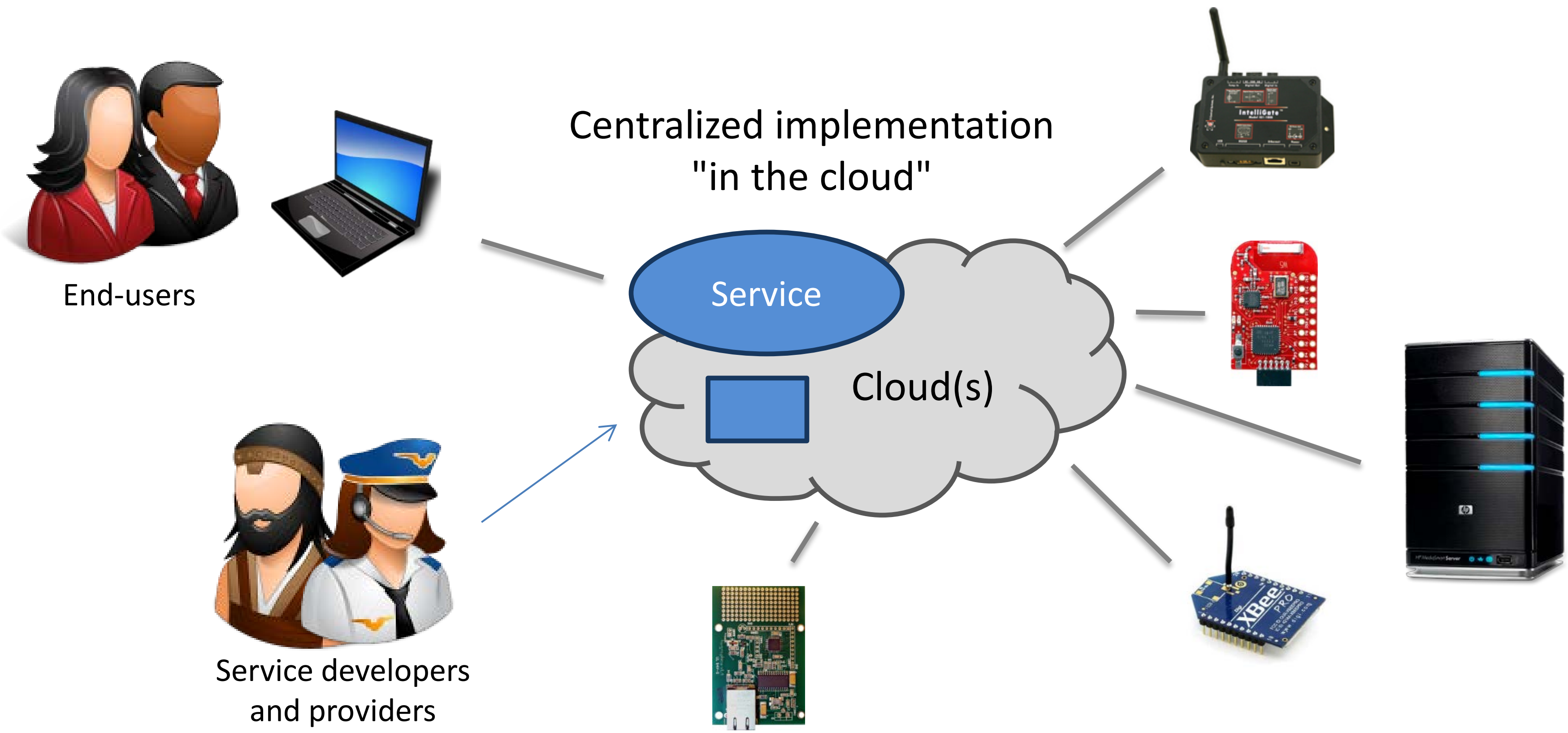
Taming Heterogeneity and Distribution in sCPS

Franck Fleurey, Brice Morin, Olivier Barais

franck.fleurey@sintef.no

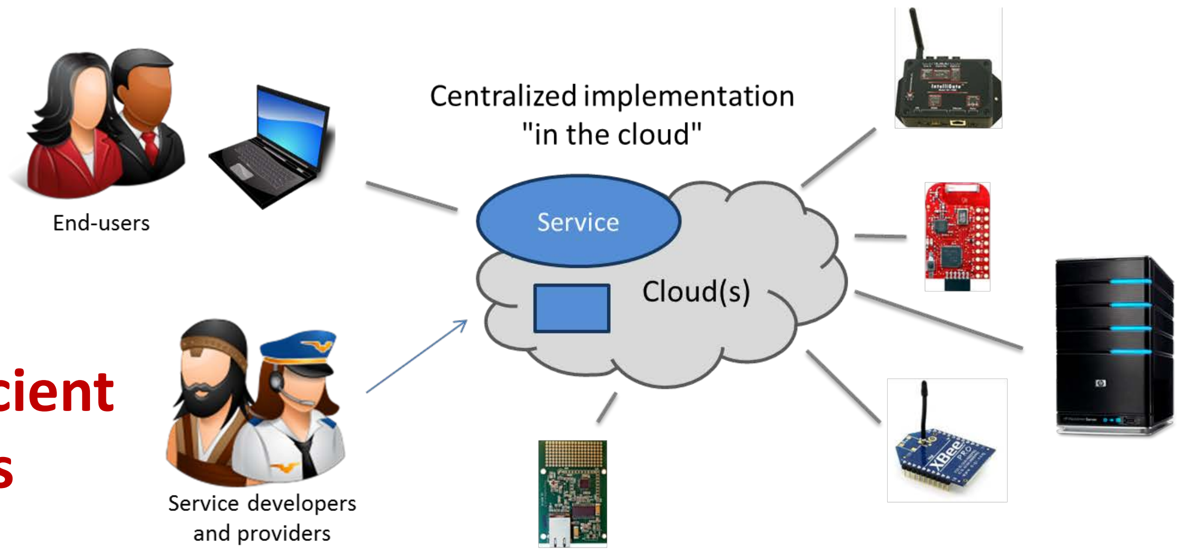
What heterogeneity and distribution?

