features that could help the designer to cope with the new needs of embedded systems. In particular, embedded systems have to be *flexible*. This is critical because engineers will have, sooner or later, to cope with various types of embedding constraints (e.g., the one of systems on chip and the one of workstations) in the same application. So, flexibility can help in the design of embedded systems either at design-time (software product line (SPL) or configuration/deployment) or at run time (adaptability).

The Flex-eWare project [1] aims at developing a solution to cope with flexibility in the design of embedded and distributed systems. This project gathered companies (Orange Labs, Schneider Electric, Teamlog, Thales, Trialog) and academics (CEA, INRIA, Telecom ParisTech, Université P. & M. Curie) from 2007 to 2010. This article presents the results of this project from both the conceptual point of view (what has to be set up in the specification) and the development process. We first elaborate a conceptual component model: Flex-eWare component model (FCM). Then we design some mappings to several technologies to assess its generality.

This article is structured as follows. Section 2 identifies the problems to be tackled by future embedded and distributed systems. Section 3 presents some existing (and usually partial) solutions proposed in the area and outlines the main concepts considered to elaborate FCM. Section 4 details our component models and its specificities. Section 5 illustrates the use of FCM in three different application domains with different underlying technologies. Finally, Section 6 concludes this article and proposes directions for future work.

2. REQUIREMENTS

This section identifies the set of requirements related to a model-based design approach for complex systems. We do so by studying the domains targeted by our work (Section 2.1). In particular, we emphasize the management of flexibility that is a key issue for future embedded and distributed systems. We then identify how flexibility management should impact the software engineering life cycle (Section 2.2) and introduce our contribution to these challenges (Section 2.3).

2.1. Requirements for future application domains

Let us first provide examples of current and future needs on software flexibility in two application domains that are emblematic of the domains targeted in our work: customer premise equipment (CPE) and automotive systems. Future needs are based on a prospective vision of these domains.

Telecommunication CPE domain. The CPE market in telecommunication refers to products installed at home, connected to an external network, and operated by business operators. Typical examples are Internet modems and 'boxes'. Such systems provide multiple services such as Internet access, IP TV, Video on Demand, and voice over IP.

Today, business operators need software architectures as well as deployment features suitable to maintain, update, extend, and configure applications for CPE such as new video encoders. Because of the various home network solutions [2, 3], they also need to support nonfunctional requirements such as QoS management. Both types of functions are needed, for instance, to replace a security component within an existing video encoder. Operators also have to cope with numerous devices (e.g., millions of Internet boxes) and therefore need features for scalable remote administration [4], deployment [5], and configuration inspection [6].

In the future, the market will move toward richer services as well as more sophisticated services involving multiple stakeholders. For instance, such services could be aggregators. Competing operators may thus coexist and have to share the management of CPE devices that would then have to cope with dynamically changing environments. In this context, service-oriented architecture [7] approaches will be required to enable new software components to be dynamically downloaded, deployed, registered, and linked to existing ones (possibly designed by other operators).

DOI: 10.1002/spe

Automotive systems. The automotive industry is currently targeting supply environments based on a vehicle manufacturer–centric relationship involving components from various sources to ensure both lower costs and lower risks of supply shortage. They are also concerned by stringent contractual and liability obligations. So, automotive systems need to support complex product diversity because either vehicles come with customer options (e.g., type of engine, accessories, etc.) or the assembly of vehicles involves multisourcing options.

Software is now a major part of these supply environments. It has been reported [8] that the development of automotive systems has already reached 40% of the total vehicle development cost, with a major part dedicated to the software part. Diversity of sources is ensured using software structuring standards such as Automotive Open System Architecture (AUTOSAR) [9], which allow the construction of systems based on the reuse of both applications and system components.

Some vehicles today include up to 70 electronic control units (ECUs). It is anticipated that for costs issues, vehicle functions will soon be deployed over a smaller number of ECUs. For instance, four cluster functions could be foreseen: power-train, body, safety, and multimedia. So, software components will have to be reused in configurations involving modifications of nonfunctional properties (NFPs). Moreover, an increasing number of external multimedia functions will also be installed in vehicles such as navigation systems, road tolling systems, or insurance systems based on usage. Integration of such functions will require more dynamicity in the underlying execution environment. This will have an impact on the way software components are developed and deployed.

Summary. On the basis of these two examples, some directions are emerging for software embedded systems. First, software flexibility must be considered all over the product life cycle (i.e., design, development, deployment, and during execution; see next section). We call *flexibility points* the specific cut-points during the development phases when variants are available to engineers (and thus flexibility of solutions can be investigated).

Second, flexibility must cope with the following needs: suitable software structuring, management of nonfunctional aspects, management of extensibility, and management of dynamically changing environments.

2.2. Requirements for future embedded systems

Section 2.1 identified future and near-future needs shared by both the automotive and CPE domains. These needs can be easily extended to other domains of embedded and distributed systems. To identify how software engineering should satisfy these requirements, we explore the way they are reflected throughout the product development life cycle. To simplify our study, we consider a rather ad hoc software life cycle, coarsely based on the Waterfall model [10] and composed of the following phases: design, development, deployment, and execution.

We extract a list of *requirements* to be fulfilled by embedded software engineering models. This list is described extensively in the remainder of this section and is summarized in Table I. Columns refer to the software life cycle phases, and rows to general needs. For example, the cell III.MD at the intersection of column *III. Deployment* and row *Management of dynamicity (MD)* provides the requirements for dynamicity during the deployment phase.

2.2.1. Requirements for the design phase. This phase deals with the specification of software requirements. In our case, this corresponds to the process of planning a solution satisfying these requirements. Designers may describe behavioral and structural aspects of a design solution using standard languages such as Unified Modeling Language (UML), formal languages such as B [11], or Architecture Description Languages (ADLs) [12] such as Wright [13]. Requirements for the design phase are reported as follows:

- I.SA: Structuring and consistency checks between system components. This includes features such as *encapsulation* with arbitrary *granularity*, strict separation of design aspects, *modularity* and *hierarchy* support to provide different system views at different abstraction levels. By expressing component needs and relate them to the associated provided services on the

DOI: 10.1002/spe

Table I. Requirements over the software life cycle.

Needs	Software life cycle			
	I. Design	II. Development	III. Deployment	IV. Execution
Software architecture (SA)	Structuring + consistency checks between system components	*Reuse of modules + support for multiple software providers*	Safe versioning and publishing + security provisioning	Traceability of software structure at run-time
Nonfunctional requirements (NF)	*Separation of concerns + code generation*	*Separation of concerns (SoC)*	*Support of existing deployment plans*	Fault tolerance
Extensibility (EX)	*Software product lines (SPL) support for models*	Reuse of legacy code + support of several languages	*Support of existing deployment technologies*	Systems open to structural/behavioral changes
Management of dynamicity (MD)	Modeling of operating modes	Environment-dependent versions of modules	Support of activities related to changes on running systems	Support of unforeseen context changes + introspection

The main objectives of Flex-eWare are outlined in italic.

invoked side, it is possible to ensure several consistency properties early in the design phase. For instance, it is possible to check that a client-side maximum allowed delay is compatible with a server-side maximum guaranteed delay.

- I.NF: Separation of concerns (SoC) and code generation. Functional and nonfunctional aspects of a system should be modeled separately at the appropriate development step. Domain-specific concerns may be abstracted and thus captured at a high level. Then code generators are able to generate the appropriate code dealing with nonfunctional requirements for the targeted domain (similarly to aspect-oriented programming [14]).
- I.EX: SPLs support for models. Current modeling languages propose features tailored for particular application domains. When unifying several languages, there are two ways to handle these variations: (i) building a unified model or (ii) build a model with flexibility points. These flexibility points enable the definition of extensions to tailor the original language to a specific need. Thus, the specification language can be designed and adapted as in an SPL.
- I.MD: Modeling of operating modes. Dynamic evolution of a system can be expressed, thanks to the definition of several operating modes and the interaction between these modes. This solution has been adopted in Architecture Analysis and Design Language (AADL) V2 [15]. Association of mode switch with mechanisms such as *introspection* (configuration discovery) or *intercession* (change on system configuration) is handled via an appropriate run-time.

2.2.2. Requirements for the development phase. This phase deals with the concrete implementation of the designed system. It also contains testing, debugging, validation, and integration of the produced systems. In some cases, design standards may require some characteristics of development process, such as code modularity or programming language. Requirements for the development phase are reported as follows:

- II.SA: Reuse of modules and support for multiple software providers. Reuse of independently developed software source code modules decreases the development effort and eases maintenance tasks through sharing of maintenance-operations experiences on independent systems. This has a direct impact over business-related metrics and, in particular, the time-to-market.

  System modules may also be implemented in parallel by several providers. Such an approach is typically used in the automotive domain where competing suppliers provide modules to more than one integrator. This requires specific support in the involved modeling languages as well as in the underlying run-time (e.g., AUTOSAR in automotive systems).

DOI: 10.1002/spe

- II.NF: SoC at a source code level. SoC is a key principle in software development. Several concerns such as run-time error treatment and communication protocols in distributed systems could be identified and separated to reduce complexity. Then they are combined by the tool chain to produce the system implementation. This approach is also similar to the one of aspect-oriented programming.
- II.EX: Reuse of legacy code and support of several languages. Complex systems may integrate pre-existing modules built using different development paradigms or no paradigm at all. Any new model or framework must consider this case and provide appropriate tools and mechanisms to integrate legacy code. There is a similar problem with programming languages because different components may have been implemented using several languages.
- II.MD: Environment-dependent versions of modules. There is a need to manage several implementations of a given module, each one being able to cope with some nonfunctional requirements. For instance, several versions of a MPEG-4 decoder may be built for different energy consumption profiles.

2.2.3. Requirements for the deployment phase. This phase deals with releasing, packaging, and installing of a system to enable its use by customers. Requirements for the deployment phase are reported as follows:

- III.SA: Safe versioning, publishing and security provisioning. There is a need to maintain consistency between versions of the various components that compose a system. For instance, backward compatibility of a component induces constraints on the versions of the depending software pieces. When publishing such systems, some dynamic linking mechanisms may be required. These mechanisms can be based on the description of provided and required, similar to Open Services Gateway initiative (OSGi) *manifests*. The identification of critical modules is important to enable safe deployment policies and protect intellectual property. For instance, AUTOSAR defines mechanisms to identify faulty components and protect modules implementation.
- III.NF: Multiple deployment policies/models support. Several deployment plans could be defined to match with several configuration requirements. For instance, according to the components installed in the host platform and the network capabilities, source code or binary content delivery may be considered.
- III.EX: Support of existing deployment technologies such as package managers, content delivery technologies, and standard file formats.
- III.MD: Support of activities related to changes on running systems, such as actions coordination, secured transmissions, and new contents (data and/or code) delivery. For instance, regarding content delivery, we identify two approaches commonly used in the CPE domain. In the *push* approach, newly released software is push onto the device by the operator. In that case, delta upload allowed by component paradigm is of paramount interest; thanks to scalability when millions of devices have to be simultaneously upgraded. In the *pull* approach, devices require new functionalities according to their needs, for example, the universal plug and play (UPnP) service discovery mechanisms.

