

HEADS IDE & Methodology



*by HEADS Consortium, Licensed under
a Creative Commons Attribution 4.0 International License.*

Table of Contents

Introduction	1.1
HEADS Actors	1.2
Platform Experts	1.2.1
Service Developers	1.2.2
Service Operators	1.2.3
HEADS IDE	1.3
@Design-time	1.3.1
@Runtime	1.3.2
Installation Guide	1.3.3
HEADS Methodology	1.4
For Platform Experts	1.4.1
Create libraries to support platform specific features / library / component	1.4.1.1
Extend the ThingML compilers to target a new platform	1.4.1.2
Implement a ThingML plugin to target a new protocol	1.4.1.3
Test the ThingML compilers and plugins	1.4.1.4
Prepare, build and flash any openWRT device to run Heads Artefacts	1.4.1.5
Prepare, build and flash any buildroot compatible device to run Kevoree	1.4.1.6
Extend Kevoree to deploy code for a new platform	1.4.1.7
Generalities	1.4.1.7.1
Kevoree Model	1.4.1.7.2
Core	1.4.1.7.3
Remote Code Loader	1.4.1.7.4
Registry Client	1.4.1.7.5
Kevoree Model generation	1.4.1.7.6
Registry Client	1.4.1.7.7
Code generator	1.4.1.7.8
Runtime	1.4.1.7.9
Kevscript Tools	1.4.1.7.10
Components	1.4.1.7.11
Component	1.4.1.7.11.1
Node	1.4.1.7.11.2
Group	1.4.1.7.11.3
Channel	1.4.1.7.11.4
Extend Kevoree to support a new communication channel	1.4.1.8
Using platforms for CEP with Apama	1.4.1.9
For Service Developers	1.4.2
Model HD-Service logic with ThingML components	1.4.2.1
Compile ThingML components to platform specific code	1.4.2.2
Model HD-Service deployment with Kevoree	1.4.2.3

Deploy platform specific code using Kevoree	1.4.2.4
Services with CEP and distributed CEP	1.4.2.5
Developing Services with Apama CEP	1.4.2.6
Migrate from Kevoree v5.3.x to v5.4.x	1.4.2.7
For Service Operators	1.4.3
Visualize the configuration and status of a running HD-Service	1.4.3.1
Modify the configuration and deployment of a running HD-Service	1.4.3.2
Re-deploy/adapt/reconfigure a running HD-Service	1.4.3.3
Operating services for CEP with Apama	1.4.3.4
Credits	1.5
License	1.6

HEADS IDE & Methodology

The main result of the HEADS project is **the HEADS IDE for HD-Services development**. The HEADS IDE and modelling languages are fully released **as open-source** using non contaminating licenses (such as LGPL or EPL) to allow both proprietary usage and extensions. The HEADS IDE is complemented by this **methodology document** which

1. guides developers of HD-Services in using the HEADS techniques and tools and
2. guides platform experts to add support for new platforms by developing plugins for the HEADS IDE.

The objective of the project is not to provide a complete set of plugins covering the whole future internet continuum but a set of selected representative plugins which

1. demonstrate the approach,
2. makes it available to third parties and
3. cover the needs of the use cases.

A HEADS plugin for a particular platform includes

- a code generator for the specific platform,
- a HEADS model library (API) exposing the capabilities of the platform and
- a mechanism for deploying code to the platform.

The plugins supporting different platforms can be released with different licences. All the generic code generators to general purpose language will be released as open-source. All the plugins developed by the academic partners will be released open-source. Plugins and specialization made by the industry partners and for the use cases might be kept proprietary. Third parties will be provided with the tools and documentation necessary to develop third party plugins.

HEADS Actors

This section introduces and defines the different HEADS actors.

As we will intensively refer to these actors in the rest of this document, **better read this whole section!**

Platform Experts

The platform Experts are responsible for creating new plugins for the HEADS IDE in order to support new platform and their specific capabilities.

The platform expert can use a number of different extension points in order to support both the design time and the runtime support of a given platform. For the design time part, ThingML and its transformations are extended to support the generation of code which can run on the target platform. For the runtime, Kevoree is extended to support the deployment and execution of the generated code on the target platform.

The tasks associated with the HEADS Platform Expert are detailed in section [3.1](#).

Service Developers

The service developer is the primary user of the HEADS IDE. The service developer is the one dealing with the requirements of the HD-Service and responsible for creating the design and implementation of the HD-Service logic.

The Service developer has to interact with a number of platform experts in order to select (or learn about) the platforms which will be used in the HD-Service infrastructure. If some platforms are not yet supported by the HEADS IDE, the platform experts need to create new plugins for the HEADS IDE in order to support and expose the capabilities of the required platforms.

The service developer also has to interact with the Service operator which is responsible for the deployment and runtime monitoring of the HD-Service.

The service developer is mainly using ThingML to create the models of the HD-Service and might create and/or use some re-usable ThingML libraries. The tasks associated with the HEADS Service Developer are detailed in section [3.2](#).

Service Operators

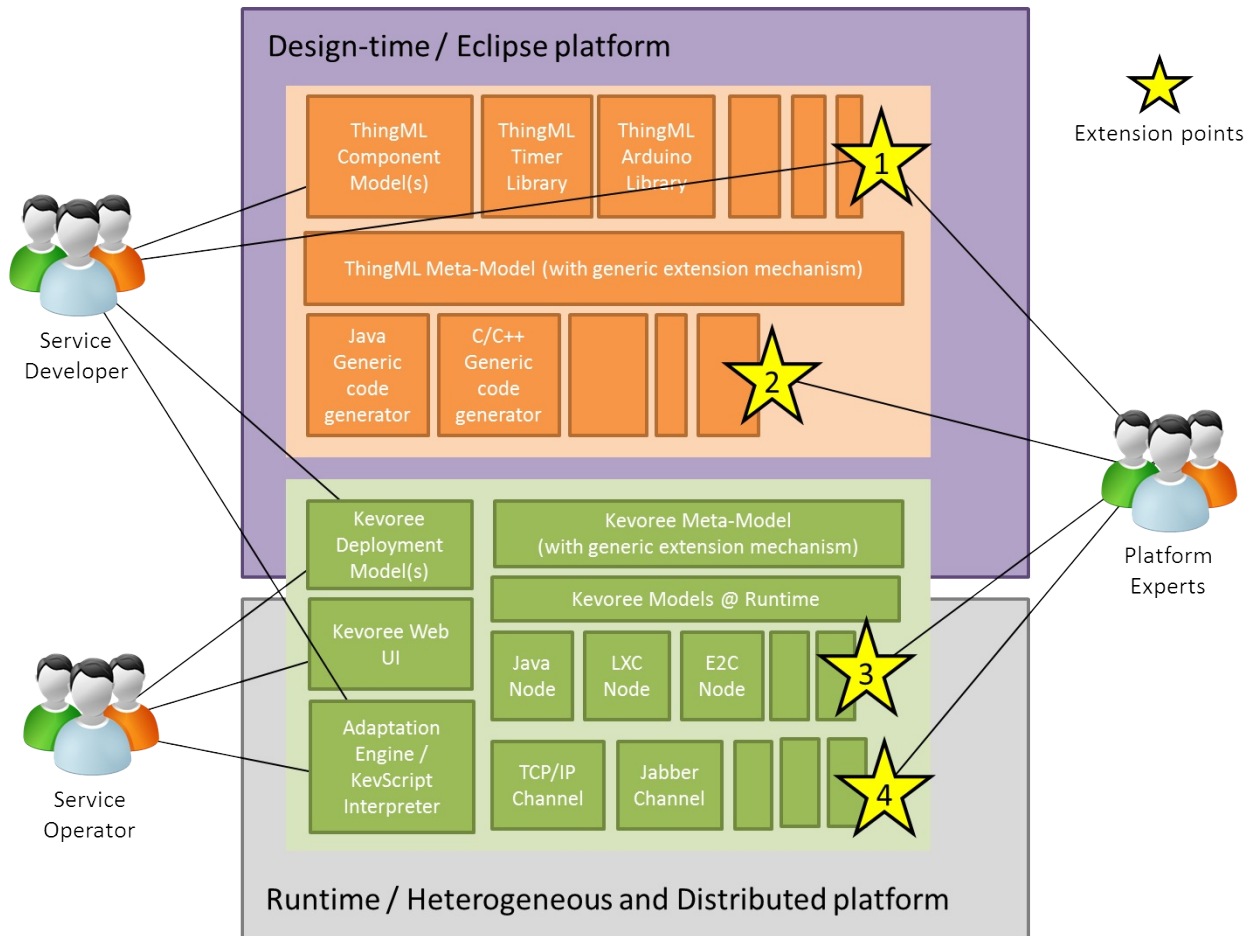
The HEADS service operator is responsible for the deployment of the HD-Service and should be able to monitor its execution, adapt it at runtime and deploy new version of the HD-Service components (provided by the HD-Service developer).

The Service operator does not necessarily have detailed knowledge of the HD-Service implementation details or platforms it is running on. It is typically an administrator or a super-user of the HD-Service itself.

The service operator is mainly using Kevoree to accomplish its tasks. The tasks associated with the HEADS Platform Expert are detailed in section [3.3](#).

HEADS IDE

The conceptual architecture of the HEADS IDE together with the main HEADS actors is depicted in the figure below:



The HEADS IDE includes 2 main parts: the design time tools in the top part of the figure and the runtime tools on the bottom part of the figure. Both parts are described in the following two sub-sections.

The HEADS IDE is meant to be used by the HD-Service developer and the HD-Service operator in order to create, deploy, monitor and evolve HD-Services. The distinction between the service developer and operator is that the developer is mainly concerned with the HEADS design time tools (based on ThingML) and the operator is mainly concerned with the HEADS runtime tools (based on Kevoree).

In addition, the HEADS IDE is meant to be extended by platform experts. The HEADS IDE is built as an open-source framework which has a set of extension points to support new platforms and communication channels. At this point, we have identified 4 extension points which are represented by yellow stars in the figure above. Each of these extension points allows supporting different aspects of a particular target platform.

@Design-time

At design-time, HEADS IDE will provide the different HEADS actors with modelling languages, editors and tools to engineer HD-services.

ThingML is as a domain-specific modeling language which includes concepts to describe both software components and communication protocols. The formalism used is a combination of architecture models, state machines and an imperative action language similar to concepts found e.g. in UML2. ThingML also provides a set of compilers (currently targeting C and Java-based platforms) in order to produce fully executable code from ThingML specifications. ThingML will both support the platform experts and service developers to define (respectively) drivers and business logic.

To express advanced analysis of sensors data, the CEP logic will be defined by the Event Processing language EPL. It contains besides descriptive event query support in an SQL-oriented style also a complete procedural language. There are a set of Eclipse plugins, which support the design of EPL specifications. This contains an Eclipse editor for EPL with features like dynamic compilation, syntax highlighting and auto completion. Special support is given for defining the structure of events by so called event types. For the communication with many common data sources adapters are available to convert the structure descriptions like XML schema, RDBMS schema to Event types. The runtime contains the corresponding support for converting the data to the internal event format of the engine.

@Runtime

todo Inria to complete this part

At runtime, the HEADS platform will be composed of a set of components and protocols supporting the execution of HD-services.

The CEP execution engine is responsible for the execution of EPL components together with adapters for conversion of external data.

The communication between two components can be performed over different channel types. HEADS IDE provides support for the following communication protocols: serial communication (RS232 and RS485), ZWave, ZigBee, OPC and MQTT. However, Platform Experts may extend the functionality and improve usability of the system by providing new communication plug-ins which will be used by Service Operator during development and configuration of the HD-Services.

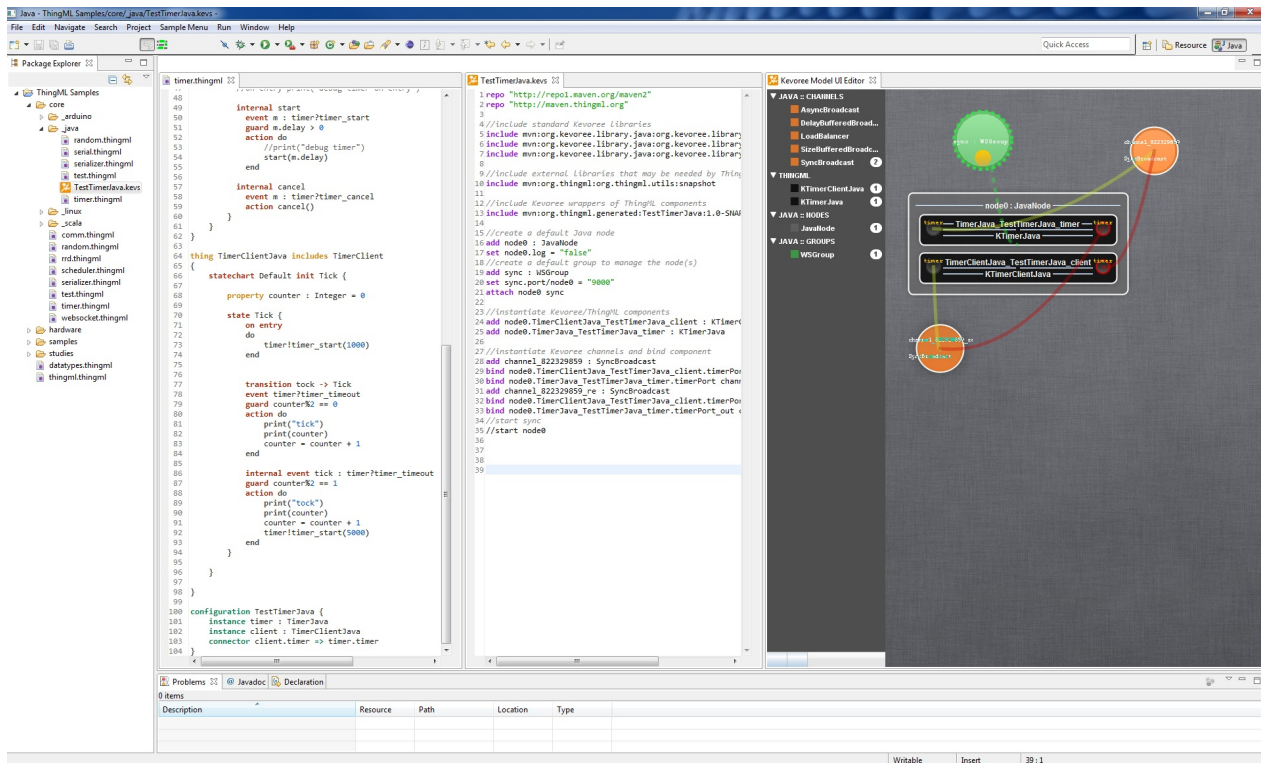
Installation Guide

The HEADS IDE is a set of Eclipse plugins, which provides languages and tools for designing and operating Heterogeneous and Distributed (HD) services.

Complete Eclipse bundles, integrating the HEADS languages, frameworks and tools are [available for download](#):

- [Linux 64 bits](#)
- [Linux 32 bits](#)
- [MacOS X \(64 bits\)](#)
- [Windows 64 bits](#)
- [Windows 32 bits](#)

Just unzip the file corresponding to your OS, and you are ready to go!



HEADS Methodology

This methodology will guide platform experts, service developers and service operators step by step in order to use the HEADS IDE.

For Platform Experts

The platform expert is responsible for writing plugins for the HEADS IDE in order to support different target platforms. The HEADS IDE has 4 different extension points which can be used to support specific platforms.

Create libraries to support platform specific features / library / component

In the case that the target language is supported by an existing transformation, support for platform specific features can be added by creating platform specific libraries in ThingML. These platform specific libraries integrated ThingML structures and code together with code written in the target language and specific to the target platform.

This section details how to implement a driver for ThingML. A driver basically wraps an existing piece of code (in Java, C, C++, ...) and exposes it as a ThingML API, which can then be seamlessly used by service designers, with no need to cope with low-level details related to Java, C, C++, ...

We will use a simple random integer generator as a running example, that we will wrap in C and also in Java.

Generating random integers could certainly be implemented directly in ThingML, however, as all programming languages already provide facilities for random generation, this would have been like... re-inventing the wheel. **Better wrap what already works!**

Defining the interface

A ThingML driver is a plain ThingML component (or *thing*). In this respect, it is usually recommended to first describe the interface of a component, and then implement it, possibly for different platforms. This is realized in two steps.

First, declare a thing fragment containing all the messages that are relevant:

```
thing fragment RandomMsg{
  message request();
  message answer(v: Integer);
}
```

As ThingML is asynchronous, **the request and the answer should be defined in two distinct messages**. In synchronous Java, this would have been a single method like `public int random()`, blocking the caller until the random is computed.

Then declare a second thing fragment, which includes the former one, and group messages into a port:

```
thing fragment Random includes RandomMsg{
  provided port random {
    receives request
    sends answer
  }
}
```

ThingML would allow defining only one thing fragment containing both the message declarations and the port. However, if a thing want to use a timer, it will need to include the timer messages. **Splitting message and port declarations favors a better reuse:**

```
thing fragment RandomUser includes RandomMsg{
  required port random {
    sends request
    receives answer
  }
}
```

Calling native code from ThingML

Calling native code, for any target language, is realized as follows:

- for pure native statement: they should be place between simple quotes, such as `'srand(time(NULL));'`

- for mixed code, typically if a call to a native function should be passed with parameters coming from ThingML, it is realized as follows: `' ' & rn & '.nextInt(Short.MAX_VALUE + 1)'`, where `rn` is a ThingML variable (holding a Java type).

The implementation (wrapping) of the random facility is given below, for C and for Java.

calling C/C++ code

```
import "../random.thingml"

thing RandomLinux includes Random
@c_headers "#include <time.h>"
{
    statechart Random init start {
        state start {
            on entry 'srand(time(NULL));'
            transition ->waiting
        }
        state waiting {
            internal waiting
            event random?request
            action random!answer('rand()')
        }
    }
}
```

First, the `@c_headers "#include <time.h>"` annotation ensure the `RandomLinux` thing includes the proper C headers. When the thing is initialized, it will initialize the random sequence by calling the C `srand(time(NULL))` function and will then wait for request and serve random integers by calling the C `rand` function.

calling Java code

```
import "../random.thingml"

datatype JavaRandom
@java_type "java.util.Random";

thing RandomJava includes Random
{
    property rn : JavaRandom = 'new java.util.Random()'
    statechart Random init waiting {
        state waiting {
            internal waiting
            event random?request
            action random!answer(' ' & rn & '.nextInt(Short.MAX_VALUE + 1)')
        }
    }
}
```

First, a datatype is created, backed by the `java.util.Random` class. This datatype is initialized in the `RandomJava` thing as follows:

```
property rn : JavaRandom = 'new java.util.Random()' .
```

The call to `new` is actually plain Java code and not a ThingML keyword, as it is placed between single quotes.

Similarly to the C thing, this thing will then wait for request and serve random integers using `' ' & rn & '.nextInt(Short.MAX_VALUE + 1)'`.

This statement mixes ThingML code: `rn` is a ThingML property (though it is mapped to a Java type), while `.nextInt(Short.MAX_VALUE + 1)` is plain Java code.

ThingML Integer are actually 2-byte long and thus cannot be mapped to Java int. They are rather mapped on Java short. The `Short.MAX_VALUE + 1` expression ensures the java int produced by `nextInt` does not overflow the ThingML Integer (*i.e.*, a Java short).

Calling ThingML code from native code

The previous example simply called native code from a ThingML program. In more advanced cases, it is however useful to be able to call ThingML code from a native API (typically when wrapping a library relying on callbacks).

in C/C++

To do so one should adapt/wrap a native C/C++ library in such a way that the library can call callbacks which execute the ThingML generated code. We propose to adapt/wrap the native (wrapped) library in a library (wrapping library) which can call the generated code in ThingML. The explanation below is given using C++, but the same approach can be used in C as well.

Context: A sensor returns a value once in a while. There is a library that handles a value update. The library provides a callback which is called on the value update. Thus, a user can implement this callback to process the value or define some logic when a new value is returned. We would like to use this library in ThingML and define logic and process the given value.

1) Create a type definition for a callback and structure which holds the callback to call from the wrapping library. The type definition and structure should look as follows:

```
typedef void (*pthingMLCallback)(void* _instance, ...);

struct ThingMLCallback {
    pthingMLCallback fn_callback;
    void* instance;

    ThingMLCallback(pthingMLCallback _callback, void* _instance):
        fn_callback(_callback),
        instance(_instance){
    };
};
```

As one may notice, we use the ellipsis ("..."). Thus, the wrapping library can call a function with any number of arguments following `_instance`. The `_instance` argument is used by ThingML to identify a thing. Therefore, `_instance` is an internal concern of ThingML. One should just make sure that a reference to a thing (`_instance`) is passed together with a reference to the callback. `ThingMLCallback` has two arguments, i.e. a reference `_callback` to the callback and void reference `_instance` to the thing, which is passed as the first argument when the callback is called.

