A dynamic component model for cyber physical systems — Source link

A Dynamic Component Model for Cyber Physical Systems

François Fouquet, Olivier Barais, Noël Plouzeau, Jean-Marc Jézéquel, Brice Morin, Franck Fleurey

A Dynamic Component Model for Cyber Physical Systems

Francois Fouquet, Olivier Barais,
Noel Plouzeau, Jean-Marc Jezequel
IRISA, University of Rennes1, France
firstname.name@irisa.fr

Brice Morin Franck Fleurey
SINTEF ICT, Oslo, Norway
firstname.name@sintef.no

ABSTRACT

Cyber Physical Systems (CPS) offer new ways for people to interact with computing systems: every thing now integrates computing power that can be leveraged to provide safety, assistance, guidance or simply comfort to users. CPS are long living and pervasive systems that intensively rely on microcontrollers and low power CPUs, integrated into buildings (e.g. automation to improve comfort and energy optimization) or cars (e.g. advanced safety features involving car-to-car communication to avoid collisions). CPS operate in volatile environments where nodes should cooperate in opportunistic ways and dynamically adapt to their context. This paper presents μ-Kevoree, the projection of Kevoree (a component model based on models@runtime) to microcontrollers. μ-Kevoree pushes dynamicity and elasticity concerns directly into resource-constrained devices. Its evaluation regarding key criteria in the embedded domain (memory usage, reliability and performance) shows that, despite a contained overhead, μ-Kevoree provides the advantages of a dynamically reconfigurable component-based model (safe, fine-grained, and efficient reconfiguration) compared to traditional techniques for dynamic firmware upgrades.

Categories and Subject Descriptors

D.2 [**Software Engineering**]; D.2.8 [**Software Engineering**]: Software Architectures—*Domain-specific architectures; Languages*

Keywords

Component-based software engineering ; Autonomic computing ; Embedded software ; Software architecture

1. INTRODUCTION

Over the last decades, the Internet has undergone dramatic changes, moving from a rather static Internet of Content, to an always more complex, dynamic and ubiquitous mix of Internet of People, Services (IoS), and Things (IoT) [1]. Based on this infrastructure, which ranges from large datacenters and cloud servers to an heterogeneous and ever growing set of things (smartphones, sensors, etc) operated by resource-constrained CPUs and microcontrollers, Cyber Physical Systems (CPS) have emerged. CPS are long living and pervasive systems that rely intensively on microcontrollers and low power CPUs, integrated into buildings and cities (automation to improve comfort, safety and energy optimization), cars (advanced safety features involving car-to-car communication to avoid collisions), and so on.

CPS operate in volatile environments where nodes should cooperate in opportunistic ways and dynamically adapt to their context. In a car-2-car scenario, 2 cars (or more) approaching the same intersection should be able to synchronize in a reasonably short delay to share information about their own context and configuration, then take distributed decisions *e.g.* on the precedence order to cross the intersection. In a building automation scenario, users working or living in the building should be able to customize their working or living environment according to their desires and needs, *e.g.* using their smartphones or tablets to adjust the intensity of the lights according to the ambient light, etc. In a factory chain scenario, robots operating on the chain should be able to adapt and cope with failures, instead of shutting down all the chain when a failure is detected. Depending on the context, it is necessary to dynamically adapt both the software and the way the CPS are configured, as things are containers that can host services.

Dynamic adaptation, pursuing IBM's vision of autonomic computing, is a very active area since the late 1990's - early 2000's [20]. However, many existing techniques concentrate on the adaptation of rather powerful nodes, which are typically able to run a Java Virtual Machine. Adaptation of resource-constrained devices such as microcontrollers has received less attention. These resource constraints prevent the use of standard operating systems, middlewares and frameworks, making the design of adaptive software for microcontroller a challenging task. In practice, microcontroller code is most of the time developed by using low-level programming languages and by following ad-hoc manual trial and error processes; these processes includes extensive testing of the resulting software in its target environment. While this might be acceptable to build static, dedicated applications, this is not a practical solution for CPS, because CPS operate in an open and dynamic environment, where the target environment cannot be foreseen at design-time.

Kevoree leverages and extends state-of-the-art approaches to scale CBSE principles horizontally (distribution between a large set of nodes) and vertically (from cloud-based nodes to microcontrollers). This paper focuses on μ-Kevoree, a mapping of Kevoree concepts for microcontrollers. μ-Kevoree pushes dynamicity and elasticity concerns directly into resource-constrained devices. In particular, this paper details the challenges of mapping such a large component model onto microcontroller-based architectures. We explain the tradeoffs that were used to obtain a useful solution coping with stringent resource constraints. This dynamic component model for resource-constrained systems has been thoroughly benchmarked against key criteria that are specific to the embedded software domain (memory usage, reliability and performance). Our model has also been applied to a real-life case study. The evaluation of μ-Kevoree for these key criteria show that, despite a contained overhead, μ-Kevoree provides a dynamically reconfigurable component-based model (safe, fine-grained, and efficient reconfiguration) with a limited overhead with respect to static approaches.

This paper is organized as follows. Section 2 presents the Kevoree component model and Section 3 details the challenges of mapping dynamic component model concepts to resource-constrained microcontrollers. Section 4 then explains how we ported Kevoree to these challenging platforms, and details the necessary tradeoffs. This new version of Kevoree is validated in Section 5 through a set of atomic benchmarks. Section 6 discusses the result and presents related work and Section 7 concludes and draw some perspectives to be addressed in future work.

2. BACKGROUND

2.1 Kevoree at a glance

Kevoree [1] is an open-source dynamic component model, which relies on models at runtime [6] to properly support the dynamic adaptation of distributed systems. Models@runtime basically pushes the idea of reflection [23] one step further by considering the reflection layer as a real model that can be uncoupled from the running architecture (e.g. for reasoning, validation, and simulation purposes) and later automatically resynchronized with its running instance.

Kevoree has been influenced by previous work that we carried out in the DiVA project [23]. With Kevoree we push our vision of models@runtime [22] farther. In particular, Kevoree provides a proper support for distributed models@runtime. To this aim we introduce the *Node* concept to model the infrastructure topology and the *Group* concept to model semantics of inter node communication during synchronization of the reflection model among nodes. Kevoree includes a *Channel* concept to allow for multiple communication semantics between remote*Components* deployed on heterogeneous nodes. All Kevoree concepts (Component, Channel, Node, Group) obey the object type design pattern [18] to separate deployment artifacts from running artifacts. Kevoree supports multiple kinds of execution node technology (e.g. Java, Android, MiniCloud, FreeBSD, Arduino, . . . [1]).