2.2.4. Requirements for the execution phase. This phase should be reduced to the interpretation of computer program instructions by a physical processor or a virtual machine. It also deals with other activities such as maintenance, update, adaptation, and evolution of the system. Requirements for the execution phase are reported as follows:

- IV.SA: Traceability of software structure at run-time. Allowing identification of submodules that are prone to change, by establishing an isomorphism between executing code and the model. By these meanings, software behavioral modifications may be expressed as structural modifications, easing localized maintenance, adaptation, and evolution activities.
- IV.NF: Fault tolerance. Changes in the execution environment may lead to new nonfunctional requirements. For instance, bad data retrieved from a broken sensor should be handled and the source redirected to obtain appropriate data from other sources (e.g., via the network). In

DOI: 10.1002/spe

that case, a new communication link must be dynamically established to maintain the system reliability.
- IV.EX: Systems open to structural/behavioral changes. Execution run-time must be able to dynamically support structural and/or behavioral extensions. Flexibility points can be used to define run-time restrictions with regard to these changes.
- IV.MD: Support of unforeseen context changes and introspection. In some case, system dynamicity cannot be specified at an early stage of its life cycle. Thus, models and frameworks should still provide tools and execution run-times enabling system adaptation to such changes. Introspection mechanisms are required to enable system adaptation. For instance, getting the quality of a given component service is required to evaluate whether or not this component may be part of a dynamic service composition.

2.3. Covered needs

Table I proposes a full view on the need for future embedded systems. This paper reports on a subset of them, which were the focus of the Flex-eWare project (noted in italic in the table) : I.NF, I.EX, II.SA, II.NF, III.NF, and III.EX.

These needs mainly deal with flexibility at design and development. One of the main goals of Flex-eWare is to encapsulate technologies into a notation dedicated on concepts and suitable for domain-specific extensions (this is detailed in Section 4). This enables the support of MDE technologies to propose various mapping as shown in Section 5 (mapping is performed on three different technologies: Fractal, embedded Component Container Connector Middleware (eC3M), and OASIS.

Two others requirements are also partially covered in the Flex-eWare project: II.EX and IV.SA. The encapsulation mechanism eases the reuse of legacy components (II.EX) and helps to increase traceability of the software architecture (IV.SA).

Other needs are more difficult to cover so far. This is in particular the case for the management of dynamicity (MD line in Table I). Needs like I.SA (consistency checks) or IV.NF (fault tolerance) are more related to methodological issues and are not in the scope of the Flex-eWare project.

3. STATE OF THE ART

This section presents some state-of-the-art projects for designing and implementing flexible embedded systems. We deliberately put some emphasis on the work that was part of the Flex-eWare project legacy (in the sense, this was technologies better known in this context). The main reason is that we took most of our inspiration from this knowledge to set up the FCM.

Sections 3.1–3.3 briefly introduce each of these building blocks: EC3M, Fractal, and OASIS. To do so, we use the criteria identified in the previous section: architecture design and development, deployment, run-time, nonfunctional aspects, and extensibility. Then Section 3.4 reviews some other existing projects that have similar objectives but were not a main source of inspiration for FCM.

3.1. Embedded Component Container Connector Middleware

The eC3M[‡] is an integrated approach for designing embedded systems. eC3M promotes a component-based approach that is aligned with the Object Management Group (OMG) Deployment and Configuration (D&C) [16] and Common Object Request Broker Architecture (CORBA) Component Model (CCM) [17] standards. Components and connectors are the two core artifacts provided by eC3M for designing embedded systems.

Architecture design and deployment. Connectors are specific kinds of components implementing interactions. The main difference is that they need to be adapted to the context in which they are

[‡]`http://www.ec3m.net`

DOI: 10.1002/spe

used; for instance, a connector implementing an asynchronous method invocation must adapt to a specific interface that is used between two application components.

Containers shield the business logic of a component from its environment. Container services may either intercept incoming or outgoing requests or implement an additional functionality that is not provided by the business logic itself (called executor in the CCM [17] terminology). An interceptor is a specific kind of connector.

The eC3M uses the UML profile MARTE [18] (Modeling and Analysis of Real-time and Embedded systems) to define a set of UML extension targeted to real-time embedded systems. It is structured into packages covering foundations, design, analysis, and annexes. The foundation package covers among other aspects NFPs. The NFPs are defined in a generic way, allowing to define specific properties by means of a standardized model library.

Nonfunctional properties such as deadlines, jitter, and memory budgets play an important role in the definition of real-time embedded systems because the correctness of the system requires that all nonfunctional requirements are met. The MARTE library standardizes frequently used properties such as durations and arrival patterns. The elements of the library are typically datatypes whose attributes may cross-reference to NFP types. An example is the real-time feature (RTF) data type that has a relative deadline attribute typed as a `NFP_duration`. Another attribute is an arrival pattern having different specializations. With respect to flexibility, it is important that NFP types are defined in a library and are thus extensible to suit domain needs.

Extensibility. Connectors and container services are not fixed; they can be defined in model libraries in a quite similar way as application components are. An application model may import the model libraries that are suitable for the application domain. The libraries are thus the primary extension mechanism in eC3M.

As already mentioned, the main difference between components and connectors is the ability of the latter to adapt themselves to a usage context. This ability is modeled by means of UML templates, that is, the possibility to refer to formal parameters like for instance a port type. In a template instantiation process, the formal template parameter is bound to an actual parameter. Implementations are instantiated as well and may be defined by means of Acceleo[§] templates.

The extensibility in terms of containers and connectors enables an adaptation to the application domain to define SPLs and to manage variability. Subcomponents within a composite may optionally be specified via a type instead of an implementation. If this is carried out, the choice of the implementation to use is delayed until the deployment phase, when instances and their allocation are defined. The implementation choice may depend on the allocation, that is, on properties of the node (such as available space, operating system (OS), and processor architecture). Another aspect is that the use of different connectors facilitates the use of different deployment architecture, for example, a deployment architecture in an automotive platform.

Run-time adaptation. The focus of eC3M is currently on statically deployed applications. It is possible to change the assembly by reconnecting ports and instantiating components at run-time. However, this must be carried out programmatically; that is, one of the application components must explicitly instantiate new components and call the port connection operations. Current work aims to express variability at model level and support automated transitions between the variants. In this context, we also seek to support the update of components implementations and the re-instantiation of existing components with a new implementation.

3.2. Fractal

Fractal [19] is a hierarchical, reflective, and open component model. Fractal components can be nested at any arbitrary level of granularity required by the modeled system, component assemblies can be navigated to discover and modify at run-time the architecture of an application, and the component containers can be programmed to customize the hosting and execution semantics. The

[§]`http://www.acceleo.org`

DOI: 10.1002/spe

Fractal component model is independent from programming languages, and run-time supports exist for Java, C, and as prototype implementations for Smalltalk, C++, .NET, and Python. Fractal is a project[¶] of the OW2 (previously known as ObjectWeb) consortium for open source middleware. Fractal/Think [20,21], which is one of the existing run-time support of the Fractal component model for the C language, is used in this article (see Section 5.1).

Architecture design and deployment. The description of the architecture and the configuration of a Fractal system is conducted with Fractal ADL [22], which is an XML-based ADL. Fractal ADL provides a language for describing component hierarchies, component communication links, and component properties. A tool chain is provided to parse, deploy, and instantiate a Fractal system. The tool chain can be extended to accommodate different needs and properties. For example, one may need to specify real-time related properties such as worst-case execution time or periodicity for a component or to specify deployment related information such as the computing node on which a component ought to be deployed.

For this extensibility to be allowed, the tool chain is divided into three parts: a loader, a compiler, and a builder parts. Each of these parts are themselves component-based with typically one component per concept of the ADL. The loader components build the abstract syntax tree (AST) corresponding to the architecture descriptor, the compiler components generate the set of instantiation and deployment tasks, and the builder components execute these tasks.

Customizing the ADL is then a matter of providing the corresponding loader, compiler, and builder components that fit the extended definition. Leclercq *et al.* [22] show how the Fractal ADL tool chain can be extended to support the design and the deployment of a heterogeneous multimedia system for video decoding. The application is composed of some legacy Java and C components and extended with Join Specification Language programs, which is a domain-specific language for specifying synchronization and concurrency constraints.

Extensibility. The Fractal component model is extensible in the sense that components can be endowed with arbitrary reflective capabilities, from plain black-box objects to components that allow a fine-grained manipulation of their internal structures. This feature has been motivated by the fact that existing component models (see for example [23] for a survey) fail from delivering a solution where components can fit various run-time environments and requirements: the model is either general purpose, for example Enterprise JavaBeans, or tailored for a precise application domain. This generality or this specialization stems from the execution semantics and the technical services that are provided by the framework to the hosted components. With Fractal, instead of mandating a particular execution semantics or a set of fixed and predefined technical services, the component containers (so-called membrane in the Fractal terminology) are open and programmable. Membranes are decomposed in controllers that implement a piece of the hosting logic. Controllers expose their services through control interfaces. Extending the Fractal component model is then a matter of providing the corresponding control interfaces, controllers, and membranes.

Run-time adaptation. The default execution semantics of a Fractal component comprises three main parts implemented as controllers: hierarchy management, binding management, and life cycle management. Each of these parts provides a set of CRUD (Create, Read, Update, and Delete) operations for managing parent–child relationships between components, communication links between components, and starting/stopping components, respectively. In addition, the framework provides a component factory for dynamically instantiating components at run-time.

3.3. The OASIS tool chain for safety-critical real-time systems

OASIS [24] is a tool chain for building safety-critical real-time multitask systems where the system behavior is independent from the asynchrony that is allowed during the execution of an application. The system behavior is therefore unvarying, unique, and independent from its realization on a

[¶]`http://fractal.ow2.org`

DOI: 10.1002/spe

target computer. Consequently, OASIS allows a deterministic and predictable execution of safety-critical real-time systems, thus guaranteeing specified dependability properties. OASIS comprises, in a programming model, its associated offline tool chain and a safety-oriented real-time kernel that implements a multiscale time-triggered execution model. The OASIS kernel is available on various architectures and is currently in use in industrial products in the nuclear field [25].

