Heterogeneous and Distributed Services: Domain-Specific Language testing

Alexandre Carteron

Research Master

SINTEF

Internship report

Heterogeneous and Distributed Services Domain-Specific Language testing

Author:
Alexandre Carteron

Supervisor:
Brice Morin
Model-Driven Engineering group

SINTEF

Abstract

The pervasiveness of embedded systems offers new opportunities to build added-value services linked to the physical world. These services rely on Highly heterogeneous components communicating within a Distributed environment. Henceforth we will refer to these services as "H&D services". Engineering H&D services is challenging as programmers need not only to describe their business logic, but also manage the diversity of the infrastructure and their programming environments. The Model-Driven Engineering paradigm promotes the modeling of services independently of the underlying platform and rely on code generation facilities. However, there exists no standard method to ensure that the abstract behaviour is consistently transposed to each of the possible concrete targets, and that the quality of service of the generated programs remains reasonable. In this Master Thesis, we describe our approach to validate both the behaviour and the efficiency of programs generated from an abstract description. User-defined tests allow checking of consistency and correctness on the different targets of the behaviour of the abstract concepts we use, while simulations of complex systems give the required performance information.

Contents

Chapter 1

Introduction

1.1 Context

New platforms and devices, from smartphones to web servers, offer more and more diverse capabilities. Depending on the field of application, challenges of different natures can appear. In some domains, such as television and voice over IP, user satisfaction depends on the reactivity of the system. Criteria such as latency require to be precisely managed. In such case, testing nun-functional properties such as performance is thus necessary. Likewise, autonomous systems such as home automation control require as few maintenance operations from the user as possible.

Testing such non-functional properties is time-consuming without dedicated tools. Indeed, each device comes with a specific programing environment designed to allow full exploitation of its capabilities. Web servers often rely on high level languages such as Java, while sensors require the treatment of low level concerns (for instance memory allocation) with languages such as C. The devices may also access data coming from the network, be it a sensor or a server. We will refer to these Heterogeneous and Distributed systems as "H&D systems".

To bring solutions to H&D concerns, such as the heterogeneity of the programing environments and communications protocols, our work rely on Model-Driven Engineering (MDE). MDE [22] is an approach which relies on describing systems through the use of models on the right level of abstraction. The goal is to ease the understanding of the systems not only during the conception, but also during the testing. This approach can thus bring answers to the concerns of functional correctness and performance of systems. For instance, MDE provides Domain-Specific Languages (DSLs) [25] and tools to combine them. DSLs, as opposed to General Programing Languages (GPLs), are designed to describe a program at the adequate level of abstraction. This allows to describe the required analysis at a high level of abstractions rather than redeveloping specific solutions for each type of device.

DSLs for H&D services rely heavily on code generation techniques. This allows to use only one language for devices as varied as sensors, smartphones and web servers, but there is a need of checking the functional behavior and the performance of the generated code. While functional testing and performance measurement have different scope, we treated them in a similar fashion. Indeed, both of these concerns are usually solved by platform-

specific solutions, which would require modifications in each compiler. We chose to solve both problems by integrating these operations in the DSL itself rather than in the different compilers. This work presents the details of our solution to automatically ensure the checking of correctness and performance of a DSL for H&D services.

Chapter 2

State of the art

This section introduces the context and the scientific background of our work. First, we introduce the Heterogeneous and Distributed (H&D) systems and their associated challenges, then present how Model-Driven Engineering (MDE) allow to tackle these challenges. This state of the art then describes Domain-Specific Languages (DSLs), one of the MDE solutions, and explains how a state-machine based DSL helps to answer some of the H&D systems concerns. Finally, we present ThingML, the DSL used during this Master Thesis.

2.1 Heterogeneous and distributed systems

H&D systems [19] are used for applications as diverse as tele-medicine, embedded systems in cars, or voice over IP. They combine heterogeneous devices performing distributed computing:

- Heterogeneity: H&D systems are composed of a variety of devices: sensors, user terminals such as smartphones, and clusters of servers (clouds). Each device has more or less limited resources, be it memory on disk, RAM or processor performance.
- Distributed computing: desktop software systems usually rely on a single computation core. On the contrary, H&D systems rather rely on a collaboration between several physical components of the system. The devices need to communicate in order to achieve a common goal, but communication protocols may vary from one device to another.

This variety of devices brings new challenges not already addressed by desktop software engineering. Recent studies, such as the one from Gokhale *et al.* [13], allow to make a state of the art of today's challenges in the H&D systems domain. We will here briefly introduce the H&D specific challenges:

- Heterogeneity: H&D systems are composed of constrained components such as sensors and larger components such as servers. Several problems emerge from this situation. First, each component has a different dedicated programing environment. For instance, Java is often used in powerful nodes, but it is not possible to use it on more constrained

components. Development of a service thus require to master these different programing environments. Next, each programing environment is susceptible to have its own message passing protocol [17]. The naive solution, building adapters for each protocols on each environment, is a tedious job.

- System adaptation: Operating conditions can change at runtime, for example with the failure of one of the components or with the upgrade of the physical architecture. The system should adapt while maintaining an acceptable quality of service and with minimal impact on the execution.

- Deployment: similarly to the adaptation to physical evolutions of the system, a software upgrade should be sent automatically to each component, rather than manually. An automatic diffusion from a unique source can reduce both the workload of the support team and the downtime of the system.

- Quality of service: This designate how the system is perceived by the user. Some metrics, such as the answer time or the uptime, can be used to measure the quality of service [30]. H&D systems can be deployed on several networks, interact with other systems, and are concurrent. It is thus hard to predict that there will not be deadlocks or to compute the maximum response time. The systems should be conceived so these predictions can be done.

2.2 Model-Driven Engineering

MDE provide tools to ease design, programing and deployment of solutions [22]. It is usually applied to develop desktop software systems, but it can also help to create H&D services.

A broad definition of modeling would be the following [20]: *"the cost-effective use of something in place of something else for some cognitive purpose"*. As we have seen in the previous section, designing H&D systems come with challenges such as managing heterogeneity and complexity. Tools to manage these challenges are commonly used in MDE to abstract the heterogeneity and to generate implementation details automatically, and to provide efficient programing environments.

More precisely, useful MDE techniques to answer H&D systems specific problems include:

- Abstraction: By abstracting the message passing protocols, the challenge of their heterogeneity is much easier to solve. For instance, the "smart messages" system [4] describe how communications content can be separated from implementation and routing concerns.

- Configuration: When two devices have similar functions but different underlying physical architectures, we want to maximize the reuse of code for the common behavior. Magee *et al.* [18] propose to separate the system into logical nodes, to be able to dynamically adapt them. Software product lines are another MDE solutions available to answer this heterogeneity challenge. This refers to methods, tools and techniques for

creating a collection of similar softwares from shared software components. For example, aspect-weaving [8] allows to describe variable parts of a software product line by aspects and to weave them automatically to a common base. Finally, code generation techniques provide an other approach: by describing the behavior of a program in an abstract language, it is possible to automatically generate the platform specific code for each targeted device.

- Adaptation: Several studies, such as the one of Bourdenas *et al.* [5] or Garlan *et al.* [11], describe how a well designed system can automatically adapt to physical modifications of the system.