- Isn't Internet of Things about having everything connected and available in the cloud?



Limitations of centralized approaches

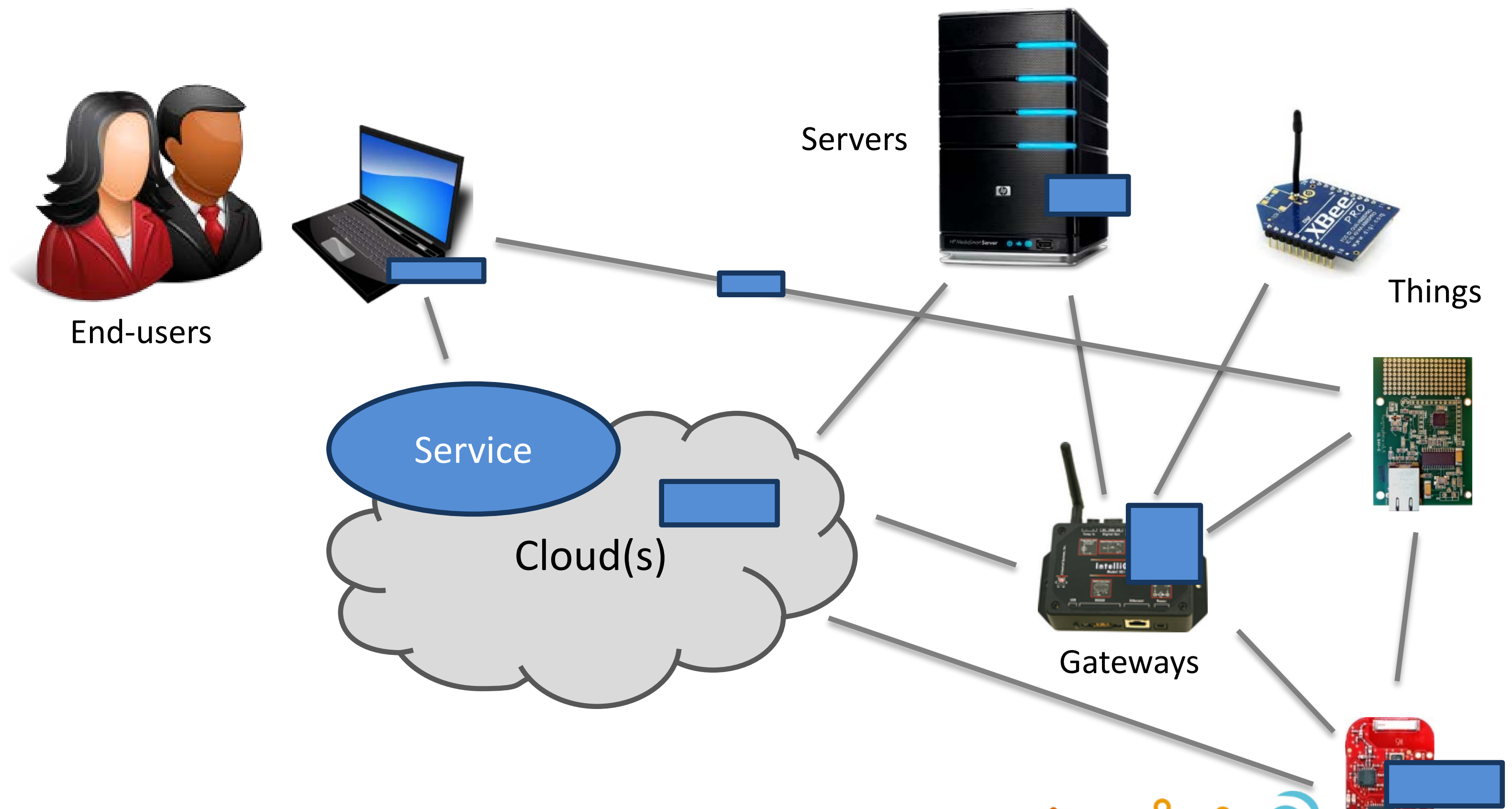
- Very easy to develop, evolve and maintain but…
  - Underexploits "Things" capabilities
  - Does not allow real-time or critical services
  - Not resource efficient (bandwidth)
  - Not robust
  - Does not scale