2) The wrapping library that calls the callback should hold a reference to an instance of `ThingMLCallback`. For example:

```
class BinarySensor {
private:
    ThingMLCallback* valueUpdatedCallback;
public:
    ...
    //set ThingML callback
    void setValueUpdatedCallback(ThingMLCallback* _callback){valueUpdatedCallback = _callback;};

    //function is called by a native library
    void valueupdate(int value);
    ...
}
```

3) Call the callback from the wrapping library as follows.

```
void BinarySensor::valueupdate(int value){
    this->valueUpdatedCallback->fn_callback(this->valueUpdatedCallback->instance, value);
}
```

Note, that the `valueupdate(int value)` function is called by the native (wrapped) library.

4) Define a thing which uses the wrapping library. The wrapping library calls the callback function `value_change_binarysensor_callback()` defined in the thing `zWaveBinarySensor`.

```

import "thingml.thingml"

datatype BinarySensor
@c_type "BinarySensor*";

thing ZWaveBinarySensor
@c_header "
#include <stdlib.h>
#include <cstdint>
#include \"BinarySensor.h\"

using namespace TinyOpenZWaveApi;
"
{
    property bs : BinarySensor

    provided port bsport {
        receives initialize
    }

    //these are two internal ports should be bound together
    provided port bsportintsend {
        sends status
    }

    required port bsportintrecv {
        receives status
    }

    function value_change_binarysensor_callback()
    @c_prototype "void value_change_binarysensor_callback(void *_instance, ...)"
    @c_instance_var_name "(ZWaveBinarySensor_Instance *) _instance"
    do
        'va_list arguments;'
        'va_start(arguments, _instance);'
        'int state = va_arg(arguments, int);'
        'va_end(arguments);'
        bsportintsend!status('state')
    end

    function init_binarysensor() do
        print "ZWaveBinarySensor: initializing ... \n"
        'ThingMLCallback* value_changed = new ThingMLCallback(value_change_binarysensor_callback, _instance);'
        bs = 'new BinarySensor();'
        '&bs&'>setValueUpdatedCallback(value_changed);'
    end

    function getState() : Integer do
        return '&bs&'>getCurrentValue()'
    end

    statechart behavior init Start {

        state Start {
            on entry do
                print "ZWaveBinarySensor: waiting for initialize command ... \n"
            end
            transition->Ready
            event bsport?initialize
            action do
                init_binarysensor()
            end
        }

        state Ready {
            on entry do
                print "ZWaveBinarySensor: ready ... \n"
            end
        }
    }
}

```

```

        internal event e : bsportintrecv?status
        action do
            // here may go some code
        end
    }

}
}

```

Note, we use the ThingML capability to blend the ThingML code and sources which are native to some platform (in this case C++). The C++ code is enclosed by the single quotes.

We have defined the callback `value_change_binarysensor_callback` with the signature that corresponds to the `typedef` from point 1. Make sure that this callback in ThingML should be annotated with `@c-prototype` and `@c-instance_var_name`. The annotation `@c-prototype` instructs ThingML to generate a function with the signature given in the double quotes, `@c-instance_var_name` casts `_instance` to the proper type. These annotations are required to perform a call of the callback on the right thing (it is the `ZWaveBinarySensor` thing in our case)

Further, we create an instance of `ThingMLCallback` that holds references to the callback `value_change_binarysensor_callback` and `ZWaveBinarySensor` thing (`_instance`), i.e. `ThingMLCallback* value_changed = new ThingMLCallback(value_change_binarysensor_callback, _instance);`. Subsequently, `value_changed` is passed to the wrapping library, i.e. `'&bs->setValueUpdatedCallback(value_changed);'`. Now if the native (wrapped) library calls the function which we have implemented in the wrapping library, i.e. `void valueupdate(int)` (see the point 2), the wrapper calls the callback function `value_change_binarysensor_callback()`. Finally, we can extract a value passed to the callback using `va_list` and `va_arg` in function `value_change_binarysensor_callback()` as well as we can use any ThingML instructions.

in Java

In Java, the easiest way to call ThingML code from a plain Java class is to

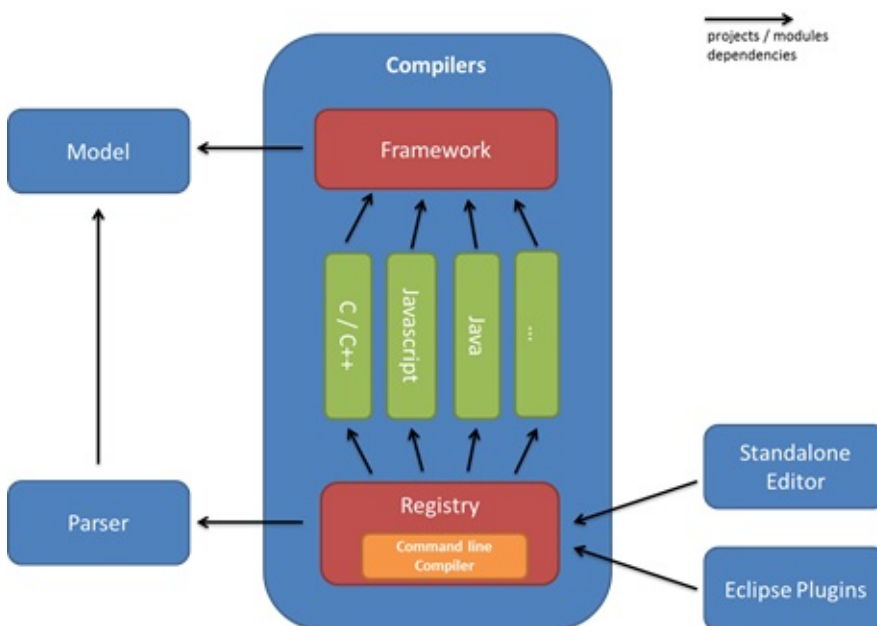
1. Make the Thing extend a Java interface. This is realized by annotating the thing: `thing MyThing @java_interface "my.package.MyInterface"`. The methods defined in this Java interface should be implemented in the thing. As ThingML functions are private by default, the function corresponding to the Java methods to be implemented need to be annotated: `function myFunction()@override "true"`. This function needs to have the exact same signature as the one defined in the Java interface.
2. In the external Java class, import the Java interface, define a pointer to that interface (a reference or a list), and call the methods of that interface in the Java class
3. Implement a registration mechanism in the Java class, so that I can actually call the class generated from the thing (and extending the interface). This can typically be done by defining an extra argument (typed by the interface) in the constructor. The plain Java class can then be created from the thing as follows: `'new my.package.MyClass(this)'`. The Java class can then hold a reference to the thingml object (`this`) and call methods on it.

Extend the ThingML compilers to target a new platform

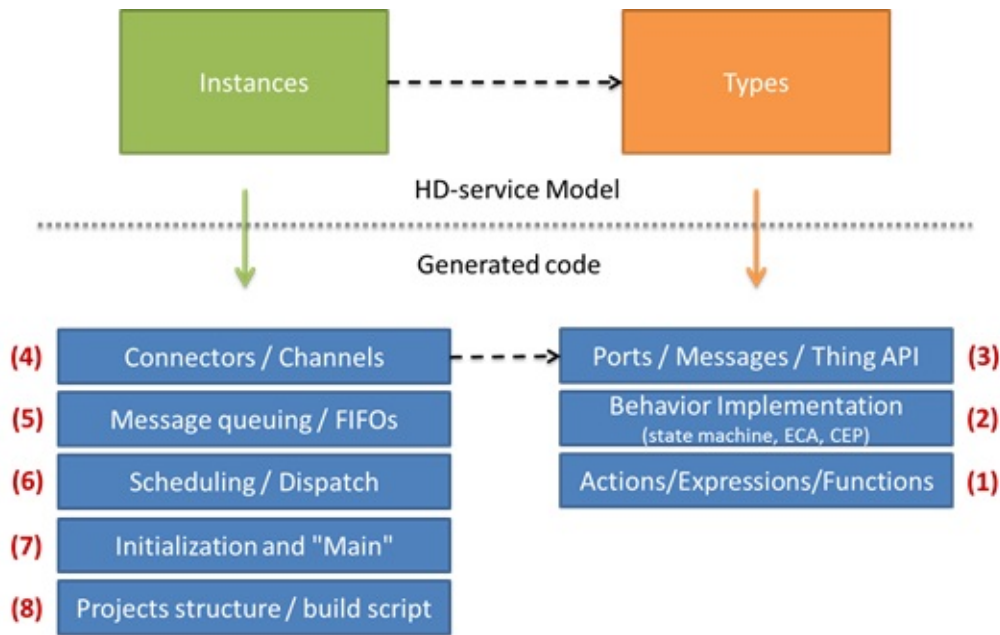
When a platform expert needs to address a new platform, he typically needs to use the HEADS transformation framework, described in detail in D2.2. This modular Object-Oriented framework defines a set of extension points, each encapsulated as a Java class. For each different language (e.g. Java, JavaScript and C) each of these extension point typically needs to be redefined. For different dialects of the same language (e.g. C for Linux and C for the Arduino microcontrollers), the platform expert can reuse already defined extension point and simply redefine a few of them to finely customize the code that is generated and accomodate with the constraints and specificities of the new platform.

Overview of the HEADS transformation framework

The HEADS code generation framework is structured in a set of modules. The figure 20 below the main sub-modules of the "Compilers" project as well as their dependencies. The idea is to have a compilation framework on top which only depends on the HEADS Model. This framework project, detailed later in this section, captures all the code and helpers to be shared between compilers. It also defines the interfaces (as abstract classes) for all the compilers. Below, individual modules correspond to the implementation of different families of compilers (Java, JavaScript, C, etc). The idea of these modules is to package together sets of compilers which have the same target languages (and typically share quite a lot of code). Finally, one the bottom, the registry module puts together all the compilers and provides a simple utility to execute them from the command line.



The idea of the code generation framework is to provide a way to independently customize different extension points. The figure below presents the 8 different extension points we have identified. Current implementation of the framework supports customizing all those extension points. However, at this point all developers are encouraged to propose and implement refactoring in order to make the APIs clear and as decoupled as possible.



The figure above presents the 8 extension points of the HEADS code generation framework. These extension points are separated in two groups: the ones corresponding to the generation of code for Types or "Things" and the ones corresponding to the generation of code for the Instances or Configuration.

Actions / Expressions / Functions

This part of the code generator corresponds to the code generated for actions, expressions and functions contained in a Thing. The generated code mostly depends on the language supported by the target platform (C, Java, etc.), and the code generators should be quite reusable across different platforms supporting the same language. The implementation of this extension point consists of a visitor on the Actions and Expressions part of the metamodel. New code generators can be created by inheriting from that abstract visitor and implementing all its methods. Alternatively, if only a minor modification of an existing code generator is needed, it is possible to inherit from the existing visitor and only override a subset of its methods.

Behavior Implementation

This part of the code generator corresponds to the code generated from the state machine structures, ECA and CEP rules contained in Things. There are main strategies and frameworks available in the literature in order to implement state machines. Depending on the capabilities, languages and libraries available on the target platform, the platform expert should have the flexibility of specifying how the behaviour is mapped to executable code. In some cases, the code generator can produce the entire code for the state machines, for example using a state machine design pattern in C++ or Java, and in other cases the code generator might rely on an existing framework available on the target platform, such as state.js for executing JavaScript state machines or ReactiveX for executing CEP queries in JavaScript or Java. To allow for this flexibility, the HEADS transformation framework should provide a set of helpers to traverse the different metaclasses responsible for modelling the behaviour and leave the freedom of creating new concrete generators and/or customizing existing code generator templates. In order to check the "correctness" of a particular code generator with respect to the language semantics, a set of reusable test cases has been created and should pass on any customized code generator.

Ports / Messages / Thing APIs

This part of the code generator corresponds to the wrapping of "things" into reusable components on the target platform. Depending on the target platform, the language and the context in which the application is deployed, the code generated for a "thing" can be tailored to generate either custom modules or to fit particular coding constraints or middleware to be used on the target platform. At this level, a Thing is a black box which should offer an API to send and receive messages through its ports. In practice this should be customized by the platform experts in order to fit the best practices and frameworks available on the target platform. As a best practice, the generated

modules and APIs for things should be manually usable in case the rest of the system (or part of it) is written directly in the target language. For example, in object oriented languages, a facade and the observer pattern can be used to provide an easy to use API for the generated code. In C, a module with the proper header with structures and call-backs should be generated.

Connectors / Channels

This part of the code generator is in charge of generating the code corresponding to the connectors and transporting messages from one Thing to the next. This is the client side of the APIs generated for the Things. In practice the connector can connect two things running in the same process on a single platform or things which are remotely connected through some sort of network (from a simple serial link to any point to point communication over a network stack). The way the code is generated should be tailored to the specific way messages should be serialized, transmitted and de-serialized. In order to customize this part of the code generator, the HEADS framework offers a set of helpers which allow listing all messages to be transported and pruning unused messages in order to generate only the necessary code. The dispatch and queuing of the messages has been separated out from the serialization and transport in order to allow for more flexibility.

Message Queuing / FIFOs

This part of the generator is related to the connectors and channels but is specifically used to tailor how messages are handled when the connectors are between two things running on the same platform. When the connectors are between things separated by a network or some sort of inter-process communication, the asynchronous nature of messages is ensured by construction. However, inside a single process specific additional code should be generated in order to store messages in FIFOs and dispatch them asynchronously. Depending on the target platform, the platform expert might reuse existing message queues provided by the operating system or a specific framework. If no message queuing service is available, like on the Arduino platform for example, the code for the queues can be fully generated.

Scheduling / Dispatch

This part of the code generator is in charge of generating the code which orchestrates the set of Things running on one platform. The generated code should activate successively the state machines of each component and handle the dispatch of messages between the components using the channels and message queues. Depending on the target platform, the scheduling can be based on the use of operating system services, threads, an active object design pattern or any other suitable strategy.

Initialization and "Main"

This part of the code generator is in charge of generating the entry point and initialization code in order to set up and start the generated application on the target platform. The HEADS transformation framework provides some helpers to list the instances to be created, the connections to be made and the set of variables to be initialized together with their initial values.

Project structure / Build script

The last extension point is not generating code as such, but the required file structure and builds scripts in order to make the generated code well packaged and easy to compile and deploy on the target platform. The HEADS transformation framework provides access to all the buffers in which the code has been generated and allows creating the file structure which fits the particular target platform. For example, the Arduino compiler concatenates all the generated code into a single file which can be opened by the Arduino IDE. The Linux C code generator creates separate C modules with header files and generates a Makefile to compile the application. The Java and Scala code generators create Maven project and pom.xml files in order to allow compiling and deploying the generated code. The platform expert can customize the project structure and build scripts in order to fit the best practices of the target platform.

How to write a (family of) compiler(s)?

For the different extension point we have presented earlier, we will use concrete compilers that we have implemented to show how to write your own compiler.

Actions / Expressions / Functions

To illustrate the HEADS Action compiler and show how to implement a family of compilers, we will take the example of the C family, composed of two compilers: Linux/POSIX and Arduino. Those two compilers share most of their code and re-define a few extension points for the parts where they differ.

The HEADS action language is fairly aligned with common programming languages, such as Java, C or JavaScript. Most of the actions and expressions can actually be factorized in a generic class:

```
public class CommonThingActionCompiler extends ThingActionCompiler {
    @Override
    public void generate(ConditionalAction action, StringBuilder builder, Context ctx) {
        builder.append("if(");
        generate(action.getCondition(), builder, ctx);
        builder.append(") {\n");
        generate(action.getAction(), builder, ctx);
        builder.append("\n");
        if (action.getElseAction() != null) {
            builder.append(" else {\n");
            generate(action.getElseAction(), builder, ctx);
            builder.append("\n");
        }
        builder.append("\n");
    }

    @Override
    public void generate(LoopAction action, StringBuilder builder, Context ctx) {
        builder.append("while(");
        generate(action.getCondition(), builder, ctx);
        builder.append(") {\n");
        generate(action.getAction(), builder, ctx);
        builder.append("\n}\n");
    }

    @Override
    public void generate(PlusExpression expression, StringBuilder builder, Context ctx) {
        generate(expression.getLhs(), builder, ctx);
        builder.append(" + ");
        generate(expression.getRhs(), builder, ctx);
    }

    @Override
    public void generate(MinusExpression expression, StringBuilder builder, Context ctx) {
        generate(expression.getLhs(), builder, ctx);
        builder.append(" - ");
        generate(expression.getRhs(), builder, ctx);
    }

    ...
}
```

The ActionCompiler class basically defines a method for each of the concepts of the HEADS action language. It is thus possible to organize a hierarchy of sub-classes that gradually redefine those methods. For example, the general C compiler just need to redefine some methods (9 in total):

```
public abstract class CThingActionCompiler extends CommonThingActionCompiler {
    @Override
    public void generate(BooleanLiteral expression, StringBuilder builder, Context ctx) {
        if (expression.isBoolValue())
            builder.append("1");
        else
            builder.append("0");
    }
    ...
}
```

Finally, the Linux/POSIX compiler only needs to redefine two methods related to printing on the standard/error output:

```
public class CThingActionCompilerPosix extends CThingActionCompiler {

    @Override
    public void generate(ErrorAction action, StringBuilder builder, Context ctx) {
        final StringBuilder b = new StringBuilder();
        generate(action.getMsg(), b, ctx);
        builder.append("fprintf(stderr, " + b.toString() + ");\n");
    }

    @Override
    public void generate(PrintAction action, StringBuilder builder, Context ctx) {
        final StringBuilder b = new StringBuilder();
        generate(action.getMsg(), b, ctx);
        builder.append("fprintf(stdout, " + b.toString() + ");\n");
    }

}
```

The same goes for the Arduino compiler:

```
public class CThingActionCompilerArduino extends CThingActionCompiler {

    @Override
    public void generate(ErrorAction action, StringBuilder builder, Context ctx) {
        final StringBuilder b = new StringBuilder();
        generate(action.getMsg(), b, ctx);

        builder.append("// PRINT ERROR: " + b.toString());
    }

    @Override
    public void generate(PrintAction action, StringBuilder builder, Context ctx) {
        final StringBuilder b = new StringBuilder();
        generate(action.getMsg(), b, ctx);
        if (ctx.getCurrentConfiguration().hasAnnotation("arduino_stdout")) {
            builder.append(ctx.getCurrentConfiguration().annotation("arduino_stdout").iterator().next() + ".print(" +
b.toString() + ");\n");
        } else {
            builder.append("// PRINT: " + b.toString());
        }
    }

}
```

In case a platform expert wants to target a language that is very different from the Java/C/JavaScript family (e.g LISP using a prefix/Polish notation where a typical/infix `a + b` would be expressed `+ a b`), he might need to redefine all the concepts, directly by inheriting from the top class `ThingActionCompiler`

Behavior Implementation

Different approaches exist when it comes to the compilation of the behavior (mostly state machine-based, with optional extensions for CEP):

- target and existing framework
- generate all code from scratch, including the code for how to dispatch event to concurrent regions, etc

Both approaches have pros and cons. Using a framework typically reduces the size and complexity of the code to be generated (and of the compilers), as most of the code is directly written in the framework. However, frameworks tend to be generic and might typically include more than what is needed, hence have a larger overhead. The full generative approach gives more flexibility and makes it possible to control each bits and bytes, and optimize the code for a particular state machine (whereas the framework needs to handle any possible state machine), but are typically more complex to implement.

The Java and JavaScript behavior compilers use a framework-based approach, which is rather idiomatic for those language, whereas the C compiler, which is expected to generate code able to run down to small micro-controllers (2KB RAM) uses a full generative approach to avoid any accidental overhead.

Because of the diversity of solutions that can be implemented for this extension point, the high level interface is rather generic so as not to constrain the platform expert:

```
public class ThingImplCompiler {

    public void generateImplementation(Thing thing, Context ctx) {

    }

}
```

Platform experts are however encouraged to implement the `generateImplementation` method in a modular way, split into several sub-methods.