[1]http://www.kevoree.org

2.2 Dynamic Adaptation with Kevoree

Kevoree aims at providing advanced adaptation capabilities to different types of nodes:

- **Level 1: Parametric adaptation.** Dynamic update of parameter values, e.g. change of sampling rate in a component that wraps a physical sensor (adaptation of instance properties).
- **Level 2: Architectural adaptation.** Dynamic addition or removal of bindings or components, e.g. replication of software components and channels on different nodes to perform load balancing (adaptation of instances graph).
- **Level 3: Dynamic provisioning of types.** Hot deployment of component types that were not foreseen before the initial deployment of the system. This allows for system evolution by enabling parametric and architectural reconfigurations, including management of instances for types that are added and managed dynamically (adaptation of types).
- **Level 4: Adaptation for remote management.** Nodes supporting level 4 adaptation participate in a remote management layer, which supervises less powerful nodes. This layer monitors remote nodes by requesting their current Kevoree model; the layer triggers dynamic adaptation of nodes by sending precomputed reconfiguration scripts to them. This remote adaptation process supports seamless management of less powerful nodes by a more powerful one, which has enough resources to build and evaluate new and appropriate configurations.

The adaptation engine relies on a model comparison between two Kevoree models to compute a script for a safe system reconfiguration; execution of this script brings the system from its current configuration to the new selected configuration [23]. Model comparison yields a delta-model defining changes (using CRUD operations) that should be applied on the source model to obtain the target model. Planification algorithms [4] use this delta-model as input in order to defined an efficient schedule of the adaptation steps. The delta-model is finally compiled into a Kevoree script. The Kevoree Script language (KevScript for short) is a core language for describing reconfiguration. KevScript is comparable to FScript for Fractal Component Model [11]. Execution of a KevScript directly adapts a Kevoree system, without the need for a full Kevoree model definition. Such adaptation scripts are written by designers, or they can be generated by automated processes (*e.g.* within a control loop managing the Kevoree system).

3. MAPPING KEVOREE ADAPTATION CONCEPTS ON MICROCONTROLLERS

3.1 Challenges

Dynamic adaptation is a key concept to build advanced CPS able to adapt to their context and to user needs. Models at runtime is an efficient approach to manage the complexity of dynamic adaptation [23] by providing control and abstraction over reflection mechanisms. Applying reflection techniques is rather straightforward on fully grown component or service models such as OSGi, or directly on top of modern object-oriented languages such as Java, as long as the execution hardware is powerful enough to run a virtual machine. The embedded software sensing and acting on the physical world (via hardware components) should be able to adapt to the needs of different (and potentially concurrent) services running on powerful nodes (in the cloud, on tablets, etc). Some services will for example subscribe to temperature alerts (the sensors being responsible to notify the service when a threshold has been reached), and then reconfigure the sensors to send almost continuous data, so that the service can precisely monitor the evolution of the temperature.

Applying models at runtime (or any dynamic adaptation technique) on execution nodes with scarce resources (e.g. microcontroller-based computation nodes) is much more difficult, for the following reasons:

1. **Downtime**: Microcontrollers often host the software that controls physical devices directly. Rebooting or freezing these microcontrollers may have severe consequences if the microcontrollers control safety critical devices, or unpleasant and noticeable effects if they control comfort devices.
2. **Volatile memory usage** (RAM): Dynamic memory allocation is the cornerstone that enables dynamic adaptation. Microcontrollers usually embed only of few kB of RAM, and this size limitations prohibits storing multiple configurations in memory at the same time.
3. **Persistent memory usage**: Persistent memory is required to ensure that the adaptation process has transaction-like properties, allowing recovery of microcontroller's state in case of a reboot after failure. EEPROM is a common type of persistent memory embedded into microcontrollers, usually with a very limited size. This type of memory also has a limited lifetime in term of numbers of writing operations. Similar to Solid State Disk [2], writes to EEPROM should be distributed among memory cells to optimize the lifetime of the overall memory.
4. **Recovery**: The ability to recover is critical for embedded systems, which are subject to failures (e.g. a temporary loss of power). Microcontrollers should reboot and restore their last configuration quickly enough to keep pace with configuration evolutions of the overall architecture.

CPS relying on a large set of autonomous sensors have cost and energy constraints that calls for cheap and power-efficient platforms able to run for long period of times with minimum on-site maintenance (*e.g.*, battery replacement). This is particularly true in the environmental monitoring domain: off-shore oil spills monitoring, flood prediction [22], air quality monitoring or radiation monitoring, for the following reasons[2]:

1. Their simplified architecture is robust and predictable: microcontrollers can operate by a wide range of temperature (typically -40 to 85 degrees Celcius), humidity, power supply, and have fixed number of cycles to execute a given operation.
2. Their energy needs (and generated heat) are very low: An 8-bit microcontroller running at 32kHz typically consumes less than 0,05W (less than 0,5W at 1MHz) excluding the need for any radiator. They can thus run for very long time on battery.
3. Their simplified architecture allows for mass production, making microcontrollers very cheap to deploy even in large numbers.

[2]See for example http://www.atmel.com/Images/doc2545.pdf for detailed facts about microcontrollers

Compared to full-fledged computation nodes, cheap microcontrollers suffer from an adaptation overhead that stems from their hardware technology in terms of adaptation time or memory wear: dynamic provisioning of component types requires writing a program in flash memory. Therefore, implementing Kevoree concepts for microcontrollers nodes relies on a precise trade-off between flexibility and typical exploitation costs. Finding a lightweight solution for each of reconfiguration level described above is one of the main challenges of μ-kevoree.

3.2 Case study

We will use a smart building case study to validate Kevoree on a set of heterogeneous nodes, including of course some microcontrollers. Different systems (relying on proprietary devices and protocols) are usually deployed in buildings to manage different aspects of the building automation, in particular comfort (lighting, air conditioning, etc), safety and security (smoke and fire detection, sprinklers, etc). The degree of flexibility offered by a building automation system is often very poor.

- These systems rely on fixed topology of communication channels. Sensors and actuators often need to be physically coupled, hindering any future reconfiguration or evolution of the system. For example, a motion sensor will trigger all the lights of the corridor.

- The architecture is organized around a central server. When the devices are not physically coupled, they usually communicate through a central server, to execute the event-driven rules that orchestrate the system. Even though updating these rules is possible, to adapt the behavior of the system, this requires access to the central server.