Good solution when possible but not sufficient in many realistic cases

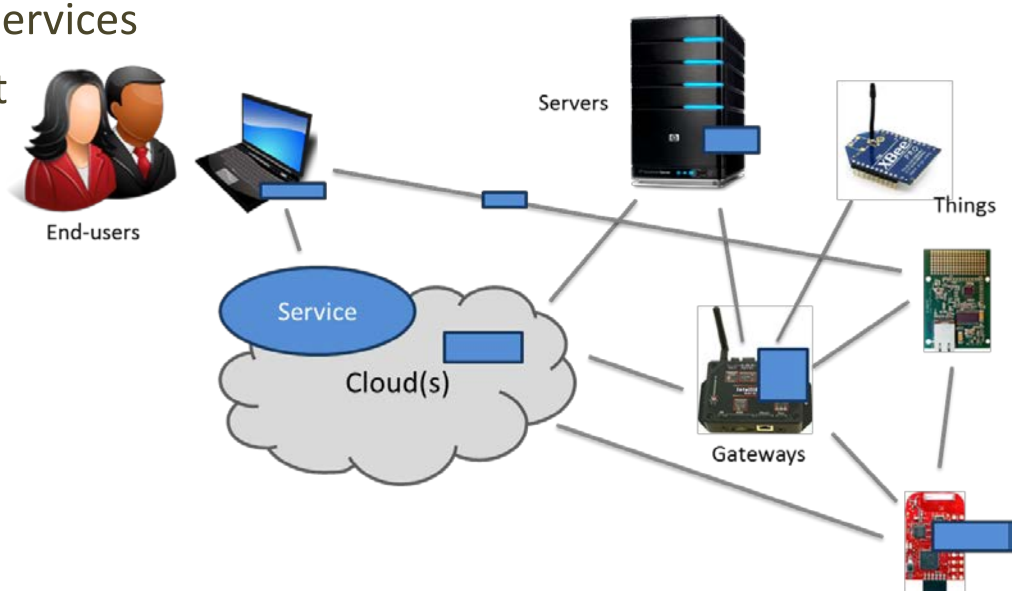
Distributing the implementation

- The service implementation is distributed to exploit the infrastructure



Benefits of HD-Services

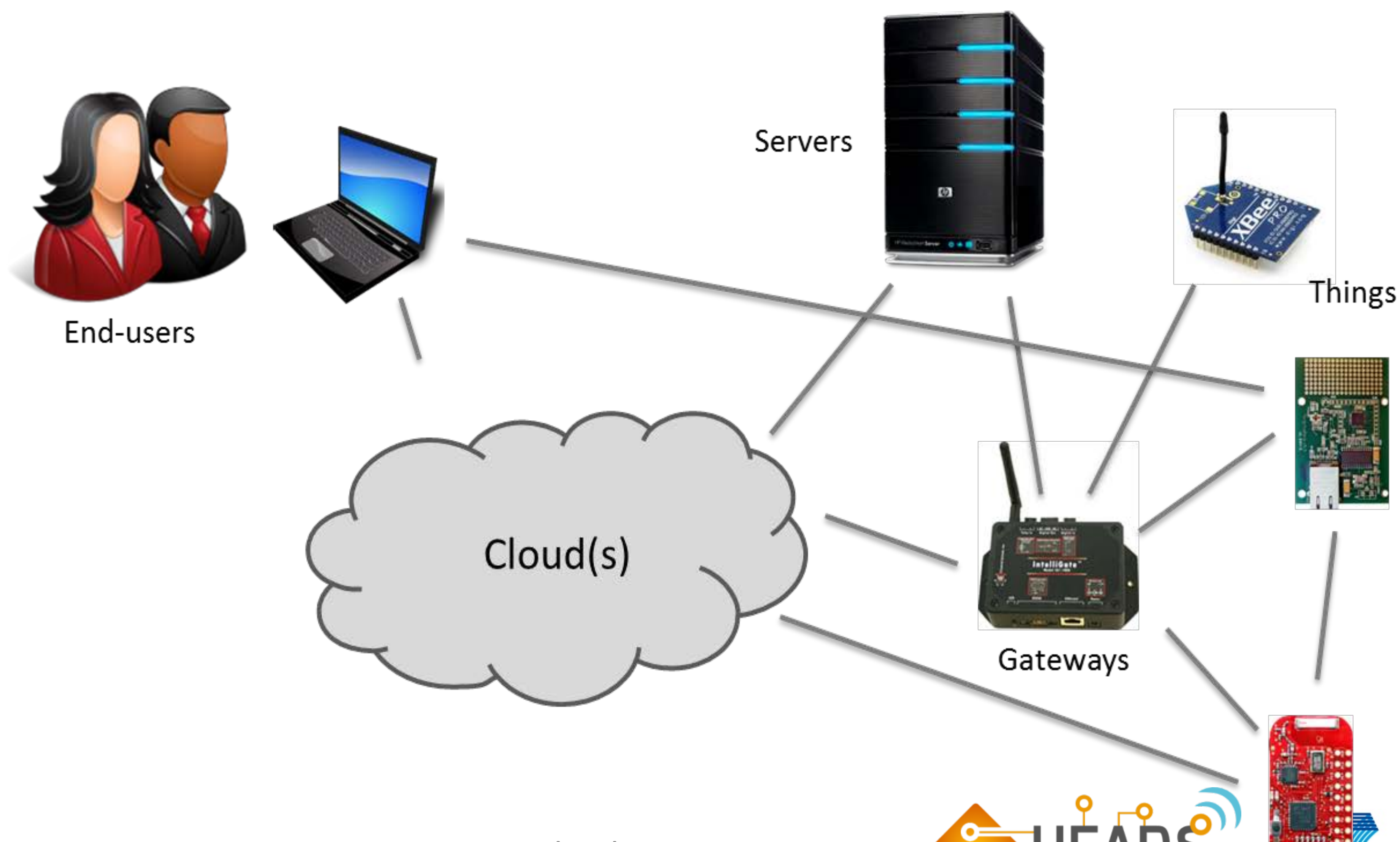
- Complex to develop, lots of different skills involved but…
  - Allows fully exploiting the features of each platforms
  - Allow for local and/or decentralized decision making
  - Robust to partial and/or temporary failures
  - Push processing close to data sources
  - Allow for real-time and critical services
  - Can scale in a "big data" context