Architecture design and deployment. A specific programming language, called ΨC, is used to describe the architecture of an OASIS application, that is, the real-time tasks called agents, their communication links, and their temporal behavior, as well as the applicative C code. An agent is composed of a set of sequential procedures, called *elementary activities* (EA), which have precedence relationships expressed through deadlines on the basis of a common physical time. The execution of an EA is bounded by its earliest starting date (the deadline of a the previous EA) and its deadline, the latest date by which it must be finished. This defines the temporal behavior of an agent. The temporal width of each EA is set by the developer with ΨC. OASIS does not introduce constraints on the manner that application are decomposed into agents, and the temporal behavior of agents can be periodic or not, regular or not.

Classical consistency checks are performed by the OASIS tool chain on communication interfaces, such as on data type. Furthermore, as the temporal behavior of an OASIS agent is fully specified, the size of buffers used to implement communications can be computed to ensure that any attempt of buffer overflows will be detected. This participates in the dependability property of the OASIS approach. Besides, the fulfillment of end-to-end temporal constraints by an application can be demonstrated by construction.

On the basis of the static description of the application, binaries are generated by the OASIS tool chain that can be used by OASIS kernels for execution. The temporal and spatial isolation mechanisms of OASIS ensure the traceability of the software structure at run-time through a strict control of the behavior of agents.

Extensibility. As communication interfaces and their temporal behaviors are fully specified, agents can be composed at both the source and binary levels. Consequently, agents can be reused in various applications and can be provided by different software suppliers. In addition, legacy code can easily be reused by encapsulating binary objects within an agent at the linking step of the construction of binaries.

In OASIS, communication latencies between agents are never considered as null. Therefore and from the programming model point of view, OASIS can be *transparently* extended to various architectures without requiring changes in the software architectures of applications. For instance, the OASIS approach has been extended from mono-processor to distributed [26] or symmetric multiprocessing architectures [27] transparently from the application developer point of view. All low-level details such as network scheduling or allocation of cores to agents are managed by the OASIS tool chain and its associated kernel.

Run-time adaptation. OASIS assumes a static description of the temporal and functional behavior of agents that are part of an application. Future work includes reconfiguration of an application to different temporal and functional behaviors in case of, for instance, software errors or hardware failures.

3.4. Other approaches

In addition to the previously identified technologies, many other approaches provide some solutions, full or partial, to the problems identified in Section 2. We briefly review some of them in the following.

Architecture Analysis and Design Language. AADL [28] is a modeling notation with both a textual and graphical representation. It provides modeling concepts to describe the run-time architecture of

DOI: 10.1002/spe

systems in terms of concurrent tasks and their interactions as well as their mapping onto an execution platform.

Architecture Analysis and Design Language offers threads as schedulable units of concurrent execution, processes to represent virtual address spaces whose boundaries are enforced at run-time, and systems to support hierarchical organization of threads and processes. AADL supports modeling of the execution platform in terms of processors that schedule and execute threads, of memory that stores code and data, of devices such as sensors, actuators, and cameras that interface with the external environment, and of buses that interconnect processors, memory, and devices. Threads can execute at given time intervals (periodic), triggered by events (aperiodic), and paced to limit the execution rate (sporadic), by remote subprogram calls (server) or as background tasks. These thread characteristics are defined as part of the thread declaration.

Architecture Analysis and Design Language offers extensibility thought the definition of new 'properties' in the model.

Open Services Gateway initiative. OSGi [5] provides a service-oriented environment initially focused on solutions for embedded Java and the networked devices markets. OSGi offers some standardized ways to manage the software life cycle and to discover services in a distributed environment. OSGi defines a framework extended by system services (i.e., log, user administration, etc). A user application is an aggregation of bundles that are described in a manifest (i.e., bundle name, provided and required interfaces, etc). Therefore, the OSGi flexibility is mainly focus on dynamic software deployment.

Some component models such as iPOJO [29] have been implemented on top of OSGi and provide means for describing and deploying a component-based architecture.

For embedded system concern, usual OSGi frameworks are not suitable for different reasons (i.e., memory management, resource sharing, scheduling mechanisms, etc). Although real-time specification for Java (RTSJ) meets these needs, executing an OSGi framework on top of RTSJ is not sufficient [30]. Some initiatives like [31, 32] are focused on the design of OSGi with RTSJ by providing, for instance, a temporal isolation.

Universal plug and play. UPnP [6] is a technology that provides an architecture for network discovery and connectivity of appliances, devices, and computing equipment of all sorts. With UPnP, a device can dynamically join a network, obtain an IP address, convey its capabilities, and learn about the presence and capabilities of other devices. Finally, a device can leave a network smoothly and automatically without leaving any unwanted state behind. UPnP covers the steps of network discovery, service description, remote invocation, and event publishing.

Automotive Open System Architecture. AUTOSAR [9] is a software architecture standardized by the automotive industry. It is the result of a development shift from ECU or ECU-based approaches, where ECUs are supplied as black boxes, to a function-based approach. It defines a basic infrastructure defining a clear separation between application software, software services, and hardware, which are typically supplied by separate stakeholders, that is, automotive manufacturers, suppliers, and systems software developers.

Automotive Open System Architecture supports a design process including a specific configuration and generation phase. Configuration involves selecting information on the overall vehicle system in which a given application component will be integrated such as the list of ECUs, the network used, and so forth. Generation involves the integration of application software with system software and configuration information into predetermined static computing configurations, an industry requirement for today's resource-constrained embedded systems.

Automotive Open System Architecture clearly advances toward component-based design, but it still lacks features to enforce suitable SoC including between functional and NFPs.

Finally, AUTOSAR flexibility is ensured to throw a clear definition of interfaces. Automotive manufacturers can easily assemble different components from different stakeholders. A drawback is that the flexibility at run-time is low (i.e., mode management).

DOI: 10.1002/spe

Component Synthesis using Model Integrated Computing. CoSMIC [33] is a tool suite to build distributed real-time embedded applications based on both the OMG CCM and D&C specifications. Applications in CoSMIC are modeled using a set of description languages: Platform Independent Component Modeling Language to describe the components and their QoS parameters, Component Descriptor Modeling Language to describe how components are deployed and configured, and Options Configuration Modeling Language to describe the middleware configuration options. The applications are built on top of the component middleware CIAO (CCM implementation over TAO), which offers capabilities to separate the development of the application from its deployment and configuration.

Component Synthesis using Model Integrated Computing supports flexibility mainly at the level of QoS options that are related to the policies of the underlying RT-CORBA ORB (TAO). The fact that CoSMIC is based on several different languages to specify an application means that each representation must be consolidated after any change on the application model. The topmost layer used to dynamically refine components properties is problematic for critical systems where all resources must be allocated statically. These drawbacks restrict the use of CoSMIC to distributed real-time embedded systems where no correctness by construction is required.

3.5. Synthesis

Table II summarizes the characteristics of the studied approaches for building flexible embedded systems. The four proposed categories are major features provided by these solutions. They are mapped from the life cycle phases identified in Section 2. These characteristics serve as input and building blocks for the FCM metamodel defined in the next section.

In terms of design and development, all studied approaches propose a software artifact introducing variability/flexibility and support code encapsulation. Even if the terms differ, the purpose is shared among all work in the state of the art.

A hierarchical vision of system design, although not provided by all approaches, seems also to be a key characteristic. This enables decomposing a system into subsystems where each subsystem can be designed independently from the other ones. This increases system flexibility by enabling designers to focus on smaller software units. For instance, eC3M allows to specify several implementations per component type. This broadens the scope of target platforms for the system because we can select the implementation that better fit a given execution context.

Finally, some approaches provide an explicit support for nonfunctional services, such as in the case of eC3M and Fractal.

In terms of deployment, all studied approaches provide some kind of descriptors (usually XML-based) to specify configuration data used when deploying a system on a target platform.

At run-time, flexibility is ensured through some mechanisms for reconfiguring dynamically the deployed system. This is achieved with an API that either modifies the assembly, like in Fractal, or enables switching between different execution modes, like in AUTOSAR or AADL.

On the basis of these characteristics and the requirements identified in Section 2, the next section proposes a software component metamodel for flexible embedded systems: FCM.

4. THE FLEX-EWARE COMPONENT MODEL

This section presents the FCM, which is our solution for designing and implementing flexible and reconfigurable embedded software systems. This model covers the life cycle phases of design, development, and deployment.

A major objective of the FCM is to be a general purpose model for embedded systems and to enable designing systems that will be later on implemented with different technologies. In this respect, Section 5 provides three case studies that illustrate how the FCM is used for Fractal, eC3M, and OASIS. Another key objective of the FCM is to be flexible, that is, being *adaptable and extensible without modifying the metamodel itself*. The main idea to achieve this goal is the use of generic elements in the metamodel that are instantiated by *model libraries*.

DOI: 10.1002/spe

Table II. Comparison of features for building blocks approaches in terms of flexibility.

Category	Requirements	eC3M	Fractal	OASIS	Other approaches
I. Design	Encapsulation	Component	Component	Agent	OSGi: bundle AUTOSAR: component UPnP: service AADL: component
	Component hierarchies	Composite component	Supported	NS	OSGi: extensions, e.g., iPOJO [29] AADL: supported
II. Development	Assembly	Connectors within composite	Architecture Description Language	Implicit	AADL: several implementation per component
	Nonfunctional aspects	Container	Membrane	Temporal	AADL: by means of 'properties'
III. Deployment	NS	Component deployment plans	Architecture Description Language	Static	OSGi: manifest file in bundle UPnP: service descriptor AUTOSAR: XML configuration data AADL: mapping of components to hardware
IV. Execution	Alternative systems	Choice between several static CDPs	Assembly re-configuration API	Static	OSGi: dynamic code deployment AUTOSAR: mode management
	Introspection	Container service	Assembly introspection API	NS	OSGi: manifest file in bundle UPnP: remote introspection of device descriptors

eC3M, embedded Component Container Connector Middleware; OASIS, Organization for the Advancement of Structured Information Standards; OSGi, Open Services Gateway initiative; AUTOSAR, Automotive Open System Architecture; UPnP, universal plug and play; AADL, Architecture Analysis and Design Language.

4.1. Underlying principles

The design of the FCM metamodel is based on four main principles. These are detailed in the following and concern the definition of components, connectors, ports, and extension mechanisms supported by the metamodel.

Distinction between component type and implementation. It is possible to provide multiple implementations of a type, for instance, with different QoS properties or suitable for a specific target (OS and/or hardware architecture). The main benefit is that an application architecture may refer only to a type whenever the implementation may vary or is deployment specific. It can be fixed at a late phase in the product life cycle (deployment, III in Table I) enhancing reusability and flexibility. A type defines a well-encapsulated entity that may own configuration properties (which are typically application-level configuration properties) and explicit interaction points called ports.