The following code snippet instantiate a composite state by using the `state.js` JavaScript library:

```
protected void generateCompositeState(CompositeState c, StringBuilder builder, Context ctx) {
    String containerName = ctx.getContextAnnotation("container");
    if (c.hasSeveralRegions()) {
        builder.append("var " + c.qname("_") + " = new StateJS.Region(\"\" + c.getName() + "\", " + containerName + ");\n");
        builder.append("var " + c.qname("_") + "_default = new StateJS.Region(\"_default\", " + c.qname("_") + ");\n");
        if (c.isHistory())
            builder.append("var _initial_ + c.qname("_") + " = new StateJS.pseudoState(\"_initial\", " + c.qname("_") + ", StateJS.PseudoStateKind.ShallowHistory);\n");
        else
            builder.append("var _initial_ + c.qname("_") + " = new StateJS.pseudoState(\"_initial\", " + c.qname("_") + ", StateJS.PseudoStateKind.Initial);\n");
        builder.append("_initial_ + c.qname("_") + ".to(" + c.getInitial().qname("_") + ");\n");
        for (State s : c.getSubstate()) {
            ctx.addContextAnnotation("container", c.qname("_") + "_default");
            generateState(s, builder, ctx);
        }
        for (Region r : c.getRegion()) {
            ctx.addContextAnnotation("container", c.qname("_"));
            generateRegion(r, builder, ctx);
        }
    } else {
        builder.append("var " + c.qname("_") + " = new StateJS.State(\"\" + c.getName() + "\", " + containerName + ");\n");
        generateActionsForState(c, builder, ctx);
        builder.append("\n");
        for (State s : c.getSubstate()) {
            ctx.addContextAnnotation("container", c.qname("_"));
            generateState(s, builder, ctx);
        }
    }
    if (c.isHistory())
        builder.append("var _initial_ + c.qname("_") + " = new StateJS.PseudoState(\"_initial\", " + c.qname("_") + ", StateJS.PseudoStateKind.ShallowHistory);\n");
    else
        builder.append("var _initial_ + c.qname("_") + " = new StateJS.PseudoState(\"_initial\", " + c.qname("_") + ", StateJS.PseudoStateKind.Initial);\n");
    builder.append("_initial_ + c.qname("_") + ".to(" + c.getInitial().qname("_") + ");\n");
}
```

Based on its extensive suite of tests, the HEADS transformation framework was able to detect a few bugs in the popular [state.js library](#) (~200 likes on GitHub and ~1000 Download a month on NPM), that were rapidly fixed by the repository maintainer.

Ports / Messages / Thing APIs

The goal of this extension point is to generate proper interface so that the generated code can easily be used and integrated by third-parties, using or not the HEADS technologies. For example a timer component which can receive two messages `timer_start` and `timer_cancel` on a port timer and can emit a `timer_timeout` message on a port timer can be addressed in Java through a couple of interface (the second one serving as a callback):

```
public interface ITimerJava_timer{
    void timer_start_via_timer(short TimerMsgs_timer_start_delay__var);
    void timer_cancel_via_timer();
}

public interface ITimerJava_timerClient{
    void timer_timeout_from_timer();
}
```

A third-party wanting to use this simple HEADS-enabled timer would thus, in plain Java implement `ITimerJava_timerClient` interface, and after instantiating a timer, register as a listener:

```
TimerJava timer = new TimerJava().buildBehavior();
timer.registerOnTimer(new ITimerJava_timerClient(){
    @Override
    timer_timeout_from_timer(){
        System.out.println("timeout!");
    }
});
timer.init();
timer.start();
timer.timer_start_via_timer(5000);//timeout! to be displayed in 5000 ms
```

Similarly in JavaScript:

```
// Public methods on the timer
TimerJS.prototype.timer_startOntimer = function(delay) {
    ...
};

TimerJS.prototype.timer_cancelOntimer = function() {
    ...
};

var timer = new TimerJS();
timer.build();
timer.getTimer_timeoutontimerListeners().push(function(){console.log("timeout!");});
timer.init();
timer.timer_startOntimer(5000);//timeout! to be displayed in 5000 ms
```

And in C:

```
void TimerLinux_handle_timer_timer_start(struct TimerLinux_Instance *_instance, int delay);
void TimerLinux_handle_timer_timer_cancel(struct TimerLinux_Instance *_instance);
void register_external_TimerLinux_send_timer_timer_timeout_listener(void (*_listener)(struct TimerLinux_Instance *));

void printCallBack(){
    fprintf("timeout!\n");
}

struct TimerLinux_Instance TestTimerLinux_timer_var;
register_TimerLinux_send_timer_timer_timeout_listener(&printCallBack);
TimerLinux_handle_timer_timer_start(&TestTimerLinux_timer_var, 5000);//timeout! to be displayed in 5000 ms
```

As this code is only structural (basically a set of methods), it is fairly easy to generate. Here is how we generate Java interfaces of components:

```
//Generate interfaces that the thing will implement, for others to call this API
for (Port p : thing.allPorts()) {
    if (!p.isDefined("public", "false") && p.getReceives().size() > 0) {
        final StringBuilder builder = ctx.getBuilder(src + "/api/I" + ctx.firstToUpper(thing.getName()) + "_" + p
.getName() + ".java");
        builder.append("package " + pack + ".api;\n\n");
        builder.append("import " + pack + ".api.*;\n\n");
        builder.append("public interface " + "I" + ctx.firstToUpper(thing.getName()) + "_" + p.getName() + "{\n");
;
        for (Message m : p.getReceives()) {
            builder.append("void " + m.getName() + "_via_" + p.getName() + "(");
            JavaHelper.generateParameter(m, builder, ctx);
            builder.append(");\n");
        }
        builder.append("}");
    }
}
}
```

Connectors / Channels

In HEADS, connectors and channels are managed by the HEADS runtime. By default the generated code is "standalone" and can be run without the HEADS runtime. To be able to run on the HEADS runtime, this require some wrappers around the implementation. Those wrappers are generated and only interact with the public interface of the component (as a developer would normally write). For example, the following code generates for the JavaScript HEADS runtime. A more conceptual view of this wrapping is provided in D2.2.

```
private void generateWrapper(Context ctx, Configuration cfg) {
    final StringBuilder builder = ctx.getBuilder(cfg.getName() + "/lib/" + cfg.getName() + ".js");
    builder.append("var AbstractComponent = require('kevoree-entities').AbstractComponent;\n");

    for (Thing t : cfg.allThings()) { //load of the fined-grained component into the coarse grained component
        builder.append("var " + t.getName() + " = require('./" + t.getName() + "');\n");
    }

    builder.append("/**\n* Kevoree component\n* @type {" + cfg.getName() + "}\n*/\n");
    builder.append("var " + cfg.getName() + " = AbstractComponent.extend({\n");
    builder.append("toString: " + cfg.getName() + ";\n");

    builder.append("construct: function() {\n");
    JSCfgMainGenerator.generateInstances(cfg, builder, ctx, true);
    for (Map.Entry e : cfg.danglingPorts().entrySet()) {
        final Instance i = (Instance) e.getKey();
        for (Port p : (List<Port>) e.getValue()) {
            for (Message m : p.getSends()) {
                builder.append("this." + i.getName() + ".get" + ctx.firstToUpper(m.getName()) + "on" + p.getName(
) + "Listeners().push(this." + shortName(i, p, m) + "_proxy.bind(this));\n");
            }
        }
    }
    builder.append("},\n\n");

    builder.append("start: function (done) {\n");
    for (Instance i : cfg.danglingPorts().keySet()) {
        builder.append("this." + i.getName() + "._init();\n");
    }
    builder.append("done();\n");
    builder.append("},\n\n");

    builder.append("stop: function (done) {\n");
    for (Instance i : cfg.allInstances()) {
        builder.append("this." + i.getName() + "._stop();\n");
    }
    builder.append("done();\n");
    builder.append("}");
}
```

```

    for (Map.Entry e : cfg.danglingPorts().entrySet()) {
        final Instance i = (Instance) e.getKey();
        for (Port p : (List<Port>) e.getValue()) {
            for (Message m : p.getReceives()) {
                builder.append("\nin_" + shortName(i, p, m) + "_in: function (msg) {\n");
                builder.append("this." + i.getName() + ".receive" + m.getName() + "On" + p.getName() + "(msg.spli
t(';'));");
                builder.append("}");
            }
        }
    }

    for (Map.Entry e : cfg.danglingPorts().entrySet()) {
        final Instance i = (Instance) e.getKey();
        for (Port p : (List<Port>) e.getValue()) {
            for (Message m : p.getSends()) {
                builder.append("\n" + shortName(i, p, m) + "_proxy: function() {this.out_" + shortName(i, p, m)
+ "_out(");
                int index = 0;
                for (Parameter pa : m.getParameters()) {
                    if (index > 0)
                        builder.append(" + ';' + ");
                    builder.append("arguments[" + index + "]");
                    index++;
                }
                if (index > 1)
                    builder.append("''");
                builder.append("});");
                builder.append("\nout_" + shortName(i, p, m) + "_out: function(msg) {/* This will be overwritten
@runtime by Kevoree JS */}");
            }
        }
    }
    builder.append("});\n\n");
    builder.append("module.exports = " + cfg.getName() + ";\n");
}

```

Message queuing / Scheduling / Dispatch

This extension point allows customizing the code generated for handling how the messages and the control are distributed among of set of component instances. From a semantic point of view, each component instance is an independent process which exchanges messages with other components in an asynchronous way. To implement this semantic, a wide range of alternatives for queuing messages, distributing them to the components can be used depending on the capabilities of the targeted platforms. Features for message exchange and multi-tasking are typically provided by operating systems or middleware platforms. In the case of resource constrained devices with no operating system, code has to be generated to fully handle the scheduling and message dispatch between components.

The API for customizing the code generator for those aspect is in class "org.thingml.compilers.configuration.CfgMainGenerator" and its sub-classes for the different platforms.

The example bellow shows how messages are queued when generating code for microcontrollers. The generated code includes a compact FIFO implementation for storing messages. The messages are serialized in the FIFO when they are emitted by a component and later processed and dispatched to the receiving components.

```
// Enqueue of messages HelloTimer::timer::timer_start
void enqueue_HelloTimer_send_timer_timer_start(struct HelloTimer_Instance *_instance, int delay){
    if ( fifo_byte_available() > 6 ) {

        _fifo_enqueue( (3 >> 8) & 0xFF );
        _fifo_enqueue( 3 & 0xFF );

        // ID of the source port of the instance
        _fifo_enqueue( (_instance->id_timer >> 8) & 0xFF );
        _fifo_enqueue( _instance->id_timer & 0xFF );

        // parameter delay
        union u_delay_t {
            int p;
            byte bytebuffer[2];
        } u_delay;
        u_delay.p = delay;
        _fifo_enqueue( u_delay.bytebuffer[1] & 0xFF );
        _fifo_enqueue( u_delay.bytebuffer[0] & 0xFF );
    }
}
```

The following listing shows how the messages are dispatched from the FIFO to the appropriate component.

```
void processMessageQueue() {
    if (fifo_empty()) return; // return if there is nothing to do

    byte mbuf[4];
    uint8_t mbufi = 0;

    // Read the code of the next port/message in the queue
    uint16_t code = fifo_dequeue() << 8;

    code += fifo_dequeue();

    // Switch to call the appropriate handler
    switch(code) {
        case 2:
            while (mbufi < 2) mbuf[mbufi++] = fifo_dequeue();
            dispatch_timer_cancel((mbuf[0] << 8) + mbuf[1] /* instance port*/);
            break;
        case 3:
            while (mbufi < 4) mbuf[mbufi++] = fifo_dequeue();
            union u_timer_start_delay_t {
                int p;
                byte bytebuffer[2];
            } u_timer_start_delay;
            u_timer_start_delay.bytebuffer[1] = mbuf[2];
            u_timer_start_delay.bytebuffer[0] = mbuf[3];
            dispatch_timer_start((mbuf[0] << 8) + mbuf[1] /* instance port*/,
                u_timer_start_delay.p /* delay */ );
            break;
        case 1:
            while (mbufi < 2) mbuf[mbufi++] = fifo_dequeue();
            dispatch_timer_timeout((mbuf[0] << 8) + mbuf[1] /* instance port*/);
            break;
    }
}
```

In the Arduino code generator, the main loop of the scheduler simply activates the components which use polling and processes messages from the queue. The component receiving a message is given the CPU for processing this message. Any message produced by the component is queued and will be later processed by the receiver. This strategy ensures that each component gets activated in turn and that the processing of a message is executed as a whole. In the case of microcontrollers, it can be interrupted by microcontroller interrupts but not by the processing of another message.

```
void loop() {
    TimerArduino_handle_Polling_poll(&TestTimerArduino_timer_var);
    HelloTimer_handle_empty_event(&TestTimerArduino_client_var);
    processMessageQueue();
}
```

Depending on the level of dynamicity required, code can be generated statically for one particular configuration (and set of connector), but even on tiny and small targets code can be generated to handle dynamically dispatching messages according to a dynamic set of connectors. The code bellow illustrates how it is done in the Arduino compiler.

```
//Dynamic dispatch for message timer_start
void dispatch_timer_start(uint16_t sender, int param_delay) {
void executor_dispatch_timer_start(struct Msg_Handler ** head, struct Msg_Handler ** tail) {
struct Msg_Handler ** cur = head;
while (cur != NULL) {
    void (*handler)(void *, int param_delay) = NULL;
    int i;
    for(i = 0; i < (**cur).nb_msg; i++) {
        if(**cur).msg[i] == 2) {
            handler = (void (*) (void *, int)) (**cur).msg_handler[i];
            break;
        }
    }
    if(handler != NULL) {
        handler(**cur).instance, param_delay);
    }
    if(cur == tail){
        cur = NULL;}
    else {
        cur++;}
}
}
if (sender == TestTimerC_client_var.id_timer) {
executor_dispatch_timer_start(TestTimerC_client_var.timer_receiver_list_head, TestTimerC_client_var.timer_receiver_list_tail);}
}
```

Initialization and "Main"

This extension point is responsible for instantiating components and properly set the attributes of these instances with proper values. It is also responsible for connecting instances together. Here is an example of a "main" in JavaScript:

```
//import types
var TimerJS = require('./TimerJS');
var SimpleTimerClient = require('./SimpleTimerClient');

//Create and initialize instances
var TestTimerJS_timer = new TimerJS("TestTimerJS_timer", false);
TestTimerJS_timer.setThis(TestTimerJS_timer);
TestTimerJS_timer.build();
var TestTimerJS_client = new SimpleTimerClient("TestTimerJS_client", 1000, 5000, true);
TestTimerJS_client.setThis(TestTimerJS_client);
TestTimerJS_client.build();

//Connect instances together
TestTimerJS_timer.getTimer_timeoutontimerListeners().push(TestTimerJS_client.receive_timer_timeoutontimer.bind(TestTimerJS_client));
TestTimerJS_client.getTimer_startontimerListeners().push(TestTimerJS_timer.receive_timer_startontimer.bind(TestTimerJS_timer));
TestTimerJS_client.getTimer_cancelontimerListeners().push(TestTimerJS_timer.receive_timer_cancelontimer.bind(TestTimerJS_timer));

//start instances
TestTimerJS_timer._init();
TestTimerJS_client._init();

//register hookup to properly stop instances
process.on('SIGINT', function () {
    console.log("Stopping components...");
    TestTimerJS_timer._stop();
    TestTimerJS_client._stop();
});
```

The platform expert needs to extend the following class to generate the main:

```
public class CfgMainGenerator {
    public void generateMainAndInit(Configuration cfg, ThingMLModel model, Context ctx) {
    }
}
```

The following code snippet illustrates how to generate the code for the connectors:

```
for (Connector c : cfg.allConnectors()) {
    for (Message req : c.getRequired().getReceives()) {
        for (Message prov : c.getProvided().getSends()) {
            if (req.getName().equals(prov.getName())) {
                builder.append(prefix + c.getSrv().getInstance().getName() + ".get" + ctx.firstToUpper(prov.getName()) + "on" + c.getProvided().getName() + "Listeners().push(");
                builder.append(prefix + c.getCli().getInstance().getName() + ".receive" + req.getName() + "on" + c.getRequired().getName() + ".bind(" + prefix + c.getCli().getInstance().getName() + ")");
                builder.append(");\n");
                break;
            }
        }
    }
    for (Message req : c.getProvided().getReceives()) {
        for (Message prov : c.getRequired().getSends()) {
            if (req.getName().equals(prov.getName())) {
                builder.append(prefix + c.getCli().getInstance().getName() + ".get" + ctx.firstToUpper(prov.getName()) + "on" + c.getRequired().getName() + "Listeners().push(");
                builder.append(prefix + c.getSrv().getInstance().getName() + ".receive" + req.getName() + "on" + c.getProvided().getName() + ".bind(" + prefix + c.getSrv().getInstance().getName() + ")");
                builder.append(");\n");
                break;
            }
        }
    }
}
```

Project structure / Build script

To make the HEADS components easily reusable, with or without the HEADS technologies, they must be properly packaged. For Java, we for example generate proper Maven projects, for JavaScript, NPM projects, and for C, Makefiles. If a platform expert would like to use Gradle instead of Maven, he would need to re-define the following extension point:

```
public class CfgBuildCompiler {

    public void generateBuildScript(Configuration cfg, Context ctx) {
        throw (new UnsupportedOperationException("Project structure and build scripts are platform-specific.));
    }
}
```

Here is for example how we generate a `package.json` for NPM projects:


```

public class JSCfgBuildCompiler extends CfgBuildCompiler {

    @Override
    public void generateBuildScript(Configuration cfg, Context ctx) {
        try {
            final InputStream input = this.getClass().getClassLoader().getResourceAsStream("javascript/lib/package.js
on");

            final List<String> packLines = IOUtils.readLines(input);
            String pack = "";
            for (String line : packLines) {
                pack += line + "\n";
            }
            input.close();
            pack = pack.replace("<NAME>", cfg.getName());

            final JsonObject json = JsonObject.readFrom(pack);
            final JsonValue deps = json.get("dependencies");
            for (Thing t : cfg.allThings()) {
                for (String dep : t.annotation("js_dep")) {
                    deps.asObject().add(dep.split(":")[0].trim(), dep.split(":")[1].trim());
                }
            }

            boolean addCEPdeps = false;
            boolean addDebugDeps = !ctx.getCompiler().getDebugProfiles().isEmpty();

            for (Thing t : cfg.allThings()) {
                if (t.getStreams().size() > 0) {
                    addCEPdeps = true;
                }
            }

            if(addCEPdeps) {
                deps.asObject().add("rx", "^2.5.3");
                deps.asObject().add("events", "^1.0.2");
            }

            if(addDebugDeps) {
                deps.asObject().add("colors", "^1.1.2");
            }

            final File f = new File(ctx.getOutputDirectory() + "/" + cfg.getName() + "/package.json");
            f.setWritable(true);
            final PrintWriter w = new PrintWriter(new FileWriter(f));
            w.println(json.toString());
            w.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

This would produce this kind of output:

```
{
  "name" : "TestTimerJS",
  "version" : "1.0.0",
  "description" : "TestTimerJS configuration generated from ThingML",
  "main" : "main.js",
  "private" : true,
  "dependencies" : {
    "state.js" : "^5.3.4",
    "colors" : "^1.1.2"
  },
  "devDependencies" : {},
  "scripts" : {}
}
```

Note that this compiler (as well as others) uses a template to simplify the code generation, since most of the content of `package.json` is fixed:

```
{
  "name": "<NAME>",
  "version": "1.0.0",
  "description": "<NAME> configuration generated from ThingML",
  "main": "main.js",
  "private": true,
  "dependencies": {
    "state.js": "^5.3.4"
  },
  "devDependencies": {
  },
  "scripts": {
  }
}
```

Lightweight extension to existing compilers

When new target languages, operating systems or core libraries need to be supported, the platform expert has to extend the ThingML compilers/transformation. The ThingML compilers are modular so that different parts can be reused and extended separately.

ThingML supports for adding annotations on most elements of the language. The platform expert can define specific annotations which are exploited in the code generator in order to support platform specific features.

For example, a thing dealing with IO typically needs to listen continuously for inputs to arrive. This behavior should be executed in a separate thread so that it does not block or slow down the execution of the core business logic. This multi-threaded behavior can be achieved in the Linux/C compiler using the `@fork_linux_thread` annotation.

