A Model-Driven Approach to Develop Adaptive Firmwares[*]

Franck Fleurey
SINTEF ICT
Oslo
Norway
Franck.Fleurey@sintef.no

Brice Morin
SINTEF ICT
Oslo
Norway
Brice.Morin@sintef.no

Arnor Solberg
SINTEF ICT
Oslo
Norway
Arnor.Solberg@sintef.no

ABSTRACT
In a near future it is expected that most things we rely on in our everyday life will contain sensors and electronic based information, have computing power, run embedded software and connect to networks. A multitude of heterogeneous things will operate in a highly dynamic environment and will collaborate with other connected systems and things to provide users with adaptable services. Constructing and controlling such adaptive things is complex. A main challenge is to cope with the dynamicity which requires the things to autonomously adapt to various execution contexts. In this paper we present an approach to develop adaptive firmwares for devices which do not have the resources to rely on advanced operating systems, middlewares or frameworks to support runtime adaptation. The paper is illustrated with the example of an adaptive temperature sensor network running on a microcontroller platform.

Categories and Subject Descriptors
D.2 [**Software Engineering**]: Miscellaneous

General Terms
Design, Performance

Keywords
Adaptive systems, embedded systems, MDE, DSL, aspects

1. INTRODUCTION
In our everyday life we are surrounded by things (e.g., household appliances, clothing, cars, traffic lights, buildings, sensors, books, mobile phones, etc) which we interact with, and which we rely on. Currently, more and more of these things are equipped with computing power and communication capabilities. Connecting these things into what is denoted Internet of Things (IoT) implying that all these things will be part of the Future Internet, provides exciting opportunities for developing new and more advanced applications in many different application domains. Examples of such application domains includes smart house, smart city, smart power grids, ambient assisted living and intelligent transport systems. Dynamicity (for example, due to mobility and change of contexts and preferences) is a key concern in such application domains, requiring the systems to dynamically adapt to their changing environments. Two main challenges for the development of these applications are to manage massively distributed things, many of which are small and have scare computing resources and the dynamic environment they have to cope with.

For typical IoT applications, the network of devices is composed of devices of different sizes and capabilities ranging from simple passive RFID tags to large servers capable of processing vast amounts of data. The approach presented in this paper focus on efficient development of high quality software that comprise advanced adaptation capabilities on relatively small thing; characterized by having too little resources in terms of available power, CPU and memory to rely on advanced operating systems, middlewares or frameworks. Examples of such things are various types of sensors or microcontroller based devices for which complex adaptive behavior has to be provided by a firmware.

The main contribution of this paper is the approach to develop firmware with adaptation capabilities for low-resource devices. The idea of the approach is to combine state of the art techniques for embedded system modeling using state machines, variability modeling using aspect oriented modeling and adaptation modeling using a combination of constraints and policy rules. The approach is based on using a set of domain specific languages to develop platform independent models specifying the functional behavior, variability and adaptation logic of the thing. These models are exploited at design time to carry out analysis, simulation and validation and are then automatically compiled to firmwares with adaptation capabilities.

The paper is organized as follows: Section 2 presents the motivations of this work and the case study of an adaptive temperature sensor which we use throughout the paper. Section 3 presents the proposed approach and details each of its steps from the models to the compiled firmware. Section 4 discusses related works and finally section 5 concludes.

[*]This work has been supported by the DiVA project (EU FP7 STREP), the VERDE project (EU ITEA), and the MODERATES and SiSaS projects (SINTEF).

SEAMS '11, May 23-24, 2011, Waikiki, Honolulu, HI, USA

2. CONTEXT AND MOTIVATIONS

We illustrate our proposed approach on low power adaptive temperature sensors. Figure 1 presents an overview of the case study setup. The sensor network is made of three types of nodes: the sensors, the gateway and the clients. This is a very typical architecture for a wide variety of sensor networks. The sensors are small battery-powered nodes which are distributed and run autonomously on batteries for extended periods of time. The gateway is a bridge between the low power communication technology used by the sensors and more standard communication technologies (in our case Bluetooth and ZigBee). Finally there can be many different client devices and applications exploiting the data provided by the sensor network through the available gateways.

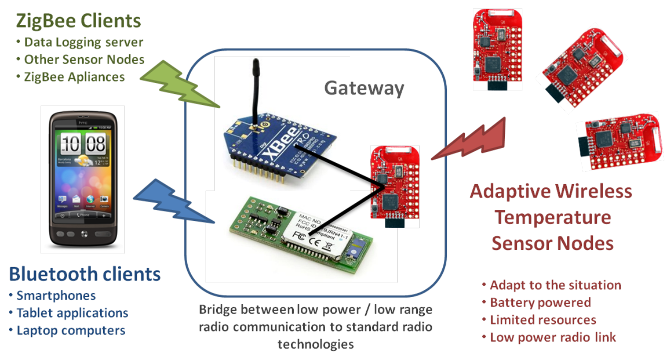


Figure 1: Sensor network used in the case study

There are many software engineering challenges related to communication, distribution or adaptation when concerning complete sensor networks. This paper however, focuses on efficient development of the software running on each of the nodes in a sensor network. These nodes typically have low power and restricted CPU and memory resources. In the case study these nodes are the adaptive wireless temperature sensor nodes depicted on the right hand side in Figure 1. The hardware for these sensors are a micro-controller board with a low power radio communication chip made by Texas Instruments (EZ-430 RF2500)[1]. The board is based on an MSP430 F2274 16 bits RISC micro-controller running at up to 16 Mhz with 1 ko of SDRAM and 32 ko of flash memory. This micro-controller has a built-in temperature sensor and the EZ-430 RF2500 board has a low power radio chip for wireless communication.

The objective is to build a firmware for this board which allow sampling the temperature from the built-in temperature sensor and transmit it to client applications. This sounds reasonably simple: we can implement it as an infinite loop which samples the temperature and transmits it using the radio chip. In practice, this would be very inefficient since this loop would drain the battery of the sensor very quickly: using 2 AAA batteries, the sensor would run only for about 20 hours. Building a more sophisticated firmware which leverages power saving capabilities of the hardware and adapts to the clients needs could allow running the sensor for over a year on the same two AAA batteries, thus, it is worthwhile exploring.

