1. HEADS Methodology

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

1. 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.

1. 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!**

* + 1. 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

}

}

* + 1. 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.

* + - 1. 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.

* + - 1. 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 nextIntdoes not overflow the ThingML Integer (*i.e.*, a Java short).

* + 1. 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).

* + - 1. in C/C++

To do so one should adapt/wrap a native C/C++ library in such a away 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 libarary 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 <cstdarg>

#include \"BinarySensor.h\"

using namespace TinyOpenZWaveApi;

"

{

property bs : BinarySensor

provided port bsport {

receives initialize

}

//these are two internal ports shoud 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 bland 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 wrrapping library, i.e. void valueupdate(int)(see the point 2), the wrapper calls the callback function value\_change\_binarysensor\_callback(). Finnally, 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.

* + - 1. in Java

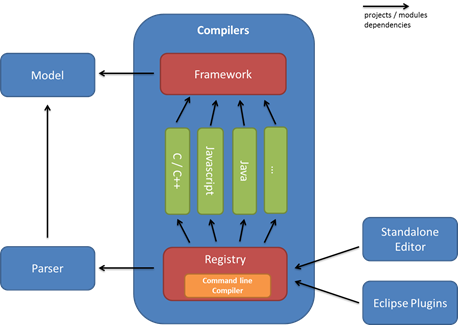
In Java, the easiest way to call ThingML code from a plain Java class is to

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.
4. Extend the ThingML transformations to compile code for a new platform

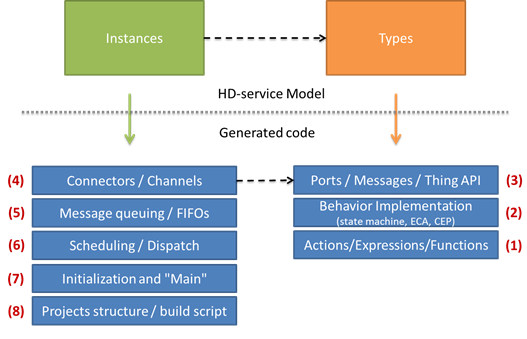
When a platform expert needs to address a new platform, he typically needs to use the HEADS transformation framework, described in detail in D2.2. This modular Object-Oriented framework defines a set of extension points, each encapsulated as a Java class. For each different language (e.g. Java, JavaScript 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 plaftorm 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 contraints and specificities of the new platform.

* 1. 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.

* + 1. 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.

* + 1. 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.

* + 1. 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.

* + 1. 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.

* + 1. 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.

* + 1. 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.

* + 1. 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.

* + 1. 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.

* 1. 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.

* + 1. 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) {//if(...) {...} else {...}

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) {//while(...) {...}

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 redfine 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

* + 1. 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) {

}

}

Plaform 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 + ")");

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](https://github.com/steelbreeze/state.js) (~200 likes on GitHub and ~1000 Download a month on NPM), that were rapidly fixed by the repository maintainer.

* + 1. 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("}");

}

}

* + 1. 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 developper 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.split(';'));\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.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");

}

* + 1. 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 in 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);}

}

* + 1. 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.receivetimer\_timeoutOntimer.bind(TestTimerJS\_client));

TestTimerJS\_client.getTimer\_startontimerListeners().push(TestTimerJS\_timer.receivetimer\_startOntimer.bind(TestTimerJS\_timer));

TestTimerJS\_client.getTimer\_cancelontimerListeners().push(TestTimerJS\_timer.receivetimer\_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 plaform 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 illustrate 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;

}

}

}

}

* + 1. Project structure / Build script

To make the HEADS components easily reusable, with our without the HEADS technologies, they must be properly package. For Java, we for example generate proper Maven project, for JavaScript, NPM projects, and for C, Makefile. If a platform expert would like to use Gradle instead of manage, 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.json");

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": {

}

}

* 1. 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 interprete this annotation and generate multi-threaded code, which [wraps the code](file:///home/barais/git/methodology/heads_methodology/README.odt/(https://github.com/SINTEF-9012/ThingML/blob/master/compilers/c/src/main/resources/ctemplates/fork.c)) 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 be 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.

1. 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](https://github.com/kevoree/kevoree-js" \l "create-your-first-component), built it and deploy it

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

* 1. Introduction

As you know Kevoree is a multiplatform distributed model tool.

Two platforms are currently maintained :

* [Java](https://github.com/dukeboard/kevoree)
* [Javascript](https://github.com/kevoree/kevoree-js)

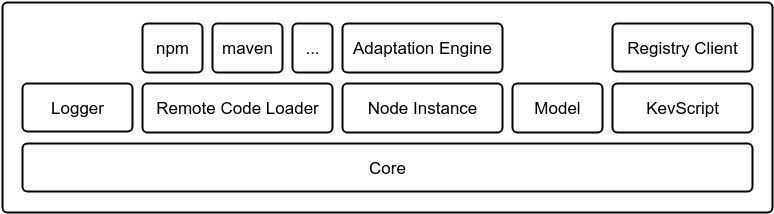
One is currently in development :

* [C#](https://github.com/kevoree/kevoree-dotnet)

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 implentation 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#.