The goal of Kevoree is to seamlessly distribute both the business logic and the dynamic adaptation capabilities on heterogeneous nodes ranging from powerful servers, to tablets, and to simple devices operated by microcontrollers. On a day-to-day basis, this would allow users to configure and reconfigure their offices on-the-fly from a smartphone, (e.g. to define a lighting environment according to the ambient luminosity, temperature, etc), while some other concerns would be managed by a central server (e.g. to turn the cameras on at night). In a crisis situation, this kind of seamless distribution would allow emergency services to cope with the failure of some nodes. Firemen could still access the data provided by low-level sensors, and compute meaningful context information on a tactical decision system despite the loss of a nodes.

4. DYNAMIC ADAPTATION FOR MICROCONTROLLERS

This section describes how Kevoree concepts are mapped to Arduino nodes. Arduino[3] is an open-source hardware and

[3]http://www.arduino.cc

software electronics prototyping platform based on an 8-bits AVR microcontroller. Arduino boards can be connected to a set of sensors and actuators and programmed in languages from the C/C++ family. While we have chosen Arduino to implement our μ-Kevoree approach, it can be applied easily to other microcontroller families (PIC, ARM, etc).

Firmware implementations are often coded manually in C using a trial and error process, with an intensive manual and automated test-based validation. More advanced techniques (such as the MDE techniques proposed by ThingML[4] [15]) aim at generating static source code for microcontrollers. Such techniques can easily be leveraged to generate the internal code of component, which is not the scope of Kevoree. Microcontroller firmware puts a strong emphasis on resource usage (such as memory, CPU and energy needs) and reliability: microcontrollers can run for long periods of time and recover in case of power or connectivity loss. We acknowledge that these properties are critical, and the benefits provided by dynamic adaptation capabilities should not jeopardize them. Our work aims at breaking the static nature of code generation while preserving all benefits of low level code design.

To this aim our approach clearly separates structure and behavior: component type behaviors are currently implemented manually in C or in the Wiring language, using state of the art practice. One core contribution of our approach is the definition of a proper abstraction system to automate management of the adaptation logic of microcontrollers, by making their business logic a separate concept. Moreover, having a clear component structure with well defined inputs and outputs also eases testing at a more abstract level.

4.1 μ-Kevoree

This sub-section describes how the main concepts of Kevoree have been ported onto microcontrollers. μ-Kevoree is totally aligned and compatible with the exiting Java and Android versions.

Types. In Kevoree component types and channel types encapsulate business logic; they are generated as C structures. *Provided* ports of component types are mapped to methods, so that client components (which require ports of the same type) can invoke these methods and eventually push data. Kevoree properties that can be dynamically updated are simply mapped onto local variables contained by this structure. A local scheduler prevents concurrent calls on these variables. *Required* ports are generated as local structures, which can optionally refer to a bound channel instance. Similarly, channel types are generated as plain C structure. Outgoing channel bindings are generated as an internal array structure, enabling dynamic allocation and storage of external provided ports references.

Asynchronous message passing. As in Kevoree's implementations for Java and Android nodes, μ-Kevoree maps each port and channel onto an actor. More precisely, a FIFO queue is generated in front of each protected method. A dispatcher (local to each component) is then in charge of dispatching messages pushed on these queues to the correct method. This local scheduler is driven by a global scheduler described below.

Instance scheduler On each node a global instance scheduler is responsible for keeping its node in a consistent state by applying the following balance strategy:

[4]www.ThingML.net

- Periodic execution: the global scheduler periodically invokes the local scheduler of each component instance that has declared a periodic execution;
- Triggered execution: the global scheduler invokes the local scheduler of each component that has a non empty message queue.

The global scheduler also periodically checks for external messages related to dynamic adaptation, as described in the next two sub-sections.

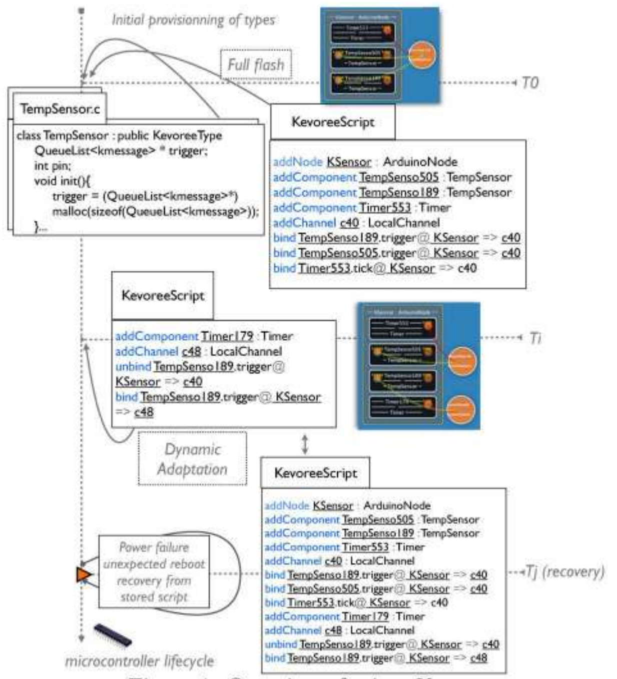
4.2 Firmware flash to handle major evolutions

Flashing a microcontroller's firmware and then rebooting the device is an easy way to implement adaptation of a microcontroller node, by replacing the implementation entirely. This adaptation technique is acceptable in some specific and controlled contexts (initial production, on-site maintenance, etc), since the device is physically connected to a more powerful node using a communication link with broad bandwidth (e.g. wired link). In this case, flashing a controller's memory is rather safe (provided that the code of the firmware is safe) and also reasonably fast: flashing the entire memory by uploading the new firmware and then rebooting the device takes a few seconds only. However, this technique is problematic when the devices are deployed remotely. Flashing the firmware over-the-air is a hazardous manipulation: firmwares are typically bulk data (compared to other data usually transmitted on wireless links) and communication errors are more likely to occur; this requires advanced protocols to cope with error handling. In practice, this approach impacts significantly the time needed to install a new firmware.

Our approach limits flashing the full firmware to cases where new component types need to be deployed. In this regard C-based microcontrollers do not provide the same flexibility than Java/OSGi nodes with respect to dynamic provisioning and class loading. This is typically required for the initial deployment of the system where all the planned component types are provisioned, or for major evolutions of the system (e.g. to handle a new type of device not foreseen before the initial deployment). In all other cases such as reconfigurations of component instances for instance, our approach performs a partial flash memory update.