In practice for more and more real-world services are HD-Services

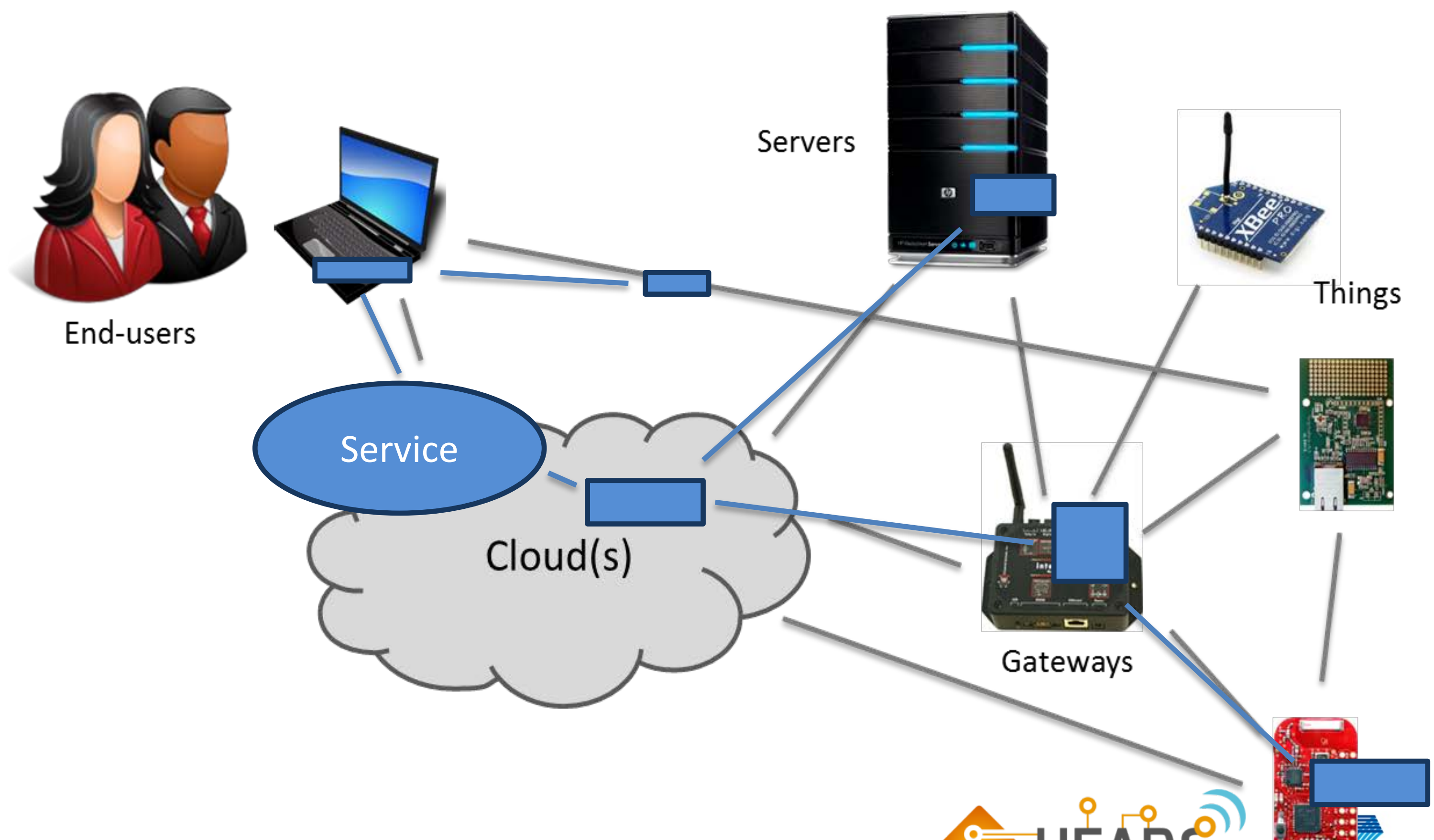
What are the problems? (1/6)

- Here is an example infrastructure



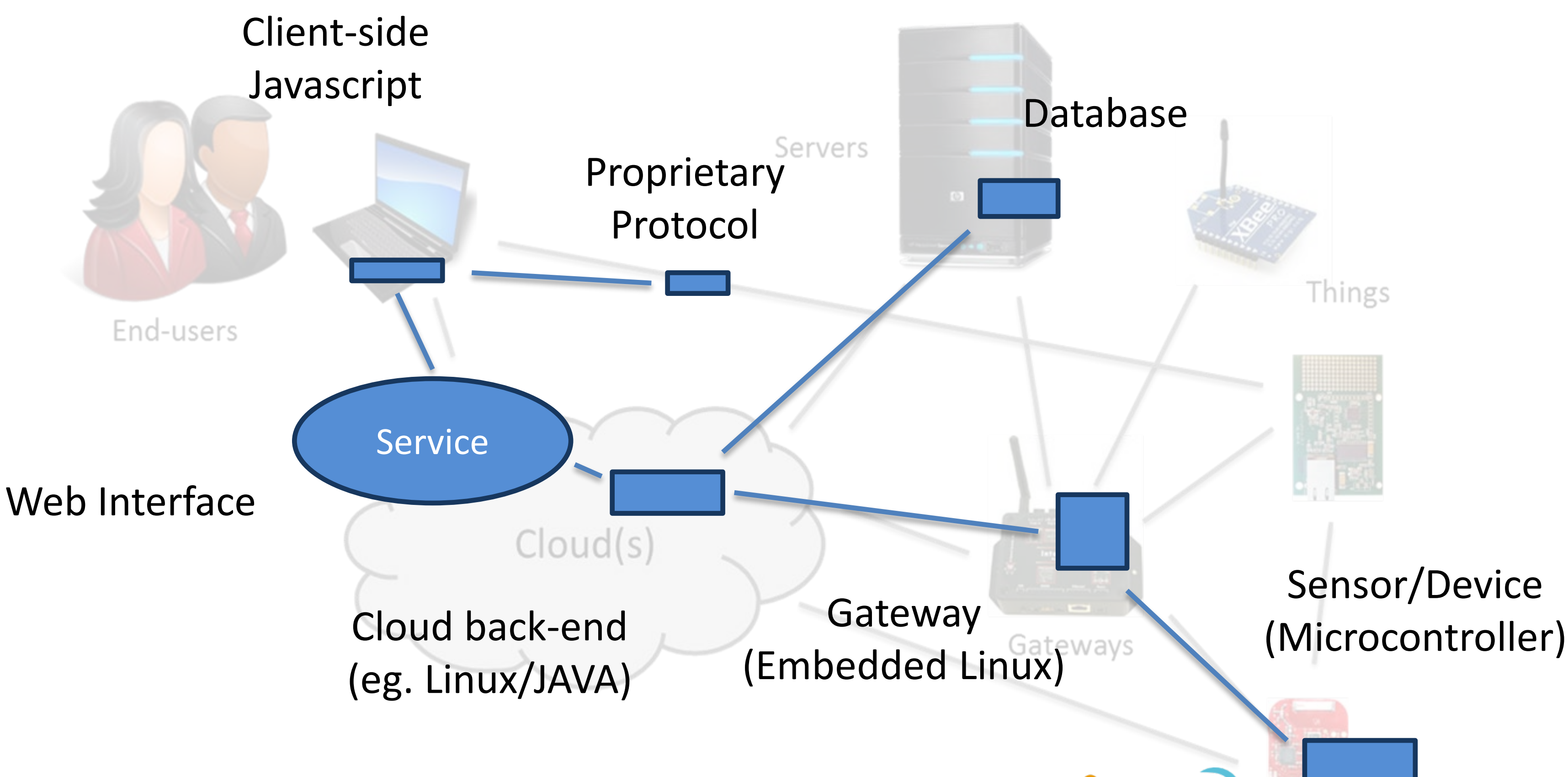
What are the problems? (2/6)