[1]http://focus.ti.com/docs/toolsw/folders/print/ez430-rf2500.html

Some features which we would like to explore for our temperature sensors are to:

- Reduce measurement accuracy and frequency to save power when it is acceptable: Depending on the situation and the applications using the sensor data it can be acceptable not to have the highest measurement frequency and accuracy. If the sensor is monitoring the temperature of a building to adjust heating it is sufficient to get temperature sample every 2 minutes with an accuracy within plus/minus one degree. This allows the sensors to conserve a lot of power. However, in case of a fire the sensor should adapt and transmit high resolution temperature data at 10 seconds interval in order to monitor the fire and help extinguishing it.

- Have the sensor record minimum and maximum measured temperature: This is a typical use case for a temperature sensor. If the sensor itself does not provide it, any application which needs this information needs to get very frequent updates from the sensor in order to store the minimum and maximum temperature. This is very inefficient since this requires frequent data transmission which can be avoided if the sensor records minimum and maximum temperature and just transmits the values when they are needed.

- Provide a way for client applications to register with the sensor and define temperature alarms. For an application which manages the heating of a set of rooms, there is no need to get the temperature from all sensors at regular intervals. The application can register with the sensors and define which are the threshold temperature for which it needs to be notified in order to turn on and off heaters. This way many unnecessary temperature transmissions (and the corresponding power) can be saved.

- Have graceful degradation of the sensor services when its battery goes low. When the battery of the sensor drains, it should not suddenly stop working. There should be a way to query the sensor for its battery status and have it run in gradually degrading mode in order to get the most out of it before its batteries need to be recharged or replaced.

In the end, even for such simple temperature sensors, combining all these features and making sure that the sensor will adapt to optimize performance in all the different contexts is a non-trivial task. This is made worse by the fact that the restricted resources available makes it unrealistic to rely on advanced operating systems, frameworks or middelware platforms to cope with the complexity. In practice, the software for these nodes is very often developed by hand directly in C or in assembly languages. The growing complexity of the required features combined with strong requirements on reliability and power efficiency makes this task tedious, costly and error prone.

The objective of the approach proposed in this paper is to combine embedded system modeling, variability modeling and adaptation modeling technique to build a complete tool-chain for handling the complexity developing adaptive firmware for low resources systems. The idea is to exploit model based abstraction techniques to formalize the functional behavior, variability and the adaptation policies for the system using , and then make use of these models to both carry out design time analysis and validations and to automatically derive the running firmware.

3. APPROACH

Figure 2 presents an overview of our approach to develop adaptive firmwares for low-resource systems such as the nodes of a sensor network. The idea of the approach is to cope with adaptation complexities and ensure efficient development of high quality firmware by capturing the system functional behavior, variability and adaptation combining a set of tiny domain specific modeling languages to produce a consistent combined model that can be validated at design time and then to automatically synthesize code from these models.

Figure 2 is divided in three horizontal layers: Tools, Models and Compilation chain. The models are the central elements which drive the approach and they are the only artifacts to be manipulated by the developers. The top layer of the figure (Tools) presents the set of tools which is used to view, edit, weave and validate the models. The bottom layer of the figure (Compilation chain) presents the 5 steps of the automated transformation process which takes the models as an input and produces firmware ready to be executed on the hardware platform. As shown in Figure 2 different types of models are used to model different concerns of the system:

- The base model (on the right in Figure 2) captures the structural and behavioral elements which are common to all configurations of the system. The formalism we use for this base model is composed of a structural view which describes the devices present in the system and the messages these devices can send and receive and a behavioral view which is expressed as a state machine.
- The aspect models encapsulate the variability of the system. They contain a set of behavioral model fragments which can be woven in the base model in order to change the behavior of the system. The idea is to capture the variability by separating all the variable pieces of behavior into aspects. A specific runtime configuration (or mode) of the system corresponds to a set of aspect models woven into the base.
- The adaptation model (on the left in Figure 2) formalizes adaptation logic of the system, i.e., which aspects should be woven depending on the context. For that, the the adaptation models contains three parts:
  - The definition of a set of features (or Variants) which link the aspect models containing their implementation.
  - The definition of the environment (or Context) for the system. It is formalized as the set of variable from the environment which are relevant for choosing the most suited configuration for the system.
  - A set of constraints and rules linking the context which define the adaptation policy by linking the context information and the features of the systems.

To work with these different types of models, the tools include editors for editing the adaptation model, the aspect models and the base model, a weaver to build the model of any arbitrary configuration of the system and simulators to validate both the adaptation behavior defined in the adaptation models and the functional behavior of any individual configuration.

Once the developer is satisfied with a set of models, the firmware is automatically derived from the model following the five step process depicted at the bottom of Figure 2. First, an exhaustive simulation of the adaptation models allow computing the optimal configuration for all possible context instances. Each of these configurations corresponds to a set of aspects to be woven in the base model. The second step consists in weaving the behavioral models for all these different configurations in order to create a set of models corresponding to each configuration of the system. The role of the third step is to compose the models for of all reachable configurations together with the context information to create a single model which implements both the behavior of all modes and the context triggers to switch between models. The complete model then serves as the basis to generate the C code for the adaptive firmware. The last step relies on a cross-compiler in order to create firmware which can run on the target platform. The following subsections detail these steps and illustrate the approach on the case study.

```
device Timer
{
    // Start the Timer
    message start(timer_id : Integer, delay : Integer);
    // Cancel the Timer
    message cancel(timer_id : Integer);
    // Notification that the timer has expired
    message timeout(timer_id : Integer);

    sends timeout
    receives start, cancel
}
```

Figure 3: Example model for a Timer device

3.1 State machine modeling / code generation

The proposed approach relies on modeling the functional behavior of the system under development using state machines. State machines have been successfully applied to model, verify and generate code for embedded systems for a long time [16, 15]. There are many different state machine modelling tools both standalone and UML-based, which can be used to model state machine and perform simulation and code generation. The particular framework we use is a domain specific language (DSL) we have developed specifically for distributed micro-controller applications. The prototype implementation of the approach relies on this DSL but the ideas could be applied with any state machine based formalism. Our DSL offers the constructs to:

- model a set of hardware devices or software components with their interfaces in terms of asynchronous messages they can exchange. Figure 3 presents the example of the model for a Timer device. The description of the Timer contains the description of the messages which can be exchanged with it and the sends and re-

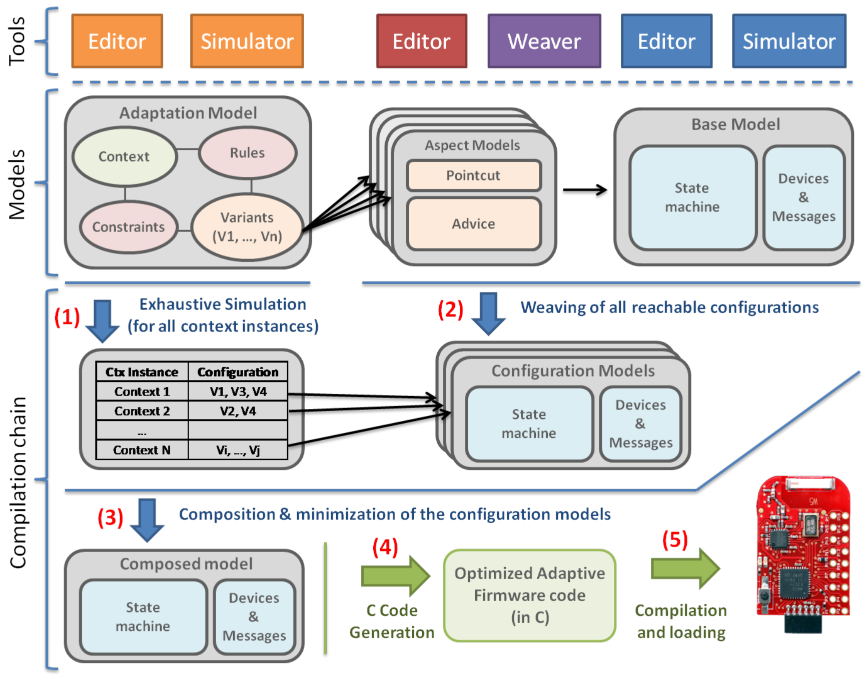


Figure 2: Overview of the proposed approach

ceives properties specify which are the messages which can respectively be sent and received by the Timer.

- model the behavior of software components using composite state machines. The state machine we use are made of states (which can be composites) and transitions. The states can contain entry and exit actions and the transition can contain an event (like the reception of a message), a guard and some actions. In order to model the actions, our domain specific language includes a simple action languages which allows for dealing with variable, basic calculations and message sending and include basic control structures such as blocks, conditional and loops. Figure 3 presents a excerpt of the state machine for a temperature sensor using the textual syntax provided by our tool.

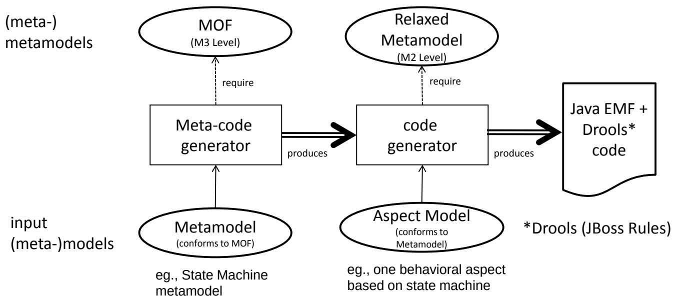
Based on the descriptions of devices and components together with their state machine, we have developed a set of template-based C code generators targeting the micro-controller platforms AVR from Atmel and MSP430 from Texas instruments. In the case study presented in this paper we use the MSP430 generators (the target platform is an MSP430F2274 micro-controller). The meta-models, tools, editors and code generators are available as open source at http://code.google.com/p/moderates/.

To implement the approach proposed in this paper the objective was to be able to reuse the modeling tools and code generator without having to modify them to take into account variability and adaptation. Instead, the idea is to capture variability and adaptation in other types of models which can automatically be composed into a model from which the existing code generator can be applied. The next two sub sections discuss how variability is captured using aspects and adaptation using a domain-specific modeling language for defining adaptation policies.

```
composite state ACTIVE init ReadSensorValues {

    state WaitingForCmd {

        on entry {
            send Timer.start('1', '10000')
        }

        transition update_data -> ReadSensorValues {
            event Timer#timeout
            guard 'timer_id == 1'
        }
        transition GetData -> WaitingForCmd {
            event WTSClient#GetData
            action send WTSClient.SensorData('temp','0','0','batt')
        }
        transition GetStatus -> WaitingForCmd {
            event WTSClient#GetStatus
            action send WTSClient.SensorStatus('interval','0','0','0')
        }
        transition GetName -> WaitingForCmd {
            event WTSClient#GetName
            action send WTSClient.SensorName('name')
        }
        transition SetName -> WaitingForCmd {
            event WTSClient#SetName
            action 'strcpy(name, new_name);'
        }
    }

    state ReadSensorValues {

        on entry {
            send LED.light_on('2') // GREEN
            send MSP430Sensor.mesure_temperature()
        }

        on exit {
            send LED.light_off('2')
```

Figure 4: Excerpt of a state machine

3.2 Variability modeling using aspects



(meta-) metamodels
MOF (M3 Level)
Relaxed Metamodel (M2 Level)
require
require
Meta-code generator
produces
code generator
produces
Java EMF + Drools* code
input (meta-)models
Metamodel (conforms to MOF)
Aspect Model (conforms to Metamodel)
*Drools (JBoss Rules)
eg., State Machine metamodel
eg., one behavioral aspect based on state machine

Figure 5: SmartAdapters chain

To capture the variability in the system under development, the idea of the approach is to use aspect-oriented modeling, i.e. capture the elements which can vary in the system in aspects. The alternative configurations of the system are then built by selecting and weaving a specific set of aspects. In practice, we use SmartAdapters to design and weave the aspects. SmartAdapters relies on generative techniques to automatically produce an operational domain-specific AOM framework for existing modeling languages. The process is illustrated in Figure 5. In previous work this technique has been successfully applied to implement aspects on top of structural models such as architecture models [10]. In this work we have applied SmartAdapters to the state machine DSL described in the previous section. This allow us to specify and weave aspects in the behavioral models expressed using this DSL. Using SmartAdaptes, the definition of an aspect is made of three parts: a pointcut, an advice and a composition protocol.