* 1. Global architecture

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


* 1. Components
* [**Generalities**](file:///home/barais/git/methodology/heads_methodology/README.odt/generalities.md): A few cross platform advices
* [**Kevoree Model**](file:///home/barais/git/methodology/heads_methodology/README.odt/model.md): How to port Kevoree's data model to your platform
* [**Remote code loader**](file:///home/barais/git/methodology/heads_methodology/README.odt/remote_code_loader.md): How to load remote code in your runtime
* [**Kevoree Model generation**](file:///home/barais/git/methodology/heads_methodology/README.odt/model_generator.md): How to quickly obtain a kevoree model code base
* [**Logger**](file:///home/barais/git/methodology/heads_methodology/README.odt/logger.md): Specifications of the logging system.
* [**Core**](file:///home/barais/git/methodology/heads_methodology/README.odt/core.md): The [Model@Runtime](mailto:Model@Runtime) conductor
* [**Generate instances of the Kevoree Metamodel**](file:///home/barais/git/methodology/heads_methodology/README.odt/model_generator.md): How to quickly obtain a kevoree model code base
* [**Registry client**](file:///home/barais/git/methodology/heads_methodology/README.odt/registry_client.md): A simple REST client for the registry
* [**Code generator**](file:///home/barais/git/methodology/heads_methodology/README.odt/code_generator.md): How to generate a Component project from a Type Definition.
* [**KevScript tool**](file:///home/barais/git/methodology/heads_methodology/README.odt/kevscript.md): Reading kevscript, generating valid kevscript...
* [**Runtime**](file:///home/barais/git/methodology/heads_methodology/README.odt/runtime.md): A Bootstrap is a runtime tool dedicated to the startup of a node instance
* **Kevoree's components**:  
  + [**Component**](file:///home/barais/git/methodology/heads_methodology/README.odt/components/component.md) : Component development guide
  + [**Node**](file:///home/barais/git/methodology/heads_methodology/README.odt/components/node.md) : Node development guide
  + [**Group**](file:///home/barais/git/methodology/heads_methodology/README.odt/components/group.md) : Group development guide
  + [**Channel**](file:///home/barais/git/methodology/heads_methodology/README.odt/components/channel.md) : 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](http://registry.kevoree.org/) by default)
* **Local kevoree editor**:
  + Node project : <https://github.com/kevoree/kevoree-web-editor>
  + Docker container : <https://github.com/kevoree/docker-image-editor>

1. Generalities
   1. 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 Kecoree Core Team if you found them hazy.

1. Every Kevoree related component should be prefixed with "kevoree" (e.g. [kevoree-book](https://github.com/kevoree/kevoree-book), [kevoree-browser-runtime](https://github.com/kevoree/kevoree-browser-runtime), [kevoree-web-editor](https://github.com/kevoree/kevoree-web-editor))
2. Every component related to a specific platform should be prefixed with "kevoree-${platform}" (e.g. [kevoree-js](https://github.com/kevoree/kevoree-js), [kevoree-dotnet](https://github.com/kevoree/kevoree-dotnet)).
3. Every generic component (i.e. described in the [platform itegration](file:///home/barais/git/methodology/heads_methodology/README.odt/index.md) part of the book) have a common name who should be followed platforms wide (e.g. [kevoree-js-kevscript](https://github.com/kevoree/kevoree-js-kevscript), [kevoree-donet-annotation](https://github.com/kevoree/kevoree-dotnet-annotation)). The following list is a uncomprehensive list of reserved keywords.

* kevscript
* core
* runtime
* model
* annotation

1. Every deployable component name should follow the form "kevoree-${platform}-${componentType}-${typeDefName}" where **componentType** is *node*, *group*, *chan* or *comp* (Respectivelly 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](https://github.com/kevoree/kevoree-dotnet-group-remotews), [kevoree-js-comp-ticker](https://github.com/kevoree/kevoree-js-comp-ticker)).
2. Kevoree model
   1. Introduction

The meta-model of Kevoree is defined [here](https://github.com/dukeboard/kevoree/blob/master/kevoree-core/org.kevoree.model/metamodel/org.kevoree.mm)  
using [KMF](http://kevoree.org/kmf/)'s modeling language.

* 1. 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.
  1. Strategies
     1. 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
  + 1. 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](file:///kevoree-schema.html).

* + 1. 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](http://www.ikvm.net/), a tool to convert JARs to DLL.

The whole process is detailed [here](https://github.com/kevoree/kevoree-dotnet-ikvm/wiki).