- Here is the software components needed for the service



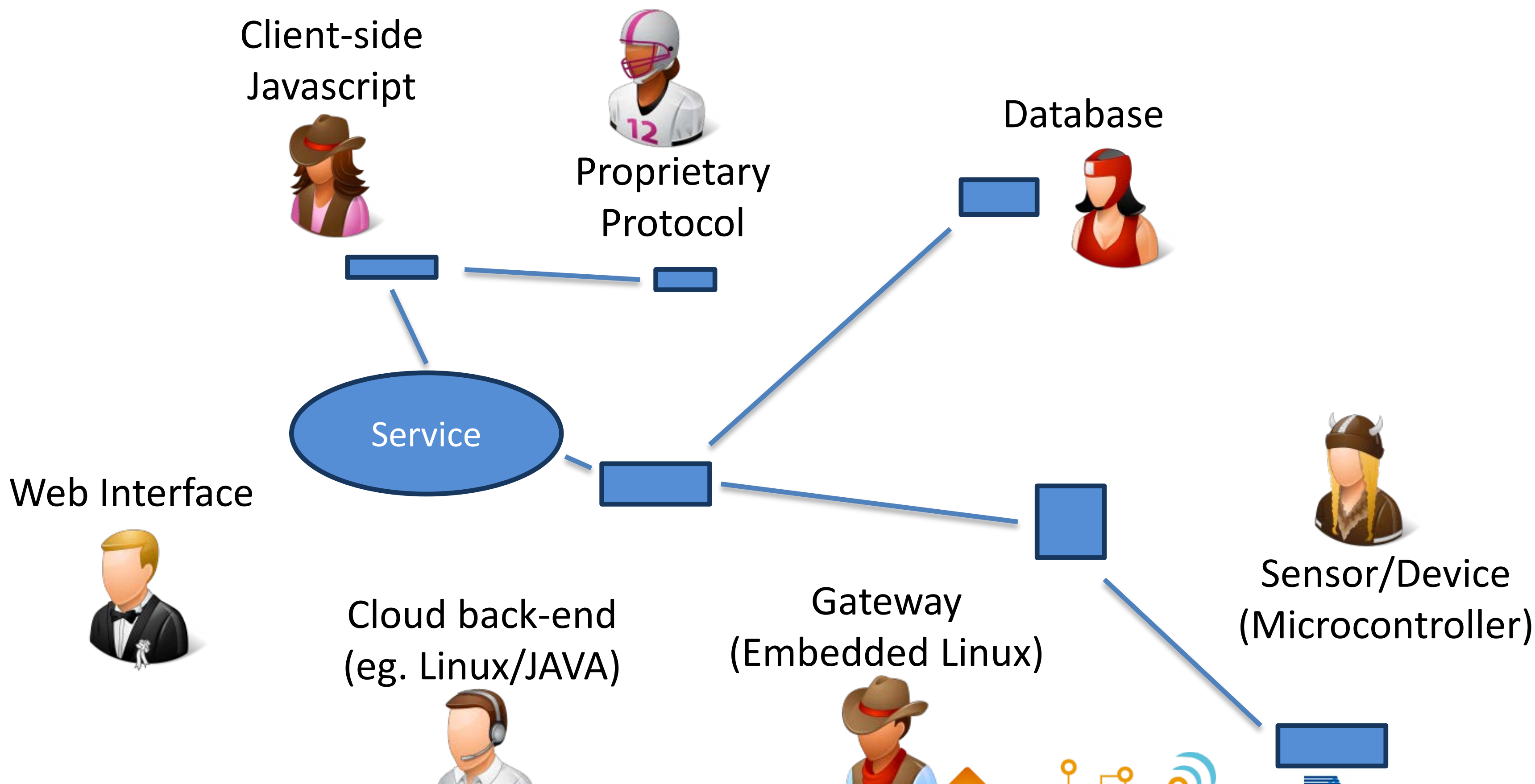
What are the problems? (3/6)