The *pointcut model* is a polymorphic[2] behavioral fragment describing **where** the aspect will be woven. The more precise is the pointcut model, the more reduced is the set of join points (places in the base model that match the pointcut). Conversely, if the pointcut is vague (*e.g.*, if the attributes of the model elements are not specified), it is likely to match a wide set of join points.

The *advice model* is model fragment (with no “abstract” elements) that specifies **what** the aspect model brings. By default, new instances of the advice model elements are introduced for each specific join point. However, it is possible to associate advice model elements with innovative strategies strategies that specifies how the elements of the aspect should actually be woven [9, 8]. These strategies are especially useful in the context of composite models (architecture, state machines, etc).

The *composition protocol* (depicted with dashed lines in Figure 5) specify **how** to weave the advice model into the pointcut model. This imperative protocol is defined with statically-typed domain-specific actions. An initial set of primitives (corresponding to CRUD operations) is automatically generated for each specialization of SmartAdapters. It is also possible to extend the hierarchy of primitive with more advanced actions (e.g complex merge).

The aspects defined in such a way can be automatically woven into a base model. First the aspects are compiled into rules in Drools Expert[3] for join points detection and Java and Eclipse Modeling Framework (EMF) code for the advice and actual weaving. Once compiled the aspect can be applied automatically applied to different base models.

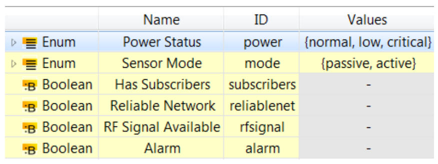
[2]After relaxing the metamodel (according to the rules defined in [13]), it becomes possible to instantiate meta-classes that were formerly abstract in the input metamodel. Such elements would match any element instance of a concrete sub-metaclass.

3.3 Adaptation Modeling

The third model required to specify an adaptive application is a model of what are the valid configurations and when to use each of them. Various approaches have been proposed to model adaptation either based on the definition of rule-based policies or based on the optimization of utility functions. The particular approach we use is a combination of both. It has been originally developed for handling large scale adaptive systems [4]. This approach has proved well-suited for describing adaptation policies in various academic and industrial adaptive applications. In all these cases, the adaptation policy is processed at runtime in order to dynamically select the best suited configurations. For low resources systems such as micro-controllers runtime reasoning is not a realistic option because of the low CPU and memory available. The idea is thus to process the adaptation model at design time in order to inject it into the final application.

The following illustrates the formalism used to model adaptation on the example of the temperature sensor case study.

Figure 6 presents the context model for the temperature sensor. The context is modeled by a set of variables related to the environment of the sensor. In this example the context is made of 6 variables. The power status is related to the state of the battery and has 3 different levels. The sensor mode corresponds to a user preference: if configured in active mode the sensor will broadcast data whereas if it is configured as passive it will only send data if a client sends a request. The 3 next variables correspond to different status for the client applications and network. The last variable, called *Alarm*, corresponds to an emergency situation such as a fire alarm which requires the highest frequency and accuracy for temperature measurements.



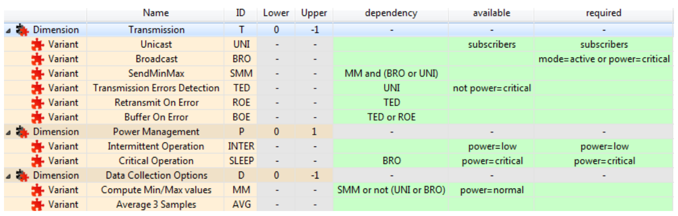
	Name	ID	Values
Enum	Power Status	power	{normal, low, critical}
Enum	Sensor Mode	mode	{passive, active}
Boolean	Has Subscribers	subscribers	-
Boolean	Reliable Network	reliablenet	-
Boolean	RF Signal Available	rfsignal	-
Boolean	Alarm	alarm	-

Figure 6: Context model

Figure 7 illustrates the variability of the temperature sensor. The sensor has 3 variability dimensions: Transmission, Power Management and Data Collection Options. All the variants of these dimensions are optional (lower bound = 0). Inside the Power Management dimension, it is possible to choose at most 1 variant. If none of the 2 variants are

[3]*a.k.a JBoss Rules*

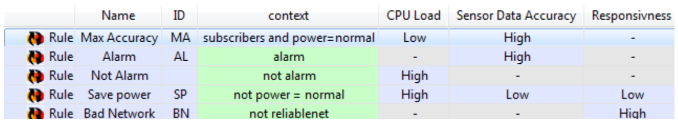
present in a configuration then the sensor is in constant operation. The Intermittent Operation variant makes the sensor hibernate after a few seconds of inactivity and wake-up periodically for measurements and interactions with client application. The Critical Operation variant aims a conserving a maximum of power but it is a degraded mode: the sensor only wakes up every 5 minutes to take a measurement and broadcast it. The two other dimension correspond to different ways of collecting and transmitting the data. The dependency columns allow setting constraints between variants. For example the Critical Operation variant depends on the Broadcast transmission variant. The two last columns allow setting constraints between the context and the variability. For example the Critical Operation variant is both available and required whenever the power situation is critical. These strict constraints restrict configuration possibilities for a specific context but are not sufficient to pin point the configuration to be used in a particular context.