4.3 Seamless dynamic adaptation of microcontrollers

Following the principles of *model@runtime*, our dynamic adaptation process is fully automated, saving designers from writing low-level adaptation scripts or from entangling adaptation logic with business logic. Prior to any adaptation, all necessary checks on the new configuration are performed on the target model. Since microcontroller nodes have limited computational power, configuration checks are performed on more powerful, Java or Android based nodes. These checks aim at detecting a mismatch between the planned configuration and the physical hardware possibilities. After this validation step, the configuration is used as input for a generator algorithm, which computes a reconfiguration script. This script is then transmitted in a compact form to dependent microcontrollers. As communication errors are frequent in wireless sensors networks, we avoid problematic microcontroller states inconsistencies by implementing a roll-back based recovery mechanism.


Initial provisionning of types
Full flash
T0
TempSensor.c
KevoreeScript
KevoreeScript
Ti
Dynamic Adaptation
KevoreeScript
Power failure unexpected reboot recovery from stored script
Tj (recovery)
microcontroller lifecycle

```
class TempSensor : public KevoreeType
    QueueList<kmessage> * trigger;
    int pin;
    void init(){
        trigger = (QueueList<kmessage>*)
        malloc(sizeof(QueueList<kmessage>));
    }...
```

```
addNode KSensor : ArduinoNode
addComponent TempSenso505 : TempSensor
addComponent TempSenso189 : TempSensor
addComponent Timer553 : Timer
addChannel c40 : LocalChannel
bind TempSenso189.trigger@ KSensor => c40
bind TempSenso505.trigger@ KSensor => c40
bind Timer553.tick@ KSensor => c40
```

```
addComponent Timer179 : Timer
addChannel c48 : LocalChannel
unbind TempSenso189.trigger@
KSensor => c40
bind TempSenso189.trigger@ KSensor
=> c48
```

```
addNode KSensor : ArduinoNode
addComponent TempSenso505 : TempSensor
addComponent TempSenso189 : TempSensor
addComponent Timer553 : Timer
addChannel c40 : LocalChannel
bind TempSenso189.trigger@ KSensor => c40
bind TempSenso505.trigger@ KSensor => c40
bind Timer553.tick@ KSensor => c40
addComponent Timer179 : Timer
addChannel c48 : LocalChannel
unbind TempSenso189.trigger@ KSensor => c40
bind TempSenso189.trigger@ KSensor => c48
```

Figure 1: Overview of micro-Kevoree

The following table lists the dynamic adaptation levels supported by the different Kevoree node technologies; the levels are defined at the beginning of Section 2. The most powerful nodes are able to run Java programs and implement all levels of adaptation features. The more constrained are the nodes, the more tradeoffs have to be addressed.

Dynamic adaptation	JavaSE	Android	Arduino
Parametric	+	+	+
Architectural	+	+	+
Dynamic provision.	+	+	+/- firm. flash
Peers management	+	+/- perf. issues	- see Future work

While severely limited resource-wise, Arduino nodes are still able to support almost all dynamic adaptation features. In most cases, complex decisions will be taken by software running on more powerful nodes. Taking adaptation decisions require processing adaptation rules, optimize goals, etc, as microcontrollers usually do not have enough computational power.

Dynamic instantiation framework.

Reflection is a fundamental principle to achieve dynamic adaptation. But the C language has no support of the primitives required to build a full-fledged reflection model; this prevents a straightforward application of the model@runtime technique on microcontroller nodes. We have removed this limitation by generating code to emulate a reflection layer on these microcontrollers. We assume that the number of types is finite when generating the framework. Adding or removing a type then implies regeneration of the whole framework as described in Section 3. With this assumption of a closed type world, we generate a flat reflection layer by using exhaustive pattern matching on instances. We generate methods that take instances as parameter; these methods provide the following basic services:

Algorithm 1 μ-Kevoree core service

```
Function  interruptScheduler(),resumeScheduler()
Function  setProperty(instanceID,propID,propValue) : Bool
Function  addBinding(channelID,componentID,portID) : Bool
Function  removeBinding(channelID,componentID,portID) : Bool
Function  createInstance(instanceID,typeID) : Bool
Function  destroyInstance(instanceID) : Bool
Function  exportCurrentState() : KevScript
```

We rely on script to export the state in order to optimize memory consumption. More precisely, every textual representation used to export the reflection model and related to type definitions (*e.g.*, port names) is locally encoded in static flash memory to save dynamic memory.

KevScript embedded interpretation.

An embedded KevScript interpreter uses the flattened reflexivity methods to implement basic services (e.g. instance life cycle, binding and parameters management).

Upon receiving a Kevoree script in a compressed format a microcontroller performs the following tasks:

Algorithm 2 KevScript Interpreter

```
Function  interpretScript(script : KevScript)
  interruptScheduler()
  if permanentMemory.size >= PermanentMemoryLimit then
    resetPMemoryIndex()
    writeToPMemory(exportCurrentState())
  end if
  lastRecoveryPoint ← createRecoveryPointInPMemory()
  ∀ st, st ∈ script.statements → writeToPMemory(st)
  if executeFromPMemory(lastRecoveryPoint) then
    closeRecoveryPoint(lastRecoveryPoint)
  else
    rollback(lastRecoveryPoint)
  end if
  resumeScheduler()
```

Scripts are checked independently of the communication context and then stored. The new configuration is committed by an atomic write in the memory once it has been validated, thereby implementing an atomic transaction mechanism. This technique prevents incomplete memory saves. In case of any problems during the KevScript interpretation, a rollback is achieved by a simple reboot of the microcontroller. The Figure 1 gives an overview of this process taking as an example a temperature sensor reconfiguration. At T0 time a full configuration is pushed containing C code and an initial KevScript. At Ti time a KevScript is pushed adding 2 instances. Between Ti and Tj time a power failure occur resulting on a recovery using memory saved KevScript.

5. VALIDATION

This section describes our experimental setups for validating our approach against the criteria identified in Section 3[5]. More precisely, our experiments measure the following parameters:

Downtime: overall time needed by the microcontroller to adapt, including uploading of the new configuration. This metric thus measures the overhead induced by the microcontroller to manage its own state with respect to domain

[5]More details on this experiment can be found `http://blog.kevoree.org/pages/kevoree-for-microcontroller-benchmark`

applications execution time.
Volatile memory usage (RAM): amount of RAM memory dedicated to dynamic allocation of component instances, channels, and bindings. This metrics thus influences the maximum number of component instances, channels and bindings that a microcontroller can manage.
Persistent memory usage: amount of persistent memory used to store reconfiguration scripts and impact of storage strategy on memory life time. Persistent memory types such as the EEPROM embedded in the 8 bits AVR have a limited number of write cycles certified for each byte, thereby limiting the amount of storable data.
Recovery reboot delay: time needed by the microcontroller to reboot and restore its last configuration, after a crash or a loss of power.