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.

* + 1. Kevoree Meta-model V5 JsonSchema

{

"$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"}

}

}

1. Core
   1. 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](mailto:model@run.time))
* **keeps track of model changes**
  + 1. 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.

* + 1. Deployment phase

Using pseudo-code, the deployment algorithm ([model@run.time](mailto: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

val 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;

}

}

* + 1. 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)

* + 1. 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

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:

var AbstractComponent = require('kevoree-entities').AbstractComponent;

module.export = AbstractComponent.extend({

toString: 'MyComponent',

doSomethingWithCore: function () {

this.core.deploy(aModel, function (err) {

if (err) {

// problem with adaptation: not done

} else {

// adaptations made

}

});

}

});

1. Remote code loader
   1. 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

* 1. Existing implementations
     1. Java

**Repo**: [maven](https://maven.apache.org/)  
**Loader**: ClassLoader named [KCL](https://github.com/dukeboard/kevoree-classloading-framework)

* + 1. Javascript

**Repo**: [npm](https://www.npmjs.com/)  
**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**

* + 1. C#

**Repo**: [Nuget](https://github.com/kevoree/kevoree-dotnet-nuget-loader)  
**Loader**:  
The code isolation is based on AppDomain and the MEF Framework.

The [Kevoree Dotnet Nuget Loader](https://github.com/kevoree/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.

1. 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](file:///home/barais/git/methodology/heads_methodology/README.odt/runtime.md)
* Node
* [Model generator](file:///home/barais/git/methodology/heads_methodology/README.odt/model_generator.md)

1. Registry client
   1. Description

The registry client is a a simple JSON-REST client for the Registry Kevoree.

It should be able to do the operations specified on this page : <https://github.com/kevoree/kevoree-registry>

It is probably one of the simplest part of the whole codebase.

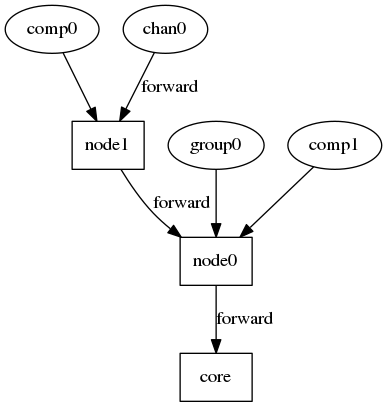
1. Logger
   1. Format

The format of a log trace is *hh:mm:ss.fff log\_level emiter message* where :

* **hh** : The current hour in the day (24h)
* **mm** : The current minute in the hour
* **ss** : The current second in the minute
* **fff** : The current millisecond in the second
* **log\_level** : The log level of the message
* **emiter** : The unique identifier of the emiter (i.e. the unique id of the component)
* **message** : The logged message
  1. Log levels :

| Name | Priority level | Comment |
| --- | --- | --- |
| TRACE | 1 |  |
| DEBUG | 2 |  |
| INFO | 3 |  |
| WARN | 4 |  |
| ERROR | 5 |  |
| NONE | 6 | This level can't be used to send a message. It exists only for filtering purpose. |

* 1. Architecture



The logging system is defined around a chain of filtering.  
A component which want to send a message will actually forward it his parent in the chain which will have responsibility to determine if it will forward it through. A message is logged only if it reach the core which is the only one allowed to print a log message.

* + 1. Filtering algorithm

A message is forwarded if the local level of logging (LOCAL\_LEVEL) is lower or equals to the level of the logged messages (MESSAGE\_LEVEL).

LOCAL\_LEVEL <= MESSAGE\_LEVEL

For example for node1 and node0 with respectively logging levels of INFO and ERROR, if comp0 send a message with level INFO it will be forwarded by node1 but will be discarded by node0 (and consequently not printed).

But if chan0 send a message of level ERROR, it will be forwarded by both node1 and node0 and then printed by the core.

1. Model generator
   1. 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.

* Publication is done by calling a POST method, documented here : [https://github.com/kevoree/kevoree-registry#push-model](https://github.com/kevoree/kevoree-registry" \l "push-model) (using the [registry client](file:///home/barais/git/methodology/heads_methodology/README.odt/registry_client.md))
* The registry's REST API expects to receive the published model formated in JSON. This is usually done by building an object structure using the [model](file:///home/barais/git/methodology/heads_methodology/README.odt/model.md) and then marshaling it in JSON.
* Implementing a model generator can be done by static code analysis or reflection.
  1. Existing implementations
     1. Java

A maven plugin ([kevoree-maven-plugin](https://github.com/dukeboard/kevoree/tree/master/kevoree-tools/org.kevoree.tools.mavenplugin)) 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).