```
function serial_receiver_process()
@fork_linux_thread "true"
do
  var buffer : Byte[256] // Data read from the serial port
  while (true) {
    //read bytes from serial port
  }
end
```

The C compiler will interpret this annotation and generate multi-threaded code, which [wraps the code](#) normally generated by the compiler without that annotation:

```
if (func.isDefined("fork_linux_thread", "true")) {
  generateCforThingLinuxThread(func, thing, builder, ctx);
} else { // Use the default function generator
  generateCforThingDirect(func, thing, builder, ctx);
}
```

If an annotation is intensively used and relevant for most compilers, the concept can be promoted directly into the language so that it can benefit from better tool support (annotations being simple string-based key/value entries). The extension of the HEADS modelling language is however beyond the scope of the HEADS project, but interested reader can read about the way we extended the language to support Complex Event Processing.

Typically, a ThingML component can be required to communicate with three type of components:

- **ThingML system:** In some situations, a developer can model applications with ThingML for both ends of the communication. In this case, communication stacks can be fully generated. The code generation framework must be extensible in order to edibility on the choice of the transport protocol.
- **Open systems:** In some other case, a ThingML component needs to communicate with another software component, whose sources are accessible to the developer. It is either possible to adapt the ThingML end, or to adapt the other one by generating code in its language. In addition to the support of a specific protocol, the code generation framework must also be adaptable in terms of message encoding. Indeed, the encoding must be understandable by the non-ThingML part of the application, or this part must be changed to understand ThingML messages.
- **Proprietary systems:** In many cases a ThingML component will have to communicate with external closed-source components, whose implementation cannot be modified. In this case, the only option is to adapt the ThingML end both in terms of encoding and transport.

Regardless of the situation, code needs to be generated (at least for the ThingML end) in order to handle the following tasks:

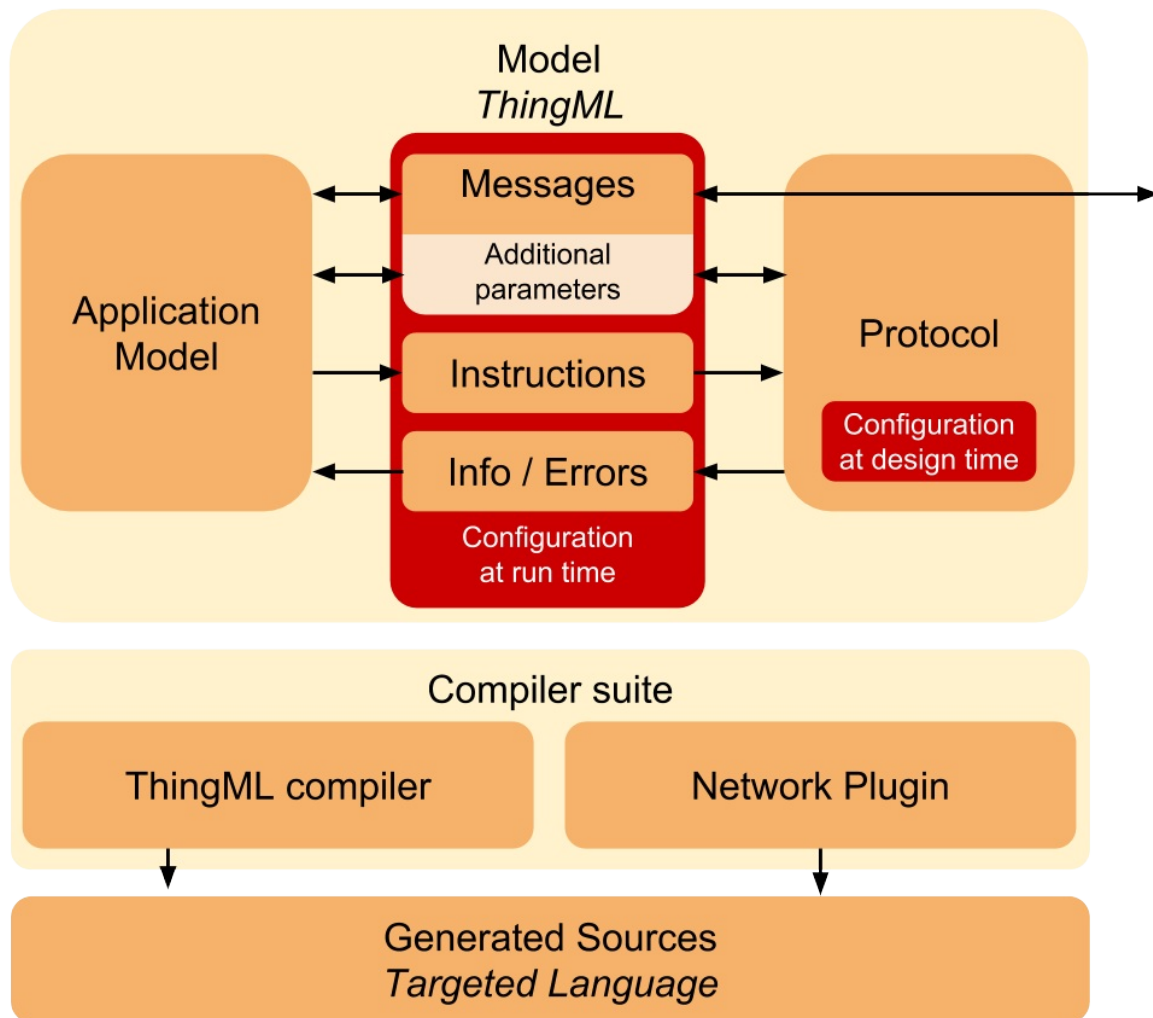
- **Encoding/Decoding:** In order to exchange data between different platforms, a serialization scheme must be chosen. While there exist numerous solutions and standards, many distributed applications need to leverage several of them. Indeed, the choice among them can be driven by various concerns (bandwidth, human readability). Furthermore, it is not always a choice, in the case where a component needs to communicate with a proprietary system, which already fixed the format for some exchanges.
- **Sending/Receiving:** Code must be generated to support both message emission and reception adapted to the targeted protocol. For most cases these operations rely heavily on a pre-existing library in the targeted language. But it can require additional aspects depending on the paradigm. For example, for synchronous communications, a message can not necessarily be sent any time, and some queuing might be required.
- **Configuration and Link management:** Before any message exchange occurs, a network interface has typically to be configured, and depending on the protocol a connection might have to be established. Furthermore, some network paradigms require some additional logic such as keeping track of connected clients.

Network Plugins

Through a set of experiments (Serial, HTTP/REST, MQTT, WebSocket, Bluetooth Low Energy, Z-Wave, etc.) it quickly appears that a part of the complexity coming from these various communication protocols can be hidden from the model, because they are not relevant to the application logic (Automatic re-sending of messages, connection establishment, a part of error management, encryption).

Network plugin can generate code handling these "technicalities". Meanwhile, a part of this complexity needs to be exposed to the ThingML code to enable a communication management flexible enough. In order to do so, the ThingML code generation framework differentiates two aspects of communication management (See Figure 10):

- **Configuration:** Through the mean of annotation on the ThingML keyword protocol, configuration at design time can be provided to the code generator in order to generate a tailor-made network interface.
- **Control:** A part of the communication handling (such as error management, client management) is inherently dynamic and hence the need for information exchange between the application logic and the generated network library at run time.



Two broad types of annotations can be used to control more precisely a generated library at run time. Annotated messages can be transformed into instructions addressed at the generated network library and not forwarded (for example, a reconnection instruction). Similarly, annotated messages can become feedback originating from the network library and transmitted to the application (for example if the connection has been lost). But additional parameters (that will not be forwarded) can also be added to messages in order to customize at run time the way those messages are forwarded (for example destination information).

```
thing fragment myMsgs { //Control at run time
  message reconnect() @websocket_instruction "reconnect";
  message connection_lost() @websocket_feedback "connection_lost";
  message msg1(Param : Float, ClientID : UInt16)
    @code "101" //ID for serialization purposes
    @websocket_client_id "ClientID";
}
protocol Websocket //Configuration at Design time
@websocket_server "true"
@websocket_max_client "16"
@websocket_enable_unicast "true"
@serialization "msgpack"

configuration myCfg {
  instance i : myThing // The thing myThing can send
  // messages through websocket in a
  connector i.myPort over Websocket //transparent way
}
```

This architecture offers a trade-off between abstraction and customizability. It allows to hide or expose parts of the communication paradigm on demand, depending on what the application requires. It provides a way to extend the ThingML code generation framework for message exchanges, while relying if necessary on serialization plugins described in the following section.

Serialization Plugins

Serialization plugins are in charge of generating both the serialization and parsing of a set of ThingML messages. Note that serialization and parsing are not necessarily perfect mirrors, as ports can be asymmetrical. ThingML provides a default binary and string-based (JSON) serialization and deserialization, available and interoperable in C, JavaScript and Java. A number of experiment has also been conducted on a number of proprietary protocols, such as the Multiwii Serial Protocol used in some drone flight controllers. In order to be flexible enough, the code generation framework must provide a way to integrate rapidly existing native library to enable the use of standard serialization schemes. But it also needs to support custom/proprietary communication protocols, by modelling their existing messages in ThingML and generating code compatible with non-modifiable software and hardware components. Moreover, to meet the various needs of compatibility, a serialization plugin needs to be usable in two different ways:

- It can either be provided to a network plugin in order to generate fully executable code for the ThingML end.-
- It can also be used to generate the methods separately in order to integrate them directly in sources written in the targeted language.

These different usages offer more flexibility on the choice of the end to adapt. One can either adapt the ThingML end of a link, but also the end running a software component written in another language. While enabling this choice, it fits the maintainability requirement as it allows regeneration of code after modification of the set of messages.

In order to validate code generators, the code generator testing framework provides an extensible set of tests, with currently 140 tests. Those are ThingML models of programs from which expected outputs (oracle) are provided. As much as possible those model demonstrate one specific aspect of the ThingML language. Therefore, the framework is able to test a compiler, and check that these models are indeed transformed into code that produces the expected outputs. By comparing the outputs of different compilers, it is also possible to detect possible misalignment in the semantics implemented by these different compilers. The testing framework can be simply extended by adding new test models.

To accelerate the execution of the test suite *140 x number of compilers*, the execution of the tests is parallelized so as to use all the available core on your local PC. An alternative setup leverages Docker so as to deploy tests on any number of nodes *e.g.* in the cloud.

Prepare, build and flash any openWRT device to run Heads Artefacts

What is OpenWrt?

OpenWrt is described as a Linux distribution for embedded devices. Instead of trying to create a single, static firmware, OpenWrt provides a fully writable filesystem with package management. This frees you from the application selection and configuration provided by the vendor and allows you to customize the device through the use of packages to suit any application. For developer, OpenWrt is the framework to build an application without having to build a complete firmware around it; for users this means the ability for full customization, to use the device in ways never envisioned.

We use in particular the lede project.

The LEDE project is a Linux-based, embedded meta-distribution based on OpenWrt, targeting a wide range of wireless SOHO routers and non-network devices. LEDE is an acronym for “Linux Embedded Development Environment”.

LEDE spun away from the mother project in May 2016, with goals of continuing to develop superior software in an open governance model and encouraging new developers to contribute to the development effort.

In this section, we will describe the different step to build a compatible firmware for the WG3526 router that has been chosen in one of the case study. The same kind of methodology can be followed for another device.

Preparing the docker environment to cross compile your firmware

First you need a docker image for crosscompiling lede firmware.

You can build it. The docker file is the following:

```
FROM ubuntu:16.04
RUN apt-get update -y && apt-get upgrade -y
RUN apt-get install git git-core build-essential libssl-dev libncurses5-dev unzip gawk file wget python svn
RUN git clone https://github.com/lede-project/source
RUN mv /opt/source /opt/lede
RUN cd /opt/lede
RUN ./scripts/feeds update -a
RUN ./scripts/feeds install -a
WORKDIR /opt/lede
CMD /bin/bash
```

You can build this image in creating a text file named Dockerfile. Put the following context and call the command

```
docker build -t barais/lede .
```

Next run the following command

```
docker run -ti barais/lede /bin/bash
#if you build your image
cd /opt/lede
```

Creating a new firmware


```
cd /opt/lede
make menuconfig
#select the correct devices.
#Select the package you need. You need at least nodejs and blockmount.
```

In particular, you need to remove support for /etc/fstab in

```
base system -> busybox -> custom busy box options -> Linux System Utilities -> Mount -> Support fstab and -a
```

You need also to include mount-utils

```
> Utilities -> mount-utils..... related (u)mount utilities
```

and

```
> Utilities -> Disc blkid..... locate and print block device attribute
```

Next configure the kernel

```
FORCE_UNSAFE_CONFIGURE=1 make kernel_menuconfig -j1 V=s
```

Select at least the option in the kernel

```
kernel FPU emulation.
Kernel Type --->
MIPS FPU EMULATOR Y
```

Do not forget to enable USBStorage NOTE: USB_STORAGE enables SCSI, and 'SCSI disk support'" "may also be needed; see USB_STORAGE Help for more information" Do not forget to enable ext4 and vfat in FS.

```

Device Drivers --->
  SCSI device support --->

## (Although SCSI will be enabled automatically when selecting USB Mass Storage,
we need to enable disk support.)
--- SCSI support type (disk, tape, CD-ROM)
<*> SCSI disk support

## (Then move back a level and go into USB support)
USB support --->

## (This is the root hub and is required for USB support.
If you'd like to compile this as a module, it will be called usbcore.)
<*> Support for Host-side USB

## (Select at least one of the HCDs. If you are unsure, picking all is fine.)
--- USB Host Controller Drivers
<*> xHCI HCD (USB 3.0) support
<*> EHCI HCD (USB 2.0) support
< > OHCI HCD support
<*> UHCI HCD (most Intel and VIA) support

## (Moving a little further down, we come to CDC and mass storage.)
< > USB Modem (CDC ACM) support
<*> USB Printer support
<*> USB Mass Storage support

## (If you have a USB Network Card like the RTL8150, you'll need this)
USB Network Adapters --->
  <*> USB RTL8150 based ethernet device support (EXPERIMENTAL)

## (If you have a serial to USB converter like the Prolific 2303, you'll need this)
USB Serial Converter support --->
  <*> USB Serial Converter support
  <*> USB Prolific 2303 Single Port Serial Driver (NEW)

## In most cases enabling RFCOMM, HIDP, HCI USB and/or HCI UART should be sufficient.

## It is also a good idea to enable the UHID (Userspace Human Interface Device) driver for Bluetooth input devices su
ch as keyboards and mice.

## Tallying up the options: CONFIG_BT, BT_BREDR, CONFIG_BT_RFCOMM, CONFIG_BT_HIDP, BT_LE, CONFIG_BT_HCIBTUSB, CONFIG_
BT_HCIUART, CONFIG_RFKILL, CONFIG_UHID

KERNEL Enable bluetooth support
[*] Networking support --->
  <*> Bluetooth subsystem support --->
    [*] Bluetooth Classic (BR/EDR) features
    <*> RFCOMM protocol support
    [ ] RFCOMM TTY support
    < > BNEP protocol support
    [ ] Multicast filter support
    [ ] Protocol filter support
    <*> HIDP protocol support
    [*] Bluetooth High Speed (HS) features
    [*] Bluetooth Low Energy (LE) features
    Bluetooth device drivers --->
      <*> HCI USB driver
      <*> HCI UART driver
    <*> RF switch subsystem support --->
  Device Drivers --->
    HID support --->
      <*> User-space I/O driver support for HID subsystem

```

Do not forget to check in the config file that is unselected. For this specific device, there is a bug with sdcard reader; See <https://dev.openwrt.org/changeset/49131>

You have also to include the following packages: bluez-libs bluez-utils usbutils.

When you are happy with your current configuration, you can build the firmware.

```
make -j1 V=s
```

If there is some questions, please answer the missing configuration part. Next, take a big big coffee. When it finishes.

Copy the bin/targets/ramips/mt7621/*.bin into your host.

```
scp bin/targets/ramips/mt7621/*.bin user@192.17.0.1:~
```

Flash the router

Next connect the router to your laptop using an ethernet cable. For the WG3526, the easiest way to update the router locally is to use a pen or something to keep the reset button pressed when starting the router. Then, when your machine gets an IP (192.168.1.X), enter 192.168.1.1, click on the big button with chinese text and choose the firmware available in your home folder. It takes a couple of minutes to flash the router, but once it is done, the router will reboot and you should get a new IP adress 192.168.1.X and everything should be fine again.

You can login to the router through ssh.

```
ssh root@192.168.1.1
```

You can connect to the luci interface [here](#) login, admin no password.

Extend the filesystem on a USB key.

First thing you have to do is to extend the File system. Plug a USB key.

You can follow this [tutorial](#)

If you put blockmount in the default package and usb in the kernel, you don-t have to install them, else install them using.

```
opkg update ; opkg install block-mount
```

```
mount /dev/sda1 /mnt ; tar -C /overlay -cvf - . | tar -C /mnt -xf - ; umount /mnt

block detect > /etc/config/fstab; \
sed -i s/option$'\t'enabled$'\t'\'0\'/option$'\t'enabled$'\t'\'1\'/ /etc/config/fstab; \
sed -i s#/mnt/sda1#/overlay# /etc/config/fstab; \
cat /etc/config/fstab;
```

You'll end up with an fstab looking something like this:

```
config 'global'
    option anon_swap      '0'
    option anon_mount     '0'
    option auto_swap      '1'
    option auto_mount     '1'
    option delay_root     '5'
    option check_fs       '0'

config 'mount'
    option target          '/overlay'
    option uuid            'c91232a0-c50a-4eae-adb9-14b4d3ce3de1'
    option fstype          'ext4'
    option enabled         '1'

config 'swap'
    option uuid            '08b4f0a3-f7ab-4ee1-bde9-55fc2481f355'
    option enabled         '1'

config 'mount'
    option target          '/data'
    option uuid            'c1068d91-863b-42e2-bcb2-b35a241b0fe2'
    option enabled         '1'
```

Check if it is mountable to overlay:

```
root@lede:~# mount /dev/sda1 /overlay
root@lede:~# df
Filesystem            1K-blocks      Used Available Use% Mounted on
rootfs                 896          244         652  27% /
/dev/root              2048         2048           0 100% /rom
tmpfs                 14708           64       14644   0% /tmp
/dev/mtdblock6         7759872      477328      7221104   6% /overlay
overlayfs:/overlay     896          244         652  27% /
tmpfs                  512           0          512   0% /dev
/dev/sda1              7759872      477328      7221104   6% /overlay
root@OpenWrt:~#
```

Note that only /overlay has grown but not the /

Reboot the router

Verify that the partitions were mounted properly: Next you can configure your router to be sure it is connected to internet.

Adding new packages

Just edit the file /etc/opkg/customfeeds.conf

```
vi /etc/opkg/customfeeds.conf
```

add the following line

```
src/gz newpackage http://downloads.lede-project.org/snapshots/packages/mipsel_24kc/packages/
```

next

```
opkg update
#to install node
opkg install wget
opkg install git
#Download the following file https://github.com/barais/WG3526Notes/blob/master/node_v4.4.5-1_mipsel_24kc.ipk?raw=true
# copy it on the router and next
opkg install --force-checksum node_v4.4.5-1_mipsel_24kc.ipk
#to install nano
opkg install nano
#to install mosquitto
opkg install mosquitto
```

Install KevoreeJS

Next you can do an update and install npm

```
npm update -g npm.
npm i -g grunt-cli
npm i -g bower
npm i -g generator-kevoree
npm i -g kevoree-cli
```

Install Kevoree NodeJS Runtime

Prefer a global install for this module as it is intended to be used that way:

```
npm i -g kevoree-cli
```

This will allow you to start a new Kevoree JavaScript runtime from the command-line by using:

```
$ kevoree
```

Usage

Usage documentation is available by using the -h flag:

```
$ kevoreejs -h
```

NB You can override the Kevoree registry your runtime uses by specifying two ENV VAR:

```
KEVOREE_REGISTRY_HOST=localhost KEVOREE_REGISTRY_PORT=9000 kevoreejs
```

Enjoy

Cross compiling node modules

Follow the [tutorial](#) to crosscompile some npm modules.

Openzwave

in your cross-compile docker container.

go to /opt/source

```

cd /opt/source

cd packages/libs
git clone https://github.com/cabal/openwrt
cd openwrt
mv openzwave ..
cd ..
rm -rf openwrt
cd ../../
make package/libs/openzwave/compile V=s
cd dl
wget http://old.openzwave.com/downloads/openzwave-1.2.919.tar.gz
cd ..
make menuconfig

#in libraries select now openzwave as module to get an ipkg
make -j1 V=s

```

in /opt/source/bin/targets/ramips/mt7621/packages you will get a file named openzwave_1.2.919-1_mipsel_24kc.ipk

just copy it and copy the libudev* on the router and install it.

```

opkg install libudev_3.2-1_mipsel_24kc.ipk --force-checksum
opkg install openzwave_1.2.919-1_mipsel_24kc.ipk --force-checksum --force-depends

```

next you have to cross-compile [openzwave-shared](#)

It is not so easy to compile it.

You have to clone the openzwave-shared in your container.

Next you can patch the binding.gyp

The linux part should be something like that.