We have used realistic configurations to assess our approach and evaluate the overhead induced by our dynamic component-based platform for microcontrollers, compared with static configurations that are updated by flashing the whole memory. All experiments were done using the Kevoree Arduino node implementation [6] running on an Arduino board with an ATMEL AVR 328P microcontroller. This processor embeds 32 KB of flash memory for storing programs, 2 KB of RAM memory and 1 KB of EEPROM. A flash-type memory (microSD) connected via an SPI bus was also used as persistent memory to assess the impact of memory type on results.

The following subsections show our specific experimental protocol and results, while the last subsection will present an industrial use case to validate the seamless integration of μKevoree devices in an existing dynamic architecture.

5.1 Downtime: How long does an adaptation freeze business logic?

Experimental setup. In this experiment we setup five different configurations, similar to the ones presented in the case study (building automation). The corresponding Kevoree models used different numbers of instances to simulate changes between the configuration used at night and a personalized configuration used during the day. More details on these models are available here [6] . In a nutshell, these models were configured with 4 nodes, hosting 0 to 10 instances each. Instances are implemented in C, with 30 lines of code each on average.

In a first step we generated the firmware of our test microcontroller, with code containing all type definitions used in this experiment. This step therefore includes code generation, compilation and writing into flash memory. Moreover, the generated code is automatically instrumented with probes to measure downtime and memory use (EEPROM and SDRAM). This step was repeated delayed of 100 ms with a new configuration that is chosen randomly; each new configuration was dynamically installed to replace the current running configuration. This random reconfiguration step was repeated 500 times. Figure 2 plots the raw data collected in this experiment. The plot on top shows that the RAM usage is constant. The second plot shows the downtime per reconfiguration, and third and bottom plots show downtime and script size respectively.

[6]http://goo.gl/Xl2z9

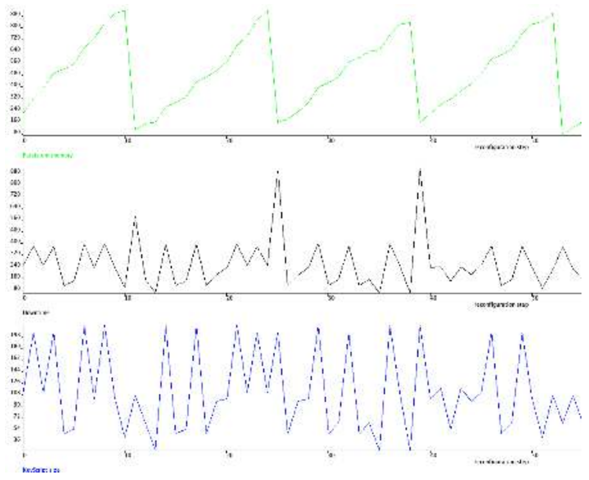


Figure 2: Experiment raw results[7]

Experimental results and analysis. Deploying a configuration by flashing the whole firmware is very costly: the downtime to deploy the initial configuration is 12.208 seconds. This high value comes mainly from the long transfer time of a full firmware but also from the time taken by the default boot loader to perform a full restart of the microcontroller. This value varies in a range of +/-2 seconds.

Results of this first experiment highlight that the size of the reconfiguration script is highly correlated with the downtime time: the Spearman correlation coefficient observed between script size and downtime is higher than 0.9. In addition, the compression algorithm used to decrease the script size in EEPROM has also an impact on downtime. We observed that the execution of this task is directly correlated with higher values of downtimes. This is discussed in the next subsection.

After 500 cycles of reconfiguration we measured the following extrema and mean values:

- minimum downtime of 58 ms, 210 (i.e. 12208 / 58) times faster than static flashing;
- maximum downtime of 916 ms, 14 (i.e. 12208 / 916) times faster than static flashing;
- mean downtime of 235 ms, 52 (12208 / 235) times faster than static flashing.

We used a distribution by percentiles graph for downtime values to better analyze these data, as shown in the following table.

Percentile(%)	0	5	25	50	75	95	100
Downtime (ms)	58	59	139	221	248	398	916

The graph in Figure 3 clearly shows that the downtime values are clustered around 220 ms. 95% of the values are below 400 ms and 75% are below 250 ms. Then, only 5% of the values are above the 400 ms, which is explained by the EEPROM compression step. The lazy compression strategy allows us to limit the number of peaks and keep the maximum value around 200 ms. The highest values for the downtime are systematically linked to a reduction of the EEPROM size (caused by the compression routine). We observed 32 compressions of the EEPROM during the 500 reconfigurations *i.e.*, 6.4% of the reconfigurations trigger a compression so that they can be completely stored into the EEPROM. The maximum value of these 32 downtime peaks is 916 ms, the minimum is 218 ms and the mean value is 580.815 ms. This

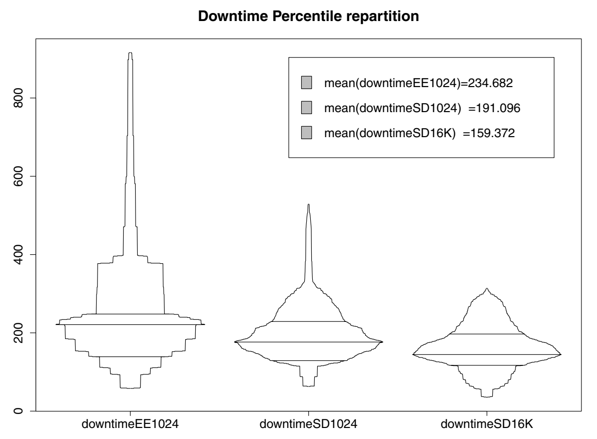


Figure 3: Flash RAM percentile downtime distribution (in ms)

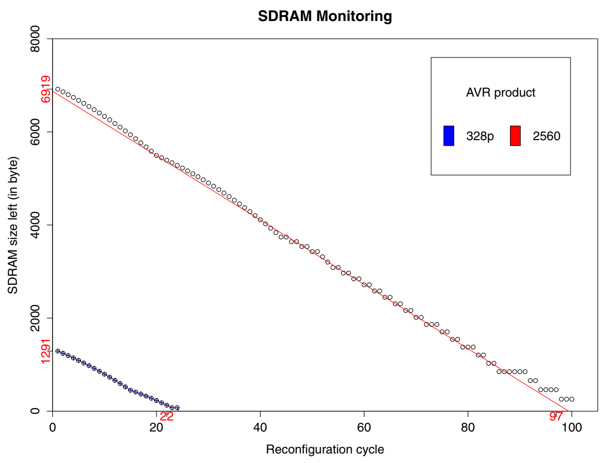


Figure 4: SDRAM capacity experiment

mean value is significantly higher than the mean value of the whole set of 500 reconfigurations (234.682 ms).