	Name	ID	Lower	Upper	dependency	available	required
Dimension	Transmission	T	0	-1	-	-	-
Variant	Unicast	UNI	-	-		subscribers	subscribers
Variant	Broadcast	BRO	-	-			mode=active or power=critical
Variant	SendMinMax	SMM	-	-	MM and (BRO or UNI)		
Variant	Transmission Errors Detection	TED	-	-	UNI	not power=critical	
Variant	Retransmit On Error	ROE	-	-	TED		
Variant	Buffer On Error	BOE	-	-	TED or ROE		
Dimension	Power Management	P	0	1	-	-	-
Variant	Intermittent Operation	INTER	-	-		power=low	power=low
Variant	Critical Operation	SLEEP	-	-	BRO	power=critical	power=critical
Dimension	Data Collection Options	D	0	-1	-	-	-
Variant	Compute Min/Max values	MM	-	-	SMM or not (UNI or BRO)	power=normal	
Variant	Average 3 Samples	AVG	-	-			

Figure 7: Variability model and constraints

The choice of a specific configuration is made by defining a set of rules as presented on Figure 8. These rules allow defining trade-offs for different contexts in terms of the properties of the system which should be optimized. For the temperature sensor, three different properties are considered: the CPU load (which directly relates to the power consumption), the sensor accuracy and its responsiveness. The adaptation rules specify what is the priority of these properties in different contexts. For example the first rule specifies that if the sensor has subscribers and if the power situation is normal then the accuracy of the sensor is the most important property while the CPU load is less important.



	Name	ID	context	CPU Load	Sensor Data Accuracy	Responsivness
Rule	Max Accuracy	MA	subscribers and power=normal	Low	High	-
Rule	Alarm	AL	alarm	-	High	-
Rule	Not Alarm		not alarm	High	-	-
Rule	Save power	SP	not power = normal	High	Low	Low
Rule	Bad Network	BN	not reliablenet	-	-	High

Figure 8: Adaptation rules

The last element of the adaptation model is a table describing the impact each variant has on the properties of the system. This last element links the variants with the adaptation rules and makes it possible to actually compare the fitness of different configuration for a particular context. This table is not presented here because of the lack of space but all the details on the adaptation modeling technique we use, the precise semantics of each elements and the details about the implementation of the tools can be found in [4]. The implementation of the adaptation modeling languages and all associated tools are part of the DiVA Studio open-source project which can be found at http://www.ict-diva.eu/.

3.4 Exhaustive Simulation

The adaptation model presented in the previous section can be used to automatically select the best configuration for any given context instance, i.e. set of values for the context variable presented in Figure 6. While the adaptation model is usually processed at runtime to select the best configurations for low resources systems the idea is to create an exhaustive simulation of the adaptation model. Exhaustive simulation is usually not an option for large adaptive systems but is made possible by the scale of the systems considered in this work. While typical adaptive systems can have billions of configurations to chose from, the number of configurations for a small embedded system like a sensor remains in the order of tens, hundreds and maybe thousands which can easily be managed by simulation tools.

To implement the approach proposed in this paper we have extended the DiVA Studio tools which implement the adaptation modeling languages with features to compute, visualize and export exhaustive simulations of adaptation models. The tool starts by enumerating all possible instances of the context variables and all the way they can be combined. This creates an exhaustive set of contexts. An adaptation simulator then computes the best configuration (i.e. the best set of variants) for each particular context. Once this is done on all contexts, the tool aggregates the result into an automaton for which each state is a particular configuration and the transitions correspond to the changes of a context variable [1].

Figure 9 presents the result of the exhaustive simulation for the temperature sensor. The label of each state corresponds to the configuration in terms of the set of variants defined in the adaptation model (Figure 7). The number in each state corresponds to the number of context instance which yield this particular configuration. The transitions are labeled with the name of a context variable. For the boolean context variable, the direction of the transitions correspond to the variable becoming true(the opposite transition has been omitted to improve the readability of the diagram).

In practice, the simulation tool is not only used for the compilation of the adaptation model (step 1 in Figure 2) but can and should also be used by the designer of the system in order to incrementally develop, correct and validate the adaptation model. To assist in these tasks the tool allows defining and checking invariant properties and can export various graphical visualizations of the simulation. As an example, the colors on Figure 7 correspond to the values of the context variable *Alarm*. The green configuration correspond to configuration which can only run if the variable is true. On the opposite red configuration can only run when the variable is false. The gray configuration can run both in some contexts for which the variable is true or false. Such visualizations are useful in order to assist the developer in the definition and validation of the adaptation models.

3.5 Weaving Aspects into State Machines

Once the developer is satisfied with a particular adaptation model, the second step in the compilation chain presented in Figure 2 is to weave the various combination of aspect corresponding to the configuration created by the simula-