* + 1. Javascript

For the JavaScript platform, the model generation is made by a [Grunt task](https://www.npmjs.com/package/grunt-kevoree-genmodel) 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**:

* **dic\_XXX**: for Dictionary Attribute
* **in\_XXX**: for input port
* **out\_XXX**: for output port

By reading this, the JavaScript model generator is able to generate the appropriate TypeDefinition and then publish it on the Kevoree registry, on demand, using another [Grunt task](https://www.npmjs.com/package/grunt-kevoree-registry).

* + 1. C#

Their 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](https://github.com/kevoree/kevoree-dotnet-model-generator) to publish the package to a Kevoree registry
3. Registry client
   1. Description

The registry client is a a simple JSON-REST client for the Registry Kevoree.

It should be able to do the operations specified on this page : <https://github.com/kevoree/kevoree-registry>

It is probably one of the simplest part of the whole codebase.

1. Code generator

The Kevoree Registry is a public application, accessible at [http://registry.kevoree.org](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.

* 1. Example

In JavaScript, the code generator is provided by the Yeoman [generator-kevoree](https://github.com/kevoree/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

.

├── browser

│   ├── kevoree-comp-ticker.html

│   └── ui-config.json

├── Gruntfile.js

├── kevs

│   └── main.kevs

├── lib

│   └── Ticker.js

├── node\_modules

│   ├── grunt

│   ├── grunt-browserify

│   ├── grunt-contrib-uglify

│   ├── grunt-contrib-watch

│   ├── grunt-kevoree

│   ├── grunt-kevoree-genmodel

│   ├── grunt-kevoree-registry

│   └── kevoree-entities

├── package.json

└── README.md

With the **Ticker** skeleton be:

var AbstractComponent = require('kevoree-entities').AbstractComponent;

var Ticker = AbstractComponent.extend({

toString: 'Ticker',

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 \*/ },

uiController: function () {

return [function () {

// ui controller

}];

}

});

module.exports = Ticker;

1. Runtime
   1. 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](http://oss.sonatype.org/service/local/artifact/maven/redirect?r=public&g=org.kevoree.platform&a=org.kevoree.platform.standalone&v=RELEASE))
* Node.js runtime, which is available on [npm](https://www.npmjs.com/package/kevoree-nodejs-runtime)
* Browser runtime, which is also targeting the JavaScript platform but runs in Web Browsers and allows components to provide a User Interface. ([Browser runtime](http://runjs.kevoree.org/))

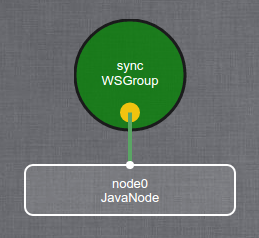
By default, every runtime must be able to start a Kevoree environment without giving any input. In such case, the runtime must create what we call a "default bootstrap model", in KevScript this model looks like this:

// default bootstrap model targeting the JavaNode

add node0 : JavaNode

add sync : WSGroup

attach node0 sync

And in the model editor:  


* 1. Dependencies

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

* [core](file:///home/barais/git/methodology/heads_methodology/README.odt/core.md)
* [model API](file:///home/barais/git/methodology/heads_methodology/README.odt/model.md)
* [kevscript interpreter](file:///home/barais/git/methodology/heads_methodology/README.odt/kevscript.md)
* [remote code loader](file:///home/barais/git/methodology/heads_methodology/README.odt/remote_code_loader.md)
* [logger](file:///home/barais/git/methodology/heads_methodology/README.odt/logger.md)

1. KevScript Tools
   1. Description

In order to manipulate models, we have created a scripting language that is called **KevScript**.  
*This language is not a* [*general purpose language*](https://en.wikipedia.org/wiki/General-purpose_programming_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.*

* 1. Implementations

The KevScript grammar is written using [Waxeye](http://waxeye.org/). 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.

* 1. 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 | AddRepo | Namespace | Start | Stop | Pause

Add <- AddToken ws NameList ws :':' ws TypeDef # add group0, group1 : WebSocketGroup

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

Move <- MoveToken ws NameList ws InstancePath # move node0.comp0, node2.\* node1

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

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

Set <- SetToken ws InstancePath ?(:'/' InstancePath) ws :'=' ws RealString # set node0.comp0.myAtt = 'foo'

Network <- NetworkToken ws InstancePath ws String2 # network node1.lan.eth0 192.168.0.1

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

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

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

Include <- IncludeToken ws String :':' String2 # include npm:kevoree-chan-websocket

NameList <- InstancePath ws \*(:[,] ws InstancePath) # node42

TypeDef <- TypeFQN ?(:'/' Version) # FooType/0.0.1 (specific vers.)