- Deployment management: This designates the way each piece of software is installed and initialized on each device. Several works have been conducted to optimize this task. For instance, Fouquet *et al.* [10] show how the deployment management can be computed in a "Just-in-Time" way among the different nodes to lighten the central server workload. The distribution of the deployment strategy is combined with a computation method to reduce the workload of the smaller nodes. This method is based on model adaptation: the models describing the system before and after an update are compared, and a "delta-model", representing the differences between the two models is created. This delta-model is used to compute the reconfiguration strategy of the H&D system.

- Quality of service: Several works exist to measure quality of service for H&D systems. For instance, Verhoef *et al.* [26] propose a solution to validate and do some performance estimations on distributed systems by using a dedicated modeling framework. Some works [6] even try to perform this QoS measures at runtime to choose the best path in a routing context, using weighted graph models.

During this Master Thesis, we focused on answering the problematics of abstraction and configuration through the use of Domain-Specific Languages, presented in the following section.

2.3 Domain-Specific Languages

Popular languages such as C or Java are General-Purpose Languages (GPLs). They can be used in a variety of domains, though the concepts offered by these languages can be unpractical for very specific domains. They often rely on external libraries or modules to bring developers more adapted concepts, but the core of the language is unchanged. For example, one can use network in C through the use of libraries such as Boost.Asio or ACE. However, such flexibility comes at a price: the necessary code to perform a specific task can be long and may require the developer to focus on unnecessary implementation details.

On the contrary, Domain-Specific Languages (DSLs) [25] are specialized to a particular application domain. While they are unusable outside of their domain, they allow to perform a specific task faster and more simply.

This is mainly explained by their simplified syntax and adapted abstraction level [16]:

- Syntax: GPLs contain an important amount of concepts, which have to be made distinct. On the contrary, DSLs only contain the concepts necessary to perform a specific task. This reduces the number of the required codes to describe the concepts present in the language.
- Abstraction level: DSLs can contain concepts not usually present in GPLs. This allows to describe more efficiently some specific behaviors. For example, Java or C do not define explicitly event-driven state-machines. While this is not required for most applications, event-driven state-machines are an efficient way to express reactive distributed programs, as we will see in the next section.

2.4 Event-driven state machines

In most popular languages, data and behavior are represented through two distinct class of concepts [1]:

Paradigm	Data	Behavior
Imperative (C)	Structures	Functions and instructions
Object-oriented (Java)	Classes	Methods and instructions

This is usually enough to represent most applications logics, but these paradigms do not allow to represent parallelism and communications easily.

Even-driven state machines [12] are a way of describing both communication and parallelism. They are especially adapted to describe reactive and distributed systems.

- Parallelism is expressed through the use of several state machines, each one representing a parallel region.

Figure 1: Parallel regions description with state machines.

- Communication can be expressed through messages sent and received when going from one state to another. The transitions which are supposed to receive a message are not performed until the message is sent by an other state machine.

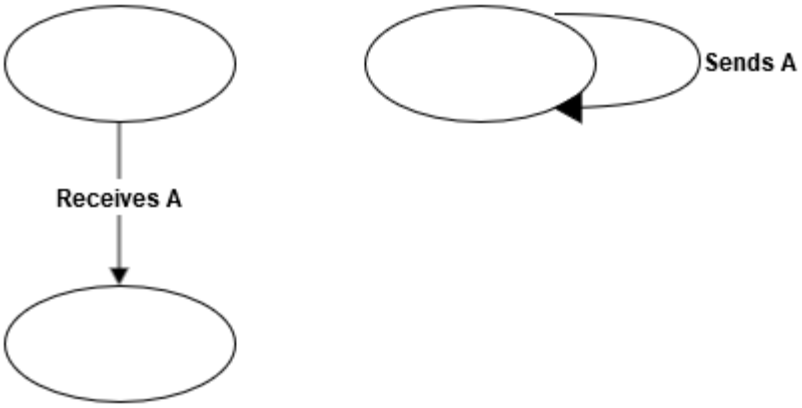

Figure 2: Message passing with state machines.

The notion of composite states allows the user to describe the hierarchy of operations. In the following example, we can see how inputs/outputs concerns and treatment were separated:

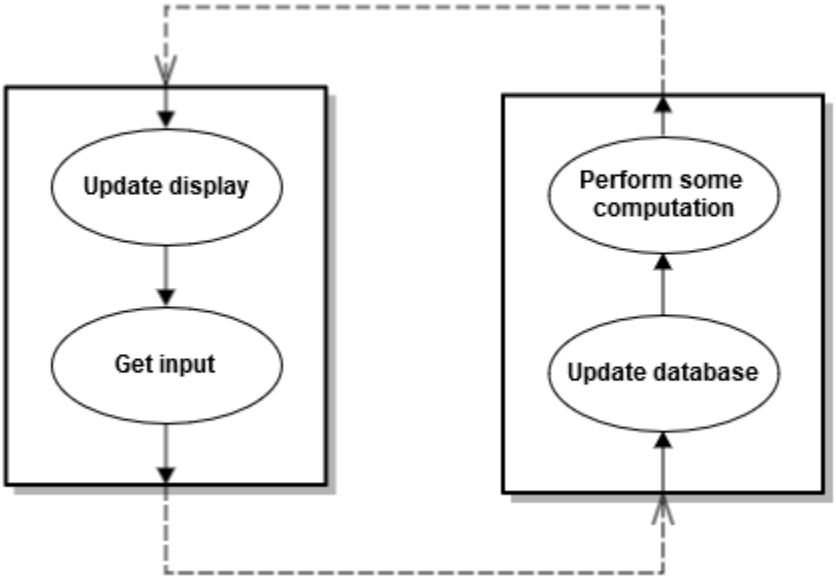

Figure 3: Example of a state machine using composite states.

It is possible to simulate event-driven state machines in GPLs, as described in annex A. The code given in the annex describes the following state machine:

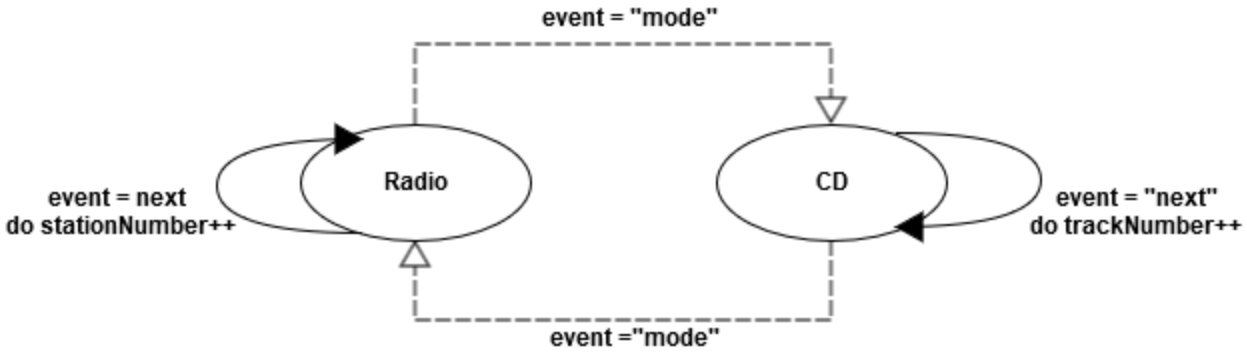

Figure 4: Example of a state machine describing a basic car radio system.