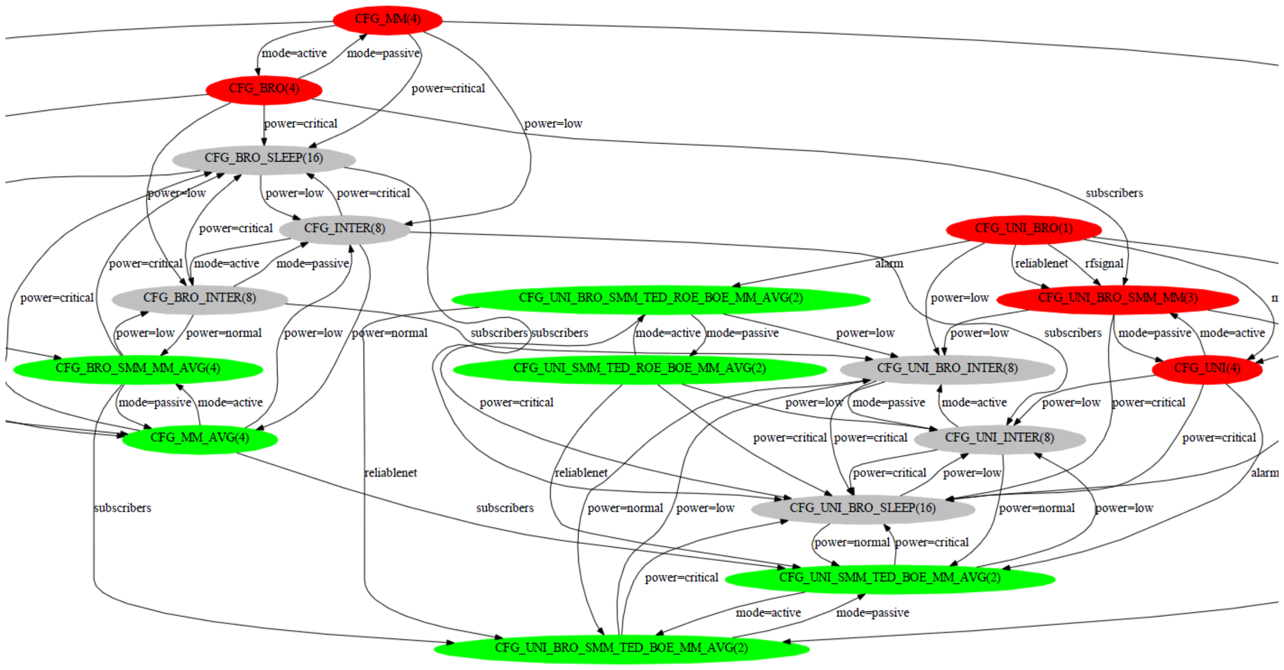


Figure 9: Distribution of values for the Alarm context variable

tion. This step yields the behavioral model of each valid configuration of the system.

As an example, Figure 10 represents the aspect which implements the *Intermittent Operation* variant. This aspect adds a *sleep* state where all the power consuming features of the device related to communication are turned off. The device is then woken up after a fixed delay. The pointcut model (depicted in dotted line) matches all the outgoing transition of the Idle state, which target any (?) state. The advice model and the composition protocol perform several actions:

1. a Sleep state is introduced. This state shut downs all the communication capabilities of the sensor (to save battery) and will exit on timeout.

2. a transition between Idle and Sleep is introduced. This transition actually starts the timer responsible for the timeout of the Sleep state.

3. the transition defined in the pointcut (between Idle and (?)) is moved, so that it will be between Sleep and (?) after weaving. Moving the transitions allows reusing all the actions associated with the transitions with no need to explicitly handle them.

Note that the Sleep state and its incoming transitions are defined as global. This means that in the case the state (?) matches several join points, the same Sleep state (and incoming transition) would be reused for all the join points.

Figure 11 illustrates the *Compute min/Max* aspect. The pointcut matches any assignment statement with a variable

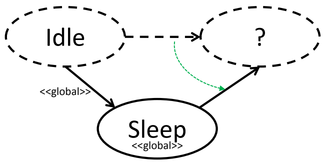


Figure 10: Aspect for the Intermittent Operation

T of type Temperature as left-hand side, and with any right-hand side (the way T is computed is not relevant for the aspect). Whenever the temperature T is updated, the aspect (advice + composition protocol) inserts the actions needed to compute the Min and Max values just after. Note that we use a Java-like ternary assignment for conciseness in the Figure, but min and max are actually computed with if-then-else constructs. Also note that we manually coded the *insert after* primitive for the specialization of SmartAdapters for state machines. It allows designer to directly use this primitive to add statements after another statement, whatever the container of the former statement is: block, state or transition. The aspect also update the Idle state with two additional transitions allowing to get the min and the max values.

3.6 Composition of the configuration models

After the exhaustive simulation and the weaving of all reachable configurations, the key step of our compilation chain is the composition and minimization of all the configuration models into a single model. This corresponds to step 3 in Figure 2. The goal is to create a composed model implementing both the adaptation and the behavior of each

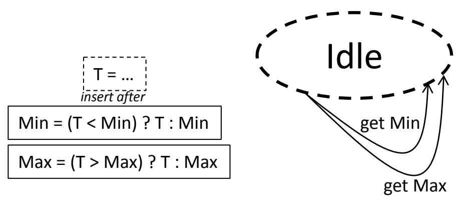


Figure 11: Aspect for the Compute min/Max variant

configuration, i.e. a composite state machine which combines the adaptation automaton depicted in Figure 9 on the behavior of all configurations. We have identified two different strategies for composing the models which yields two different models which are semantically equivalent but structurally different.

3.6.1 *One composite state per configuration*

This first strategy is the easiest to implement. It consist of using the adaptation automaton as a starting point and replacing each configuration by a composite state which contains the state machine which implements its expected behavior. The set of device and messages supporting this global state machine is the union of all devices and messages present in the configuration models.

The main benefit of this composition strategy is its simplicity and the readability of the models it produces. The model allow browsing throw the configurations and examining individually the behavior of each of them. The clear separation between the adaptation logic (top-level composite states and transitions) and the logic of the sensors (inside each top-level composite state) makes it possible to visualize, simulate, debug, test, etc each mode separately. The drawback is that since all the configurations are based on the same base models an only differ by a few aspects, their models are usually quite similar. This means that the resulting composite state machine contains many occurrences of the same pieces of behavior in the composites corresponding to different configurations. Thus generating code from this state machine would produce important code duplications which is not acceptable when targeting target platforms with very limited resources.

3.6.2 *Generating the Optimized State Machine*

To produce efficient code we need to create a state machine having the same semantics as the one produced by the previous strategy but avoiding the duplication of model elements it creates. The idea for that is to weave all the aspects into the base model to obtain a merged state machine that mixes the adaptation logic with the logic of the sensors. Figure 12 presents an example for such a composed model. The elements of the models are augmented with guards from the adaptation models which allow enabling the elements only if they should be in the current context.