```

[ "OS=='linux'", {
  "variables": {
    "OZW_LIB_PATH" : "/opt/source/build_dir/target-mipsel_24kc_musl-1.1.15/openzwave-1.2.919/",
    "OZW_INC"      : "/opt/source/dl/openzwave-1.2.919/cpp/src/",
  },
  "defines": [
    "OPENZWAVE_ETC=/usr/local/etc/openzwave",
    "OPENZWAVE_DOC=/usr/local/share/doc/openzwave",
    "OPENZWAVE_SECURITY=0"
  ],
  "link_settings": {
    "libraries": ["-lopenzwave"]
  },
  "include_dirs": [
    "<!(node -p -e \"require('path').dirname(require.resolve('nan'))\")>",
    "<(OZW_INC)>",
    "<(OZW_INC)/value_classes"
  ],
  "cflags": [ "-Wno-ignored-qualifiers -Wno-write-strings -Wno-unknown-pragmas" ],
}],

```

Next set the following envs variable

```

export CC=/opt/source/staging_dir/toolchain-mipsel_24kc_gcc-5.4.0_musl-1.1.15/bin/mipsel-openwrt-linux-gcc
export CXX=/opt/source/staging_dir/toolchain-mipsel_24kc_gcc-5.4.0_musl-1.1.15/bin/mipsel-openwrt-linux-g++
export STAGING_DIR=/opt/source/staging_dir
export PKG_CONFIG_PATH=/opt/source/build_dir/target-mipsel_24kc_musl-1.1.15/openzwave-1.2.919
export npm_config_arch=mips
# path to the node source that was used to create the cross-compiled version
export npm_config_nodedir=/opt/source/build_dir/target-mipsel_24kc_musl-1.1.15/node-v4.4.5/

```

Please install nodejs in version 4.4.5 in your docker container. You can do it using nvm.

next install the module

```
npm i -g node-gyp
```

in the folder where you clone node-openzwave-shared

```
npm install --unsafe-perm
```

next put the resulting folder in a node_modules and copy it to your global npm folder on the router.

Great, we can do a kevoree component that use this library.

node-usb, noble, bleno

I just prepare the pre-compiled libraries for noble and bleno.

You have to install the [libusb ipk](#) which is in this github repository. Copy it on the router, copy also the [node-noble.tgz](#) on the router.

Install the libusb-1.0_1.0.20-1_mipsel_24kc.ipk file.

```
opkg install --force-checksum libusb-1.0_1.0.20-1_mipsel_24kc.ipk
```

next uncompress the node-noble.tgz in /root and copy all the content of the node_module folder in /usr/lib/node_modules

```
cd ~
rm -rf node_modules
tar -czf node-noble.tgz
mv node_modules/* /usr/lib/node_modules
```

Just for information, to create this node_modules, I had to cross-compile the node-usb.

```
git clone --recursive https://github.com/nonolith/node-usb.git
cd node-usb
npm i
```

next edit the binding.gyp file.

```
--      'use_system_libusb%': 'false',
++      'use_system_libusb%': 'true',

-- ['use_system_libusb=="true"', {
--     'include_dirs+': [
--         '<!(pkg-config libusb-1.0 --cflags-only-I | sed s/-I//g)'
--     ],
--     'libraries': [
--         '<!(pkg-config libusb-1.0 --libs)'
--     ],
--     }],

++['use_system_libusb=="true"', {
++    'include_dirs+': [
++        '/opt/source/build_dir/target-mipsel_24kc_musl-1.1.15/libusb-1.0.20/libusb/', '/opt/source/staging_dir
/target-mipsel_24kc_musl-1.1.15/usr/include/'
++    ],
++    'libraries': [
++        '-lusb-1.0'
++    ],
++    }],
```

```
#link the libusb library
ln -s /opt/source/build_dir/target-mipsel_24kc_musl-1.1.15/libusb-1.0.20/ipkg-install/usr/lib/libusb-1.0.so /opt/source/staging_dir/toolchain-mipsel_24kc_gcc-5.4.0_musl-1.1.15/mipsel-openwrt-linux-musl/bin/../../../../toolchain-mipsel_24kc_gcc-5.4.0_musl-1.1.15/lib/

#chck if ld can find it
/opt/source/staging_dir/toolchain-mipsel_24kc_gcc-5.4.0_musl-1.1.15/lib/gcc/mipsel-openwrt-linux-musl/5.4.0/../../../../mipsel-openwrt-linux-musl/bin/ld -lusb-1.0 --verbose

#Compile node-usb
export CC=/opt/source/staging_dir/toolchain-mipsel_24kc_gcc-5.4.0_musl-1.1.15/bin/mipsel-openwrt-linux-gcc
export CXX=/opt/source/staging_dir/toolchain-mipsel_24kc_gcc-5.4.0_musl-1.1.15/bin/mipsel-openwrt-linux-g++
export STAGING_DIR=/opt/source/staging_dir
export PKG_CONFIG_PATH=/opt/source/build_dir/target-mipsel_24kc_musl-1.1.15/openssl-1.2.919
export npm_config_arch=mips
# path to the node source that was used to create the cross-compiled version
export npm_config_nodedir=/opt/source/build_dir/target-mipsel_24kc_musl-1.1.15/node-v4.4.5/
npm install --unsafe-perm

#prepare the package
rm -rf .git
cd ..
mv node-usb usb
mkdir test
cd test
mkdir node_modules
mv ../usb node_modules
mv node_modules/usb/node_modules/* node_modules
npm i bleno
npm i noble

#Great it works you can zip the test folder.
```


Prepare, build and flash any buildroot compatible device to run Kevoree

Buildroot is a simple, efficient and easy-to-use tool to generate embedded Linux systems through cross-compilation.

- Buildroot can handle lots of things: Cross-compilation toolchain, root filesystem generation, kernel image compilation and bootloader compilation.
- Buildroot is easy to use. Thanks to its kernel-like menuconfig, gconfig and xconfig configuration interfaces, building a basic system with Buildroot is easy and typically takes 15-30 minutes.
- Buildroot supports hundreds of packages: X.org stack, Gtk3, Qt 5, GStreamer, Webkit, OpenJDK, nodeJS a large number of network-related and system-related utilities are supported.
- Buildroot is for Everyone. It has a simple structure that makes it easy to understand and extend. It relies only on the well-known Makefile language.

Install buildroot

[Download](#) build root and unzip it on a linux. You can also use docker.

Prepare your buildroot configuration

```
# in the unzipped folder (**buildroot root**)
make raspberrypi3_defconfig #if you want to create an image for a raspberry pi 3.
```

```
## Customize your buildroot configuration
```

Add the KevoreeJS Feature

```
```bash
mkdir package/kevoreejs
cd package/kevoreejs
touch kevoreejs.mk
touch Config.in
nano -w Config.in
```

```
config BR2_PACKAGE_KEVOREEJS
 bool "kevoreejs"
 help
 KevoreeJS package: a runtime for Kevoree Javascript component
 see http://kevoree.org for more on this software
```

```
nano -w ../Config.in
```

```
menu "Custom packages"
 source "package/kevoreejs/Config.in"
endmenu
```

```
nano -w kevoreejs.mk
```

```
#
kevoreejs
basic building rules
#
KEVOREEJS_VERSION = 1.1

KEVOREEJS_INSTALL_STAGING = YES

define KEVOREEJS_BUILD_CMDS
$(MAKE) CC="$(TARGET_CC)" LD="$(TARGET_LD)" -C $(@D) all
endef

define KEVOREEJS_INSTALL_STAGING_CMDS
$(MAKE) DESTDIR=$(STAGING_DIR) -C $(@D) install
endef

define KEVOREEJS_INSTALL_TARGET_CMDS
$(INSTALL) -D -m 0755 $(@D)/helloworld \
$(TARGET_DIR)/bin/helloworld
npm update -g npm && npm i -g grunt-cli bower generator-kevoree kevoree-cli@latest && npm cache clean
endef

$(eval $(call package,kevoreejs))
```

in board/yourboard/ (for example board/raspberrypi3/)

create a file named S99Kevoreejs.sh

```
nano -w S99Kevoreejs.sh
```

Put the following content

```
#!/bin/sh
#
Start the kevoreejs...
#

case "$1" in
 start)
 echo "Starting KevoreeJS..."
 /usr/bin/kevoree
 ;;
 stop)
 echo -n "Stopping kevoreeJS..."
 killall -9 node
 ;;
 restart|reload)
 "$0" stop
 "$0" start
 ;;
 *)
 echo $"Usage: $0 {start|stop|restart}"
 exit 1
esac

exit $?
```

```
nano -w config.json
```

Put the following content

```
{
 "registry": {
 "host": "registry.kevoree.org",
 "port": 443,
 "ssl": true,
 "oauth": {
 "client_secret": "kevoree_registryapp_secret",
 "client_id": "kevoree_registryapp"
 }
 }
}
```

You have to select some feature, in particular nodejs

Next, you have to ensure that kevoreejs will boot

edit the board/yourboard/postbuild.sh

```
#change yourboard with raspberrypi3 for example
nano -w board/yourboard/postbuild.sh
```

```
copy custom-files
cp board/raspberrypi3/S99Kevoreejs.sh "$TARGET"/etc/init.d/
chmod a+x "$TARGET"/etc/init.d/S99Kevoreejs.sh
mkdir -p "$TARGET"/root/.kevoree/
cp board/raspberrypi3/config.json "$TARGET"/root/.kevoree/config.json
```

Next do make menuconfig and add **node**, **WCHAR**

```
-> toolchain
-> WCHAR
...
-> Target packages
 -> Interpreter languages and scripting
 -> node, npm express and add kevoree-cli in the additional modules
```

## Build your image

Next, build all.

```
make -j4 all
take a coffee
```

in the output/images folder, you will get your image for your device and kevoreejs runtime will start during the boot process, you can now deploy your own Heads application.

## Extend Kevoree to deploy code for a new platform

HEADS is not specific to any runtime platform. The platform expert is responsible for adding support to deploy code to a new platform.

Kevoree currently supports 2 platforms:

- **JVM-based**  
ThingML can generate Java code and automatically wrap it into components that can be managed by Kevoree
- **JavaScript-based**  
See here how you can [create your own component](#), build it and deploy it

The following are the guidelines on how to integrate a new target platform for Kevoree.

## Introduction

As you know Kevoree is a multiplatform distributed model tool.

Two platforms are currently maintained :

- [Java](#)
- [Javascript](#)

One is currently in development :

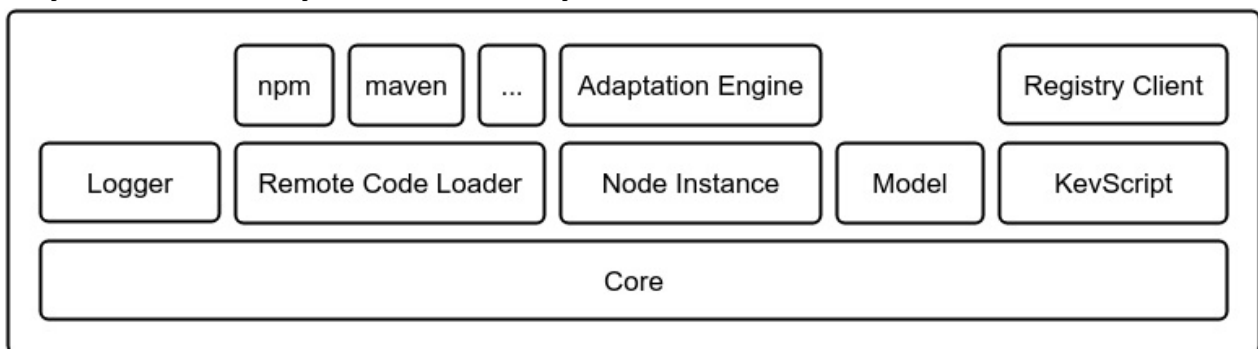
- [C#](#)

Each of those platforms are based on the same concepts and are split in the same way.

In the rest of this chapter we will detail the architecture of an implementation of the Kevoree runtime. It aims to be useful if you want to write Kevoree in another language (python, haskell, ruby, erlang, you name it) but will be based on our experience with Java, JavaScript and C#.

## Global architecture

This picture shows the different parts involved in a Kevoree platform:



## Components

- **Generalities:** A few cross platform advices
- **Kevoree Model:** How to port Kevoree's data model to your platform
- **Remote code loader:** How to load remote code in your runtime
- **Kevoree Model generation:** How to quickly obtain a kevoree model code base
- **Core:** The Model@Runtime conductor
- **Generate instances of the Kevoree Metamodel:** How to quickly obtain a kevoree model code base

- **Registry client**: A simple REST client for the registry
- **Code generator**: How to generate a Component project from a Type Definition.
- **KevScript tool**: Reading kevsript, generating valid kevsript...
- **Runtime**: A Bootstrap is a runtime tool dedicated to the startup of a node instance
- **Kevoree's components**:
  - **Component** : Component development guide
  - **Node** : Node development guide
  - **Group** : Group development guide
  - **Channel** : Channel development guide
- **Useful development tools**:
  - **Local kevoree registry**:
    - Java project : <https://github.com/kevoree/kevoree-registry>
    - Docker container : <https://github.com/kevoree/docker-image-registry-replica> (clone the [official registry](#) by default)
  - **Local kevoree editor**:
    - Node project : <https://github.com/kevoree/kevoree-web-editor>
    - Docker container : <https://github.com/kevoree/docker-image-editor>

## Generalities

### Naming rules

The following rules have not been followed by the past. Don't be surprise to find some unconventional names from time to time. They will be kept that way for legacy. The java platform in particular follow its own naming rules.

The following rules are also in a draft state. Feel free to contact the Kevoree Core Team if you found them hazy.

1. Every Kevoree related component should be prefixed with "kevoree" (e.g. [kevoree-book](#), [kevoree-browser-runtime](#), [kevoree-web-editor](#))
2. Every component related to a specific platform should be prefixed with "kevoree- $\{platform\}$ " (e.g. [kevoree-js](#), [kevoree-dotnet](#)).
3. Every generic component (i.e. described in the [platform itegration](#) part of the book) have a common name who should be followed platforms wide (e.g. [kevoree-js-kevsript](#), [kevoree-dotnet-annotation](#)). The following list is a uncomprehensive list of reserved keywords.
  - kevsript
  - core
  - runtime
  - model
  - annotation
4. Every deployable component name should follow the form "kevoree- $\{platform\}$ - $\{componentType\}$ - $\{typeDefName\}$ " where **componentType** is *node*, *group*, *chan* or *comp* (Respectively for the Node, Group, Channel and Component). The **typeDefName** should be consistent among the various implementations of the same type definition (e.g. [kevoree-dotnet-group-remotews](#), [kevoree-js-comp-ticker](#)).

# Kevoree model

## Introduction

The meta-model of Kevoree is defined [here](#) using [KMF](#)'s modeling language.

## Requirements

- A model must be serializable and unserializable to/from a JSON structure. It is useful for the communication of models by the Groups or the publication of models to a Kevoree registry.

## Strategies

### Adding a generator for the targeted platform

You have two choices here:

- create a KMF generator that targets your language
- create a model from scratch in the target language

### Transpiling from an existing model

A real life scenario is the development of the C# platform. No generator exists for this platform, but C# paradigms are very close from Java's one so we had been able to use [IKVM](#), a tool to convert JARs to DLL.

The whole process is detailed [here](#).

From our experience, the generated DLL is working really well but we had a hard time figuring out how to integrate it with the isolated contexts needed to load components into a node.

### Kevoree Model's JSON Schema

What matters in the end is that the target platform is able to (de)serialize Kevoree models from (and to) JSON strings.

The JSON format must comply with this JSON Schema:

```
{
 "$schema": "http://json-schema.org/draft-04/schema#",
 "additionalProperties": false,
 "type": "object",
 "definitions": {
 "Group": {
 "additionalProperties": false,
 "type": "object",
 "properties": {
 "metaData": {
 "type": "array",
 "items": {"$ref": "Value"}
 },
 "dictionary": {"$ref": "Dictionary"},
 "typeDefinition": {"$ref": "TypeDefinition"},
 "name": {"type": "string"},
 "fragmentDictionary": {
 "type": "array",
 "items": {"$ref": "FragmentDictionary"}
 },
 "started": {"type": "boolean"},
 "subNodes": {
```

```

 "type": "array",
 "items": {"$ref": "ContainerNode"}
 },
 "required": ["started"]
},
"Dictionary": {
 "additionalProperties": false,
 "type": "object"
},
"FragmentDictionary": {
 "additionalProperties": false,
 "type": "object",
 "properties": {"name": {"type": "string"}}
},
"NetworkInfo": {
 "additionalProperties": false,
 "type": "object",
 "properties": {
 "values": {
 "type": "array",
 "items": {"$ref": "Value"}
 },
 "name": {"type": "string"}
 }
},
"Channel": {
 "additionalProperties": false,
 "type": "object",
 "properties": {
 "metaData": {
 "type": "array",
 "items": {"$ref": "Value"}
 },
 "dictionary": {"$ref": "Dictionary"},
 "typeDefinition": {"$ref": "TypeDefinition"},
 "bindings": {
 "type": "array",
 "items": {"$ref": "MBinding"}
 },
 "name": {"type": "string"},
 "fragmentDictionary": {
 "type": "array",
 "items": {"$ref": "FragmentDictionary"}
 },
 "started": {"type": "boolean"}
 },
 "required": ["started"]
},
"MBinding": {
 "additionalProperties": false,
 "type": "object",
 "properties": {
 "hub": {"$ref": "Channel"},
 "port": {"$ref": "Port"}
 }
},
"Port": {
 "additionalProperties": false,
 "type": "object",
 "properties": {
 "bindings": {
 "type": "array",
 "items": {"$ref": "MBinding"}
 },
 "name": {"type": "string"},
 "portTypeRef": {"$ref": "PortTypeRef"}
 }
},
"PortTypeRef": {
 "additionalProperties": false,
 "type": "object",

```



```

 "properties": {
 "ref": {"$ref": "PortType"},
 "mappings": {"$ref": "PortTypeMapping"},
 "noDependency": {"type": "boolean"},
 "name": {"type": "string"},
 "optional": {"type": "boolean"}
 },
 "required": [
 "optional",
 "noDependency"
]
 },
 "DeployUnit": {
 "additionalProperties": false,
 "type": "object",
 "properties": {
 "requiredLibs": {
 "type": "array",
 "items": {"$ref": "DeployUnit"}
 },
 "hashCode": {"type": "string"},
 "name": {"type": "string"},
 "filters": {
 "type": "array",
 "items": {"$ref": "Value"}
 },
 "version": {"type": "string"},
 "url": {"type": "string"}
 }
 },
 "DictionaryType": {
 "additionalProperties": false,
 "type": "object"
 },
 "TypeDefinition": {
 "additionalProperties": false,
 "type": "object",
 "properties": {
 "superTypes": {
 "type": "array",
 "items": {"$ref": "TypeDefinition"}
 },
 "metaData": {
 "type": "array",
 "items": {"$ref": "Value"}
 },
 "deployUnits": {
 "type": "array",
 "items": {"$ref": "DeployUnit"}
 },
 "dictionaryType": {"$ref": "DictionaryType"},
 "name": {"type": "string"},
 "version": {"type": "string"},
 "_abstract": {"type": "boolean"}
 },
 "required": ["_abstract"]
 },
 "Repository": {
 "additionalProperties": false,
 "type": "object",
 "properties": {"url": {"type": "string"}}
 },
 "PortTypeMapping": {
 "additionalProperties": false,
 "type": "object",
 "properties": {
 "paramTypes": {"type": "string"},
 "beanMethodName": {"type": "string"},
 "serviceMethodName": {"type": "string"}
 }
 },
 "Value": {

```

```

 "additionalProperties": false,
 "type": "object",
 "properties": {
 "name": {"type": "string"},
 "value": {"type": "string"}
 }
 },
 "ComponentInstance": {
 "additionalProperties": false,
 "type": "object",
 "properties": {
 "metaData": {
 "type": "array",
 "items": {"$ref": "Value"}
 },
 "dictionary": {"$ref": "Dictionary"},
 "typeDefinition": {"$ref": "TypeDefinition"},
 "provided": {
 "type": "array",
 "items": {"$ref": "Port"}
 },
 "name": {"type": "string"},
 "fragmentDictionary": {
 "type": "array",
 "items": {"$ref": "FragmentDictionary"}
 },
 "started": {"type": "boolean"},
 "required": {
 "type": "array",
 "items": {"$ref": "Port"}
 }
 }
 },
 "required": ["started"]
},
"ContainerNode": {
 "additionalProperties": false,
 "type": "object",
 "properties": {
 "networkInformation": {
 "type": "array",
 "items": {"$ref": "NetworkInfo"}
 },
 "metaData": {
 "type": "array",
 "items": {"$ref": "Value"}
 },
 "components": {
 "type": "array",
 "items": {"$ref": "ComponentInstance"}
 },
 "dictionary": {"$ref": "Dictionary"},
 "hosts": {
 "type": "array",
 "items": {"$ref": "ContainerNode"}
 },
 "typeDefinition": {"$ref": "TypeDefinition"},
 "name": {"type": "string"},
 "host": {"$ref": "ContainerNode"},
 "fragmentDictionary": {
 "type": "array",
 "items": {"$ref": "FragmentDictionary"}
 },
 "groups": {
 "type": "array",
 "items": {"$ref": "Group"}
 },
 "started": {"type": "boolean"}
 },
 "required": ["started"]
},
"Package": {
 "additionalProperties": false,

```

```

 "type": "object",
 "properties": {
 "deployUnits": {
 "type": "array",
 "items": {"$ref": "DeployUnit"}
 },
 "typeDefinitions": {
 "type": "array",
 "items": {"$ref": "TypeDefinition"}
 },
 "name": {"type": "string"},
 "packages": {
 "type": "array",
 "items": {"$ref": "Package"}
 }
 }
 },
 "PortType": {
 "additionalProperties": false,
 "type": "object",
 "properties": {"synchrone": {"type": "boolean"}},
 "required": ["synchrone"]
 }
},
"properties": {
 "nodes": {
 "type": "array",
 "items": {"$ref": "#/definitions/ContainerNode"}
 },
 "repositories": {
 "type": "array",
 "items": {"$ref": "#/definitions/Repository"}
 },
 "groups": {
 "type": "array",
 "items": {"$ref": "#/definitions/Group"}
 },
 "packages": {
 "type": "array",
 "items": {"$ref": "#/definitions/Package"}
 },
 "mBindings": {
 "type": "array",
 "items": {"$ref": "#/definitions/MBinding"}
 },
 "hubs": {
 "type": "array",
 "items": {"$ref": "#/definitions/Channel"}
 },
 "generated_KMF_ID": {"type": "string"}
}
}

```

# Core

## What is it?

The **Core** is the *director* of a Kevoree runtime:

- it is in charge of a **node** (Kevoree NodeType)
- it has to execute adaptations on deployments (model@run.time)
- keeps track of model changes

## Bootstrap

The first step of the core is to **bootstrap** the runtime. To do so, the core is in charge of creating the node instance defined with a **unique** name at the start of the Core.