Explicit connectors and connector types. The FCM provides the ability to model *connectors*, specific variants of components that describe interactions as well as their implementation. Thus, new interaction mechanisms can be added by extending model libraries that define connector types and implementations. This makes it possible to tailor interaction mechanisms to domain needs, for example, provide synchronous calls with configurable timeouts or implementations that are optimized for a specific real-time operating system such as OSEK [34] in the automotive domain. Connectors have

DOI: 10.1002/spe

a role within a composition, carry a type and can be realized by one or more implementations. The main difference is that they are typically defined within a template because they have to be able to adapt themselves to the context in which they are used.

A uniform way of defining new kind of ports. Instead of fixed kinds of ports (e.g., one for events and another for invocations), a port in FCM is characterized by its type and its *port kind*. The port kind is part of a general or domain-specific model library and is associated with a certain (informal) semantics. A mapping rule associated with the port kind describes how to derive provided or required interfaces from the port type.

Extension mechanism. The objective of the extension mechanism is to identify elements that may need to change to react for instance to domain requirements or new underlying technologies. These elements are ports, interaction semantics and their implementation, and container services.

A constraint is that extensions should be possible *without modification of the metamodel* because this would require an adaptation of modeling environments (tools). Thus, extensions are specified via modeling libraries: domain-specific connector types and implementations in connection with suitable port kind definitions enable the customization of interaction mechanisms. Specific components and connectors (interceptors) defined in a model library extend the available choice of services within a container.

4.2. Architecture of the metamodel

The FCM metamodel identifies two main packages: `BasicFCM` and `CompleteFCM`. The diagram of these two packages is depicted in Figure 1. Concepts in the `BasicFCM` package mainly address issues related to composition, and concepts in the `CompleteFCM` package address issues related to deployment.

The `BasicFCM` package defines the basic concepts associated with a FCM component. Many elements in our component model have a name. The FCMCore subpackage defines a specific meta model element called NamedElement that reflects this: it is a common superclass for all model elements that have a name. For related elements to be organized, a common concept is to introduce name spaces, that is, to *package* related elements. But not all elements can be owned directly by a package (e.g., attributes are owned by a component, as shown in the next paragraph). As for the named element, we introduce a superclass that captures the concept of elements that 'can be packaged'. Please note that the package is itself a packageable element, enabling arbitrary nesting.

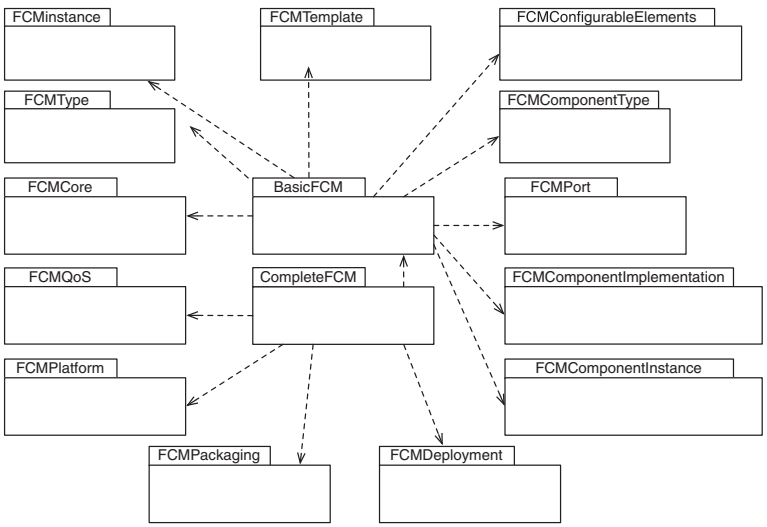

Figure 1. The Flex-eWare component model architecture.

DOI: 10.1002/spe

A component is an entity of encapsulation. It is characterized by (1) its name, (2) a set of well-defined interaction points (called ports), (3) the set of configuration attributes that it owns, and (iv) its behavior. As said before, an important and quite common concept is the separation between a component type and its implementation. The first three aspects define a component type that is specified in the package `FCMComponentType` (while relying on the specification of ports and attributes within other packages). The fourth aspect is only relevant for a component implementation. In the sequel, we state which packages of the metamodel deal with component characteristics (2) to (4).

(2) Ports (see `FCMPort` package) are a fundamental concept of component modeling. This common encapsulation mechanism exists in most component-oriented frameworks, even though the name and the semantics given to this concept may vary according to the framework. A major characteristic of ports in FCM is that they are not only characterized by a type but also by a kind (PortKind). The kind carries an informally specified semantics and a rule that characterizes the port in terms of provided and required interfaces (mapping rule). This mechanism enables the extensibility of ports: instead of defining a specific metamodel element for each kind of port (e.g., a port that provides an interface, a port that consumes events, etc.), a single generic port is used. New port kinds can be defined in modeling library (i.e., without modifying the metamodel), along with a mapping rule for provided and required interfaces. Because of their role in the context of ports, interfaces are introduced as a set of operation signatures within FCM. They are actually the only concrete kind of 'types' defined by the `FCMType` package that introduces the generic notion of a type and typed elements, that is, elements such as ports that have a type. The metamodel remains voluntarily generic about what a type is, except for interfaces.

(3) The component type owns a set of configuration attributes. The basic idea is that an *instance* of a component fixes the value of such an attribute. The ability to have attributes is inherited via the superclass `FCMConfigurableElement` defined in the package `FCMConfigurableElements`. Besides the component type, other elements (notably port kinds) inherit from this metamodel element or its variant `ConfigurableElementInstance`.

(4) A component implementation (see package `FCMComponentImplementation`) is a realization of a component type. The implementation is either monolithic or described as an *assembly* of parts (i.e., some manifestation of component types or implementations assembled together). In case of the latter, an implementation owns a set of parts and a set of *connectors* that connect the ports of these parts. A connector has a type (see connector type mentioned previously) and an implementation (ConnectorImplementation), which is a specification of a component implementation.

Connectors are a specific variant of components that are responsible for interactions. In the metamodel, a ConnectorType inherits from ComponentType without adding any particular properties. Likewise, a connector implementation inherits from a component implementation. This concept is important for extensibility: instead of having a predefined set of interaction mechanisms, a connector type describes interaction patterns, and a connector implementation is a possible realization of this pattern. A specific property of connectors is that their definition is not fixed because they need to adapt to the context in which they are used; for example, a connector port may be typed with a placeholder type that is later replaced by concrete component type. This mechanisms is captured with the `FCMTemplate` package. The idea is to be able to capture generic model elements (i.e., with explicit template parameters) that are representative of a particular application-domain and/or particular target technologies. These generic elements can then be made application specific by simple and systematic parameter bindings.

`FCMInstance` and `FCMComponentInstance`. The `FCMInstance` package introduces mechanisms for specifying statically (i.e., at design-time) run-time instances. An instance specification has a set of slots that associate a model element with a value (ValueSpecification). These mechanisms are inspired by UML2. The FCMComponent instance package defines an extension of the generic instance specification in case of components, that is, specific slots that reference contained parts, ports, or connectors. For instance, a PartSlot associates the parts of a component with a value that they have within a specific instance.

DOI: 10.1002/spe

The `CompleteFCM` package defines extended features related to deployment associated with a FCM component in the following packages:

- `FCMQoS`. QoS definitions within the FCM are based on QoS aware types and QoS expression. A component implementation owns a set of QoS expressions. However, there is no concrete mechanism on how QoS expressions are formed because the QoS definition (nonfunctional aspects in general) should not make use of a particular formalism. This enables the use of existing means to define QoS properties. A mechanism that is intended to be used in conjunction with FCM is the UML profile MARTE that features a library with basic NFP types and a value specification language.
- `FCMPackaging`. Packaging allows to bundle one or more implementations of the same component type within a single unit. The basic idea is to have a single artifact that represents a component to facilitate component deployment and installation without fixing a certain technology how the contained parts are stored (e.g., in a ZIP file).
- `FCMServices`. This package offers the possibility to define so-called services that intercept interactions through a port (before and/or after an invocation). Similar to a connector, a service is an extension of a normal component; that is, it has a separation between type and implementation and can be defined in a model library. The latter implies that the set of services can be extended depending on domain needs. A service is typically realized within a *container*. However, the concept of a container is not part of FCM itself because from a modeling viewpoint, it is sufficient to specify which services should be activated for a component instance.
- `FCMPlatform`. A platform (FCM domain) is characterized by a set of elements that are either processing resources (Node) or communication resources (e.g., Bus). This concept can be extended as required for certain domains (e.g., to add specific communication resources).
- `FCMDeployment`. This package defines primarily the concept of a static allocation of component instances on nodes (Node). This information is captured by a DeploymentPlan (adopting CCM terminology) that owns a set of deployments that associate instance and node.

4.3. Dimensions of flexibility

As mentioned in Section 4.1, a major design criterion of the FCM metamodel has been to enable flexibility without a *modification of the metamodel* but with introducing model libraries. This section summarizes the four main flexibility dimensions that are achieved by the FCM.

- New component *ports*. New component ports enable the implementation of new interaction mechanisms. The ability to extend ports via the definition of a port kind element within a model library is the first building block for flexible interactions and is enabled via the package `FCMPort`.
- New *connectors*. The second flexibility dimension is provided by the ability to define new interaction components along with their realization in a model library. As previously mentioned, connectors with the FCM are variants of components. They are thus specified as specialization of component types (package `FCMType`) and component implementations (package `FCMComponentImplementation`).
- New *NFPs*. The `FCMQoS` package does not assume a particular language for defining QoS expressions. This enables the use of languages or approaches tailored to a particular application domain as long as the definition of specific NFPs.
- New *containers*. The `FCMServices` package allows the definition of new container services by means of a model library. Because services are components embedded into the container, there is also a separation between type and implementation.

Overall, the FCM metamodel provides a common ground for designing and implementing component-based systems where the concepts such as component, port, and connector can be specialized to match the specificities of run-time platforms. We illustrate this in the next section with three case studies on three different platforms: Fractal/Think, eC3M, and OASIS.

DOI: 10.1002/spe

5. CASE STUDIES

This section illustrates the use of the FCM that has been presented in the previous section, on three use cases: wireless sensor networks (WSNs; Section 5.1), distributed client/server applications (Section 5.2), and control systems for electrical devices (Section 5.3). For each use case, the concepts defined in the FCM are mapped onto different technologies: the Fractal/Think component framework [20, 21], the eC3M middleware, and the OASIS tool chain [24].

This section covers a broad range of usages and technologies for embedded systems and wishes to demonstrate the adequacy of the FCM in all these cases. Furthermore, each use case emphasizes a particular aspect related to the design and implementation of embedded systems: reconfiguration, low memory footprint, and software component reuse, respectively.

All three presentations follow the same pattern. We start by introducing briefly the case study and the platform. We then present the mapping of FCM concepts onto the platform, give an overview of the toolchain associated with the platform, and report on some experimental data. We concluded by highlighting the flexibility dimensions of the FCM that have been put into practice by the case study.