- Heterogeneous infrastructure and technologies are needed



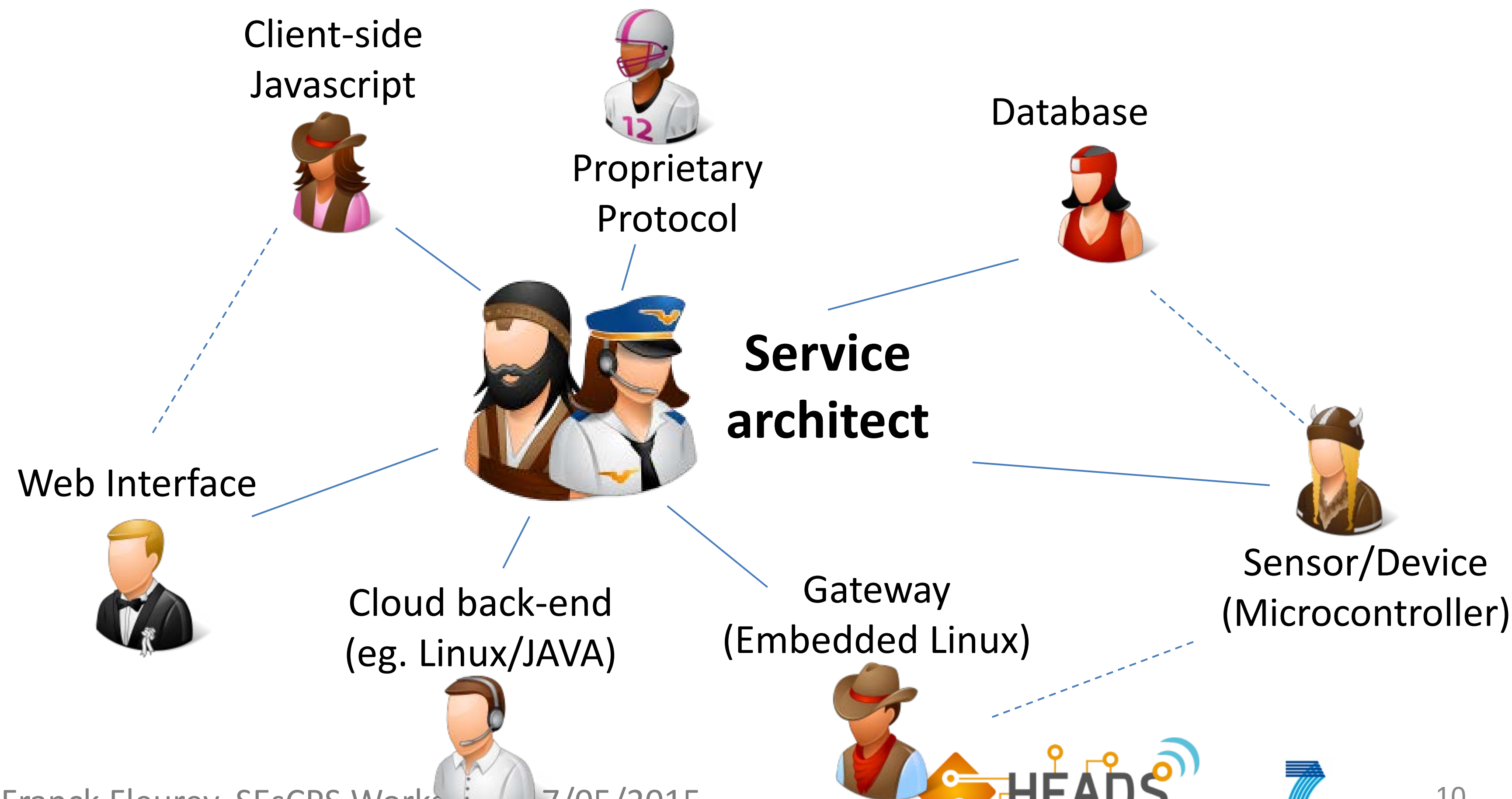
What are the problems? (4/6)

- A lot of different expertise are needed
  - Both for development and runtime deployment/maintenance



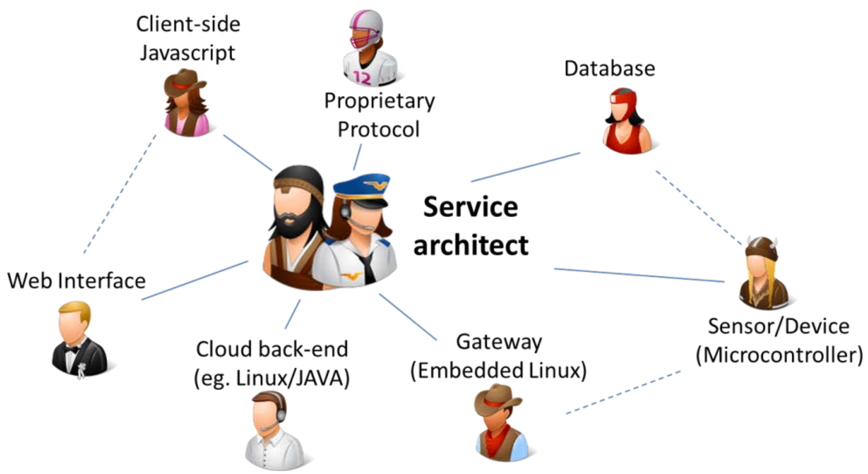
What are the problems? (5/6)

- Someone needs to coordinate all experts
  - Design the different components, their functionality and interractions



What are the problems? (6/6)

- Large heterogeneous teams need to collaborate
  - A service architect / developer
  - Many "platform experts"
  - Complex and expensive
  - Unavailable to small actors
- Service maintenance and evolutions
- Infrastructure is dynamic
  - Constant evolution/adaptation
- (Early) Validation?
- Software reuse?