In order to know which NodeType to instantiate and with what properties, the core must have a model containing those data.

This model is known as the **bootstrap model**. Using this model the core will be able to create a new ContainerNode instance and keep a reference to it in order to ask that particular node *how* to do runtime adaptations on deployment phases.

## Deployment phase

Using pseudo-code, the deployment algorithm (model@run.time) looks like this:

```
fun deployNewModel(newModel: KevModel) {
 // check the validity of the new model
 if (isValid(newModel)) {
 // compare new model with current model
 const compareResult: AdaptationModel = node.compare(currentModel, newModel);
 // execute a list of command to adapt the current system
 // according to the new model
 if (compareResult.execute()) {
 // keep track of current model (history)
 saveModel(currentModel);
 // use new model as current model
 setCurrentModel(newModel);
 } else {
 // if an adaptation fails, rollback to previous state
 // which means, execute the successfully applied command
 // backwards to go back to the previous state
 compareResult.rollback();
 // notify error
 throw error;
 }
 } else {
 // if the new model is not a valid model, discard it and notify
 throw error;
 }
}
```

## Error handling

If an error occurs while processing the adaptations the Core is in charge of putting the runtime state back to the previous one. This is known as the **rollback** phase.

A rollback is an execution of all the already processed adaptations but backwards (cf. **Deployment phase** algorithm)

## Using the Core within the running components and fragments

From a component or fragment perspective, one might want to apply reconfigurations on the running system on its own. To do so, each **Component**, **Channel**, **Group** and **Node** must be able to get a reference to the runtime **core**.

In **Java**, accessing the runtime core can be done like that:

```
@Component(version = 1)
public class MyComponent {

 @KevoreeInject
 private ModelService modelService;

 public void doSomethingWithCore() {
 modelService.update(aModel, new UpdateCallback() {
 @Override
 public void run(Boolean success) {
 if (success) {
 // adaptations made
 } else {
 // problem with adaptation: not done
 }
 }
 });
 }
}
```

In **JavaScript**, each instance can access the core locally:

```
const AbstractComponent = require('kevoree-entities/lib/AbstractComponent');

module.export = AbstractComponent.extend({
 toString: 'MyComponent',
 tdef_version: 1,

 doSomethingWithCore: function () {
 this.getKevoreeCore().deploy(aModel, function (err) {
 if (err) {
 // problem with adaptation: not done
 } else {
 // adaptations made
 }
 });
 }
});
```

# Remote code loader

## Introduction

A Kevoree model represents a set of Components, Nodes, Groups and Channels that can be connected together.

At some point the nodes have to load the Components, Groups fragments and Channels fragments from a shared code repository.

Once loaded in a node, a piece of code must be isolated from the rest of the application.

For example if a Node "A" have a dependency to a library **Z** in version **1.0.0** and have to load Component "B" with a dependency to the same library **Z** but in version **2.0.0**, the action of loading "B" in the context of "A" should not override **Z** in its version 2.0.0

## Existing implementations

### Java

**Repo:** [maven](#)

**Loader:** ClassLoader named [KCL](#)

### Javascript

**Repo:** [npm](#)

**Loader:** Node.js module (CommonJS standard)

Code isolation is a structural feature of this platform because modules are loaded based on their full path location and versioning is handled by **npm**

### C#

**Repo:** [Nuget](#)

**Loader:** The code isolation is based on [AppDomain](#) and the [MEF Framework](#).

The [Kevoree Dotnet Nuget Loader](#) is a component which combines Nuget, AppDomain and MEF. It take a Nuget name and version and return an isolated context containing the remote component code.

## Advices

This component can be pretty tricky to implement according to the default features of the targeted language.

You should implement it as soon as possible because it will impact the following code parts :

- [Bootstrap](#)
- Node
- [Model generator](#)

## Registry client

### Description

The registry client is a a simple JSON-REST client for the [Registry Kevoree](#).

The Kevoree Registry source-code is available here: <https://github.com/kevoree/kevoree-registry>

# Model generator

## Introduction

In order to declare publicly a component you have to publish its `TypeDefinition` and its `DeployUnit` to a registry. To do so you have to generate a model for your component.

- Model generation is handled by platform-specific tools
  - [Maven plugin](#) - *java*
  - [Grunt task](#) - *js*
- Publication is handled by platform-specific tools
  - [Maven plugin](#) - *java*
  - [Grunt task](#) - *js*
- A model generator statically analyses code or uses reflection in order to create a Kevoree model composed of a `TypeDefinition` and its related `DeployUnit`
- The created model is then published to the Kevoree registry for a specific **user** in a **namespace** using a specific **name** and **version** for the `TypeDefinition` & `DeployUnit`.

## Existing implementations

### Java

A maven plugin ([kevoree-maven-plugin](#)) is in charge of publishing collectively a component to a maven repository and to a Kevoree registry.

The code analysis is done by reflection (mostly by annotations scanning).

### Javascript

For the JavaScript platform, the model generation is made by a [Grunt task](#) that will reflect on the Node.js module code. The reflection is mostly done by reading the provided properties of the class. The properties that have a meaning in Kevoree are prefixed using **naming conventions**:

- **tdef\_version**: for the `TypeDefinition` version
- **dic\_XXX**: for Dictionary Attribute
- **in\_XXX**: for input port
- **out\_XXX**: for output port

The `TypeDefinition` is created by reading those properties.

The `DeployUnit` is created by reading the `package.json` in order to know how to download the module from the **npm registry**.

Publication can then be handled, on demand, using another [Grunt task](#).

### C#

There is no integrated tool to do all in once in *c#* yet.

The process is split in two steps:

1. publish the package to a nuget registry
2. use the C# [Kevoree Model Generator](#) to publish the package to a Kevoree registry





## Code generator

The Kevoree Registry is a public application, accessible at <http://registry.kevoree.org>.

This application is basically a map of TypeDefinition <-> DeployUnit links. It is used to resolve the components, nodes, channels and groups binaries based on their TypeDefinition.

With that in mind, when adding a new platform to the Kevoree eco-system, it might be convenient to provide a code generator that is able to scaffold projects for the new targeted platform language.

The idea behind that is to ease the creation of a new DeployUnit for a specific TypeDefinition. Because the registry knows all about the available TypeDefinitions, and because a TypeDefinition contains all the necessary information to create a code skeleton (type, dictionary attributes, inputs, outputs). One could create a library that takes developer inputs to determine a TypeDefinition name and version, and then create the code skeleton for that specific TypeDefinition using the targeted language paradigm.

## Example

In JavaScript, the code generator is provided by the Yeoman [generator-kevoree](#). This generator is a command-line Node.js application that prompts questions to the developer in order to, in the end, create a **kevoree-js** project.

```
$ yo kevoree

Kevoree Project Generator:

[?] Would you like to start from an existing TypeDefinition from the Kevoree Registry? Yes
[?] Specify a TypeDefinition fully qualified name (eg. Ticker or my.company.MyType) org.kevoree.library.Ticker
[?] Which version would you like to use? (40 total versions) 5.2.10
[?] Choose your NPM module name: kevoree-comp-ticker
[?] Do you want this to be runnable by the browser runtime? No
[?] What is the license of your module? (MIT) LGPL-3.0

... keep on answering
```

And finally, the generator will create a clean project based on the Kevoree Registry TypeDefinition named **Ticker** in version **5.2.10** in the example:

```
$ tree -L 2
.
├─ Gruntfile.js
├─ kevs
│ └─ main.kevs
├─ lib
│ └─ Ticker.js
├─ package.json
├─ README.md
└─ (...)
```

With the **Ticker** skeleton be:

```
var AbstractComponent = require('kevoree-entities/lib/AbstractComponent');

var Ticker = AbstractComponent.extend({
 toString: 'Ticker',
 tdef_version: 1,

 dic_random: {
 optional: true,
 defaultValue: false,
 },
 dic_period: {
 optional: true,
 defaultValue: 3000,
 },
 start: function (done) {
 this.log.debug(this.toString(), 'START');
 done();
 },
 stop: function (done) {
 this.log.debug(this.toString(), 'STOP');
 done();
 },
 out_tick: function (msg) { /* noop */ }
});

module.exports = Ticker;
```

The Yeoman code generator v3.5.2 only complies with Kevoree v5.3.x or less

# Runtime

## Usage

A runtime is a Kevoree executable application. It is used to actually start a Kevoree core targetting a specific platform.

Currently, Kevoree has 3 runtimes:

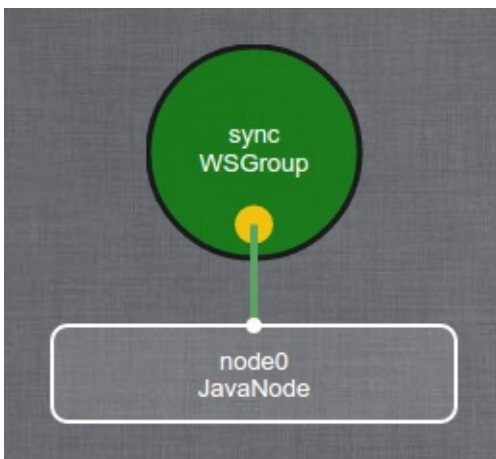
- Java runtime, which is an executable JAR file. ([Download](#))
- Node.js runtime, which is available on [npm](#)
- Browser runtime, which is also targeting the JavaScript platform but runs in Web Browsers and allows components to provide a User Interface. ([Browser runtime](#))

By default, every runtime must be able to start a Kevoree environment without giving any input. In such case, the runtime must create what we call a "default bootstrap model", in KevScript this model looks like this:

```
// default bootstrap model targeting the JavaNode
add node0 : JavaNode
add sync : WSGroup

attach node0 sync
```

And in the model editor:



## Dependencies

A runtime depends, at least, on the following Kevoree parts:

- [core](#)
- [model API](#)
- [kevscript interpreter](#)
- [remote code loader](#)

# KevScript Tools

## Description

In order to manipulate models, we have created a scripting language that is called **KevScript**.

*This language is not a [general purpose language](#)*

A KevScript engine will take a script and a model as inputs, and return a new model modified according to the script.

*In other word, KevScript is only a Kevoree-specific transformation language.*

## Implementations

The KevScript grammar is written using [Waxeye](#). Waxeye is a parser generator based on parsing expression grammars (PEGs) and it is able to generate parsers in many different languages, such as:

- Java
- JavaScript
- Python
- C
- Ruby
- Scheme

If you want to create your own KevScript interpreter, maybe waxeye can already generate the parser for you.

## KevScript grammar in Waxeye

```
KevScript grammar

Rules
=====
KevScript <- ws *((Statement | :Comment eol) ws) :?Comment

Statement <- Add | Remove | Move | Attach | Detach | Set | AddBinding | DelBinding | Include | Network | Ad
dRepo | Namespace | Start | Stop | Pause

Add <- AddToken ws NameList ws ':' ws TypeDef # kevs.add group0, gr
oup1 : WebSocketGroup

Remove <- RemoveToken ws NameList # remove node0, node0
.comp1, sync

Move <- MoveToken ws NameList ws InstancePath # move node0.comp0, n
ode2.* node1

Attach <- AttachToken ws NameList ws InstancePath # kevs.attach node0,
node1 group0

Detach <- DetachToken ws NameList ws InstancePath # kevs.detach node0,
node1 group0

Set <- SetToken ws InstancePath ?(: '/' InstancePath) ws ':' ws (CtxVar | GenCtxVar | RealString)
set node0.comp0.myAtt = 'foo'
set sync.myAtt/node
0 = "foo"
set node0.started =
"false" -> is used to define if the instance is started or not

Network <- NetworkToken ws InstancePath ws (CtxVar | String2) # network node1.lan.e
th0 192.168.0.1
```

```

AddBinding <- BindToken ws InstancePath ws InstancePath # bind node1.comp0.se
ndMsg chan42

DelBinding <- UnbindToken ws InstancePath ws InstancePath # unbind node1.comp0.
sendMsg chan0

AddRepo <- RepoToken ws RealStringNoNewLine # repo "http://org.so
natype.org/foo/bar?a=b&c=d"

Include <- IncludeToken ws String ':' String2 # include npm:kevoree
-CHAN-WEBSOCKET

 # include mvn:org.kev
oree.library.javase:websocketgrp:2.0.5-SNAPSHOT

NameList <- InstancePath ws *(:[,] ws InstancePath) # node42
 # node0, node0.comp1,
node42

TypeDef <- TypeFQN ?(:/' Version) # FooType/0.0.1 (spec
ific vers.)
 # FooType (last
vers.)
 # org.kevoree.Foo/0.4
2.0 (fully qualified name)

TypeFQN <- String3 *([.] String3)

Namespace <- NamespaceToken ws String # namespace sp-ace_0

Start <- StartToken ws NameList # start host.child

Stop <- StopToken ws NameList # stop child, child1

Pause <- PauseToken ws NameList # pause node0.comp

InstancePath <- (Wildcard | String | CtxVar | GenCtxVar) *(:[.] (Wildcard | String | CtxVar | GenCtxVar))
 # node0.*.att
 # %%ctxVar%%
 # node0.%%genVar%%

Wildcard <- '*'

CtxVar <- :'% ' String :'% '

GenCtxVar <- :'%%' String :'%%'

String <- +[a-zA-Z0-9_-]

String2 <- +[a-zA-Z0-9-:_%@_-]

String3 <- +[a-zA-Z0-9_]

Version <- (TdefVersion ?(:/' DuVersion))
TdefVersion <- Integer | Latest | CtxVar
DuVersion <- Release | Latest | CtxVar

Integer <- +[0-9]
Release <- ReleaseToken
Latest <- LatestToken

Line <- +(!eol .) # anything but EOL

RealString <- :['] *(NewLine | Escaped | SingleQuoteLine) :[']
 | :["] *(NewLine | Escaped | DoubleQuoteLine) :["]

Escaped <- [\](!eol .)
SingleQuoteLine <- +(!['] ![\] (!eol .))
DoubleQuoteLine <- +(!["] ![\] (!eol .))

RealStringNoNewLine <- :['] *([\](!eol .) | !['] ![\] (!eol .)) :[']
 | :["] *([\](!eol .) | !["] ![\] (!eol .)) :["]

```

```
NewLine <- :'\r\n' | :'\n' | :'\r'
=====
End Rules

Void Non-terminals
=====
RepoToken <- 'repo'
IncludeToken <- 'include'
AddToken <- 'kevs.add'
RemoveToken <- 'remove'
MoveToken <- 'move'
SetToken <- 'set'
AttachToken <- 'kevs.attach'
DetachToken <- 'kevs.detach'
NetworkToken <- 'network'
BindToken <- 'bind'
UnbindToken <- 'unbind'
NamespaceToken <- 'namespace'
StartToken <- 'start'
StopToken <- 'stop'
PauseToken <- 'pause'
LatestToken <- 'LATEST'
ReleaseToken <- 'RELEASE'
Comment <- '//' ?Line
eol <- :'\r\n' | :'\n' | :'\r'
ws <- '([\t] | eol)'
=====
End Void Non-terminals
```

## Components

- [Component](#)
- [Node](#)
- [Channel](#)
- [Group](#)



# Component

## Description

A component is a piece of code which can tack data in input and produces data in ouput.

## Examples

- Ticker : Emit a random value every N seconds.
- ConsolePrinter : Print everything it receive.
- ArduinoController : Can be used as a proxy to an [arduino device](#).

## Interface

A component interact with the rest of a kevoree system through its interface. It is composed of different elements, usually described by annotated field and method (at least in our existing implementations in java, javascript/typescript and C#).

### Input

A field annotated with input will receive data from any channel connected to it.

### Output

A field annotated with output offer the ability to send data through channels connected to it.

### Start

The method annotated with Start will be called when an instance of the component need to be started.

### Stop

The method annotated with Stop will be called when an instance of the component need to be stopped.

### Update

The method annotated with Update will be called when an instance of the component need to be updated.

### Param

A field annotated with Param will be instantiated with a value provided by the model.

## KevoreeInject

A few interfaces to the external system are provided to the components by injection of interfaces instances. The core will match every KevoreeInject annotated field and will look for a instance of its interface.

The provided interfaces are :

- Logger : Offers a way to log messages.
- Context : Provider accessors to the node node, the instance path and the instance name.
- ModelService : Offers operators on the node's model. For example you can use it to publish a new model, which will be adapted by the core.

## Naming rules reminder

As defined in the [generalities](#) part, every component must follow this name rule : kevoree- $\{platform\}$ -comp- $\{componentName\}$  (e.g kevoree-js-comp-ticker, kevoree-dotnet-comp-consoleprinter, kevoree-java-comp-arduino-controller).

# Node

## Description

A node is a component which receive a set of traces (i.e. a list of differences between the current model and the targeted model) and will adapt its state to match it with the targeted model.

For now a single implementation of Node is done by language. Mostly because is it one of the most complex part of a Kevoree implementation.

## Interface

A node interact with the rest of a kevoree system through its interface. It is composed of different elements, usually described by annotated field and method (at least in our existing implementations in java, javascript/typescript and C#).

### Start

The method annotated with Start will be called when an instance of the node need to be started.

### Stop

The method annotated with Stop will be called when an instance of the node need to be stopped.

### Update

The method annotated with Update will be called when an instance of the node need to be updated.

### Param

A field annotated with Param will be instantiated with a value provided by the model.

## KevoreeInject

A few interfaces to the external system are provided to the components by injection of interfaces instances. The core will match every KevoreeInject annotated field and will look for a instance of its interface.

The provided interfaces are :

- **Logger** : Offers a way to log messages.
- **Context** : Provider accessors to the node node, the instance path and the instance name.
- **ModelService** : Offers operators on the node's model. For example you can use it to publish a new model, which will be adapted by the core.

# Group

## Description

A group is a piece of code which synchronize a model between the nodes connected to it. It is also the interface to load a living model into an external tool (e.g an editor).

## Examples

- RemoteWSGroup : The nodes are connected to a shared WebSocket broker which broadcast every received messages.
- WSGroup : A WSGroup have the same behaviour as a RemoveWSGroup but one of the group fragment is running the WebSocket broker.

## Interface

A group interact with the rest of a kevoree system through its interface. It is composed of different elements, usually described by annotated field and method (at least in our existing implementations in java, javascript/typescript and C#).

### Start

The method annotated with Start will be called when an instance of the group need to be started.

### Stop

The method annotated with Stop will be called when an instance of the group need to be stopped.

### Update

The method annotated with Update will be called when an instance of the group need to be updated.

### Param

A field annotated with Param will be instantiated with a value provided by the model.

## KevoreeInject

A few interfaces to the external system are provided to the groups by injection of interfaces instances. The core will match every KevoreeInject annotated field and will look for a instance of its interface.

The provided interfaces are :

- Logger : Offers a way to log messages.
- Context : Provider accessors to the node node, the instance path and the instance name.
- ModelService : Offers operators on the node's model. For example you can use it to publish a new model, which will be adapted by the core.

## Naming rules reminder

As defined in the [generalities](#) part, every group must follow this name rule : kevoree- $\{platform\}$ -group- $\{groupName\}$  (e.g kevoree-js-group-ws, kevoree-dotnet-group-remotews).