5.1. Case study 1: FCM over Fractal

Our first case study is in the domain of WSN. This domain is rather broad, going from city automation services (e.g., smart public lighting, waste management) to personal healthcare services (e.g., continual medical monitoring) and to CPE. The target execution platform for this case study is the Fractal/Think [20, 21] component framework, which is a C implementation of the Fractal component model (see Section 3.2).

The Think compiler supports a set of *flexible-oriented properties* [35] for designing reconfigurable wireless embedded systems. These properties are used to configure the Think compilation process: first, to generate the metadata allowing to reify the Fractal component concepts at run-time (e.g., to retrieve a component attribute or the descriptor of a bound interface); second, to generate the standard Fractal controller implementations over these metadata [36]. As dynamic reconfiguration may not be necessary for all system components, the Think framework provides fine-grained mechanisms to specify whether a simple component attribute, a single component, or a subset of system components is not likely to evolve at execution time.

These features allow to generate minimal reconfiguration infrastructures, optimizing available resources usage in accordance with application domain needs [37, 38]. These are typical non-functional concerns that can be expressed by extension mechanisms provided within the FCM metamodel.

5.1.1. Mapping of FCM concepts. We briefly outline below the mapping between FCM and Fractal. Readers may refer to [39] for further details.

Mapping of generic FCM ADL concepts. The mapping between FCM and Fractal ADL is straightforward because the latter is a building block for defining the FCM metamodel presented in Section 4. Therefore, most of the FCM concepts can be directly mapped toward Fractal model entities, apart from two features not handled by Fractal: (i) An FCM model relies on three levels of architecture's specification – *type*, *implementation*, and *instance* – whereas Fractal ADL focuses on the latter level. Thereby, a FCM instance model is the only entry point of the mapping process between FCM and Fractal. (ii) The concept of *port* is not supported by Fractal, where component interactions are only specified by a *binding* between a single *required interface* and a single *provided interface*. As the concept of FCM interface is isomorphic to the one of Fractal, the mapping rule merely consists in translating each FCM *connector instance* into a set of Fractal *bindings*, according to the set of required and provided interfaces attached to both FCM *port ends*.

Mapping of reconfiguration features. We rely on the extension mechanism defined in the FCM and presented in Section 4 for specifying reconfiguration capabilities:

DOI: 10.1002/spe

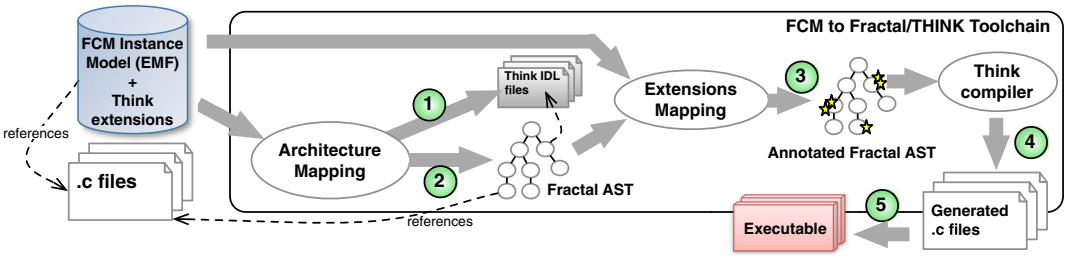

Figure 2. Flex-eWare component model to FRACTAL/THINK process.

- Within the FCM model library, we define the set of services implemented by the Fractal reconfiguration controllers. The FCM developer then specifies which *component containers* of its application must provide local reconfiguration capability at run-time.
- The flexible-oriented properties defined by Think are modeled at FCM level by means of *QoSExpressions*. These expressions can be attached to any FCM elements and can be configured to be interpreted recursively by the Think compiler (e.g., for specifying with a single *QoSExpression* attached to a composite that it must be applied to all of its subcomponents).

5.1.2. Overview of the process and associated tools. The mapping from an FCM instance model to a Think executable is sketched out in Figure 2. The numbered steps correspond respectively to the following treatments:

1. For each FCM interface signature, a corresponding file is generated in the Think Interface Description Language (IDL).
2. The FCM architecture description is translated into Fractal AST nodes, which is the internal architectural representation used by the Think compiler.
3. The *flexibility-oriented properties* set by the FCM developer as *QoSExpressions* are interpreted, and the corresponding fine-grained properties expected by the Think compiler are inferred in consequence. The same mechanism is used to set the Fractal *containers* specified at FCM level. This step outputs an annotated Fractal AST that feeds the compiler.
4. The Think compiler maps architectural elements to C variables in implementation code, transforms existing functional code, and produces metadata and Fractal controller implementations according to the annotations attached to the AST nodes. In addition, it generates the code implementing the bootstrap process of the system.
5. Finally, the set of C source files generated by Think are compiled and linked by a classical C compiler.

5.1.3. Experiments. We designed a typical WSN infrastructure whose purpose is to monitor and manage a group of sensors and/or actuators deployed in the field (e.g., buildings, factories, forests). These devices form a Zigbee network that is administrated via an asymmetric digital subscriber line or general packet radio service Internet connection. Measured data are sent to an oBIX [40] server and are available for consultation via a web-based graphical interface. Additionally, administrators are able to remotely modify device architectures. The left side of Figure 3 shows a simplified version of our infrastructure.

Because we are interested in evaluating run-time reconfiguration capabilities in resource-limited systems, we focus on the Zigbee network devices, which typically expose this kind of constraint. In our case study, the Zigbee network is mainly composed of AVRRAVEN boards including an Atmega1284p processor (8-bit AVR, 128 KB of Flash memory, 16 KB RAM, 4 K EEPROM) and a Atmega3290 processor dedicated to liquid crystal display (LCD) management. These devices are coordinated by a RZUSBSTICK board (90USB1287 8-bit processor) bound to the Zigbee/HTTP gateway. On each sensor node, an FCM architecture is deployed, as illustrated in the right part of

DOI: 10.1002/spe

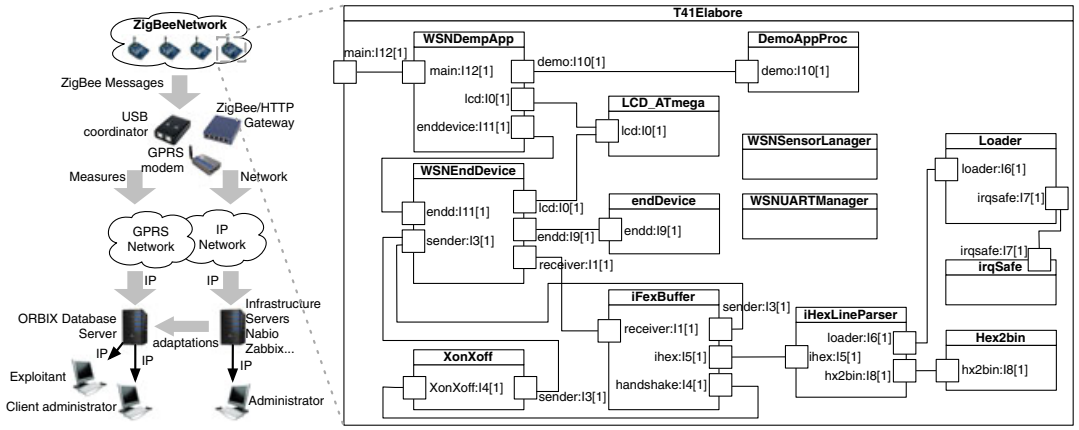

Figure 3. Global infrastructure and Flex-eWare component model architecture instance deployed on each sensor node.

Figure 3. This architecture implements the required services to dynamically update or change its components sent in a binary form via the Zigbee network.

Dynamic reconfiguration experiment. One of the reconfiguration examples we ran was motivated by the need for changing the way remote devices present data to in-site users through its embedded display. The goal is to replace the LCD manager component `LCD_ATmega` (see Figure 3) by a new version of it, `newLCD_ATmega`, during system execution. For it to be accomplished, the following operations are executed:

1. The Zigbee network coordinator sends a predefined message to the device, which passes to a special *Reconfiguration* mode.
2. The Zigbee network coordinator sends the `newLCD_ATmega` component to the device. This new component was previously converted into an Intel HEX format. Code and Data are sent through the network line-by-line.
3. Once the transmission is completed, a reference to a Fractal interface implemented by the container of `newLCD_ATmega` and allowing its run-time introspection is retrieved by the `WSNEndDevice` component. This introspection service allows to retrieve the provided interfaces of the uploaded component.
4. The initial bindings to `LCD_ATmega` are destroyed and replaced to bound its uploaded instance, thanks to a Fractal controller implemented by the container of the `WSNEndDevice` component.
5. The device returns to a *Nominal* execution mode. In this particular case, the device is rebooted. This could be avoided if component containers expose and implement the Fractal life cycle controller that ensures a safe transition between *Reconfiguration* and *Nominal* modes. However, providing this service at run-time has a non-negligible impact in terms of memory footprint, which is the most critical performance issue for WSN applications, as discussed in the next section.

Low resources usage experimental results. Table III presents the memory footprint of a binary generated from the FCM model instance shown in Figure 3 intended to be deployed in a sensor node. 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 read only memory, whereas data are generally placed in random access memory. Table III(a) presents the footprint of the application code compared with the code generated by the Think framework. We consider three scenarios. (i) If none of the FCM extensions presented in Section 5.1.1 are used to explicitly specify the reconfiguration points of the architecture, Think generates by default metadata and Fractal controllers for the whole system (Table III(b)). (ii) In the second scenario, the

DOI: 10.1002/spe

Table III. Memory footprint sizes (in bytes) of the case study, for the functional part and the Think Framework part.

	Functional part	Think framework part		
	(a)	(b)	(c)	(d)
Scenarios	–	Highly flexible	Highly flexible	Static
	–	All controllers	Required controllers	Required controllers
Code	5462	+59.8%	+25.0%	+10.8%
Data	619	+87.0%	+66.6%	+36%

Think metadata are generated for the whole system, but only the mandatory Fractal controllers are deployed to implement the reconfiguration scenario with the `newLCD_ATmega` component presented previously (Table III(c)). (iii) Finally, only the mandatory Fractal controllers and metadata are deployed (Table III(d)).

These results show that a fine-grained tuning of the architecture reconfiguration points is a required feature to fulfill the constraints of wireless embedded systems. By the use of FCM extensions, we provide to the developer high-level mechanisms to explicitly deploy only the mandatory services required by a reconfiguration scenario. The induced overheads are then paid only where necessary.