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) \*(:[.] (Wildcard | String)) # node0.\*.att

Wildcard <- '\*'

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

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

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

Version <- +[a-zA-Z0-9.\_-]

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 <: 'add'

RemoveToken <: 'remove'

MoveToken <: 'move'

SetToken <: 'set'

AttachToken <: 'attach'

DetachToken <: 'detach'

NetworkToken <: 'network'

BindToken <: 'bind'

UnbindToken <: 'unbind'

NamespaceToken <: 'namespace'

StartToken <: 'start'

StopToken <: 'stop'

PauseToken <: 'pause'

Comment <: '//' ?Line

eol <: '\r\n' | '\n' | '\r'

ws <: \*([ \t] | eol)

# =================

# End Void Non-terminals

1. Components

* [Component](file:///home/barais/git/methodology/heads_methodology/README.odt/component.md)
* [Node](file:///home/barais/git/methodology/heads_methodology/README.odt/node.md)
* [Channel](file:///home/barais/git/methodology/heads_methodology/README.odt/channel.md)
* [Group](file:///home/barais/git/methodology/heads_methodology/README.odt/group.md)

1. Component
   1. Description

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

* + 1. 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](https://www.arduino.cc/).
  1. 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#).

* + 1. Input

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

* + 1. Output

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

* + 1. Start

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

* + 1. Stop

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

* + 1. Update

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

* + 1. Param

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

* + 1. 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. For more detail read the [Logger](file:///home/barais/git/methodology/heads_methodology/logger.md) page
* 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.
  1. Naming rules reminder

As defined in the [generalities](file:///home/barais/git/methodology/heads_methodology/generalities.md) 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).

1. Node
   1. Description

A node is a component which receive a set of traces (i.e. a list of differencies 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.

* 1. 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#).

* + 1. Start

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

* + 1. Stop

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

* + 1. Update

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

* + 1. Param

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

* + 1. 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. For more detail read the [Logger](file:///home/barais/git/methodology/heads_methodology/logger.md) page
* 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.

1. Group
   1. 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).

* + 1. 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.
  1. 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#).

* + 1. Start

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

* + 1. Stop

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

* + 1. Update

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

* + 1. Param

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

* + 1. 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. For more detail read the [Logger](file:///home/barais/git/methodology/heads_methodology/logger.md) page
* 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.
  1. Naming rules reminder

As defined in the [generalities](file:///home/barais/git/methodology/heads_methodology/generalities.md) part, every group must follow this name rule : kevoree-${platform}-group-${groupName} (e.g kevoree-js-group-ws, kevoree-dotnet-group-remotews).

1. Channel
   1. 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.

* + 1. 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.
  1. 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#).

* + 1. Start

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

* + 1. Stop

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

* + 1. Update

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

* + 1. Param

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

* + 1. 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. For more detail read the [Logger](file:///home/barais/git/methodology/heads_methodology/logger.md) page
* 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.
  1. Naming rules reminder

As defined in the [generalities](file:///home/barais/git/methodology/heads_methodology/generalities.md) part, every channel must follow this name rule : kevoree-${platform}-chan-${channelName} (e.g kevoree-js-chan-ws, kevoree-dotnet-chan-remotews).

1. 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

@Library(name = "Java")

public class MyFirstChannel implements ChannelDispatch {

@KevoreeInject

ChannelContext channelContext;

@Override

public void dispatch(final Object payload, final Callback callback) {

for (Port p : channelContext.getLocalPorts()) {

p.call(payload, callback);

}

}

}

This naive implementation basically implements a direct call. However, instead of a simple p.call it is possible to call third-party API 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 component are deployed.

You can follow this tutorial on how to [make your own Kevoree channel](http://kevoree.org/practices/level5/) for more details about channels.

1. For Service Developers

We have identified 4 different tasks for the Service Developer.

1. 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.

* 1. 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

}

}

* 1. 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)