However, GPLs do not define these state machines explicitly. The corresponding GPL code is long and hard to read and maintain, as it relies, for this example in C, in a hierarchy of switch instructions. On the contrary, a DSL dedicated to those state-machines would describe the same behavior with the following code:

```
State Radio, CD
Radio -> CD when event == mode
CD -> Radio when event == mode
Radio -> Radio when event == next do stationNumber++
CD -> CD when event == next do trackNumber++
```

Event-driven state machines have however some drawbacks: simple instruction flow structures such as "while" are harder to represent.

For an application getting inputs until the input is the character "q", the code in a GPL would look like:

```
input = ""
while(input != "q"){
        input = readline()
}
```

A state machine describing the same behavior would look like:

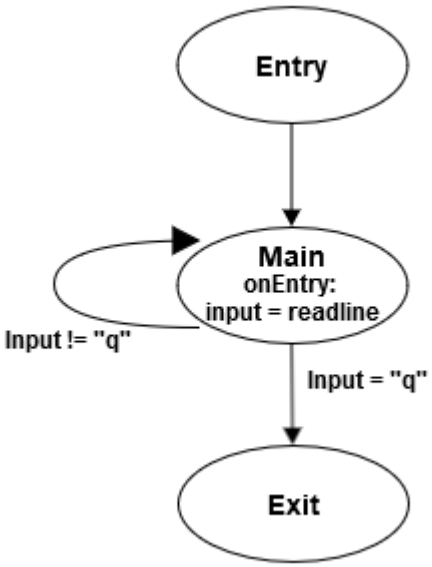

Figure 5: While loop description with a state machine.

The DSL-like code necessary to describe it would be:

```
State entry, main, exit
String input
entry -> main
main -> loop when input != "q"
main -> exit when input == "q"
main.onEntry.code = {input = readline()}
```

For simple behaviors like this one, state machines are not adapted.

Event-driven state machines help to solve some problems inherent to H&D systems, but lack some of the features of more classical languages. As we will see in the next section, the DSL we worked with uses both state machines to describe the reactive and distributed part of the system and an action language to describe more common behaviors.

2.5 Case study: ThingML

ThingML [24] is a DSL dedicated to small devices ranging from sensors to smartphones and small servers. Its name refers to the "internet of things" [2]. It is the idea that every object in our environment is susceptible to be connected to a network, even passively through the use of RFID. This means that we need means to describe systems composed of multiple objects, sensors and more classical computing nodes.

ThingML approach rely on abstracting each component of the system as a "thing" and is based on two components [9]:

- a state machine based language to describe both the possible internal states of a thing and the communications between things.
- an action language, similar to imperative languages such as C, to describe simple flow control structures and instructions, ie the behavior of the "thing".

Once the behavior is described, the DSL code is used to automatically generate platform-specific code adapted to the chosen device. For example, it is possible to generate Java or POSIX C for Linux servers, or a platform-specific C for Arduino. However, this versatility comes at a price: each targeted language requires its own compiler. Each compiler has to be hand-written, bringing a potential for undetected errors. The generated code can also have platform-specific performance issues, which have to be detected before they can be corrected.

The detection of problems has to be integrated in the management tools already present in this project. This project is hosted on a Git forge and uses Maven to manage dependencies. This provides the following advantages:

- Collaboration: each user is able to work on a part of the project with a local copy, and synchronize his work on the web server once he is satisfied. This synchronization is done through tools detecting conflicts in edited files and displaying differences between versions of a same file.
- History of modifications: should an update break important functionalities, it is always possible to come back to an earlier version of a file or even the project.
- Dependency management: a complex project such as ThingML contains compilers for several languages, each one of these relying on many dependencies. Maven is able to automatically download and configure such dependencies, be it a dependency to an external tool or a subproject. For instance, this internship relied on several subprojects linked one between the other and to other subprojects of ThingML

This Master Thesis, by providing an automated testing approach will bring agility to the developement of ThingML.

The goal is to be able to have a real-time feedback each time a user makes an update, as it will be described in the following sections.

2.6 Addressed research problems

This Master Thesis addresses two categories of problems: functional testing and performance measurement in the context of a DSL for H&D services. While their scope is different, functional testing and performance measurement are similar in the way that they are difficult to perform due to the variety of compilers. Our approach rely each time on integrating these operations in the DSL itself rather than in the different compilers. This allows us to adapt much more quickly any new compiler to our testing environment.

Functional testing ensures that the implementation conforms to the specifications, and regroups several large categories of methods. Manual methods can be classified in a first group. It ranges

from user feedback to beta testing [15], but since each feature has to be tested by hand, it is time-consuming and has to be redone after each update of the language. On the other hand, mathematical approaches described by formal methods [7] give strong certainties on the properties of a program. However, these methods rely on restrictive hypotheses which are unpractical when creating a DSL for H&D services. Methods such as unit testing are an intermediary that rely on writing tests manually, to be then executed automatically after each update. In this context, unit testing appears to be a reasonable trade-off.

However, unit testing [21] is generally supported by platform specific frameworks such as Junit for Java, while DSLs rely on code generation using a different compiler for each target. It would require significant important efforts to embed such frameworks inside each compiler, this is why existing unit testing framework are not adapted to DSLs. This document presents an approach based on execution trace comparison to automatically perform functional tests.

Performance measurement [27] is used to understand the execution time, memory use and even energy consumption of programs or some of their components. It can also help to detect if an implementation is slower than expected and brings helpful data to identify the origin of the problem. Performance measurement framework are again platform specific, however it is possible to collect a significant amount of informations in a platform-independent way, and later use the frameworks to give detailed informations on the execution of a specific implementation. This can be used to give users guidelines on the most adapted compiler for a given platform, to understand what makes a program fast or slow depending on its characteristics and, whenever a compiler is generating inefficient code, to diagnose the cause of the problem.

Moreover, writing benchmarks is time consuming. To allow assessment of performances of programs of variable complexities and size, we worked on generating state machines simulating real-life programs. These generated programs respects several sets of constraints and parameters to be as close as possible of real-life programs. This document presents our approach combining generation of state machines and platform-independent measures of performances.

Chapter 3

Functional testing

In this section, we first explain what is functional testing and its role during the development of a programing language. We present several existing methods to perform this functional testing and we show their limits in the scope of the development of a DSL such as ThingML. We then explain how testing based on execution traces would anwser some of these concerns. We describe the work we performed to prove the validity of this approach and the results we obtained, as well as the informations it was able to provide for the development team.

3.1 Scope of functional testing

A programing language is expected to provide a certain number of features, depending on the scope of this language. We call correctness the fact that a feature does not contain unwanted behavior, and it is an especially important property.

Functional testing is a general term to describe techniques used to ensure that a program behaves as intended. Used on simple use cases, it allows to check correctness of features of a language.

Once a bug is detected, defect tracking tools allow the development team to keep a record of all detected mistakes and whether they were corrected or not. It is also possible to create unit tests to make sure further development of the language do not break any existing feature.

3.2 Existing solutions

To perform functional testing, development teams usually rely on a combination of manual and automated solutions.

For example, beta testing [15] is an important source of information on the correctness of the language, but is time consuming and not sufficient to detect all bugs, since it would require to retest the entire set of functionalities after each update of the language. User feedback is also a major source of information, however bug correction takes time and leave the user unable to use the project in the meantime. The limits of manual solutions calls for more automated ways of testing the correctness of the features of a language.