Probes also monitored SDRAM during this experiment. Neither memory leaks nor memory fragmentation occurred. Although the SDRAM is stable, it is necessary to check that the overhead induced by the framework actually allows for the dynamic creation of a realistically high number of instances, to match concrete use case needs. Our next experiment aimed at evaluating the capacity (in terms of dynamic instances) of our test microcontroller.

Experimental results and analysis with a different setup. We used the same protocol of 500 iterations, but we replaced the EEPROM with a 2 GB external flash memory (SD card), connected on an SPI bus. This experience is repeated twice: with only 1kb (same size as EEPROM), and with 16kb; results are shown in the following table.

Percentile(%)	0	5	25	50	75	95	100
DowntimeSD 1K(ms)	63	88	129	176	229	324	529
DowntimeSD 16K(ms)	35	56	117	145	197	297	314

Flash memory has a longer initialization time, which explains that the lowest values for the flash experiment are higher than the lowest value in the EEPROM experiment. However, writing speed of flash memory is high, resulting in a homogenization of downtime, which is under 200 ms most of the time. One can notice that the large increase of persistent memory (16 kb) clips the downtime peaks and therefore improves average downtime value. However this does not change the distribution of main values significantly.

5.2 Volatile memory usage: how many instances?

Experimental setup. The purpose of this experiment was to precisely determine the maximum number of instances that can fit into the SDRAM. An initial configuration was created with three instances: a timer, a switch and a default channel. Every 100 ms, the configuration was expanded by adding a new switch instance. Probes were injected to monitor the SDRAM.

Experimental results and analysis.
Figure 4 shows that SDRAM memory is full after 22 cycles, *i.e.* our test microcontroller can manage 25 instances (the 3 initial ones plus 22 additional instances). In practice, the count of devices controlled by one single microcontroller is approximately equal to the number of pins they have. In addition, microcontrollers should be able to run some code to orchestrate these devices and perform some computation. Kevoree is able to manage 25 instances (both for wrapping physical devices and for defining some orchestration) on our test microcontroller (22 pins). Despite a memory overhead, our approach is compatible with current practices.

Experiments results and analysis with a different setup. In this setup we used an ATMEL 2560 (4 KB of SDRAM) as our test microcontroller. The AVR 2560 accepts a load of 100 instances *i.e.* an improvement of+300% instances with +300% SDRAM. Again, the number of instances is larger than the number of pins (80) of the AVR 2560.

5.3 Persistent memory usage: How many certified reconfigurations?

Experimental setup. We used the setup of Section 5.1.

Experimental result and analysis. We observed that $500/32 = 15.625$ reconfigurations can happen before the EEPROM (1 KB) is full, requiring a compaction using a new initial state. As for Solid State Disk [2], write operations to EEPROM should to be distributed throughout the memory in order to distribute wear. Our algorithm writes every byte before computing a new initial state. Each byte of this memory is certified for 100,000 writes [7]. Therefore if we assume 100 reconfigurations every day (which is much more than what is needed in most case studies), each byte of the EEPROM will be written 6.4 times a day on average, ensuring 15,625 days (i.e. about 43 years) of certified lifetime for the EEPROM.

Experiments results and analysis with a different setup.
In average, we can serialize the reconfiguration scripts of our experiment using 16 bytes. Since our algorithm ensures that every byte is written before the memory needs to be compressed, we can use the reasoning of the previous paragraph to compute the lifetime with a larger memory.

[7]http://arduino.cc/en/Reference/EEPROMWrite

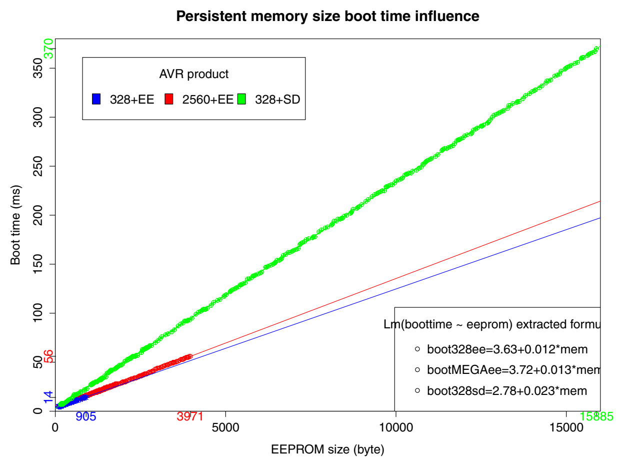


Figure 5: Persistent memory size boot time influence

5.4 Recovery reboot delay: How long to recover?

Experimental setup. This experiment used the configuration set described in Section 5.1, with only 50 cycles of reconfiguration performed every 2 seconds. The microcontroller was physically rebooted between each reconfiguration, and a new probe was inserted to measure the configuration restore time the after booting.

Experimental results and analysis.
Figure 5 displays the results of this second experiment. It appears that in the worst case with this AVR product (when the 1 KB EEPROM memory is almost full) the boot time is approximately 15 ms. In the best case (EEPROM almost empty) the boot time is approximately 3 to 4 ms. This value is mainly due to the slow read speed of the EEPROM embedded in the AVR. However, the time to restore a configuration is reasonable for most use cases, even in the worst case. We can therefore infer that the script size in EEPROM has a small impact on boot time with respect to save time. Therefore the best strategy is to use all available memory.

Experimental results and analysis with a different setup.
We used the same setup, but we replaced the EEPROM with a flash memory (1 KB and 16 KB). Using SD memory instead of EEPROM implies a longer boot time. We noticed a coefficient value of 0.012 for the EEPROM, and of 0.023 for the SD. This comes from the extra computation needed to read and write SD card. Unlike the EEPROM, the communication bus to access the SD flash is external to the microcontroller. The initialization time taken by flash memory configuration is linearly distributed to the limit of 16 KB. Above this size, the initialization time becomes greater than 360 ms. This is significantly higher than the reconfiguration time.

5.5 Discussion

The choice of persistent memory type and size depends on the use case needs. In some cases, there is a real need for traceability, and the history of the system should be kept *e.g.*, for *post mortem* analysis in case of failure. In other cases, performance of adaptation and boot time after a failure is more important. The benchmark developed in this approach can be useful to determine empirically which memory setup to use.