The strategy for creating the adaptation guards is not completely trivial since the aspects we use are not limited to the addition of new elements into the base model. We also allow

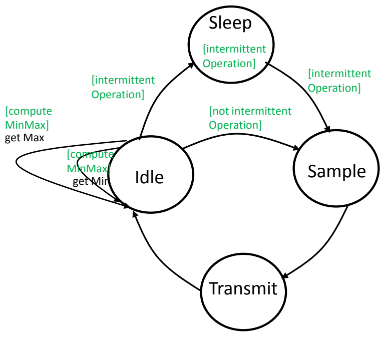


Figure 12: Flat Optimized State Machine (Excerpt)

aspect models to remove or update elements so that we can describe expressive deltas around the base model. We now describe how we handle the addition, removal and update of model elements into the optimized state machine:

- **Addition**: All the elements (states, transitions) added by the aspect are guarded with the following condition: I & C, where I is the initial condition of the element as defined in the advice and C is the condition describing all the contexts where the aspect can be woven. C is basically a conjunction of "var_i==value_i" conditions.
- **Removal**: All the elements that should be removed from the nominal mode are actually kept in the merged state machine. These elements are however guarded with the following condition: I & not(C). This means that we simply ignore these elements instead of removing them. This way, these elements are still present in all the other contexts.
- **Update**: All the updated elements are guarded with the following condition: (I & not(C)) || (N & C), where I is the condition defined in the nominal mode, N is the new condition defined in the updated element, and C is the context where the aspect is active. We apply the same technique for the actions associated with these elements.

The guards associated with the elements of this state machine are simplified using classic boolean algebra rules (and further optimized by the compiler after the code generation step). This strategy thus allows generating compact code implementing both the functional behavior and the adaptation rules which allow switching between configurations.

3.6.3 *Safely switching configurations*

Like in any adaptive system the switch from one configuration to another is a critical moment. It is important that the system is able to migrate from one valid running configuration to another without going through invalid states. In the case of component based systems it means shutting down properly the components to remove and instantiating and starting the components to be added. In the case of state machines it means switching from one configuration to the other only when the active part of the state machine remains stable. In future work we would like to investigate exactly the extent of the active part of the state machine which needs to be untouched to allow for a particular reconfiguration.

For the moment we are using a more restrictive policy: in the current implementation of the approach the reconfigurations are only allowed to be triggered in specific states of the base model which are specified by the designer. In the temperature sensor case the Idle state is the only state which allows for reconfiguration. This mean that whenever something changes in the environment, the actual reconfiguration of the sensor will occur the next time it reaches its Idle. For sensor and micro-controller applications this is an acceptable restriction since most of these systems usually have a state which is both an initial state and periodically visited. In practice this restriction is implemented by only changing the context variable used by the state machine when it is in the Idle state (or one of the Idle states in the case of the "one composite state per configuration" strategy).

4. RELATED WORK

This section presents a brief overview of related works.

4.1 Complex Adaptive Systems

DiVA is an European project that provides a new tool-supported methodology with an integrated framework for managing dynamic variability in adaptive systems. It leverages Model-Driven Engineering (MDE) and Aspect-Oriented Modeling (AOM) techniques both at design-time and runtime to tame the complexity of large adaptive systems [9, 7]. DiVA considers adaptive systems as Dynamic-Software Product-Lines (DSPL) [5]. At design-time, the variability, the environment, the context and the adaptation logic (using simple rules and high-level goals) are captured by different domain-specific modeling languages, and can be partially simulated and validated [4]. An exhaustive simulation would be too time and resource consuming and would generate a huge amount of data that would be difficult to efficiently leverage at runtime. Each feature of the variability model is then refined into an aspect model (a fragment of architecture). At runtime, the aspect models corresponding to the features adapted to the current contexts (according to a reasoner component) are woven together to obtain a global architecture. This global architecture model is then validated and compared to the current architecture. This comparison produces a safe reconfiguration script that is applied to the running system to actually adapt it.

In this paper, we rely on the DiVA metamodels to design our adaptive embedded systems. Instead of refining features into architectural aspects, we refine them into behavioral aspects (fragment of state machines), which provide a finer-grain support for the dynamic adaptation of small embedded systems. Moreover, since the systems we target are small embedded systems, we perform an exhaustive simulation/validation at design-time, which generates two equivalent state machines: one composite state machines that clearly separates the adaptation logic from the logic of the sensors, and one merged state machine to produce efficient C code. At runtime, all the adaptive logic is thus hard-coded with no need for reasoning.

4.2 Aspect-Oriented Modeling and Weaving of State Machines

In [12], Nejati *et al.* propose a heuristic-based approach to match and merge hierarchical statecharts. The matching leverages static matching (typography, linguistic and depth) and behavioral matching based on a quantitative bisimilarity (and not just qualitative). The merging step takes the mappings produced by the matching step as input. If first performs a sanity check on these mappings and then compose the two statecharts to obtain a merged model containing the two behaviors. In [6], Klein *et al.* propose an approach implemented in Kermeta [11] to weave aspects models into scenarios. Both the matching and the weaving consider the semantic of scenarios to achieve a consistent and meaningful weaving.

The works of Nejati *et al.* and Klein *et al.* (among others) highlight the need for considering the semantic of behavioral models when weaving aspects. While the EDAP specialization of SmartAdapters (fully generated with no hand-written customizations) proves to be sufficiently expressive to realize our case studies, it could definitely benefits from these approaches in a close future (when we will realize more complex case studies). Another weaver dedicated to state machine that our approach could benefit from is the Motorola WEAVR [3].

4.3 Adaptive Sensor Networks

In [2], Bourdenas and Sloman present the Starfish framework for specifying and dynamically managing policies in sensor nodes. They particularly emphasize the use of policies for the self-healing of sensor networks *i.e*, policies to define strategies for dealing with sensor errors, component failures and the appropriate reconfigurations. The policy management system is based on Finger2, a policy enforcement system that simplifies the development of the adaptive behavior of sensors. Finger2 is a lightweight version of Ponder2[4] based on TinyOS, dedicated to sensor nodes that cannot run JVM due to resource constraints. Finger2/Ponder2 provides two different types of rules: authorizations and obligations. Authorizations permit or deny actions based upon the action, the source of the action and the target of the action. Obligations must perform certain actions when certain events occur, similarly to Event-Condition-Action rules. The Starfish framework provide functionalities that are commonly used in wireless sensor networks: sensor sampling, schedulers, buffering, networking, etc, which can be leveraged by the adaptation rules (*e.g.* to change the size of buffers is a node is overloaded, to change networking routes in case of a failure, etc). The rules are interpreted (and not hard-coded into the sensors), which makes it possible to dynamically update the set of rules driving the adaptation logic of a node. Starfish also provides a graphical development environment to specify policies, missions (set of policies), roles (subjects or targets of authorization rules) and configurations (the initial code to be loaded in the sensor, including all the possible roles that this node might play in the future).