Formal methods [7] are an intensive research field, and rely on a mathematical approach to give strong certainties on the properties of a program. The general idea is to prove these properties on a set of hypotheses, rather than testing these properties on a finite set of test cases. For instance, VDM++ is a model-based specification language, on which studies have been conducted to model

distributed embedded real-time systems [26]. The aim is to be able to perform analyzes such as communication delay computation [28]. However, these methods rely on restrictive hypotheses. For instance, the specifications of timed VDM++ are about 100 pages long, which is unpractical when creating a DSL for H&D services. Moreover, the size and complexity of H&D services would generate time and resource consuming validation cycles often incompatible with the rapid and agile cycles promoted by the DSLs.

3.3 Unit testing

Unit testing is a popular method of performing functional testing in the industry [21]. For each use case of a program, or in the case of a language, for each feature, a test has to be written. This test executes a sample of code and contains an assertion about the state of the program after running the code sample. If the assertion is not true, a problem occurred during the test and the tested feature contains a bug.

Tests are usually launched automatically each time the project is updated. This allows to check the update did not break any of the existing functionalities. Moreover, since each feature should have its own tests, it is easy to verify that every aspect of the project has been tested.

However, unit testing is not flawless. For example, testing a feature only guaranties it works in the condition of the test, and may miss some problematic use cases. A common solution is to reuse a same test with different inputs and expected outputs to increase the covering of use cases, a solution which was implemented during this work. Moreover, in the case of a DSL compiling to different environments, it is unpractical to integrate a framework in each of the targets and each framework inside the project management tools.

3.4 Improvement of unit testing

This Master Thesis focused on improving integration of unit testing in the context of a DSL compiled to different target. For example, code written using ThingML can be compiled to general purpose languages such as C or Scala. Unit testing frameworks are different for each of these languages, and are difficult to integrate at the same time in the project manager.

3.4.1 Method description

To better understand the scope of this work, we will briefly describe the state of the ThingML project before this Master Thesis: several tests had already been written using the ThingML language, covering a large part of its features. However, these tests had to be run manually and there was no standard way of sending inputs to them.

Tests are now structured in two parts:

- The test itself is described by a state machine or a group of state machines. It uses the feature of the DSL to be tested. The state machines are waiting for inputs to progress among their states, and send outputs at key moment of their execution.
- A feeding state machine sends the required inputs to the test.

The outputs sent by the test are written to a file to form the execution trace of the test. This produces a unified output for every target environment, be it C, Scala, etc.

Using this structure of tests, integration of unit testing can be done in two steps.

- Execution: tests are compiled and executed on each of the target environments.
- Recovery of the result: the produced output files are accessed and execution traces regrouped and compared.

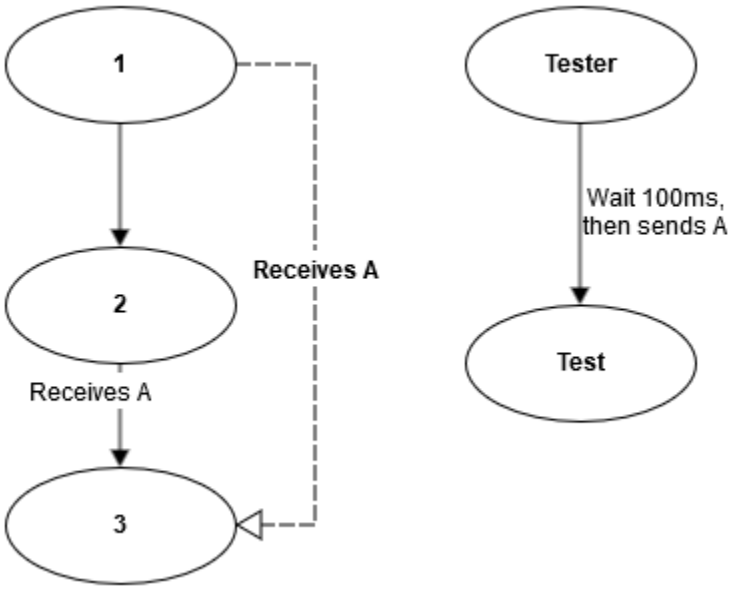

Figure 6: Example of a test, described in annex B.

The figure 6 presents an example of a test used for the ThingML language. It is used to detect if the transitions which receive no message ("empty transitions") are well implemented. This test starts in state 1, and should go directly in state 2, and then in state 3 after 100ms. The expected trace is "123". However, if the empty transitions are not supported, this test will go from state 1 to state 3 after 100ms, the trace will be "13", allowing to detect a mistake in the compiler.

3.4.2 Expected improvements

Traces open opportunities to collect more precise informations about the execution of the test. Indeed, unit testing only collect informations about the state of the program at a given point of the execution, while the traces can give informations about the progress of the program. For example, path of execution can be obtained by generating an output at the entry of each state.

It is also possible to verify consistency between different compilers by comparing traces. This gives a stand back about the bug location:

Tests results	Analysis
Test fails with compiler X	Compiler X is defective
All tests fail with the same trace	Test has been incorrectly written, or the DSL itself is defective
Tests fail with different traces	Non conclusive, but at least one compiler is defective

Finally, we also expect to conserve all the expressiveness of usual unit testing frameworks. Indeed, they rely on asserting some properties on objects. Traces can be used to simulate such a behavior by asserting these properties at a given point in the test code, then giving as an output a certain code for success, and another code for a fail.

3.5 Results

The work we performed on this subject was focused on giving the maximum of informations with the minimal amount of effort from the development team.

To add a new test, the user has to write and send to the server a ThingML state machine implementing a "Test" interface, and use the TestIn and TestOut ports provided by this interface. The user also adds an annotation indicating the inputs and the associated expected outputs, possibly using regular expressions in inputs and outputs.

The file is then automatically compiled to the different targets and executed. The results are then integrated in the project build informations, and a HTML file containing a summary of all tests results is generated. An example of tests results is available in annex D.

The following table show how the method presented previously gave valuable informations to the development team.

Test name	Expected output	C output	Java output	Analysis
After	EA*	EAEAEA	EAEAEA	Error in the test itself, EA* should have been written (EA)*
Arrays	e1827[...]	CompilationError	e1827[...]	C compiler crashes when using arrays
CompositeEvent	12b3c4	12b3a5e12	12b3c4	C compiler does not handle composites properly.

The third example (CompositeEvent) contains a trace which allows to understand what was the flow of execution of the program, and thus the cause of the bug. As you can see in the actual code given in the annex C, it shows that the composite captured an event instead of one of its internal states.

The suite of tests was integrated in such a manner that the tests are run after each update of the project. This allow to ensure the updates do not break existing functionalities, and also proved to be helpful to monitor the project progress:

- The first version of automatic bug tracking revealed that 37 of the 54 tests were failing. This number is high, however this is easily explained by the fact that many tests were not correct, be it the inputs, expected outputs or the code itself. The number of failed tests dropped to 21 after correcting the tests.

- Bug tracking can help to prioritize bug fixing, since sometimes a single error can have an effect on many tests. For instance, after fixing a single ordering bug in the Scala compiler, 4 more tests went successful.
- This suite of tests proved to be useful when adding a new compiler, to determine when it is ready to use. For instance, when the Java compiler was created, only 3 Java tests on 27 were successful. After two weeks, only 2 tests were still failing, showing that the compiler was mostly ready.