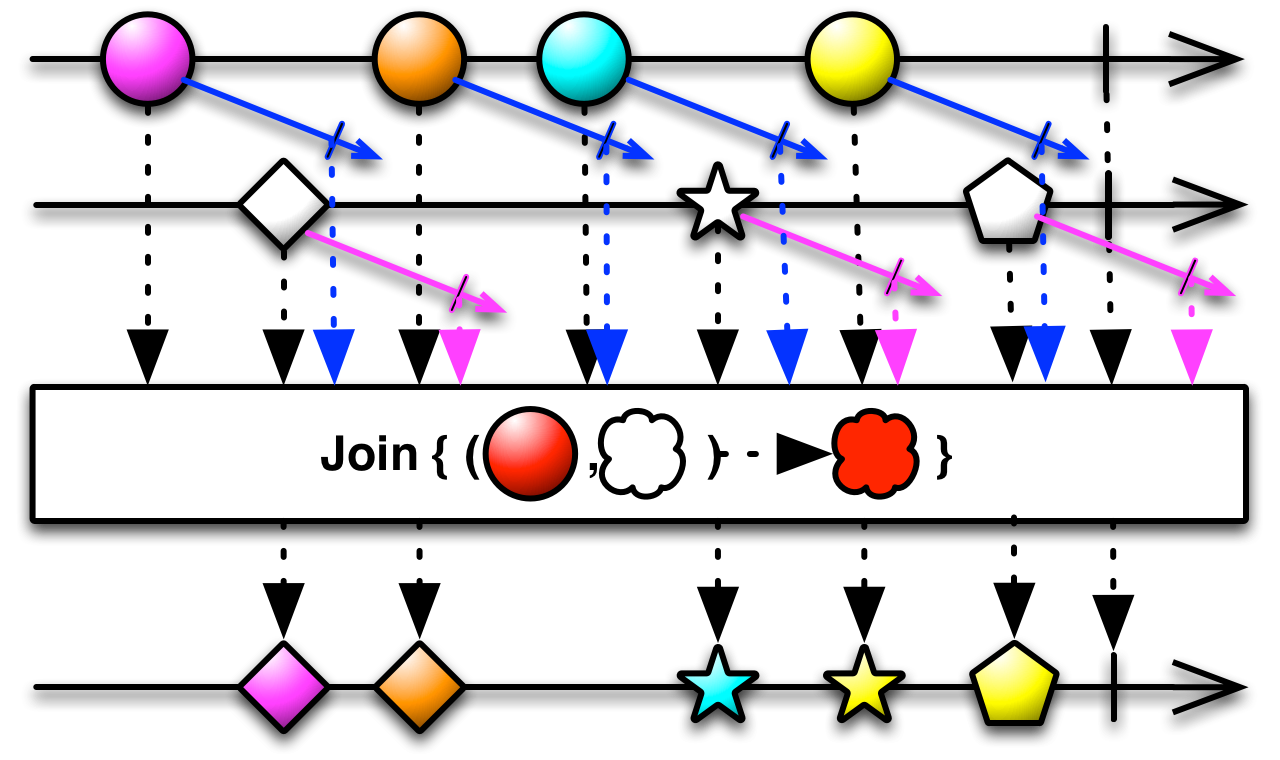
In addition to the "normal" statements common with those language, HEADS provides:

* send a message myPort!myMessage(), myPort!myMessage2(a, b, 0) for asynchronous message passing between components
  1. 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](http://reactivex.io/).

* + 1. Join

The [join operator](http://reactivex.io/documentation/operators/join.html) "*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); },

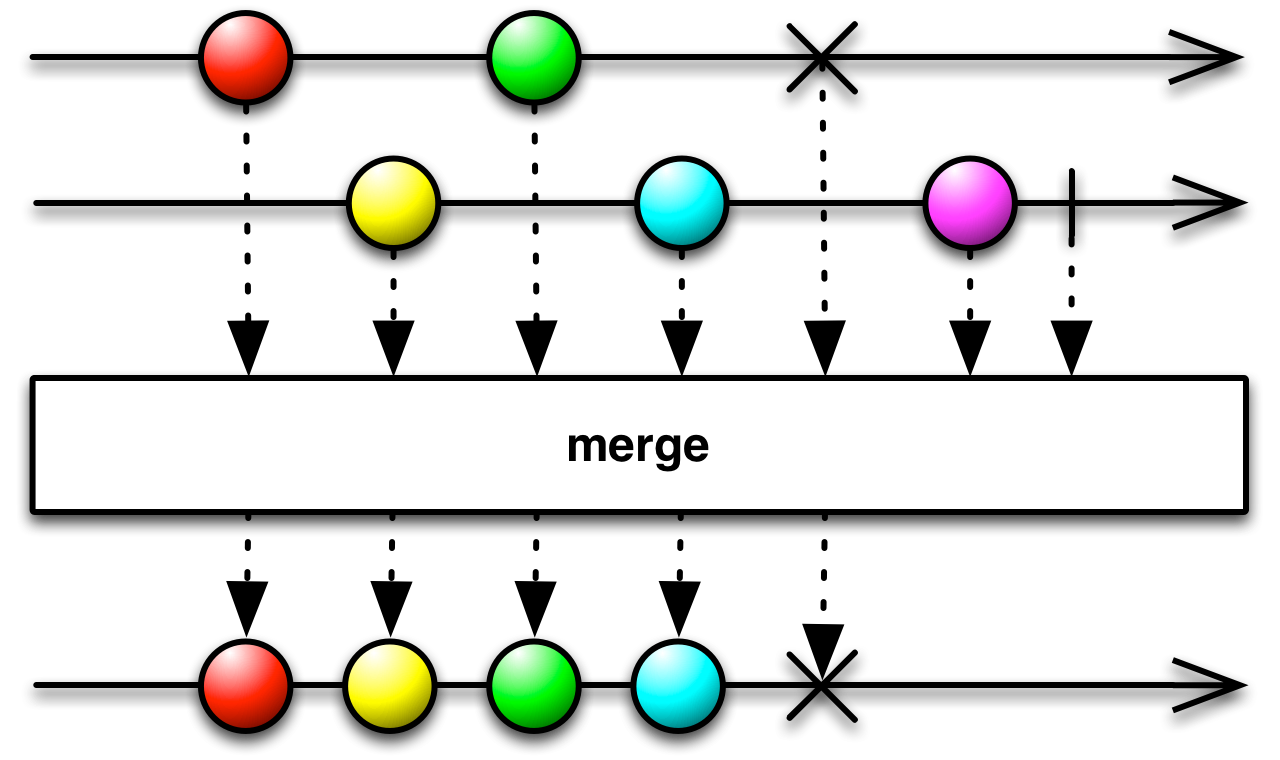
function (err) { console.log('Error: ' + err); },