# Channel

## Description

A channel is a piece of code which forward messages from components with an output port connect to it to components with an input port connected to it.

## Examples

- RemoteWSChan : The components are connected to a shared WebSocket broker which broadcast every received messages.
- WSChan : A WSChan have the same behaviour as a RemoteWSChan but one of the channel fragment is running the WebSocket broker.

## Interface

A channel interact with the rest of a kevoree system through its interface. It is composed of different elements, usually described by annotated field and method (at least in our existing implementations in java, javascript/typescript and C#).

### Start

The method annotated with Start will be called when an instance of the channel need to be started.

### Stop

The method annotated with Stop will be called when an instance of the channel need to be stopped.

### Update

The method annotated with Update will be called when an instance of the channel need to be updated.

### Param

A field annotated with Param will be instantiated with a value provided by the model.

## KevoreeInject

A few interfaces to the external system are provided to the channels by injection of interfaces instances. The core will match every KevoreeInject annotated field and will look for a instance of its interface.

The provided interfaces are :

- Logger : Offers a way to log messages.
- Context : Provider accessors to the node node, the instance path and the instance name.
- ModelService : Offers operators on the node's model. For example you can use it to publish a new model, which will be adapted by the core.
- ChannelContext : offer an access to the input and output of connected components.

## Naming rules reminder

As defined in the [generalities](#) part, every channel must follow this name rule : kevoree- $\{platform\}$ -chan- $\{channelName\}$  (e.g kevoree-js-chan-ws, kevoree-dotnet-chan-remotews).



## Extend Kevoree to support a new communication channel

To support different communication protocols, HEADS relies on the definition of communication channels. Kevoree allows defining channel types which can then be deployed between components. The channel types can wrap low level protocols (such as binary on a serial link) as well as high level protocols (such as emails or skype calls).

A channel in Kevoree is implemented as follows:

```
@ChannelType(version = 1, description = "First attempt at creating a channel")
public class MyFirstChannel implements ChannelDispatch {

 @KevoreeInject
 private ChannelContext channelContext;

 @Override
 public void dispatch(final String payload, final Callback callback) {
 for (Port p : this.channelContext.getLocalPorts()) {
 p.send(payload, callback);
 }
 }
}
```

This naive implementation basically implements a direct call to the local ports. However, instead of a simple `p.send` it is possible to use third-party APIs, for example publishing a message on a MQTT topic.

The channel context gives you access to the model. The dispatch method is called automatically when a message is received by one of the channel fragment. You must have in mind that this channel is instantiated for any node on which bound components are deployed.

You can follow this tutorial on how to [make your own Kevoree channel](#) for more details about channels.



## Platform Experts for CEP

In order to deploy code generated from ThingML CEP on Apama CEP engines, we need to have Apama EPL as a supported platform in the HEADS transformation framework. A platform expert with Apama CEP knowledge is needed for writing a plug-in for this HEADS transformation framework in order to support Apama EPL as a target platform. This plug-in transforms ThingML CEP into Apama EPL code. It is based on the mapping between ThingML CEP and Apama EPL described in detail in the deliverable D2.3 of the HEADS project. The main parts of this mapping are:

- Event type <-> Message
- Channel <-> Port
- Monitor <-> Thing
- Stream query <-> Stream query

## For Service Developers

We have identified 4 different tasks for the Service Developer.

## Model HD-Service logic with ThingML components

The service developer uses ThingML to define the components of the HD-Service and implement the logic of those components.

State machines are a common formalism to express reactive behavior that needs to react on some events, correlate events, and produce some new events. A state machine-based programming language, and ThingML in particular, is thus a good candidate to implement Kevoree components and write the logic that orchestrates the different ports of this component.

### Define interfaces:

Let's consider a simple ThingML program made of two things, basically involving message to deal with a timer:

```
thing fragment TimerMsgs {
 // Start the Timer
 message timer_start(delay : Integer);
 // Cancel the Timer
 message timer_cancel();
 // Notification that the timer has expired
 message timer_timeout();
}
```

First, a timer:

```
thing fragment Timer includes TimerMsgs
{
 provided port timer
 {
 sends timer_timeout
 receives timer_start, timer_cancel
 }
}
```

### Implement the logic using state machines and action languages

A simple client using the timer could be:

```
thing HelloTimer includes TimerMsgs {

 required port timer {
 receives timer_timeout
 sends timer_start, timer_cancel
 }

 readonly property period : Integer = 1000
 property counter : Integer = 0

 statechart behavior init Init {

 state Init {
 on entry do
 timer!timer_start(period)
 end

 transition -> Init //this will loop on the Init state, and start a new timer
 event timer?timer_timeout
 action do
 print "hello "
 print counter
 print "\n"
 counter = counter + 1
 end
 }
 }
}
```

This simple, platform-independent service basically outputs a "hello n" every second. This is realized by starting a timer when entering the `Init` state. On timeout, a `timer_timeout` message is received, triggering the prints, before it re-enters the `Init` state.

Basically, this would produce the following outputs:

```
hello 0
hello 1
hello 2
```

Alternatively, the previous example can be refactored in the following way to keep the state machine clean:

```

thing HelloTimer includes TimerMsgs {

 required port timer {
 receives timer_timeout
 sends timer_start, timer_cancel
 }

 readonly property period : Integer = 1000
 property counter : Integer = 0

 function printHello() do
 print "hello "
 print counter
 print "\n"
 counter = counter + 1
 end

 statechart behavior init Init {

 state Init {
 on entry do
 timer!timer_start(period)
 end

 transition -> Init //this will loop on the Init state, and start a new timer
 event timer?timer_timeout
 action printHello()
 }
 }
}

```

More generally the general syntax for a function is:

```

function myFunction(param1 : ParamType1, param2 : ParamType2) : Returntype do
 ...
end

```

Like in most programming languages, functions are particularly useful to encapsulate code that is called from multiple places, to avoid duplication.

The HEADS action and expression language is fairly aligned with major programming languages such as Java, JavaScript or C:

- variable definitions and affectations `var i : Integer = 0 ,`
- algebraic ( `+` , `-` , etc) and boolean operators ( `and` and `or` )
- control structures `if (true) do ... end else do ... end , while(true) do ... end`
- `print "hello"` and `error "alert!"`
- function calls `myFunction(0, 1)`

In addition to the "normal" statements common with those language, HEADS provides:

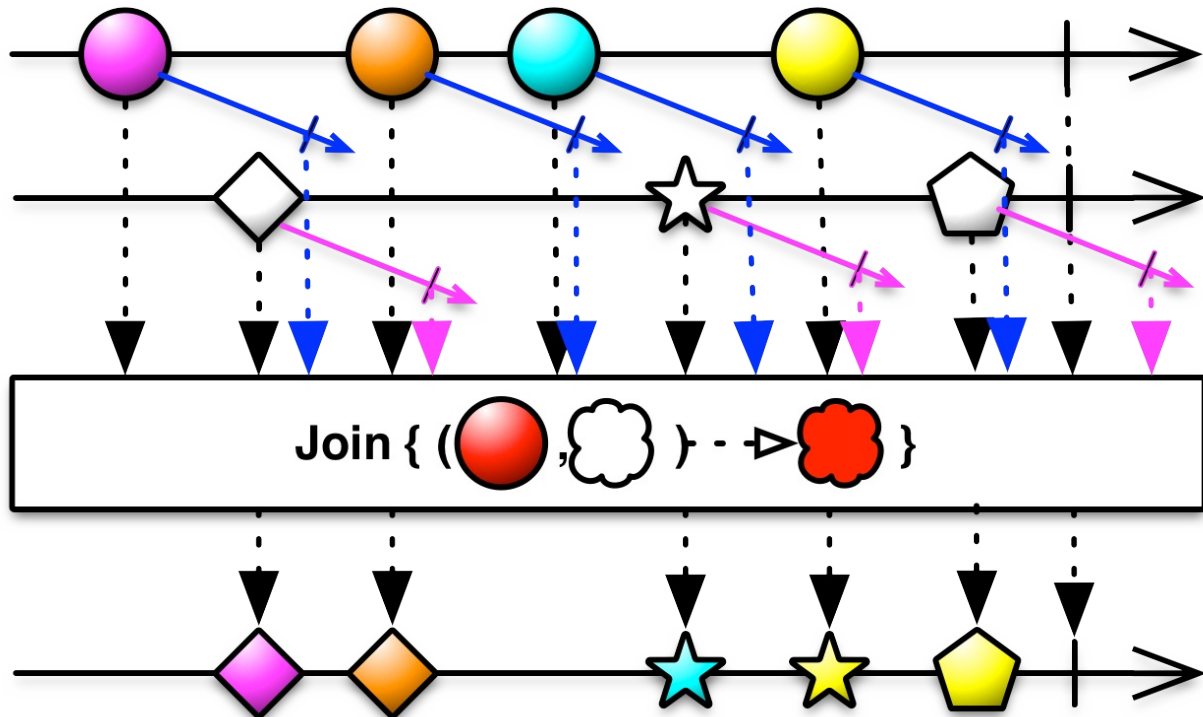
- send a message `myPort!myMessage()` , `myPort!myMessage2(a, b, 0)` for asynchronous message passing between components

## Implement advanced logic with Complex Event Processing

The HEADS modelling language has been extended with CEP concepts (See D2.2 for more details). CEP complements state machines and provides more powerful abstractions to handle streams of events, for example to compute the average of the values provided by a sensor on a given time window, or to when some behavior should be triggered when two events happen "at the same time". While this can be expressed with state machines, this typically implies instantiating timers and arrays (to manage time windows), managing different interleaving, etc, *i.e.* this generates accidental complexity. Those CEP concepts are mapped to the ones provided by [ReactiveX](#).

### Join

The **join operator** "combines items emitted by two Observables whenever an item from one Observable is emitted during a time window defined according to an item emitted by the other Observable". See the figure below (taken from ReactiveX documentation) to get an idea of how it works.



This is expressed in the HEADS modelling language using this syntax:

```
stream simpleJoinWithParams @TTL "100" do
 from [e1 : receivePort?m1 & e2 : receivePort?m2 -> cep1(e1.v1 + e2.v1)]
 select a : #0, b : #1
 action sendPort!cep1(a, b)
end
```

Whenever a message `m1` and a message `m2` are received within 100 ms, it will produce a `cep1` message.

The same query expressed directly using the ReactiveX API would require about 15 lines of code (in Java or here in JavaScript):

```
//Code sample taken from ReactiveX documentation
var xs = Rx.Observable.interval(100)
 .map(function (x) { return 'first' + x; });

var ys = Rx.Observable.interval(100)
 .map(function (x) { return 'second' + x; });

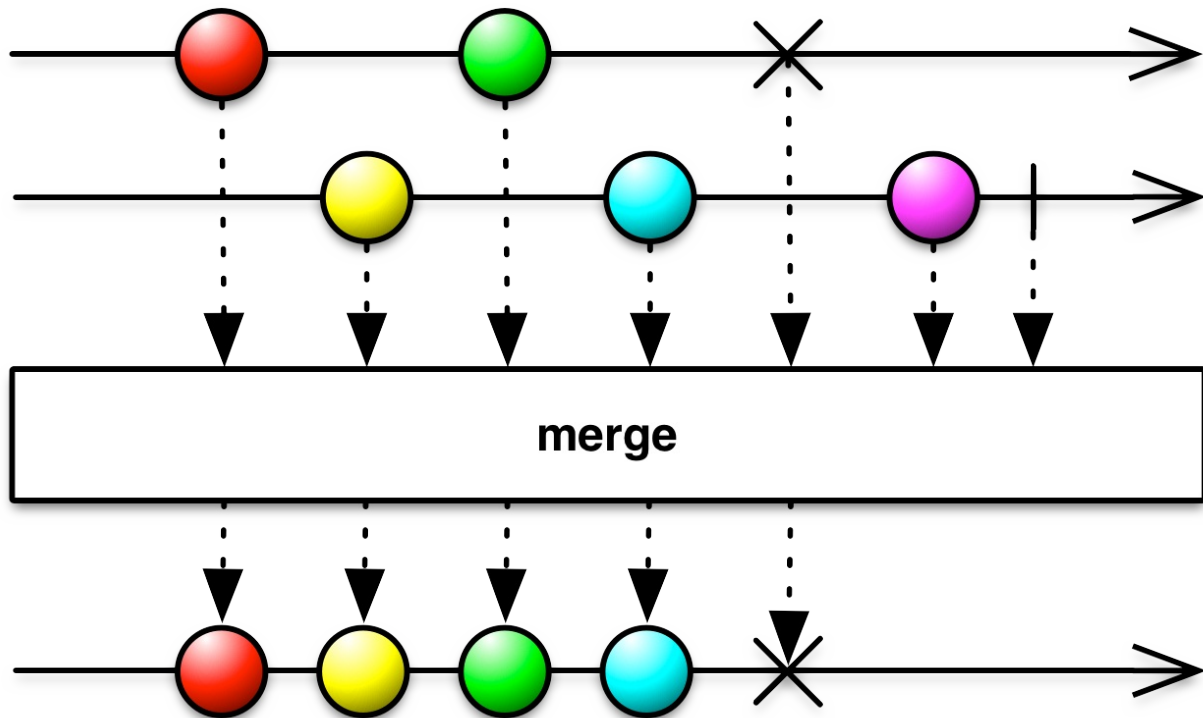
var source = xs
 .join(
 ys,
 function () { return Rx.Observable.timer(0); },
 function () { return Rx.Observable.timer(0); },
 function (x, y) { return x + y; }
)
 .take(5);

var subscription = source.subscribe(
 function (x) { console.log('Next: ' + x); },
 function (err) { console.log('Error: ' + err); },
 function () { console.log('Completed'); });
```

The declarative syntax of the HEADS modelling language for CEP concepts hence greatly simplifies the expression of CEP queries. Moreover, the stream expressed above can be compiled to Java or JavaScript with no modification, the compiler taking care of mapping to the Java and JS version of the ReactiveX APIs.

## Merge

The [Merge operator](#) "combine multiple Observables into one by merging their emissions". See the figure below (taken from ReactiveX documentation) to get an idea of how it works.



This is expressed in the HEADS modelling language using this syntax:

```
stream simpleMerge do
 from [e1 : receivePort?m1 | e2 : receivePort?m2 -> cep1()]
 action sendPort!cep1()
end
```

The same query expressed directly using the ReactiveX API would require about 15 lines of code (in JavaScript or here in Java):

```
//Code sample taken from ReactiveX documentation
Observable<Integer> odds = Observable.just(1, 3, 5).subscribeOn(someScheduler);
Observable<Integer> evens = Observable.just(2, 4, 6);

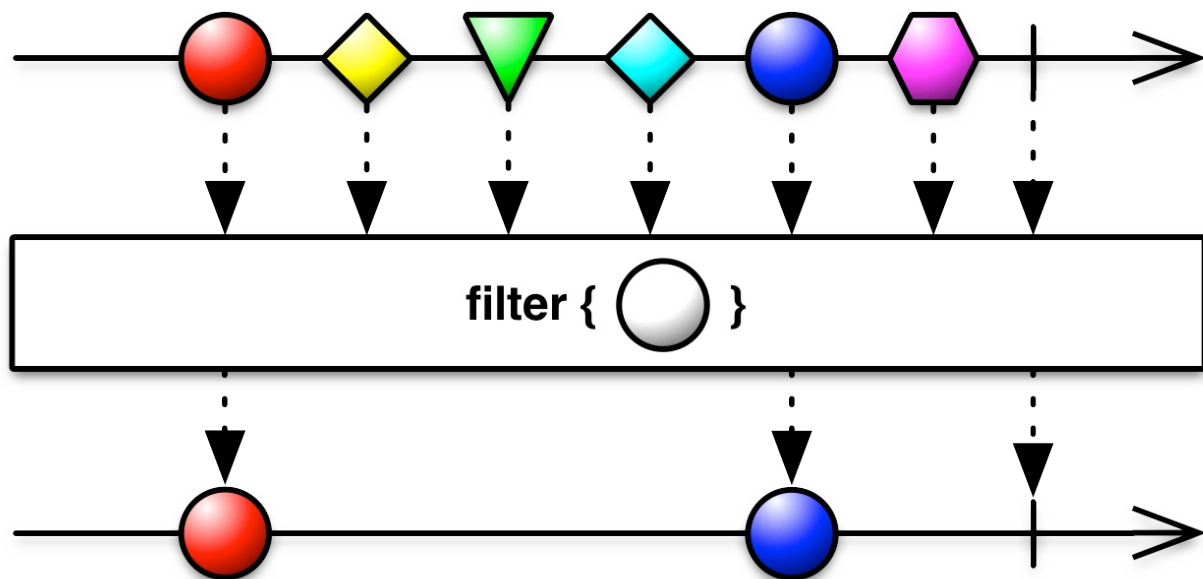
Observable.merge(odds, evens)
 .subscribe(new Subscriber<Integer>() {
 @Override
 public void onNext(Integer item) {
 System.out.println("Next: " + item);
 }

 @Override
 public void onError(Throwable error) {
 System.err.println("Error: " + error.getMessage());
 }

 @Override
 public void onCompleted() {
 System.out.println("Sequence complete.");
 }
 });
```

## Filter

A **Filter** "emits only those items from an Observable that pass a predicate test". See the figure below (taken from ReactiveX documentation) to get an idea of how it works.



This is expressed in the HEADS modelling language using this syntax:

```
operator lessThan4(m : m1) : Boolean
 return m.x < 4

stream filterLessThan4 do
 from m : [e1 : rcv?m1 -> res(e1.x)]::filter(lessThan4(m))
 select a : #0
 action send!res(a)
end
```

The same query expressed directly using the ReactiveX API would require about 15 lines of code (in JavaScript or here in Java):



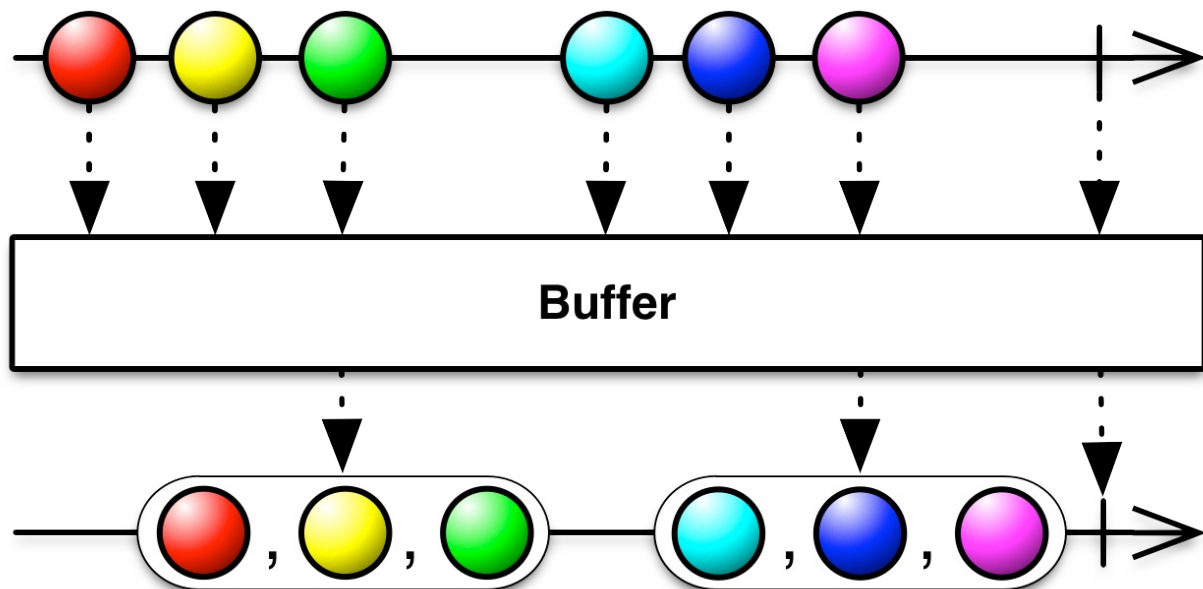
```
//Code sample taken from ReactiveX documentation
Observable.just(1, 2, 3, 4, 5)
 .filter(new Func1<Integer, Boolean>() {
 @Override
 public Boolean call(Integer item) {
 return(item < 4);
 }
 }).subscribe(new Subscriber<Integer>() {
 @Override
 public void onNext(Integer item) {
 System.out.println("Next: " + item);
 }

 @Override
 public void onError(Throwable error) {
 System.err.println("Error: " + error.getMessage());
 }

 @Override
 public void onCompleted() {
 System.out.println("Sequence complete.");
 }
 });
```

## Aggregator and Windows

A [Window \(or Buffer\)](#) "periodically gathers items emitted by an Observable into bundles and emit these bundles rather than emitting the items one at a time". See the figure below (taken from ReactiveX documentation) to get an idea of how it works.