3.6 Discussion

Using traces to perform functional testing proved to be a useful tool to monitor the project status and help solving bugs. However, this method has several limitations:

- Traces do not describe the whole state and execution of a program. It is then possible that a bad execution produces a good trace. The error would then remain undetected. The tests should then be as simple as possible, covering only a specific feature, to ensure these false positives do not occur.
- The test framework is compiled to ThingML. An error during the compilation of the framework is then possible. False positives and negatives can potentially appear this way. The test framework has then to remain as simple as possible, using only a minimal set of the functionalities of the language. Moreover, since each compiler is written independently, it is unlikely that the same mistake is reproduced each time, limiting the probability of such false positives and negatives.

Chapter 4

Performance measurement

In this section, we first explain what is the goal of performance measurement and its role during the development of a programing language. We present the existing methods to perform this performance measurement and we show they can be used and improved to fit in a project such as the development of ThingML. We describe the work we performed and the results we obtained, as well as the information it provided for the development team.

4.1 Generated code performance

Performance is the measure of how fast a program is executed. A DSL such as ThingML should ensure that the code generated by each compiler has acceptable performance and should provide ThingML users guidelines on when to use which compiler. In this section, we first describe the different metrics which can be used to assess the performance of a language, then show how they are used in benchmarks. We finally discuss the additional informations which can be extracted from the comparison between different generated code.

4.1.1 Metrics

Performance can be measured in several ways [27]. Some methods are more precise, but have a higher cost at execution: it is a trade-off between precision and fidelity of the measure.

- Execution time: the simplest metric is the execution time of a given program. While this gives a first information about performance, it is limited since it gives no information on the cause of any slowness. This method does not modify the execution time, at the cost of a limited usefulness.
- Time spent in each part of the program: Several frameworks, such as Gperftools and Yourkit, allow to measure the time spent in each part of a program. This is more precise since it helps understanding which part of the program is slow, but may slightly increase the execution time.

To measure performances of C, Java and Scala, we used the following frameworks:

- Gperftools [14]: this C framework interrupts the execution of the program at regular intervals and stores the current executed function. This method is only an approximation since

interruptions may miss the execution of some functions. It is then best used on applications running for a sufficient long time.

- Yourkit [29]: this Java and Scala framework rely on the Java virtual machine (JVM) to extract precise informations on the number of calls of each method and their execution time. Being embedded in the already heavy-load JVM, its overhead is usually not significant (1 to 10 %)

Once the tools to measure the performances of programs are integrated, it is necessary to write performance tests. These tests are named "benchmarks", and are presented in the following section.

4.1.2 Benchmarks

Benchmarks [23] are programs written to compare the execution speed of a given set of operations. They usually rely on executing a given task with slight variations a high number of times. For instance, matrix product computation is a classic benchmark when the goal is to measure the efficiency of basics operations such as addition, product and memory allocation.

This example is interesting since it exhibits one of the necessary properties of benchmarks: a sufficient execution time. Indeed, very short programs tend to describe more the overhead of a given language (almost none for C, long for Java who depends on the initialization of the JVM) than the actual performances of the operations they contain.

However, this problem can not be solved by repeating a high number of times a given operation, since compilers may detect this behavior and perform optimizations that couldn't occur in real life.

The following example:

```
x=0
while(x<1000){
        x=x+1
}
```

could be translated by some compilers in this code:

```
x=1000
```

which would not be the behavior we actually want to measure.

It is then necessary to think about how representative of real-life programs is a given benchmark.

Moreover, it is important to write different benchmarks with varied sets of operations, since execution speed may vary greatly from one language to another for several reasons:

- Implicit behavior: similar codes may implicitly perform some operations in some languages. For instance, Java checks that an array is not accessed out of its bounds, while C leaves this testing task to the developer. These implicit checks, while reducing risks of errors, increase the execution time.

- Implementation: languages running on virtual machines are usually slower than compiled programs. However, virtual machines allows optimization at runtime which balance this slowness, and may even be faster in some circumstances [3].

An open question is how to write a code which is running enough time, possibly by performing repetitive tasks, and which is still representative of real programs.

4.1.3 Heterogeneity

Benchmarks are in general a useful tool during the development of a new language or DSL. They are especially important for ThingML, since it is compiled to different targets such as C and Java. For instance, C targets small Linux components, while Java can be run on any component with sufficient memory and processing power to run a Java Virtual Machine. The goal is to provide faster and memory-light environments for components with limited hardware, while also improving portability through the use of possibly slower environments such as Java.

In order to get a coherent DSL, it is important to detect if a compiler produces code significantly slower than the other compilers. End-users of the DSL may also want to know which language is more efficient for which use case.

Usually, benchmarks are hand-written and a pseudo-code is translated to each target languages. A DSL compiling to several targets such as ThingML opens the possibility of writing one benchmark with the DSL itself, and then comparing generated codes in each languages. A number of informations can be gathered this way, assuming we are testing a specific feature in a given benchmark:

- If the benchmark is slow for all languages, the tested feature is not well defined in the language, not used right or slow by nature.
- If a specific language is significantly slower, the compiler is probably producing inefficient code for the tested feature.

4.2 Automatic generation of benchmarks

We presented the interest of measuring performances and of comparing how a same benchmark behaves with different compilers. However, writing benchmarks is a time-consuming task, and we decided to investigate the possibility of automatically generating those test programs.

4.2.1 Proposed approach

Since ThingML is state-machine based, the generation of a program is similar to the generation of a graph. Moreover, these benchmarks should mimic as close as possible the behavior of a real-life program. However, there exists no metric to judge of the "realism" of a program. We tried to identify key properties of real-life programs, and translate them into rules. We then based the generation on the respect of rules valid at all time, and on the variation of parameters describing more or less complex programs.

4.2.2 Necessary properties of the generated code

Real-life programs are extremely varied, but some rules are valid for all of them.

- No blocking states: since we are doing a benchmark, we expect the program to be active for all the duration of the state. Reaching a blocking state would mean the program is inactive, the collected data would then be unrepresentative.

- No unreachable state: we want to simulate a big program, and unreachable states can be interpreted as dead code, which occurs in real-life programs. However, if there is no reachability check, there is the risk that only a few states are reachable, the generated program would be a simulation much smaller than expected.

The benchmark generator has to ensure these rules are respected for the generated state machines.

4.2.3 Parameterization

To simulate programs of different size and complexity, the generated state-machines can be configured to have variations on several parameters. These parameters are transmitted to the generator and allow to control the complexity and the size of the generated code.

To introduce variability, we decided to model these parameters as intervals to allow the generator to randomly pick values within these intervals.

We chose to test our approach with the following set of parameters:

- Number of parallel regions: Regions describe parallelism inside a program. Some programs can be sequential, while other can be heavily parallel.
- Number of states per regions: complex operations are described by a higher number of states than simpler ones.
- Number of transition per state: complex operations usually have a higher number of possible outputs.
- Depth and proportion of composite states: this describes how complex is the hierarchy of operations.

4.2.4 Generator algorithm

We describe here the algorithm we used to generate the state machines. Note that we use variables such as maxRegion in the pseudo-code for the sake of simplicity. In the actual code, we pick a random value in a range configured beforehand each time, maxRegion can be for instance a value between 2 and 4.