5.1.4. Flexibility dimensions. This case study puts into practice two of the four flexibility dimensions identified in Section 4.3 and their associated model libraries – *container* and *NFPs* – reifying at model level the specificities of the Think run-time platform. The container dimension enables dealing with the reconfiguration controllers. The set of reconfiguration services supported by Think has been therefore defined as an FCM model library directly usable within the end-user's specifications. The NFPs dimension concerns the *QoSExpressions* for Think *flexibility-oriented properties*. This extension mechanism provided by FCM offers a straightforward mean to decorate model artifacts with annotations. These annotations are in turn used to drive the interpretation process leading to the generation of Think executables.

5.2. Case study 2: FCM over eC3M

The second case study is in the domain of distributed client/server applications with the eC3M middleware platform presented in Section 3.1.

5.2.1. Mapping of FCM concepts. Because eC3M is directly based on the FCM profile, no mapping is required. Readers may refer to [39] for further details. Application models typically contain additional information in the form of MARTE stereotypes to specify real-time aspects. An example is the RTF of the MARTE 'High-Level Application Modeling' (HLAM) approach. In general, the MARTE value specification language and the standardized NFP library (Annex D of the MARTE specifications) are used to specify nonfunctional parameters, notably durations (NFP_duration).

5.2.2. Overview of the process and associated tools. An overview of the eC3M toolchain is provided in Figure 4. The main specification artifact is a UML model enriched by information from the profiles FCM and MARTE. A *set of model transformations* is executed to transform the component-based model into an object-oriented model on which standard UML to code generators, in particular UML to C++ generators, can be applied.

These transformations include the following:

- The reification of connectors – that is, replacing FCM connectors (stereotyped UML connectors) with interaction components that are adapted to the application context; that is, use

DOI: 10.1002/spe

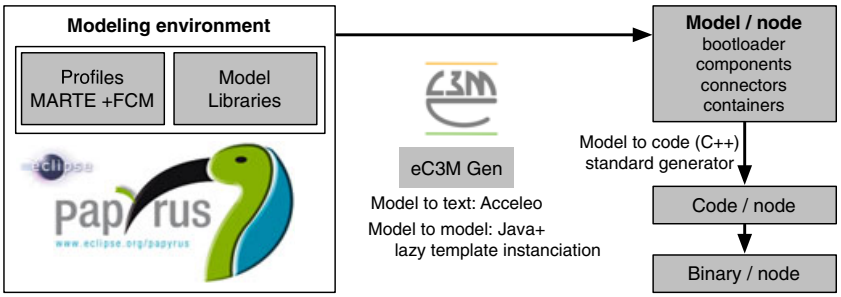

Figure 4. The embedded Component Container Connector Middleware toolchain.

port types that are compatible with those of the application components (and implementations adapted to these port types as well).

- The implementation of the container pattern – that is, redirecting connections to an application component with connections to the container that embeds the application component.
- Apply standard design patterns that transform components into standard classes – that is, replace ports with functions related to manipulate connections and obtain references. This function is a bit similar to CCM IDL3 toward IDL2 mapping.
- Create a subset of the model per node on which an application is deployed. Each of these models contains a bootloader that is responsible for instantiating the components that are deployed on this node (in the context of a static deployment).

5.2.3. Experiments. The eC3M model has so far been used for some sample applications, including a data acquisition system. In the sequel, we examine a very simple system consisting of a client and a server component, as shown in Figure 5 (the interface ICompute consists of two operations: *add* and *mult*). For initial activities to be started, eC3M uses a simple convention: the client owns a port providing the standard eC3M interface called *IStart*. This interface includes the operation *run* (similar to the Java *Runnable* interface) that is automatically invoked during the system start-up.

The client can use the 'standard' FCM port kind UseInterface resulting in a derived required interface that corresponds exactly to the interface that types the port – in this case, *ICompute*. In real-time applications, the caller may want to pass for instance a period length (to enable automatic cyclic invocations) and a relative deadline along with the operation invocation. The MARTE RTF is a standardized data structure for these real-time properties. One option to pass the RTF property with a call is to simply add it as an additional parameter to an operation. Instead of manually modifying the interface, a client developer can change the kind of port and use a variant that calculates a derived interface with an additional RTF parameter automatically. In this case, the interaction between client and server needs to be realized by a connector evaluating the RTF parameter and calling the unmodified interface of the server.

Another quite frequent need is that the client is not blocked while waiting for a result. CORBA calls this asynchronous messaging (AMI) and standardizes two options: either a modified operation signature returns 'poller' objects that can be queried later for result data or the client is called

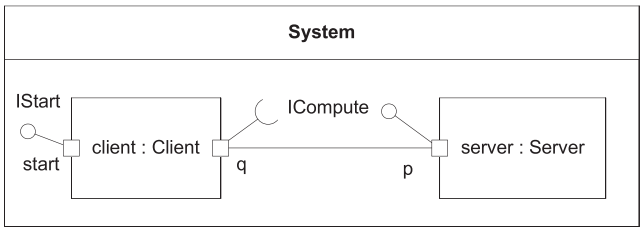

Figure 5. Simple example with different deployment options.

DOI: 10.1002/spe

back once data arrive. The need for these calls in component-based applications is reflected by the recent OMG RfP (request for proposal) AMI4CCM. Both variants are available via FCM port kinds available in an eC3M model library that compute the associated provided/required interfaces.

These two examples (passing RTF and using CORBA AMI) show the flexibility provided by the FCM port and connector mechanisms.

Deployment and footprint. To show that the modeled example allows for different deployments, we will examine three variants (and the achievable footprint). In the first variant, the components are deployed on the same node and interact via a direct local invocation. This is also the case for the second option, but the server component is protected against concurrent access by means of a declarative container rule. In the last variant, the system is distributed on two different nodes: the client is deployed on *NodeA*, the server on *NodeB*, and the interaction is implemented by means of a small connector component on top of sockets provided by the OS.

Table IV shows the footprint figures for the first deployment variant with a direct connection between client and server. There is a column for the code size on an ARM processor (a frequently used processor in the embedded world), another for a x86 processor, and a third for the data size (random access memory) of a component instance. The code has been produced by a gcc 4.4 compiler with the -Os (optimize space) flag. Because both processors have 32 bit architectures, there is no difference with respect to the size of each instance. This size is quite small: (i) the server requires a virtual function table entry ($1x4$ bytes) for implementing interface *ICompute*; (ii) the client component requires $3x4$ bytes, $1x4$ for required port `q`, and $2x4$ for the provided port `start`; and (iii) the system requires $2x4$ bytes for storing the two part references. For this deployment variant, the overheads of component mapping are in the order of a few bytes.

In the second variant, the only change is to declare the use of a container service that serializes concurrent accesses. The declaration is compatible with the MARTE stereotype protected passive unit (PpUnit; see HLAM chapter of [18]). The realization of container services in eC3M is based on the delegation to an executor, as shown in Figure 6. Table V shows the footprint of the additional components, namely the container itself (called Server_cc, the postfix is a shorthand for component container) and the interceptor PpUnit. Please note that the container itself is only added, if there is at least one container service declared.

Table VI shows the footprint of the last variant, the distributed client/server system, in which client and server are deployed on different nodes and interact via a socket (for simplicity, only the footprint of the client is shown because the server part has a similar size and structure). The adaptive

Table IV. Local deployment of client/server system.

Code ARM	Code x86	Data size	Description
2843	3840		Binary
256	346	24	BootLoader.o
340	443	4	ComponentModel/Server/Server.o
392	571	12	ComponentModel/Client/Client.o
52	110	8	ComponentModel/System/System.o

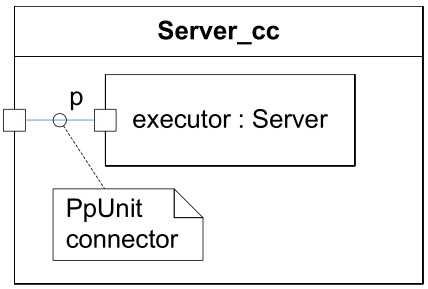

Figure 6. Container encapsulating protected passive unit server implementation.

DOI: 10.1002/spe

Table V. Client/Server system with protected passive unit container.

Code ARM	Code x86	Data size	Description
88	183	8	ComponentModel/Server/Server_cc.o
616	852	40	methodCall_ICompute/PpUnit/PpUnit.o

Table VI. Distributed deployment, client node.

Code ARM	Code x86	Data size	Description
9448	11290		Binary
620	695	276	BootLoader.o
60	108	8	ComponentModel/System/System.o
392	571	12	ComponentModel/Client/Client.o
352	487	–	SocketRuntime/ASN.o
1628	1902	20	SocketRuntime/Socket.o
1998	2008	220	SocketRuntime/SocketRuntime.o
260	359	20	methodCall_ICompute/AsyncCall/Socket.o
404	533	16	methodCall_ICompute/AsyncCall/CStub.o

part of the socket connector can be easily identified because it has been generated into a package that has the interface name as postfix (methodCall_ICompute). Because this part contains stubs performing parameter marshaling, its size depends on the interface, that is, the number of operations and parameters – in our scenario, the server provides two operations with two parameters each. In this case, the adaptive part is quite small (500 bytes) compared with the fixed part in SocketRuntime (4 K).

In this section, we have shown the overhead of the eC3M mechanisms (container, port reifications at run-time) is very small. A more important overhead is implied by the implementation of the interaction mechanisms. However, these are defined in a model library and can be tailored toward the application needs. In case of the shown example, the achieved footprint is very small because its reduction on a simple marshaling an activation mechanism. For instance, an ORB supporting heterogeneous platforms, different transports, and server activation policies (object adapter) would be much larger – too large for some system requirements. With eC3M, there is a choice to use a very simple interaction mechanism with a low footprint or – if required – a connector based on ORB implementations.

5.2.4. Flexibility dimensions. With the eC3M platform, the four flexibility dimensions identified in Section 4.3 are available. As seen in the simple client/server example, a client may specify NFPs such as, for instance, the frequency of server invocation. Client calls can be made asynchronous via AMI ports and connectors. In these cases, *port* and *connector* flexibility facilitate the use of *NFPs*.

Another variant that has not been shown in the example is data-flow-oriented communication, for example, a sensor producing data that is consumed by a controller. In this case, the consumer may either actively pull data or be notified whenever new data arrives. Each variant of flow ports in eC3M is represented by an SCM ports with the appropriate ports kind. The associated connectors buffer a configurable volume of data.

Containers may be defined to implement technical services including for instance simple trace/logging mechanisms, on demand instantiation of component instances, or distributed mechanisms such as fault detectors.

5.3. Case study 3: FCM over OASIS

The third case study concerns a medium voltage protection relay, namely the Sepam 10 product, from Schneider Electric (Paris, Rueil-Malmaison, France). The software part of this embedded system has been designed with the FCM and implemented with OASIS toolchain, which has been presented in Section 3.3.