This is expressed in the HEADS modelling language using this syntax:

```
stream lengthW do // compute min, max, average of m2.x on windows of 5 seconds
 from e : [recv?m2]::timeWindow(5000, 5000)
 select avg : average(e.x[]), min : min(e.x[]), max : max(e.x[])
 action send!res2(avg, min, max)
end
```

Aggregators are normal functions that takes arrays (corresponding to the content of a window/buffer) as parameter:

```

function average(x : Integer[]) : Float do
 var i : Integer = 0
 var sum : Integer = 0
 while (i < x.length) do
 sum = sum + x[i]
 i = i + 1
 end
 return sum / x.length
end

```

Those aggregators are typically defined in a reusable libraries that any stream can then use directly.

## Debugging

Traces are a common way of understanding the execution of a program, identify and solve bugs. Traces can automatically be added to trace:

- the initialization of instances, with values for all attributes. Those traces appear in light blue in the figure below.
- the execution of the state machine (which states are being entered/exited, which transition are triggered, etc). Those traces appear in yellow in the figure below.
- the emission/reception of messages on ports. Those traces appear in green in the figure below.
- the affectation of variables. Those traces appear in magenta/purple in the figure below.
- the execution of functions. Those traces appear in dark blue (not present in the figure below).

Using `@debug "true"` and `@debug "false"` the service developer can finely filter the elements he wants to trace.

```

INIT: instance HelloTimer
 period = 1000 counter = 0
TestTimerJS_client(HelloTimer): enters behavior
TestTimerJS_client(HelloTimer): enters behavior:Init
TestTimerJS_client (TimerClient) : timer!timer_start(1000)
TestTimerJS_client (HelloTimer): timer?timer_timeout()
TestTimerJS_client(HelloTimer): exits behavior:Init
TestTimerJS_client(HelloTimer): on timer?timer_timeout from behavior:Init to beh
avior:Init
hello 0
TestTimerJS_client(HelloTimer): property counter changed from 0 to 1
TestTimerJS_client(HelloTimer): enters behavior:Init
TestTimerJS_client (TimerClient) : timer!timer_start(1000)
TestTimerJS_client (HelloTimer): timer?timer_timeout()
TestTimerJS_client(HelloTimer): exits behavior:Init
TestTimerJS_client(HelloTimer): on timer?timer_timeout from behavior:Init to beh
avior:Init
hello 1
TestTimerJS_client(HelloTimer): property counter changed from 1 to 2
TestTimerJS_client(HelloTimer): enters behavior:Init
TestTimerJS_client (TimerClient) : timer!timer_start(1000)

```

## Compile ThingML components to platform specific code

In order to produce executable code, a ThingML configuration should be defined, where components (both platform-independent and platform-specific) need to be instantiated and connected together:

```
configuration TestTimerJava {
 instance timer : TimerJava
 instance client : TimerClientJava
 connector client.timer => timer.timer
}
```

### Test your *things* in a standalone mode

This ThingML program can then be compiled (using the standalone editor) to Java. In the compiler Menu, select Java/JASM. This will generate the Java code, wrap it into a Maven project, compile it and run it. In the terminal/console, you should see:

```
tick
0
tock
1
tick
2
tock
3
```

### Wrap your *things* into Kevoree components

Now, to wrap this program into Kevoree, just select, in the compiler menu, Java/Kevoree. This will generate a set of wrappers that exposes the ThingML components (things) as Kevoree components, and will update the pom.xml file to include the necessary Kevoree plugins. In addition, it will also generate a Kevscript file corresponding to the initial configuration of the system, as described in the ThingML configuration:

```
repo "http://repo1.maven.org/maven2"
repo "http://maven.thingml.org"

//include standard Kevoree libraries
include mvn:org.kevoree.library.java:org.kevoree.library.java.javaNode:release
include mvn:org.kevoree.library.java:org.kevoree.library.java.channels:release
include mvn:org.kevoree.library.java:org.kevoree.library.java.ws:release

//include external libraries that may be needed by ThingML components
include mvn:org.thingml.org.thingml.utils:snapshot

//include Kevoree wrappers of ThingML components
include mvn:org.thingml.generated:TestTimerJava:1.0-SNAPSHOT

//create a default Java node
add node0 : JavaNode
set node0.log = "false"
//create a default group to manage the node(s)
add sync : WSGroup
set sync.port/node0 = "9000"
attach node0 sync

//instantiate Kevoree/ThingML components
add node0.TimerClientJava_TestTimerJava_client : KTimerClientJava
add node0.TimerJava_TestTimerJava_timer : KTimerJava

//instantiate Kevoree channels and bind component
add channel_1324969411 : SyncBroadcast
bind node0.TimerClientJava_TestTimerJava_client.timerPort_out channel_1324969411
bind node0.TimerJava_TestTimerJava_timer.timerPort channel_1324969411
add channel_1324969411_re : SyncBroadcast
bind node0.TimerClientJava_TestTimerJava_client.timerPort channel_1324969411_re
bind node0.TimerJava_TestTimerJava_timer.timerPort_out channel_1324969411_re
start sync
start node0
```

## Deploy and adapt with Kevoree, as usual

After you recompile the project (using `mvn clean install`), you can open this KevScripts into the Kevoree editor and deploy it using Kevoree as a normal Java node.

## Model HD-Service deployment with Kevoree

Please, follow the [Kevoree Book](#)

Here we should discuss the integration between the ThingML implementation models and the Kevoree deployment models.

**todo** Inria to complete this part

# Deploy platform specific code using Kevoree

After the platform specific binaries have been compiled, Kevoree should be used to deploy, initialize and monitor the application.

## Java environment

### The Kevoree Maven Plugin

The Kevoree Maven plugin extracts the Component-Model from the annotations placed in your code, and stores it into a Kevoree Model packed along with the compiled class files. Also, for the ease of use, the Kevoree Maven plugin embeds a Kevoree runner. This runner launches a Kevoree runtime using a KevScript file. This file is supposed to be `src/main/kevs/main.kevs` in the project. Alternatively, you can specify the location of the KevScript file you want to use in the configuration of the plugin. You can also specify the name of the node you want to launch.

```
<project>
 <!-- classic pom.xml file content ... (this hasnt changed) -->
 <build>
 <plugins>
 <plugin>
 <groupId>org.kevoree.tools</groupId>
 <artifactId>org.kevoree.tools.mavenplugin</artifactId>
 <!-- ${kevoree.version} must be v5.4.0-SNAPSHOT or greater -->
 <version>${kevoree.version}</version>
 <executions>
 <execution>
 <goals>
 <goal>generate</goal>
 <goal>deploy</goal>
 </goals>
 </execution>
 </executions>
 <configuration>
 <!-- your Kevoree registry namespace -->
 <namespace>mynamespace</namespace>
 <nodeName>anotherName</nodeName>
 <kevscript>src/main/kevs/anotherModel.kevs</kevscript>
 </configuration>
 </plugin>
 </plugins>
 </build>
</project>
```

### Available actions

Mainly the Kevoree maven plugin is automatically executed at compile time in order to put additional informations in the JAR. (*Mainly model fragment of currently developed type definitions*).

Additionally, user can execute directly the root kevScript file refered in the configuration using the following command:

```
mvn kev:run
```

This will execute Kevoree directly in the maven environnement.

## Javascript

Kevoree-js has 2 different Grunt tasks to process and deploy your project:

- **grunt-kevoree-genmodel:** parses your sources in order to create the corresponding Kevoree model
- **grunt-kevoree:** starts a Kevoree JavaScript runtime using `kevs/main.kevs` KevScript file and `node0` as a default node name.

Those Grunt tasks must be defined in a `Gruntfile.js` at the root of your project.

```
module.exports = function (grunt) {
 require('load-grunt-tasks')(grunt);

 grunt.initConfig({
 kevoree_genmodel: { main: {} },
 kevoree: { main: {} },
 kevoree_registry: {
 src: 'kevlib.json'
 }
 });

 grunt.registerTask('default', 'build');
 grunt.registerTask('build', 'kevoree_genmodel');
 grunt.registerTask('kev', ['build', 'kevoree']);
 grunt.registerTask('publish', ['build', 'kevoree_registry']);
};
```

You can get more details on their own repos [grunt-kevoree](#) and [grunt-kevoree-genmodel](#)

## Service Developers for CEP

A service developer uses ThingML CEP to develop things which should run on CEP engines. With the HEADS IDE he generates the code for the device. Assume, a platform expert provided a plug-in for the HEADS transformation framework to generate Apama EPL (Event Processing Language) code. Then the service developer can generate Apama EPL code for things modeled in ThingML.

A second approach is to develop a service using Apama EPL and create the service as one query. In a second step use the CEP Recommender (described in deliverable D4.3 of the HEADS project).

## Join and Filter

This example joins two streams (equi-join). Each of these is filtered with a condition. The example demonstrates two aspects. First, filter operations and join operations can be executed on different CEP engines. Second, since the filter operations depend only on one stream, the filter operation can be executed before the join operation. This lowers network traffic since only events relevant for the join have to use network connections.


### Original Query

The following query is executed on a central CEP engine.

```
from
 tEvent in all TemperatureEvent() within 10.0
join
 pEvent in all PressureEvent() within 10.0
on
 tEvent.sensorId equals pEvent.sensorId
where
 tEvent.temperature < 20.0 and pEvent.pressure < 50.0
select
 SensorEvent(pEvent.sensorId, pEvent.pressure,
 tEvent.temperature): sensorEvent {
 send sensorEvent to "Output";
 }
```

## Distributed Queries


With the CEP Recommender the query above is split into three parts. Each part can run on its own CEP engine. CEP engine 1 is connected through "CHAN1" with CEP engine 3. CEP engine 2 is connected through "CHAN2" with CEP engine 3.

Part 1  Filter temperature events, executed on CEP Engine 1

```
from tEvent in all TemperatureEvent()
where tEvent.temperature < 20.0
select tEvent: tEvent1 {
 send tEvent1 to "CHAN1";
}
```

Part 2  Filter pressure events, executed on CEP Engine 2

```
from pEvent in all PressureEvent()
where pEvent.pressure < 50.0
select pEvent : pEvent1 {
 send pEvent1 to "CHAN2";
}
```

Part 3  Join operation, executed on CEP Engine 3



```
monitor.subscribe("CHAN1");
monitor.subscribe("CHAN2");
from
 tEvent in all TemperatureEvent() within 10.0
join
 pEvent in all PressureEvent() within 10.0
on
 tEvent.sensorId equals pEvent.sensorId
select
 SensorEvent(pEvent.sensorId, pEvent.pressure,
 tEvent.temperature): sensorEvent {
 send sensorEvent to "Output";
 }
```

## How to develop CEP Queries within the HEADS IDE

This section describes how to get started with complex event processing (CEP) development and how to use the recommender to partition queries into smaller queries for deployment to distributed services.

## How to use the Apama Studio

There are four components in Apama Studio for CEP development: The EPL Editor and debug environment, the Query Builder, the Event Modeler, and the Dashboard Builder. This How To describes the preferred components to use inside the HEADS IDE.

### Components of Apama Studio

- Developing EPL with Apama Studio: EPL Editor

To get started with EPL inside the Apama Studio use the tutorial on EPL monitor scripts. In Apama Studio open the Welcome page, go to the Tutorials section. There are three tutorials. The last one with the title 'Developing an Application with MonitorScript' is the most important one. It explains the Apama Event Programming Language (EPL), called "MonitorScript". A Cheat Sheet guides you through this tutorial.

In addition there are several samples introducing topics of Apama. In Apama Studio open the Welcome page, go to the Samples section. These six samples can be imported into an Eclipse workspace as Apama projects. From the Apama Workbench perspective you can launch these samples, one at a time. The Readme files suggest small exercises to modify these samples. The source of the samples is the folder 'demos' in the Apama installation. The folder 'samples' contains more in-depth examples for components of Apama and the connection with other products like Universal Messaging or Terracotta BigMemory.

Steps to perform for a sample:

- Import project into workspace
- Explore files in the Apama Workbench perspective. Toggle on the 'Show All Folders' button in the toolbar of the 'Workbench Project View'.
- Launch the sample with the 'Launch Control Panel'. This starts a correlator (Apama CEP runtime process) listening on port localhost:15903 (default). In addition a dashboard viewer is started in a separate window and events are sent to the correlator.

There are three perspectives in Apama Studio, Apama Developer, Apama Runtime, and Apama Workbench. Apama Developer is similar to the Java perspective in JDT. Apama Workbench is a simplified combination of Apama Developer and Apama Runtime. It is tailored to work on a single Apama project. This can be selected in the Workbench Project View.

- Using the Query Builder in Apama Studio

The Query Builder is a combined graphical and textual editor for Apama Queries. These queries are new with Apama 5.3 and they allow a purely descriptive way to define stream queries. These queries are capable of large time windows.

- Event Modeler

This component helps to model state machines in EPL. This is not the focus inside the HEADS IDE and therefore not described in detail. Refer to the Apama documentation which is included in the Eclipse Help.

- Dashboard Builder

The dashboard builder is used for the development of dashboard to visualize stream data. This is not part of the HEADS IDE and not in the focus of HEADS.

## How to use the CEP Recommender

Suppose a service developer implements an EPL query. This query can be partitioned into parts which define new streams of events which are filtered, joined, projected and otherwise transformed. The parts of the query can run on other nodes than one central CEP node which runs on a powerful machine. With the help of platform experts the service developer can run the CEP Recommender to get a recommendation for a set of EPL queries which run on several nodes according to their CEP capabilities. The platform experts have to describe the CEP capabilities of the nodes, the CPU capacities. In addition the service developer has to describe the CPU capacity consumption of the operations in the query and the event rates on the input streams and the selectivity of filter operations.

With this information the CEP Recommender calculates recommended partitions with model transformations through Triple Graph Grammars. The algorithmically steps are

- Build the syntax model, a kind of an abstract syntax tree from the EPL syntax.
- Build the Query Operator Tree with Xtend from the syntax model.
- Transform the Query Operator Tree with eMoflon using Triple Graph Grammars into the Meta Model. The Meta Model adds information for the partitioning to the Query Operator Tree.
- Transform the Meta Model with eMoflon using Triple Graph Grammars into a partitioning of the query. This partitioning is one of a list of partitionings since the Tripple Graph Grammar transformation is not unique. The result of this algorithm is a list of recommendations, each partitioning the original query into a set of simpler queries. The user has to decide, which partitioning fits best. Each partitioning is a model instance, containing a partitioned query, i.e. a set of simpler queries which have together the same functionality than the input query. Such a set of queries can be deployed to the nodes.

This approach is also described in deliverable D4.2, section 4.2.

The steps for the different roles in the HEADS IDE are:

- The service developer develops query logic (EPL text, Apama query language is not implemented yet). This is the main input, which is put into a text file with the extension epl.
- The service developer starts with the context menu of this file, entry CEP Recommender, the algorithm. The result is the Meta Model, which is visible in the Meta Data Editor.
- In this editor, the service developer adds statistical data: event rates, event sizes, selectivity of filters, etc.
- In the same editor the platform expert provides nodes info: CPU capacity of the nodes and CEP capabilities of the nodes.
- When all the input is collected the service developer or the platform expert starts the generation of the recommendations with the same context menu on the epl file. The recommendations are in a folder besides the epl file.

The service developer or the platform expert has to decide which recommendation fits best to be deployed on the distributed nodes.

- Future developments for the rest of the project

The CEP Recommender should also work on the Apama query language, which is a feature first included in Apama 5.3. We will work on the EPL generation of the EPL queries of a recommendation. In addition automatic deployment to the distributed nodes is a topic for the future development.

# Migrate from Kevoree v5.3.x to v5.4.x

We have rewritten the Kevoree Registry in 2016 in order to strengthen our *database* and to add *user authentication* and a *namespace* concept. Those changes impacted all the different platforms. They had so much impact that v5.4.x **is no longer** backward-compatible with v5.3.x and below.

## Kevoree Registry: users & namespaces

In order to give more control over what is published on the registry, we have added **users** and **namespaces** to it.

This means that from now on, valid credentials are mandatory to publish models to the registry. To get those credentials, an account has to be created on the registry, [here](#).

Creating a new user will automatically create a namespace with the **same name** so you can directly publish Kevoree models using those credentials.

But it also allows you to create new namespaces (if not already created), from the [registry website](#) using the **New namespace** button when authenticated.

## One configuration file for all platforms

The Kevoree registry connection information and user credentials can now be stored independently from the Kevoree platform you are using. It has to be located in `$HOME/.kevoree` in a JSON file named `config.json` :

```
{
 "user": {
 "login": "YOUR_NAME_HERE",
 "password": "YOUR_PASSWORD_HERE"
 },
 "registry": {
 "host": "registry.kevoree.org",
 "port": 443,
 "ssl": true,
 "oauth": {
 "client_secret": "kevoree_registryapp_secret",
 "client_id": "kevoree_registryapp"
 }
 }
}
```

For security reasons, this file should have limited read access on your operating system

This file will be read by the KevScript interpreter in order to connect to the registry and resolve TypeDefinitions & DeployUnits.

If you want to connect to a third-party registry, just modify the content of the `$HOME/.kevoree/config.json` file according to your registry location and credentials.

## The new KevScript resolver

We have introduced two new keywords in the language in order to solve DeployUnit resolutions by the KevScript interpreter:

- LATEST
- RELEASE

TypeDefinition versions can now be specified using whether an **integer** or **LATEST** which will indicate to the KevScript interpreter which version the service developers expect. For the DeployUnits, the possibilities are whether **LATEST** or **RELEASE**. Refer to the table at the end of this [wiki page](#) for more details about those keywords.

## Platform specific details

You can read more about the migration for a specific platform here:

- [Java platform migration](#)
- [JavaScript platform migration](#)

**For Service Operators**

## Visualize the configuration and status of a running HD-Service

Please follow the tutorials <http://kevoree.org/practices/level0> and <https://github.com/kevoree/kevoree-eclipse-plugin/>

**todo** Inria to complete this part

## Modify the configuration and deployment of a running HD-Service

<http://kevoree.org/practices/level2>

<http://kevoree.org/practices/level3>

**todo** Inria to complete this part



## Re-deploy/adapt/reconfigure a running HD-Service

In order to reconfigure a Kevoree model and trigger adaptations of your system after re-deployment you need a [Kevoree Editor](#) ([Java Editor](#) or [Web Editor](#)) and some knowledge of the Kevoree Script ([KevScript](#)) language.

### Using the Web Editor

You need to have a group providing a WebSocket endpoint in order to pull/push your models with the Kevoree Web Editor (i.e WSGroup - Java platform or WebSocketGroup - JavaScript platform)

#### Pulling your model

Open a Web browser to this location: `http://editor.kevoree.org/?host=SOME_HOST&port=SOME_PORT`

This will automatically pull the model using WebSocket and the `host:port` you specified in the URL parameters.

Now you should have your model displayed in the editor.



#### Reconfigure your model

##### Using KevScript

If you open the KevScript editor (using the button in the top panel) the content is updated to reflect your current model content.

You can edit this KevScript directly and then press **Run** so that it modifies your model.

Once you are done reconfiguring your model you are going to want to deploy it. (You can skip to **Re-deploy your model**)

#### Re-deploy your model

To deploy your model you just have to click on the node you want to deploy the model to.

You now have its properties displayed in a pop-up window (instance name, dictionary attributes, network settings). Because you pulled your model directly from your node in the first you don't have to specify the network settings to push a model to it.

You can just simply press the **Push** button, and your reconfigured model will be send to your Kevoree platform and it will make the necessary adaptations.

## Service Operators for CEP

A service operator has to deploy Apama as a CEP engine on

- cloud-nodes
- Nodes powered by Raspberry Pi 2/3
- Nodes powered by Intel-Atom like Dell Edge devices In addition, the code written in EPL (Event Processing Language) has to be deployed by injection on these CEP engines. This can be EPL code generated by the CEP Recommender or originally developed by a service developer. If the event processing network contains nodes which are not supported by a CEP engine, C code or similar has to be generated with the HEADS transformation framework for this device.

## Credits

This work has been funded by [EU FP7 HEADS project](#).



## Contributors

By alphabetical order (family name):

- Olivier Barais (Inria and University of Rennes 1)
- Vladimir Cvetkovic (M2M)
- Franck Fleurey (SINTEF)
- Brice Morin (SINTEF)
- Martin Skorsky (SAG)
- Maxime Tricoire (Inria)
- Anatoly Vasilevskiy (SINTEF)
- Walter Waterfeld (SAG)

# License

## HEADS Methodology (this document)



HEADS Methodology by [HEADS Consortium](#) is licensed under a [Creative Commons Attribution 4.0 International License](#).



## HEADS IDE

- [ThingML](#): LGPL-3.0
- [Kevoree](#): LGPL-3.0
- [Software AG's Apama \(Community Edition\)](#): Free with limitations, see <http://www.apamacommunity.com/terms-conditions/>