- State machine structure creation: we first create the hierarchy of the states, composite states and regions. This steps includes marking at least one state within each composite as final, to allow the transition from one composite to another. As described in the next step, the final states will have an extra transition to a FinalState. FinalStates are special states which, when reached, indicate that the composite will perform the next transition instead of its internal states.

```
FOR i = 0 TO maxRegion
  r = new Region
  tree.add(r)
  r.init(0)
DEF init(currentDepth)
  FOR j = 0 TO maxState
```

```
    IF rand()<compositeRatio AND currentDepth < maxDepth THEN
      s = new CompositeState
      self.add(s)
      s.init(currentDepth+1)
    ELSE
      s = new State
      self.add(s)
  self.states.finalState = new FinalState()
  //end of FOR j
  FOR k = 0 TO maxFinals
    self.states.getOneRandom().mayBeFinal = TRUE
  //end of FOR k
```

- Transition generation: we generate the flow of execution of the program. First, we define the number of inputs and outputs of each state, then we connect the states.

```
FOR region IN tree.regions:
  region.initTransitions()
DEF initTransitions:
  FOR currentState IN container:
        FOR i = 0 TO maxOutputs
          create new Transition FROM currentState TO
              container.getRandomState()
          IF currentState IS container: //Regions and composites are
            currentState.initTransitions() //containers, simple states
                                           //are not.
        //end of FOR i
        IF currentState.mayBeFinal THEN
          create new Transition FROM currentState TO container.finalState
```

- Constraint solving: we have to ensure that there are no blocking or unreachable states. This rely on repetitively selecting a problematic state and modifying one of its input or output, until all the detected problems are solved.

```
orphans = tree.getBlockingStates()+tree.getUnreachableStates()
WHILE orphans.size() > 0
  currentState = orphans.getOneRandom()
  IF currentState.isBlocking THEN
    newOutput = currentState.parent.outputs.getOneRandom()
    currentState.outputTransitions.getOneRandom().setTo(newOutput)
  ELSE
    newInput = currentState.parent.inputs.getOneRandom()
    currentState.inputTransitions.getOneRandom().setFrom(newInput)
```

4.3 Results

Generation of state machines allowed us so far to measure how big a state machine can be, how fast they are to perform transitions in the different language and the associated disk space and RAM costs.

The generated state machines can have slightly variable sizes for a given set of parameters, and perform their transitions randomly. To ensure the consistency of the results, we decided to respect the following protocol:

- For each given set of parameters, several state machines are generated to ensure results are not dependent of a specific structure.
- For a given state machine, tests are performed several times to ensure the results are not dependent on a specific path of performed transitions.
- Whenever a parameter is not specified, we assume that the state machine contains 100 states and performs 1000 transitions.
- The tests were run on a single core machine with 512MB of memory, running on Linux. The last part of this section describes how the results of the tests change with other configurations.

The performance measures are gathered in automatically generated HTML files, such as the one in annex E. To ease readability, we separated results in several charts, that we are going to analyse now.

In the following graphs, we separated C and Scala which are on too different scales, keeping Java as a reference:

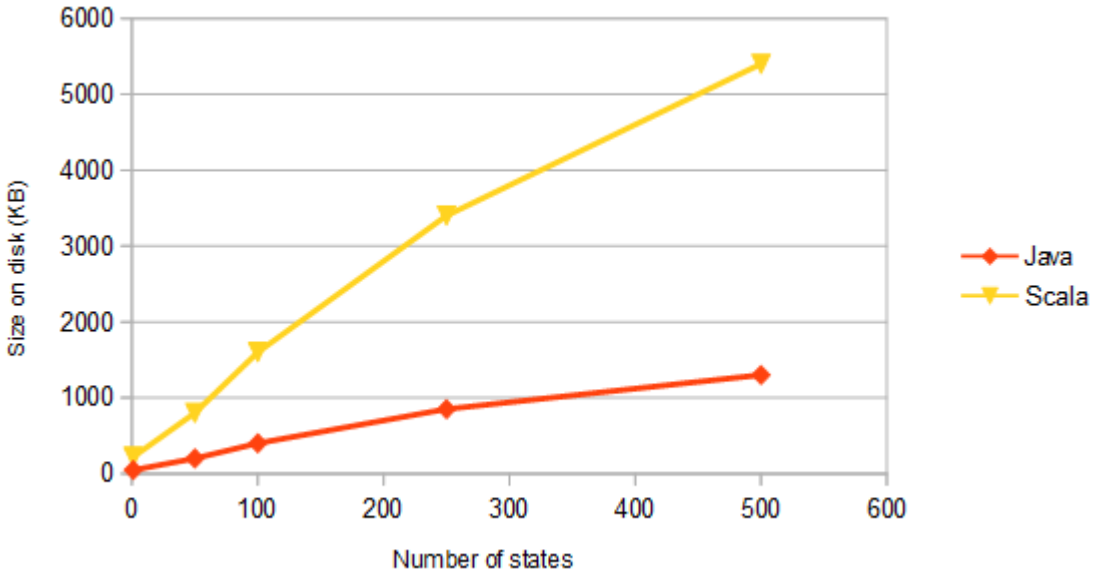

Figure 7.1: Size of the generated program on the disk depending on the size of the state machine for Scala and Java

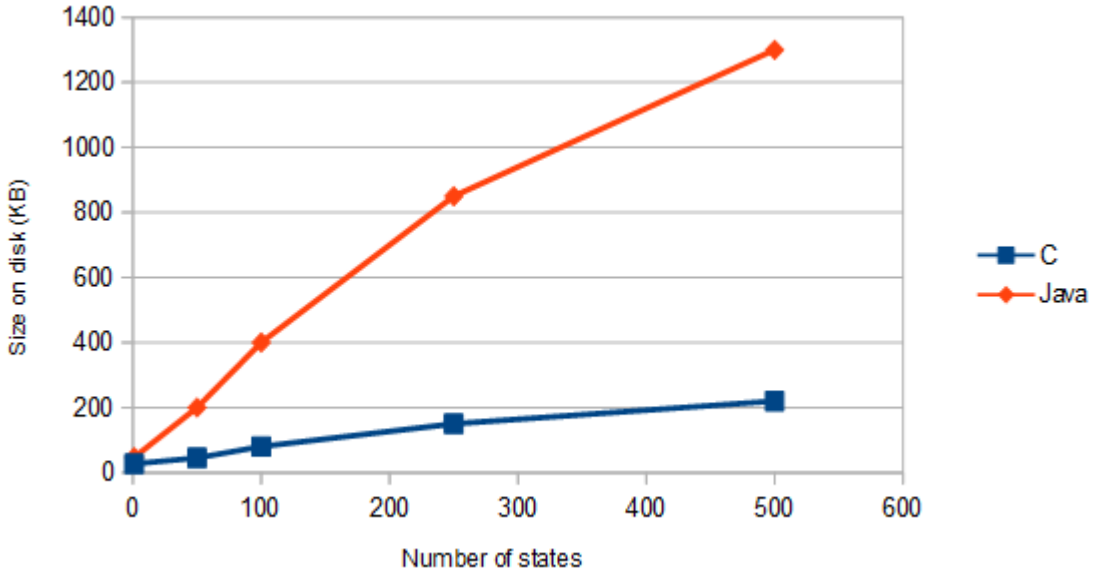

Figure 7.2: Size of the generated program on the disk depending on the size of the state machine for Java and C

The size of the generated program on the disk is roughly proportional to the size of the state machine. Java generates a program roughly 5 times bigger than C, while Scala is from 10 to 25 times bigger than C. C is therefore more adapted to platforms with a very low hard-drive capacity.

This test also revealed that the precompiler is running out of memory for state machines with more than 500 states on a machine with 512MB of RAM. However, it is possible to increase this limit by increasing the RAM.

Table 8: RAM consumption depending on the size of the state machine and on the language.