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.

* + 1. Merge

The [Merge operator](http://reactivex.io/documentation/operators/merge.html) "*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() {

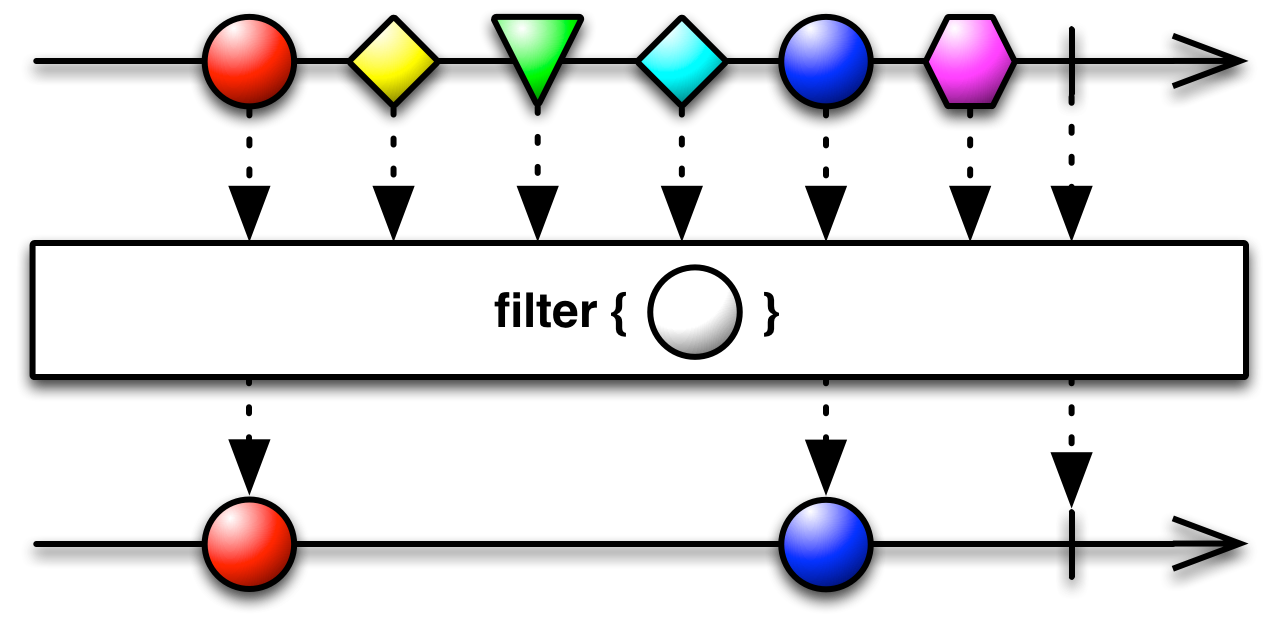
System.out.println("Sequence complete.");

}

});

* + 1. Filter

A [Filter](http://reactivex.io/documentation/operators/filter.html) "*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 : recv?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() {