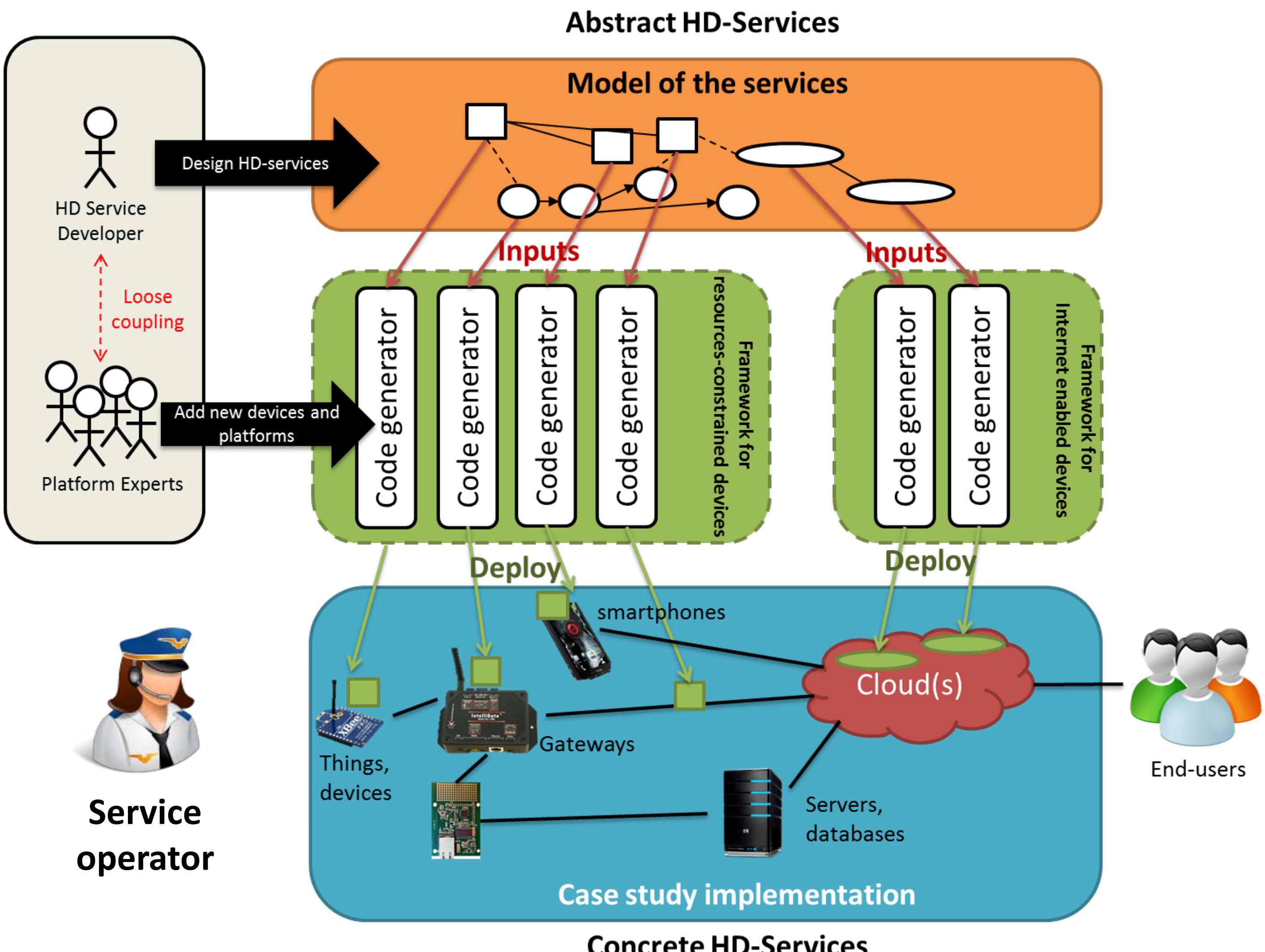


Challenging and expensive

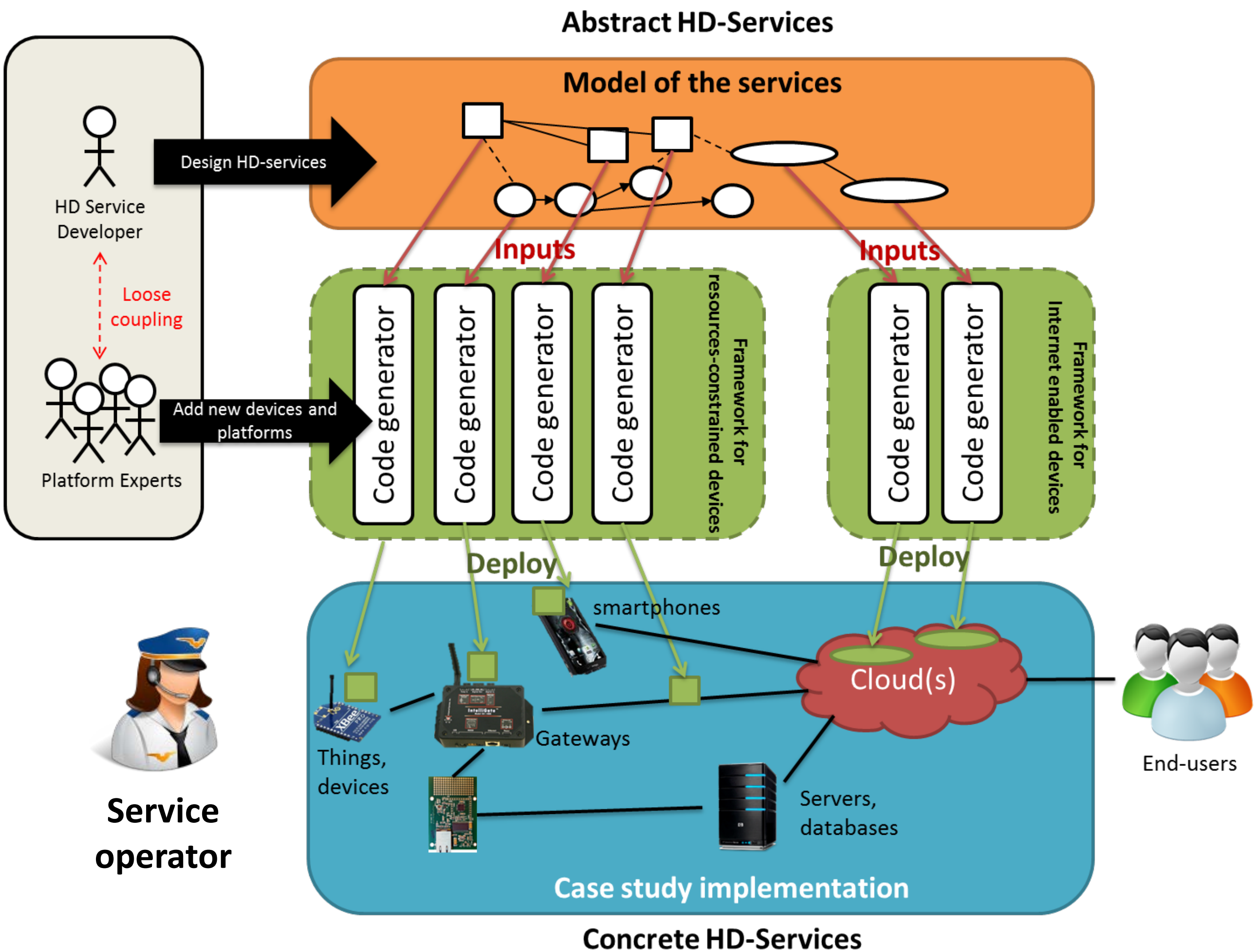
State of the practice

- **State of the art / practice**
  - Solution 1: Centralized service which uses devices "as-is"
    - Most common practice. Simple but restrictive.
  - Solution 2: Avoid problems by carefully selecting platforms
    - For which software frameworks pre-exist (eg. Arduino libs / shields)
  - Solution 3: Hide behind an homogeneous software layer
    - OS + generic or specific middleware platforms (eg. JAVA/JVM)