However, using an external memory significantly increases the price of such a platform. In addition, ensuring atomicity of reconfigurations is more difficult because of the asynchrony of transfer protocols. Existing protocols for interaction with SD cards (like MMC) are not suitable for storing reconfiguration scripts. More precisely, these scripts are much shorter (around 25 bytes in our experiments) than the minimum frame size (512 bytes) required by these protocols, and this leads to a significant overhead. In practice, most embedded devices combine EEPROM and flash memories. Kevoree allows designers to combine different memory types for different purposes.

μ-Kevoree overhead. Our framework adds several overhead sources, especially for the management of dynamic instance creation of components and channels. In order to quantify overhead, we measured volatile and program memory sizes on an HelloWorld program, using plain C and a Kevoree firmware setup on a 328P AVR microcontroller. The plain C version left 1842 free bytes after boot sequence, while the Kevoree version left 1604 bytes free. This represents an overhead about 11% of the total available RAM (242 bytes out of 2048). The plain C version used 2.3 KB of firmware memory, while the Kevoree use was 7.3 KB, giving an overhead of about 15% of the total 32kb available. The impact of Kevoree scheduler on processing cycles is highly program dependent, and we are currently experimenting further to compute this overhead.

Our synchronization and communication layer introduced an overhead under 15% on both memories, which is an acceptable value in the case of our IoT application. However, this impact should be evaluated in more depth for hard realtime applications, with a special attention to components needs in terms of processing cycles.

6. RELATED WORK

Software architecture aims at reducing complexity through abstraction and separation of concerns by providing a common understanding of component, connector and configuration [10, 21, 32]. One of the remaining challenges, strengthened by the Future Internet and CPS [24], is to properly manage dynamic architectures. SCA [8] is a standard that highlights modular software architecture concepts. It provides a model for composing applications that follow Service-Oriented Architecture principles. Frascati [29] is a SCA runtime that allows developing highly configurable applications. However, SCA focuses on rather heavy nodes typically able to run a JVM whereas μ-Kevoree also manages the dynamic adaptation of microcontroller-based systems, in addition to Java nodes, with a contained overhead.

Many approaches have highlighted the need for dynamic architectures to implement pervasive computing. A common approach consists in building a middleware to hide the heterogeneity of networks, hardware, operating systems, and programming languages.Rellermeyer *et al.* [26] for example provides an architecture for flexible interaction with electronic devices. Based on OSGi to implement a dynamic module system, their approach provides an abstraction layer

[8]`http://osoa.org/`

for device independence. Their architecture has the following non-functional benefits: scalability and ease of administration, flexibility, security, and efficiency. In a similar vein Escoffier *et al.* proposed iPOJO [13], a service component runtime that simplifies the development of OSGi applications. iPOJO has mainly been used in home-automation to implement service-oriented pervasive applications [7]. AutoHome is a middleware that extends the iPOJO component model, to create a framework to host autonomic home applications. Gaïa [27] is a CORBA-based meta-operating system for ubiquitous computing, built on top of a classical operating system aimint at abstracting the heterogeneity and complexity associated with ubiquitous environments. Olympus [25] proposes a high-level DSL to ease the development of Gaïa applications. Cassou *et al.* [9] proposes a generative programming approach to provide programming, execution and simulation support dedicated to the pervasive computing domain. They also demonstrate how abstraction can help to guide and verify the development of pervasive applications. Again, all these approaches rely on a reconfigurable middleware (often OSGi-based), and this restricts their deployment to powerful nodes, e.g. powerful enough to run a Java virtual machine. Our goal is to provide the same level of abstraction to develop pervasive and adaptive applications for powerful nodes and also for resource-constrained devices.

Several approaches have shown the benefits of using Model Driven Engineering (MDE) to design and reconfigure pervasive applications. Model-based approaches such as Matlab [9], Charon [3], UMLh [8], HyRoom [31], Masachio[16], Mechatronic UML [28], HyVisual [12], or SysML [10] propose a model to code development process with verification techniques to design modular embedded systems. However, none of these approaches support dynamic adaptation of a running system without first designing the adaptation at a business level. This, in practice, significantly reduces the number of configurations these approaches can manage. Indeed, these approaches provide no means to manage the combinatorial explosion of the number of configurations typically encountered in CPS: all configurations need to be explicitly designed.

Several approaches have shown the need of dynamic reconfiguration capabilities for embedded systems. Reconfigurable intelligent sensors are now able to confront major challenges in the design of cost-effective, energy-efficient, customizable systems, for example in the domain of health monitoring systems adaptable to individual users [19], or in the domain of operating system kernels [5]. Run-time reconfiguration can be achieved through programmable logic reconfiguration and/or software adaptation. In the first case, reconfigurable System on Chips [30] are a promising solution. To support software adaptation of embedded software, other works reuse a software architecture-based approach to the construction of embedded systems. For example, The Koala model [32], used for embedded software, allows late binding of reusable components with no additional overhead. Think [14] defines a component-based framework to support different mechanisms for dynamic reconfiguration and to select between them at build time, with no changes in operating system and application components. Different from these approaches, Fleurey *et al.* [15] present an approach based on state machines and an adaptation model to derive adaptive firmwares for microcontrollers. The approach relies on automatically enumerating configurations by exploring a set of adaptation rules defined at design time and compiling the resulting state-machine (which merges the business logic and the adaptation logic) into an optimized, yet static, firmware. In [17], Hofig *et al.* highlight the use of models@runtime for resource-constrained devices. They provide a UML state machine interpreter for AVR microcontrollers and compare the performance overhead with static code generation: model@runtime interpretation is adequate for the majority of situations, except when dealing with high-throughput or delay-sensitive data. Influenced by these approaches, Kevoree leverages models@runtime for microcontrollers and proposes new mechanisms to support dynamic reconfigurations and to select between them at runtime.

[9]`www.mathworks.com/products/matlab/`

[10]`http://www.sysml.org/`

7. CONCLUSION AND PERSPECTIVES

This paper presented μ-Kevoree, which pushes dynamicity and elasticity concerns directly into resource-constrained devices, based on the notion of models@runtime. This modeling layer that micro-controllers expose at runtime, enables the efficient and safe reasoning (by other Kevoree nodes: Java or Android) to adapt microcontroller-based nodes. In particular, this paper focused on the challenges met when mapping Kevoree and models@runtime features to low power microcontrollers, and on the required tradeoffs because of the stringent resource constraints.