Size of the state machine	**RAM consumption (C)**	**RAM consumption (Java)**	**RAM consumption (Scala)**
1 states	0.28MB	29MB	32MB
50 states	0.30MB	29MB	32MB
100 states	0.35MB	31MB	36MB
250 states	0.38MB	31MB	38MB
500 states	0.40MB	32MB	40MB

The RAM use of C programs is significantly smaller than the Java and Scala equivalents, approximately 100 times. We can notice that the size of the state machine has a low impact on the RAM consumption. This indicates that most of the RAM use is independent of the size of the state machine.

In the following tests, Scala have been deactivated for more than 10^5 transitions, and Java for 10^7 transitions: this is not a limit of the language or of the implementation, but only a question of reasonable testing time.

Table 9: Execution time depending on the size of the state machine and on the language.

Transitions performed	Execution time for C (s)	Execution time for Java (s)	Execution time for Scala (s)
10^2	0.005	0.05	10
10^3	0.01	0.2	20
10^4	0.04	0.8	120
10^5	0.25	10	-
10^6	3	120	-
10^7	25	-	-

The execution time is roughly proportional to the number of performed transitions. C is 10 to 40 times faster on this benchmark than Java, which is itself 100 faster than Scala. C is then preferable when performances are critical.

Table 10: RAM consumption depending on the number of transitions performed.

Transitions performed	RAM consumption for C	RAM consumption for Java	RAM consumption for Scala
10^2	0.4MB	30MB	32MB
10^3	0.4MB	30MB	32MB
10^4	0.4MB	30MB	32MB
10^5	0.4MB	30MB	-
10^6	0.4MB	30MB	-
10^7	0.4MB	-	-

Performing tests with variable number of performed transitions can help detect memory leaks if the RAM consumption increases. We did not detect any significant RAM consumption increases over time, which means that the transitions do not induce majors memory leaks.

The tests were useful to provide the following guidelines:

- On very constrained devices, be it the hard-drive, RAM or processor, C is able to provide improved performances.
- On more powerful devices, the users can choose Java, Scala and C, depending on their programing preferences.

4.4 Discussion

The state machine generation described is this report is only a basic proof of concept. We intend to improve this generation by trying to find relevant metrics to measure the quality of the generated state machine. These measures should allow us to assess the usefulness of some improvements we considered.

4.4.1 Quality of generation measures

A new language such as ThingML does not possess programs big enough to be used as a reference to measure its quality.

However, since we are trying to simulate a real-life program, analyzing the state machine can help us decide if the generation is good enough. In the case of a state-machine based programs, the following informations can be measured:

- Variance of parameters: a program usually relies on a combination of simple and complex operations. A high variance of the states depths, of the number of state by region and of the number of transitions per state can be interpreted as a variance of the complexity of operations. This measure can be performed once the state-machine is generated.
- Number of time each transition is performed: we can expect that some transitions will be much more performed than other. This measure can be performed at execution time.

4.4.2 Quality of generation improvement

To get closer to a real life program, we considered the following leads:

- Insert more diverse concepts: the currently generated state machines do not contain operations described by the action language. This can be done by inserting operations from the action language inside the states.
- Parameterize the inserted concepts: by inserting only a given set of operations, it is possible to detect which ones are slow. This would allow to bring guidelines on which concepts should be used in priority.
- Randomize the generated concepts: as for the state machine structure, we fear that inserting always the same pieces of code might lead some compilers to optimize this in a way that would false the results. We want to know if bringing variability in the inserted pieces of code would help tackle this concern.

Chapter 5

Conclusion

In this Master Thesis, we tackled the problem of testing a DSL for H&D services. The techniques usually available to test programs and languages are not adapted to this kind of heterogeneous developing environments, since it requires to re-implement a test harness for each platform. We proposed an approach relying on performing testing operations in the DSL itself, rather than implementing these operations in each compiler. More specifically, we tackled two categories of problem: functional testing and performance measurement.

Functional testing is difficult because the DSL is compiled on a variety of platforms, using different programing languages. By embedding unit testing directly in the DSL, our approach makes it seamless for the DSL designers and users to test the DSL and their H&D services. Indeed, from a platform independent test specification, our approach enables to test it automatically on all the available platforms.

Moreover, our solution relies on comparing execution traces. This opens possibilities of understanding more finely what went wrong during the execution of the tests, but also allows to compare traces of different compilers to locate the origin in the problem in the compiler or in the test itself.

Performance measurement often relies on benchmarks. Developing benchmark is tedious, this is why we decided to generate automatically benchmarks of different sizes and complexities. This generation ensures constraints such as reachability of all states, and uses variable sets of parameters to introduce variability, such as the depth of the composites states or the number of regions.

The generated benchmarks allow us to assess the performances of the code produced by the different compilers. This provides the DSL users guidelines about which compiler should be used depending on the use case, and the DSL developers guidelines about how to optimize their compilers.

While this approach gave us an initial assessment of the performances of each compilers, we expect to improve these benchmarks by inserting operations from the action language of our DSL.

This document presented an approach for the functional and performance testing of a DSL compiled to different environments. This approach proved to contribute to improve the ThingML DSL both in terms of reliability and efficiency.

Bibliography

[1] Allen L. Ambler, Margaret M. Burnett, and Betsy A. Zimmerman. Operational versus definitional: A perspective on programming paradigms. *Computer*, 25(9):28–43, 1992.

[2] Kevin Ashton. That internet of things thing. *RFiD Journal*, 22:97–114, 2009.

[3] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[4] Cristian Borcea, Deepa Iyer, Porlin Kang, Akhilesh Saxena, and Liviu Iftode. Cooperative computing for distributed embedded systems. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 227–236. IEEE, 2002.

[5] Themistoklis Bourdenas and Morris Sloman. Starfish: policy driven self-management in wireless sensor networks. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 75–83. ACM, 2010.

[6] Jorge Cardoso, Amit Sheth, John Miller, Jonathan Arnold, and Krys Kochut. Quality of service for workflows and web service processes. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(3):281–308, 2004.

[7] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[8] Franck Fleurey, Brice Morin, and Arnor Solberg. A model-driven approach to develop adaptive firmwares. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 168–177. ACM, 2011.

[9] Franck Fleurey, Brice Morin, Arnor Solberg, and Olivier Barais. Mde to manage communications with and between resource-constrained systems. In *Model Driven Engineering Languages and Systems*, pages 349–363. Springer, 2011.

[10] Francois Fouquet, Olivier Barais, Noel Plouzeau, Jean-marc Jezequel, Brice Morin, and Franck Fleurey. A Dynamic Component Model for Cyber Physical Systems Categories and Subject Descriptors. *15th International ACM SIGSOFT Symposium on Component Based Software Engineering (2012)*, pages 135–144, 2012.

[11] D. Garlan, SW Cheng, and AC Huang. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, pages 276–277, 2004.

[12] Alain Girault, Bilung Lee, and Edward A Lee. Hierarchical finite state machines with multiple concurrency models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(6):742–760, 1999.

[13] Aniruddha Gokhale, Krishnakumar Balasubramanian, Arvind S. Krishna, Jaiganesh Balasubramanian, George Edwards, Gan Deng, Emre Turkay, Jeffrey Parsons, and Douglas C. Schmidt. Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. *Science of Computer Programming*, 73(1):39–58, September 2008.

[14] Google. Gperftools, code.google.com/p/gperftools.

[15] Matti A Kaulio. Customer, consumer and user involvement in product development: A framework and a review of selected methods. *Total Quality Management*, 9(1):141–149, 1998.