In our approach, we use different types of formalisms to describe the adaptation logic of sensors: rules (available and

[4]http://www.ponder2.net/

require) and goals. As noticed in [4], it rapidly becomes difficult to specify and understand an adaptive systems using low-level rules only (because rules can have overlapping contexts, interact with each others, etc). The combination of rules and goals helps in mastering this complexity. Another difference is the way we handle the reconfiguration. In Starfish, the "action" part of a rule is an imperative script that directly affects the behavior of the sensor. In our approach, we use behavioral aspect models that mixes declarative (pointcut and advice models) and imperative (composition protocol) styles. This reduces the manual and error-prone inputs from the designer. Moreover, aspect models are woven at the model level, independently from the running system. This avoids "the complexity, danger and irreversibility of reality" [14]. Our approach indeed allows designer to simulate both the adaptive logic [4] and the logic of the sensor *e.g.*, by executing the model. It is either possible to simulate the behavior of each mode independently (using the composite state machine), or the behavior of the merged state machine that is the basis for generated the code of the sensor node.

5. CONCLUSION AND FUTURE WORK

This paper presented a Model-Driven approach to engineer adaptive firmwares. System with restricted resources, like micro controller-based systems, are a challenging target because of the very low amount of CPU and memory they offer. Software engineering techniques based on the use of advanced operating systems, frameworks, middlewares or high level programming languages are most of the time not applicable in this context. On the other hand the relative limited size of the systems can be exploited to use techniques which would not scale to other types of systems.

In particular, we propose to run exhaustive simulations of the adaptation at design time and generate efficient code based on the simulation results. While this is not realistic for complex adaptive systems with millions or billions possible configurations, this makes sense for small adaptive systems with less than a few thousands configurations. The resulting approach is based on using state of the art techniques to model adaptation policies and generate micro-controller code from state machines. From the result of this exhaustive simulation, we automatically generate (using behavioral aspect weaving) two semantically equivalent state machines:
i) a high-level state machine which clearly highlights the different modes of the system and separates the adaptation logic from the business logic. This representation is well suited for documentation, and makes it possible to validate and simulate each mode independently.
ii) a low-level state machine which merges the adaptation logic into the business logic. This representation is well suited for code generation, since it avoids duplication of state and transitions.

The approach is implemented in open-source tools and the case study of an adaptive temperature sensor has been implemented. One issue discussed in Section 3.6.3 is that we currently only allow the system to reconfigure in states that have been specified by the designer (typically the Idle state). We plan to provide support to be able to automatically detect quiescent states [16], where the system can safely be reconfigured.

6. REFERENCES

[1] Nelly Bencomo. *Supporting the Modelling and Generation of Reflective Middleware Families and Applications using Dynamic Variability.* PhD thesis, Lancaster University, 2008.

[2] T. Bourdenas and M. Sloman. Starfish: policy driven self-management in wireless sensor networks. In *SEAMS'10: ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 75–83, Cape Town, South Africa, 2010. ACM.

[3] T. Cottenier, A. van den Berg, and T. Elrad. The Motoroal WEAVR: Model Weaving in a Large Industrial Context. In *AOSD'07: 6th International Conference on Aspect-Oriented Software Development - Industry Track*, Vancouver, Canada, 2007.

[4] F. Fleurey and A. Solberg. A Domain Specific Modeling Language supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In *MODELS'09: ACM/IEEE 12th International Conference on Model-Driven Engineering Languages and Systems*, Denver, Colorado, USA, oct 2009.

[5] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *IEEE Computer*, 41(4), April 2008.

[6] J. Klein, L. Hélouet, and J-M. Jézéquel. Semantic-based Weaving of Scenarios. In *AOSD'06: 5th International Conference on Aspect-Oriented Software Development*, Bonn, Germany, 2006. ACM.

[7] 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.

[8] B. Morin, J. Klein, J. Kienzle, and J-M. Jézéquel. Flexible Model Element Introduction Policies for Aspect-Oriented Modeling. In *MODELS'10: ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems*, Oslo, Norway, October 2010.

[9] Brice Morin. *Leveraging Models from Design-time to Runtime to Support Dynamic Variability.* PhD thesis, Université Rennes 1, 09 2010.

[10] Brice Morin, Olivier Barais, Grégory Nain, and Jean-Marc Jézéquel. Taming Dynamically Adaptive Systems with Models and Aspects. In *31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, May 2009.

[11] P.A. Muller, F. Fleurey, and J. M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *MoDELS'05: 8th Int. Conf. on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, Oct 2005. Springer.

[12] S. Nejati, M. Sabetzadeh, M. Chechik, S.M. Easterbrook, and P. Zave. Matching and Merging of Statecharts Specifications. In *ICSE'07: 29th International Conference on Software Engineering*, pages 54–64, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[13] R. Ramos, O. Barais, and J. M. Jézéquel. Matching Model Snippets. In *MoDELS'07: 10th Int. Conf. on Model Driven Engineering Languages and Systems*, page 15, Nashville USA, October 2007.

[14] J. Rothenberg, L. E. Widman, K. A. Loparo, and N. R. Nielsen. The Nature of Modeling. In *in Artificial Intelligence, Simulation and Modeling*, pages 75–92. John Wiley & Sons, 1989.

[15] J. Zhang and B.H.C. Cheng. Using temporal logic to specify adaptive program semantics. volume 79, pages 1361–1369, 2006.

[16] Ji Zhang and Betty H. C. Cheng. Model-based Development of Dynamically Adaptive Software. In *ICSE'06: 28th International Conference on Software Engineering*, pages 371–380, Shanghai, China, 2006. ACM Press.