This new version of Kevoree has been thoroughly evaluated with benchmarks in order to assess its usability in realistic setups. Despite an overhead with respect to static (non adaptive) code, these benchmarks have shown that 75% of the transactional reconfigurations can be performed in less than 250 ms, which is an acceptable value in many case studies. This is definitely faster (by a factor of almost 50) than a full memory rewrite of the firmware. Also, these benchmarks have shown that the time needed to reboot a microcontroller and restore its previous configuration is a linear function of the script size. For example, booting using a 1 kB EEPROM memory takes between 3 to 15 ms, while this memory size is large enough to store the script of 15 successive reconfigurations before needing compaction. Finally, the benchmarks have shown that Kevoree enables the deployment of software component instances in a number greater than the available pin count on the microcontroller. It is therefore possible to bind a software component to each physical device controlled by the microcontroller, and to deploy an extra component to coordinate these components.

In the future, we will improve the reliability of reconfigurations by making the computation of the initial state step transactional (compression of the persistent memory) *e.g.*, and by exploiting a circular rolling buffer on the persistent memory. Our scheduling algorithm is another area for improvement. Based on existing opportunistic garbage collectors (*e.g.* Java) we will leverage the computational cycles not used by hosted components to trigger the compression of the persistent memory in a lazy way, rather than waiting for a "memory full" event. We will also investigate further optimizations to reduce the memory consumption and the energy consumption. Finally we are planning to integrate simple reasoners in microcontrollers driven by μ-Kevoree so that they can operate in a fully autonomous mode, with no need to delegate the reasoning to a larger node.

Acknowledgment

This work has been funded by the EU FP7 projects: FI-PPP ENVIROFI, S-Cube NoE and MODERATES.

8. REFERENCES

[1] The Internet of Things Meets The Internet of People. http://www.harborresearch.com/_literature_60961/The_Internet_of_Things_Meets_The_Internet_of_People.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70. USENIX Association, 2008.

[3] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specification of hybrid systems in charon. In *Proceedings of the Third International Workshop on Hybrid Systems: Computation and Control*, HSCC '00, pages 6–19, London, UK, 2000. Springer-Verlag.

[4] F. André, E. Daubert, N. Grégory, B. Morin, and O. Barais. F4plan: An approach to build efficient adaptation plans. In *MobiQuitous*. ACM, 2010.

[5] S. Bagchi. Nano-kernel: a dynamically reconfigurable kernel for WSN. In *1st international conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications*, MOBILWARE '08, pages 10:1–10:6, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[6] G. S. Blair, N. Bencomo, and R. B. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009.

[7] J. Bourcier, A. Diaconescu, P. Lalanda, and J. A. McCann. AutoHome: An Autonomic Management Framework for Pervasive Home Applications. *ACM Trans. Auton. Adapt. Syst.*, 6:8:1–8:10, February 2011.

[8] S. Burmester, H. Giese, and O. Oberschelp. Hybrid uml components for the design of complex self-optimizing mechatronic systems. In J. BRAZ, H. ARAÃŽJO, A. VIEIRA, and B. ENCARNAÃĞÃČO, editors, *INFORMATICS IN CONTROL, AUTOMATION AND ROBOTICS I*, pages 281–288. Springer Netherlands, 2006.

[9] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications. In *33rd International Conference on Software Engineering (ICSE'11)*, pages 431–440, Honolulu, US, 2011. ACM.

[10] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *24th International Conference on Software Engineering*, ICSE '02, pages 266–276, New York, NY, USA, 2002. ACM.

[11] P.-C. David, T. Ledoux, M. Léger, and T. Coupaye. FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *Annales des Télécommunications*, 64(1-2):45–63, 2009.

[12] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[13] C. Escoffier, R. S. Hall, and P. Lalanda. iPOJO: an Extensible Service-Oriented Component Framework. In *IEEE SCC*, pages 474–481. IEEE Computer Society, 2007.

[14] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller. Think: A Software Framework for Component-based Operating System Kernels. In *General Track of the annual conference on USENIX Annual Technical Conference*, pages 73–86, Berkeley, CA, USA, 2002. USENIX.

[15] F. Fleurey, B. Morin, and A. Solberg. A Model-Driven Approach to Develop Adaptive Firmwares. In *SEAMS'11@ICSE: Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Honolulu, Hawai, USA, 2011.

[16] T. Henzinger. Masaccio: A formal model for embedded components. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. Mosses, and T. Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, volume 1872 of *Lecture Notes in Computer Science*, pages 549–563. Springer Berlin / Heidelberg, 2000.

[17] E. Höfig, P. H. Deussen, and I. Schieferdecker. On the performance of UML state machine interpretation at runtime. In *6th international symposium on Software engineering for adaptive and self-managing systems (SEAMS '11)*, pages 118–127, New York, USA, 2011. ACM.

[18] R. Johnson and B. Woolf. The Type Object Pattern, 1997.

[19] E. Jovanov, A. MilenkoviÄĞ, S. Basham, D. Clark, and D. Kelley. Reconfigurable Intelligent Sensors for Health Monitoring: A Case Study of Oximeter sensor. In *26th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 4759–4762, 2004.

[20] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[21] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Softw. Eng.*, 26:70–93, January 2000.

[22] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg. Models@ Run.time to Support Dynamic Adaptation. *Computer*, 42(10):44–51, 2009.

[23] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel. Taming Dynamically Adaptive Systems with Models and Aspects. In *ICSE'09: 31st International Conference on Software Engineering*, Vancouver, Canada, May 2009.

[24] E. D. Nitto, C. Ghezzi, A. Metzger, M. P. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Autom. Softw. Eng.*, 15(3-4):313–341, 2008.

[25] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas. Olympus: A High-Level Programming Model for Pervasive Computing Environments. In *Third IEEE International Conference on Pervasive Computing and Communications*, pages 7–16, Washington, DC, USA, 2005. IEEE Computer Society.

[26] J. S. Rellermeyer, O. Riva, and G. Alonso. AlfredO: an architecture for flexible interaction with electronic devices. In *9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 22–41, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[27] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing*, 1:74–83, October 2002.

[28] W. Schäfer and H. Wehrheim. Graph transformations and model-driven engineering. chapter Model-driven development with Mechatronic UML, pages 533–554. Springer-Verlag, Berlin, Heidelberg, 2010.

[29] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, 2011.

[30] H. Singh, M.-H. Lee, G. Lu, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho. Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.*, 49:465–481, May 2000.

[31] T. Stauner, A. Pretschner, and I. Péter. Approaching a discrete-continuous uml: Tool support and formalization. In *Workshop of the pUML-Group on Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists*, pages 242–257. GI, 2001.

[32] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.