[16] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[17] G. Lawton. Machine-to-machine technology gears up for growth. *Computer*, 37(9):12–15, 2004.

[18] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing distributed systems in conic. *Software Engineering, IEEE Transactions on*, 15(6):663–675, 1989.

[19] Dan Nessett. Massively distributed systems: Design issues and challenges. In *USENIX Workshop on Embedded System*, 1999.

[20] J Rothenberg. *The nature of modeling*. 1989.

[21] Per Runeson. A survey of unit testing practices. *Software, IEEE*, 23(4):22–29, 2006.

[22] Douglas C Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[23] Susan Elliott Sim, Steve Easterbrook, and Richard C Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering*, pages 74–83. IEEE Computer Society, 2003.

[24] SINTEF. ThingML, www.thingml.org.

[25] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages. *Centrum voor Wiskunde en Informatika*, 5, 2000.

[26] Marcel Verhoef, PG Larsen, and Jozef Hooman. Modeling and validating distributed embedded real-time systems with VDM++. *FM 2006: Formal Methods*, 2006.

[27] Elaine J Weyuker and Filippos I Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE transactions on software engineering*, 26(12):1147–1156, 2000.

[28] Ti-Yen Yen and Wayne Wolf. Communication synthesis for distributed embedded systems. In *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on*, pages 288–294. IEEE, 1995.

[29] LLC Yourkit. Yourkit profiler, www.yourkit.com.

[30] Weibin Zhao, David Olshefski, and Henning G Schulzrinne. Internet quality of service: An overview. 2000.

Appendix A

Example of event-driven state machine

```
/*****************************************************************/
#include <stdio.h>

/*****************************************************************/
typedef enum {
       ST_RADIO,
       ST_CD
} STATES;

typedef enum {
       EVT_MODE,
       EVT_NEXT
} EVENTS;

EVENTS readEventFromMessageQueue(void);

/*****************************************************************/
int main(void)
{
  /* Default state is radio */
  STATES state = ST_RADIO;
  int stationNumber = 0;
  int trackNumber = 0;

  /* Infinite loop */
  while(1)
  {
    /* Read the next incoming event. Usually this is a blocking function. */
    EVENTS event = readEventFromMessageQueue();
```

```
    /* Switch the state and the event to execute the right transition. */
    switch(state)
    {
      case ST_RADIO:
        switch(event)
        {
          case EVT_MODE:
            /* Change the state */
            state = ST_CD;
            break;
          case EVT_NEXT:
            /* Increase the station number */
            stationNumber++;
            break;
        }
        break;

      case ST_CD:
        switch(event)
        {
          case EVT_MODE:
            /* Change the state */
            state = ST_RADIO;
            break;
          case EVT_NEXT:
            /* Go to the next track */
            trackNumber++;
            break;
        }
        break;
    }
  }
}
```

Appendix B

Example of a simple ThingML test

```
thing TestEmptyTransition includes Test
@test "ttt # IJKKK"
{
	statechart TestEmptyTransition init I {
		state I {
			on entry harnessOut!testOut('I')

			transition -> J

			transition -> K
			event m : harnessIn?testIn
			guard m.c == 't'
		}
		state J {
			on entry harnessOut!testOut('J')

			transition -> K
			event m : harnessIn?testIn
			guard m.c == 't'
		}
		state K {
			on entry harnessOut!testOut('K')

			transition -> K
			event m : harnessIn?testIn
			guard m.c == 't'
		}
	}
}
```

Appendix C

Example of failing ThingML test

```
thing TestCompEventCapture includes Test
@test "00 # 12b3c4"
@test "000 # 12b3c4a5"
{
	statechart TestCompEventCapture init s1{
		composite state s1 init s2 {
			on entry harnessOut!testOut('1')

			transition -> s5
			event m : harnessIn?testIn
			guard m.c == '0'
			action harnessOut!testOut('a')

			state s2 {
				on entry harnessOut!testOut('2')

				transition -> s3
				event m : harnessIn?testIn
				guard m.c == '0'
				action harnessOut!testOut('b')
			}

			state s3 {
				on entry harnessOut!testOut('3')

				transition -> s4
				event m : harnessIn?testIn
				guard m.c == '0'
				action harnessOut!testOut('c')
			}

			state s4 {
```

```
                        on entry harnessOut!testOut('4')
                }

        }

        state s5 {
                on entry harnessOut!testOut('5')

                transition -> s1
                event m : harnessIn?testIn
                guard m.c == '0'
                action harnessOut!testOut('e')
        }
    }
}
```

Appendix D

Example of functional tests results

Test name	Compiler	Result
testArrays2	C	Success
testArrays2	Scala	Success
testArrays2	Java	Success
testArrays	C	ErrorAtCompilation does not match e1827364554637281e1w2w3w4w5w6w7w8w for input x (x)
testArrays	Scala	ErrorAtCompilation does not match e1827364554637281e1w2w3w4w5w6w7w8w for input x (x)
testArrays	Java	Success
testAutoTransition	C	Success
testAutoTransition	Scala	Success
testAutoTransition	Java	Success
testCompEventCapture	C	Success
testCompEventCapture	Scala	Success
testCompEventCapture	Java	Success
testCompositeStates	C	Success
testCompositeStates	Scala	01230 does not match 012320 for input nni (nni)
testCompositeStates	Java	Success
testCompStatesEntry	C	Success
testCompStatesEntry	Scala	Success
testCompStatesEntry	Java	Success
testCompStatesExit	C	Success
testCompStatesExit	Scala	I1211CI does not match I1212CI for input t212xt (t212xt)
testCompStatesExit	Java	Success
testDeepCompositeStates	C	012abc3de does not match (012abc|abc012)(3d|d3) for input na (na)
testDeepCompositeStates	Scala	012345 does not match (012abc|abc012)(3d|d3)(4e|e4)(5f|f5) for input naaa (naaa)
testDeepCompositeStates	Java	012abc35df does not match (012abc|abc012)(3d|d3) for input na (na)

Appendix E

Example of performance tests measures

Compiler	Test	Memory	Binary size	Execution time	States number
C	PerfTest0 0	0.3 MB	72506	0.008567	122
C	PerfTest0 0	0.3 MB	72506	0.007500	122
C	PerfTest0 0	0.3 MB	72506	0.010001	122
Scala	PerfTest0 0	34.0 MB	1847008	36 s	122
Scala	PerfTest0 0	37.0 MB	1847008	36 s	122
Scala	PerfTest0 0	33.0 MB	1847008	32 s	122
Java	PerfTest0 0	29.0 MB	452982	0.394s	122
Java	PerfTest0 0	29.0 MB	452982	0.567s	122
Java	PerfTest0 0	32.0 MB	452982	0.492s	122
C	PerfTest1 0	0.3 MB	129850	0.015646	219
C	PerfTest1 0	0.3 MB	129850	0.013588	219
C	PerfTest1 0	0.3 MB	129850	0.013445	219
Scala	PerfTest1 0	33.0 MB	3264123	34 s	219
Scala	PerfTest1 0	37.0 MB	3264123	35 s	219
Scala	PerfTest1 0	35.0 MB	3264123	36 s	219
Java	PerfTest1 0	37.0 MB	802231	0.402s	219
Java	PerfTest1 0	35.0 MB	802231	0.43s	219
Java	PerfTest1 0	34.0 MB	802231	0.416s	219