DOI: 10.1002/spe

The safety function of the software part of Sepam 10 protection relays is first to detect any faults within the supervised power network and then ask the tripping of the circuit breakers to isolate the faulty portion of the network. The decision to ask or not the tripping of the circuit breaker is taken by protection algorithms. Note that differences between medium protection relays mainly consist in the set of protection algorithms that are embedded in the device. Typical power network faults conditions are overloads, short circuits, insulation faults, and so on. It is required that detection and isolation of faults must occur within a given time, as specified by the IEC 60255 standard and noted detection delay. Using the OASIS approach, we have define a software platform called OASISepam to develop a deterministic Sepam 10 protection relay, through a by construction fulfillment on the specified end-to-end detection delay [41].

5.3.1. Using FCM concepts to map OASIS entities. We consider only the structural aspects of an OASIS application. Neither behavioral aspects nor temporal behavior aspects are considered. The OASIS ΨC language defines the following keywords: application, agent, clock, global, and body. An application is composed of agents and clocks. An agent contains global variables (expressed using the global keyword), bodies, and communication interfaces. A body is defined as a sequence of so-called EAs (for elementary actions; see section 3.3), and the OASIS language provides instructions to switch between bodies.

We grouped OASIS entities in two packages: component and communication packages. The component package contains the definition of the application, clocks, global variables, and body elements, as well as the relations between these elements. The communication package includes the definition of communication mechanisms and their associated interfaces, as well as clocks used to specify temporal constraints on these communications. OASIS provides two communication mechanisms: temporal variables and messages. However, we focus on temporal variables only as OASISepam uses exclusively this communication mechanism. A temporal variable is an implicit, one-to-several real-time data flow. The task owner of the temporal variable updates this flow at a predetermined rhythm, specified through the OASIS ΨC programming language.

Component package. As an OASIS application consists of communicating agents, it is hierarchically the highest component of the design. An application has a name and an initial time (keyword `inittime`). The initial time is relative to the clock associated to the application. It defines the value of the time when the application is started. An Application class is defined and the stereotype FCMComponentType is applied to this class. The attributes of Application class are `inittime` and `name`. The Application class refers to a clock with the role applicationClock.

An Agent Class represents an OASIS agent. This class has three attributes: `name`, `starttime`, and `stacksize`. `starttime` defines the first activation date of the agent, and `stacksize` defines the size of the stack associated to this agent. The Agent class is also stereotyped by FCM-ComponentType because an agent is an autonomous and reusable entity in OASIS and it can communicate with other agents. The relation that an agent is contained by an application is already included by the FCM component definition through the composite relation. The Agent class owns bodies. The Body class is again stereotyped by FCMComponentType. The Body class can refer other bodies, which shows the chain of agent behavior. An Agent class refers to a StartBody class, which indicates the object Body from which an object Agent starts its behavior. The Agent owns also PsyCGlobalVariable, which contain all global variables of an agent. The Agent class refers to a class Clock with two roles: as startClock and as agentClock. Figure 7 shows the Agent entity within the FCM-OASIS metamodel library.

Communication package. An agent is a communicating entity in OASIS. We focus on communication mechanisms based on temporal variables. For temporal variables, the interaction points of the communicating agents are directly used inside the bodies of the communicating agents. Consequently, the ports representing these interaction points are attached to the communicating agents and to the bodies of these agents.

As the Agent and the Body classes are stereotyped by FCMComponentType, they may have ports. Each port corresponds to an interaction point. A FCMPort stereotype is applied to all ports.

DOI: 10.1002/spe

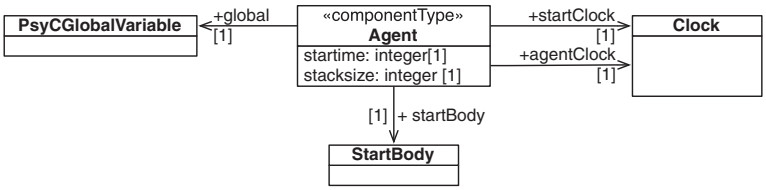

Figure 7. OASIS agent entity stereotyped by FCMComponentType within the OASIS metamodel described using Flex-eWare component model.

The FCMPort stereotype refers to a PortKind; consequently, each port refers also to this PortKind. PortKind defines specific rules to precisely map a given behavior. Each communication mechanism of OASIS is expressed through a specified PortKind. In the remainder of this section, we present two different PortKinds involved for temporal variables: TemporalVariable and ConsultInterface port kinds.

TemporalVariable port kind. The TemporalVariable class is defined to represent OASIS communication behavior on the owner side of the temporal variable. It refers to a clock to specify the rhythm of the temporal variable (i.e., the sampling rate of the data flow). In addition, the TemporalVariable class has a pastValue attribute, which defines the number of values the owner wishes to read from the data flow.

ConsultInterface port kind. The ConsultInterface class is defined to answer the need of representing the behavior of the reader of a temporal variable. It does not refer to a clock as the rhythm of the temporal variable is defined on the owner side. Similar to the TemporalVariable PortKind, the ConsultInterfance has a pastValue attribute.

5.3.2. Overview of the process and associated tools. Figure 8 shows the big picture of the OASIS-FCM development process. On the left side of the figure, elements developed by the designer of the OASIS-FCM library are shown: the OASIS-FCM metamodel library and the code generator tool. On the right site of the figure, how an OASIS application developer can use the OASIS-FCM metamodel to build applications and generate associated ΨC code is shown. Acceleo is used to generate the ΨC code. Then from the model written in the UML format and by using the code generator, a ΨC code corresponding to described application is generated. The code generator uses a template of a ΨC code. The template contains a set of scripts. Each script can visit and evaluate a structural element of the model, such as class, association, connector, port, and instance, to produce the corresponding ΨC code of the described application. Note that the temporal behavior of agents must be specified and included in the generated ΨC code by the application designer without relying on FCM concepts. Finally, the classical OASIS tool chain can be used to build applications.

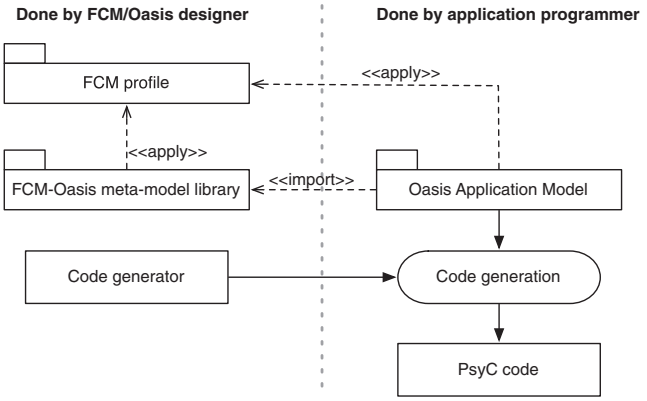

Figure 8. OASIS-FCM development process.

DOI: 10.1002/spe

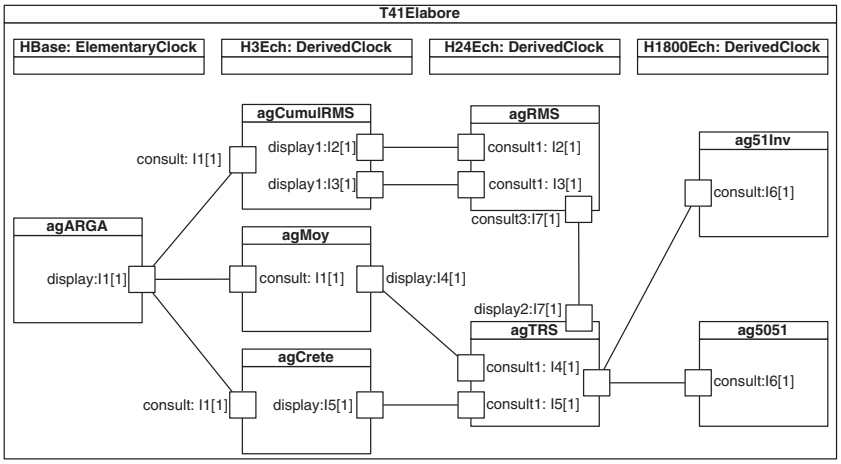

Figure 9. Medium voltage protection relay application modelized using the OASIS-FCM metamodel.

5.3.3. Experimentation using OASIS-FCM metamodel. Figure 9 shows the model we designed of the Sepam 10 medium voltage protection relay using the OASIS-FCM metamodel. The Sepam 10 is made of three stages, namely the acquisition, the measurement, and the protection stages. The acquisition stage produces new voltage data based on information collected by sensors. The measurement stage applies various signal processing algorithms. Results of this measurement stage are used by the protection stage to ask or not the tripping of the circuit breakers. The acquisition stage is made of one task, whereas other stages are made of several tasks. The agARGA component defines the acquisition stage. Components agMoy, agCumulRMS, agRMS, agCrete, and agTRS define the measurement stage. Finally, components ag5051 and ag51Inv define the protection stage. On the basis of this model, the application designer can generate the corresponding OASIS initial code that must be fulfilled with temporal constraints and functional code. This application was successfully executed on an STR710 board (ARM7-based processor) running the OASIS kernel.

One of the advantages of using FCM is the use of graphical interfaces to design OASIS-based applications. The reusability of agents at the binary level, which is possible in OASIS, can therefore be facilitated. Consequently, the design of various Sepam-based products, in which protection algorithms encapsulated in components are removed or added, is made easier.

5.3.4. Flexibility dimensions. The OASIS case study puts into practice one of the flexibility dimensions identified in Section 4.3 and its associated model library: *port*. Communicating through OASIS temporal variables is indeed represented by a special kind of FCM ports that define appropriate rules to precisely map the specific behavior of this communication mechanism. A temporal variable is a real-time data flow associated with an internal variable of a agent. An agent that wishes to access a temporal variable must specify the number (i.e., depth) of a value it needs to consult from the flow. Therefore, both TemporalVariable and ConsultInterface classes have a mandatory pastValue attribute to express this behavior. The illustration of this flexibility dimension of the FCM for expressing temporal variables mechanisms can be generalized to other OASIS communication mechanisms.

5.4. Synthesis

As a matter of synthesis on these case studies, two main points are worth noticing.

First, these case studies demonstrate that a high-level, MDE-based approach for designing component-based systems does not conflict with stringent requirements in terms of resources (memory, CPU, etc.) such as in the case of the WSN experiment in Section 5.1, the experiment on ARM processors in Section 5.2, or time constraints such as in the case of the voltage protection relay in Section 5.3.

DOI: 10.1002/spe

Second, the flexibility dimensions that have been introduced in the FCM in terms of ports, connectors, containers, and NFPs are adequate for the supporting the requirements of diverse application use cases such as the one presented in this section. Even if all these flexibility dimensions are not available for each target platform, and in this case cannot be exploited, we believe that this is still valuable to include them in the model to capitalize on some common know-how for embedded systems.