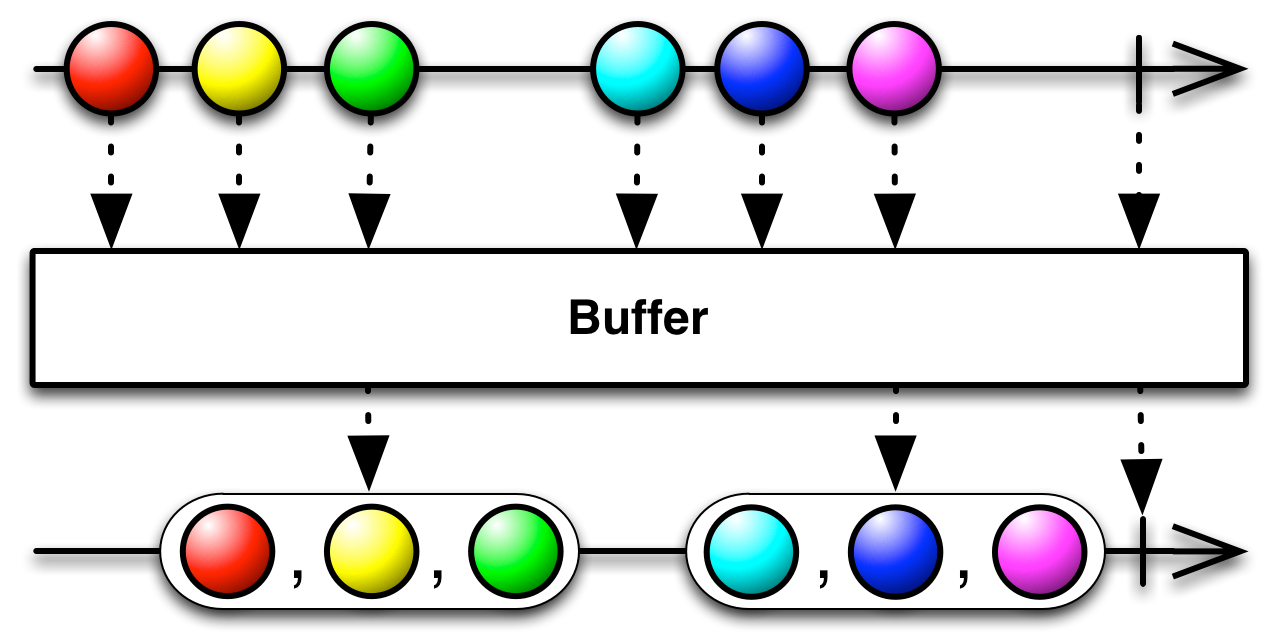
System.out.println("Sequence complete.");

}

});

* + 1. Aggregator and Windows

A [Window (or Buffer)](http://reactivex.io/documentation/operators/buffer.html) "*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

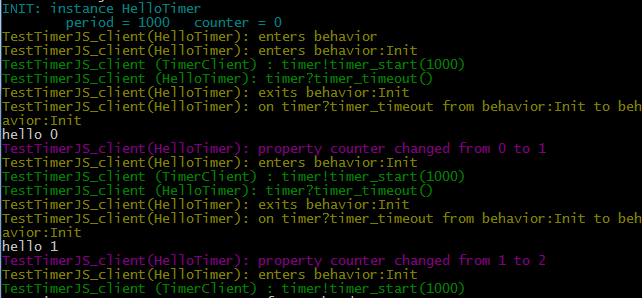
Those aggregators are typically defined in a reusable libraries that any stream can then use directly.

* 1. 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.



1. 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 togather:

configuration TestTimerJava {

instance timer : TimerJava

instance client : TimerClientJava

connector client.timer => timer.timer

}

* + 1. 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

* + 1. 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

* + 1. 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.

1. Model HD-Service deployment with Kevoree

Please, follow the [Kevoree Book](http://doc.kevoree.org/)

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

**todo** Inria to complete this part

1. 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.

1. Java environment
   1. 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/resources/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.

<plugin>

<groupId>org.kevoree.tools</groupId>

<artifactId>org.kevoree.tools.mavenplugin</artifactId>

<version>${kevoree.version}</version>

<executions>

<execution>

<goals>

<goal>generate</goal>

</goals>

</execution>

</executions>

<configuration>

<nodename>MyNode</nodename>

<model>src/main/kevs/main.kevs</model>

</configuration>

</plugin>

* + 1. 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 environement.

1. 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) {

grunt.initConfig({

// grunt-kevoree-genmodel

kevoree\_genmodel: {

main: {

options: {

quiet: false,

verbose: true

}

}

},

// grunt-kevoree

kevoree: {

options: {

name: 'node0'

},

run: {

kevscript: 'kevs/main.kevs'

}

}

});

};

You can get more details on their own repos [grunt-kevoree](https://github.com/kevoree/kevoree-js/blob/master/tools/grunt-kevoree/README.md) and [grunt-kevoree-genmodel](https://github.com/kevoree/kevoree-js/blob/master/tools/grunt-kevoree-genmodel/README.md)

1. C++

todo

1. For Service Operators
2. 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

1. 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

1. 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](http://oss.sonatype.org/service/local/artifact/maven/redirect?r=public&g=org.kevoree.tools&a=org.kevoree.tools.ui.editor&v=RELEASE)  
or [Web Editor](http://editor.kevoree.org/)) and some knowledge of the Kevoree Script ([KevScript](http://kevoree.github.io/kevoree-book/kevoree_script_aka_kevscript/README.html)) language.

* 1. Using the Java Editor

**todo**

* 1. 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)

* + 1. 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.



* + 1. Reconfigure your model
       1. 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**)

* + - 1. Using the graphical editor

**todo**

* + 1. 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.