6. CONCLUSION

This article has presented an MDE approach for designing and implementing embedded systems. Models are widely recognized as an efficient way for capturing high-level requirements and architecture design choices. Associated with techniques based on model transformation and code generation, they provide an efficient approach for reasoning about complex embedded system architectures, abstracting the implementation details, and easing the porting between different versions of host target platforms.

The work presented in this article is the result of a collaborative project between academic and industrial partners sharing expertise in telecommunication and automotive industries. The aim is to foster the adoption of MDE solutions for designing and implementing embedded systems. The work presented here is organized around three main activities: requirement elicitation, metamodeling, and run-time solutions.

Section 2 has identified a set of requirements coming from the telecommunications and automotive industries. These requirements have been organized in terms of design, development, deployment, and execution, which are the four main phases of the software development life cycle. They put forward the necessity to incorporate flexibility points as soon as possible in the software development life cycle of embedded systems. This is a key characteristic to obtain systems that are agile and flexible enough to accommodate change and evolution. This requirement elicitation phase has been complemented by a study of some state-of-the-art middleware and component-based solutions for implementing embedded systems (see Section 3).

On the basis of these inputs, we have proposed in Section 4 the FCM model for designing embedded systems. The two main characteristics of FCM is to be component-based and to introduce flexibility points implemented as model libraries that extend the FCM. This model has been put into practice with three case studies that are reported in Section 5: WSNs, distributed client/server applications, and control systems for electrical devices. In all three cases, the execution platforms used to operate the applications were different: the Fractal component-based platform, the eC3M CCM-based middleware platform, and the OASIS toolchain for safety-critical real-time systems, respectively.

This study has shown the adequacy and the maturity of MDE solutions for designing and implementing industrial strength case studies. Models are appropriate solutions for capturing the variability and the flexibility needed by modern embedded systems.

In future work, we plan to push the use of models a step further by using them at run-time. The main expected benefit will be to better support co-evolution of code and models and to be able to reflect seamlessly changes that are applied on the applications either at run time or at design-time. This objective raises several difficult challenges such as providing an efficient solution for encoding efficiently models for resource-constrained embedded systems and reconciling divergent changes that are applied concurrently at design-time and run-time versions of the models. Yet, we believe that this objective will provide a major step toward providing more agility in the design and implementation of embedded systems. Several other objectives can be mentioned for future works.

First, we plan to enable interoperability between different platforms, thanks to a common FCM-based design and some gateways (e.g., connectors) to be developed. Second, the FCM can be a common base for reuse and sharing technological assets between different platforms. For example, a connector for a given communication protocol can be specified in terms of FCM, mapped, and reused across different target platforms. Third, current extension models in the FCM are purely additive. It may happen that conflicts arise when applying several extensions. At some point, this is a concern that is shared by other studies. For example, nonconflicting aspects are a domain of

DOI: 10.1002/spe

research per se in the aspect-oriented software development community [14]. In future work, we plan to address this concern at the extension model level.

ACKNOWLEDGEMENTS

The work presented in this article has been partially funded by Agence Nationale de la Recherche (ANR) under grant ANR-06-TLOG-017 Flex-eWare. We thank the reviewers for their comments which helped in improving the quality of the article.

REFERENCES

1. Flex-eWare Consortium. The Flex-eWare project home page, 2010. http://www.flex-eware.org.
2. KNX. The KNX standard for home and building control, Jan 2010. http://www.knx.org.
3. Zigbee Alliance. The Zigbee alliance, Jan 2010. http://www.zigbee.org.
4. Broadband Forum. Tr-069: CPE WAN management protocol v1.1, 2007. http://www.broadband-forum.org.
5. OSGi Alliance. Osgi service platform core specification release 4, Aug 2005. http://www.osgi.org.
6. UPnP Forum. UPnP device architecture version 1.1, Oct 2008. http://www.upnp.org.
7. Papazoglou MP. Service-oriented computing: concepts, characteristics and directions. In *Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE'03)*. IEEE Computer Society: Los Alamitos, 2003; 3–12.
8. Grimm K. Software technology in an automotive company: major challenges. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*. IEEE Computer Society: Washington, DC, USA, 2003; 498–503.
9. Autosar. AUTOSAR: automotive open system architecture, Jan 2010. http://www.autosar.org.
10. Royce WW. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th International Conference on Software Engineering*. IEEE Computer Society Press: Los Alamitos, CA, USA, 1987; 328–338.
11. Abrial J-R. *The B Book – Assigning Programs to Meanings*. Cambridge University Press: Cambridge, New York, 1996.
12. Medvidovic N, Taylor R. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering* Jan 2000; **26**(1):70–93.
13. Allen R, Garlan D. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*. IEEE Computer Society Press: Los Alamitos, CA, 1994; 71–80.
14. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J-M, Irwin J. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-oriented Programming (ECOOP'97)*, Vol. 1241, Lecture Notes in Computer Science. Springer: Berlin, 1997; 220–242.
15. SAE. AADL Standard, V2. *Technical Report*, Society of Automotive Engineers. approved in Nov. 2008.
16. OMG. Deployment and configuration of component based distributed applications, v4.0, 2006. OMG document formal/2006-04-02.
17. OMG. CORBA component model specification, Version 4.0, 2006. OMG Document formal/2006-04-01.
18. OMG. A UML profile for MARTE: modeling and analysis of real-time embedded systems, Beta 2, 2008. OMG document ptc/2008-06-09.
19. Bruneton E, Coupaye T, Leclercq M, Quéma V, Stefani J-B. The fractal component model and its support in Java. *Software Practice and Experience (SPE)* 2006; **36**(11-12):1257–1284.
20. Fassino J-P, Stefani J-B, Lawall J, Muller G. Think: a software framework for component-based operating system kernels. In *Proceedings of the USENIX Annual Technical Conference*. USENIX: Berkeley, CA, 2002; 73–86.
21. Anne M, He R, Jarboui T, Lacoste M, Lobry O, Lorant G, Louvel M, Navas J, Olive V, Polakovic J, *et al*. Think: view-based support of non-functional properties in embedded systems. In *2nd International Conference on Embedded Software and Systems*. IEEE Computer Society: Los Alamitos, CA, USA, 2009; 147–156.
22. Leclercq M, Ozcan A, Quéma V, Stefani JB. Supporting heterogeneous architecture descriptions in an extensible toolset. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. ACM Press: New York, NY, 2007; 209–219.
23. Lau KK, Wang Z. Software Component Models. *IEEE Transactions on Software Engineering* Oct 2007; **33**(10): 709–724.
24. David V, Delcoigne J, Leret E, Ourghanlian A, Hilsenkopf P, Paris P. Safety properties ensured by the OASIS model for safety critical real time systems. In *Proceedings of the 17th International Conference on Computer Safety, Reliability and Security (SAFECOMP'98)*, Vol. 1516, Lecture Notes in Computer Science. Springer: Heidelberg, Germany, 1998; 45–59.
25. David V, Aussaguès C, Louise S, Hilsenkopf P, Ortolo B, Hessler C. The OASIS based qualified display system. *4th American Nuclear Society Int. Topical Meeting on Nuclear Plant Instrumentation, Controls and Human-Machine Interface Technologies (NPIC & HMIT)*, Columbus, Ohio, USA, 2004.

DOI: 10.1002/spe

26. Chabrol D, David V, Aussaguès C, Louise S, Daumas F. Deterministic distributed safety-critical real-time systems within the OASIS approach. *Proceedings of the Internationl Conference on Parallel and Distributed Computing Systems (PDCS 2005)*, Phoenix, AZ, USA, 2005; 260–268.
27. Aussaguès C, Ohayon E, Brifault K, Dinh Q. Using multi-core architectures to execute high performance-oriented real-time applications. *International Conference on Parallel Computing (PARCO 2009)*, 2009.
28. As-2 Embedded Computing Systems Committee SAE. Architecture analysis & design language (AADL), Nov 2004. SAE Standards AS5506.
29. Escoffier C, Hall R. Dynamically adaptable applications with iPOJO service components. In *Proceedings of the 6th International Symposium on Software Composition (SC'07)*, Vol. 4829, Lecture Notes in Computer Science. Springer: Berlin, Germany, 2007; 113–128.
30. Richardson T, Wellings AJ, Dianes JA, Díaz M. Providing temporal isolation in the OSGi framework. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded systems*, JTRES '09. ACM: New York, NY, USA, 2009; 1–10.
31. Richardson T, Wellings A. An admission control protocol for real-time OSGi. In *Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '10. IEEE Computer Society: Washington, DC, USA, 2010; 217–224.
32. Kung A, Hunt JJ, Gauthier L, Richard-Foy M. Issues in building an ANRTS platform. In *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '06. ACM: New York, NY, USA, 2006; 144–151.
33. Lu T, Turkay E, Gokhale A, Schmidt DC. CoSMIC: an MDA tool suite for application deployment and configuration. *Proceedings of the OOPSLA 2003 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Anaheim, CA, 2003.
34. OSEK/VDX Consortium. OSEK operating system, version 2.2.3, 2005. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf.
35. Lobry O, Navas J, Babau JP. Optimizing component-based embedded software. *Computer Software and Applications Conference, Annual International* 2009; **2**:491–496.
36. Polakovic J, Mazare S, Stefani JB, David PC. Experience with implementing safe reconfigurations in component-based embedded systems. In *10th International ACM Symposium on Component-Based Software Engineering (CBSE'07)*. Springer: Berlin, Germany, 2007; 242–257.
37. Loiret F, Navas J, Babau JP, Lobry O. Component-based real-time operating system for embedded applications. In *Proceedings of the 12th International Sigsoft Symposium on Component-Based Software Engineering (CBSE'09)*, Vol. 5582, Lecture Notes in Computer Science. Springer: East Stroudsburg, Pennsylvania, USA, 2009; 209–226.
38. Navas JF, Babau JP, Lobry O. Minimal yet effective reconfiguration infrastructures in component-based embedded systems. In *Proceedings of the 2009 ESEC/FSE workshop on Software Integration and Evolution @ Runtime (SINTER'09)*. ACM: New York, NY, USA, 2009; 41–48.
39. ANR Flex-eWare Project. Flex-eWare component model mappings, 2009. https://srcdev.lip6.fr/trac/research/flex-eware/wiki/FCM.
40. oBIX. oBIX open building information exchange, Feb 2010. http://www.obix.org.
41. Jan M, Lalande J, Pitel M, David V. Usage of the safety-oriented real-time OASIS approach to build deterministic protection relays. In *Symposium on Industrial Embedded Systems (SIES 2010)*. IEEE Computer Society: Washington, DC, 2010; 128–135.

DOI: 10.1002/spe