  - Solution 4: Custom develop manually all pieces of software
    - Can exploit full potential but very expensive (eg. automotive)
  - Solution 5: Fully fledged Model-Driven "PIM/PSM" approach
    - Good separation of concerns but impractical and too exclusive
- Non of the above allow exploiting the full continuum of platforms to its full potential (and at a reasonable cost)

HEADS Approach



HEADS Approach



(1)

Domain Specific Modelling Language

(2)

Code Generation Framework

(3)

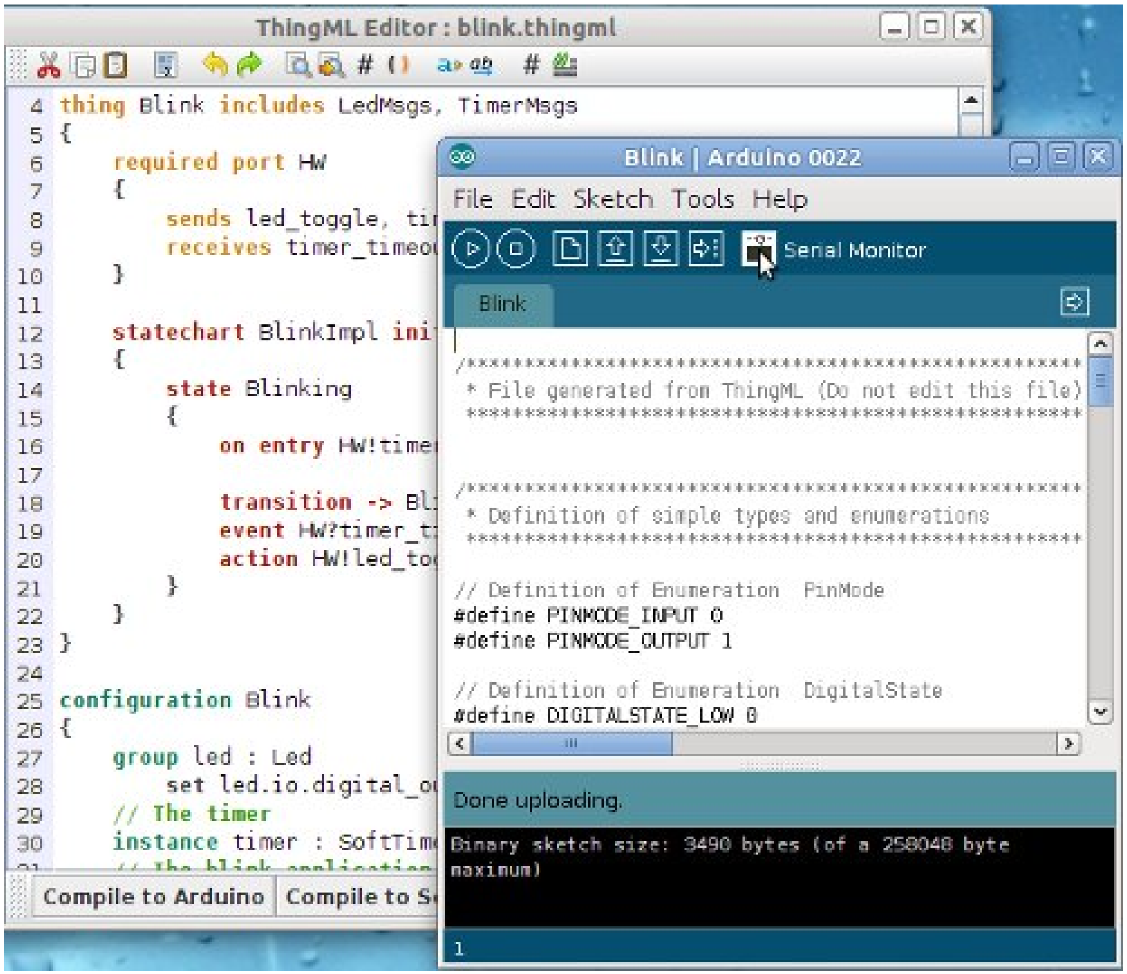
Deployment and Runtime

1. Domain Specific Modelling Language

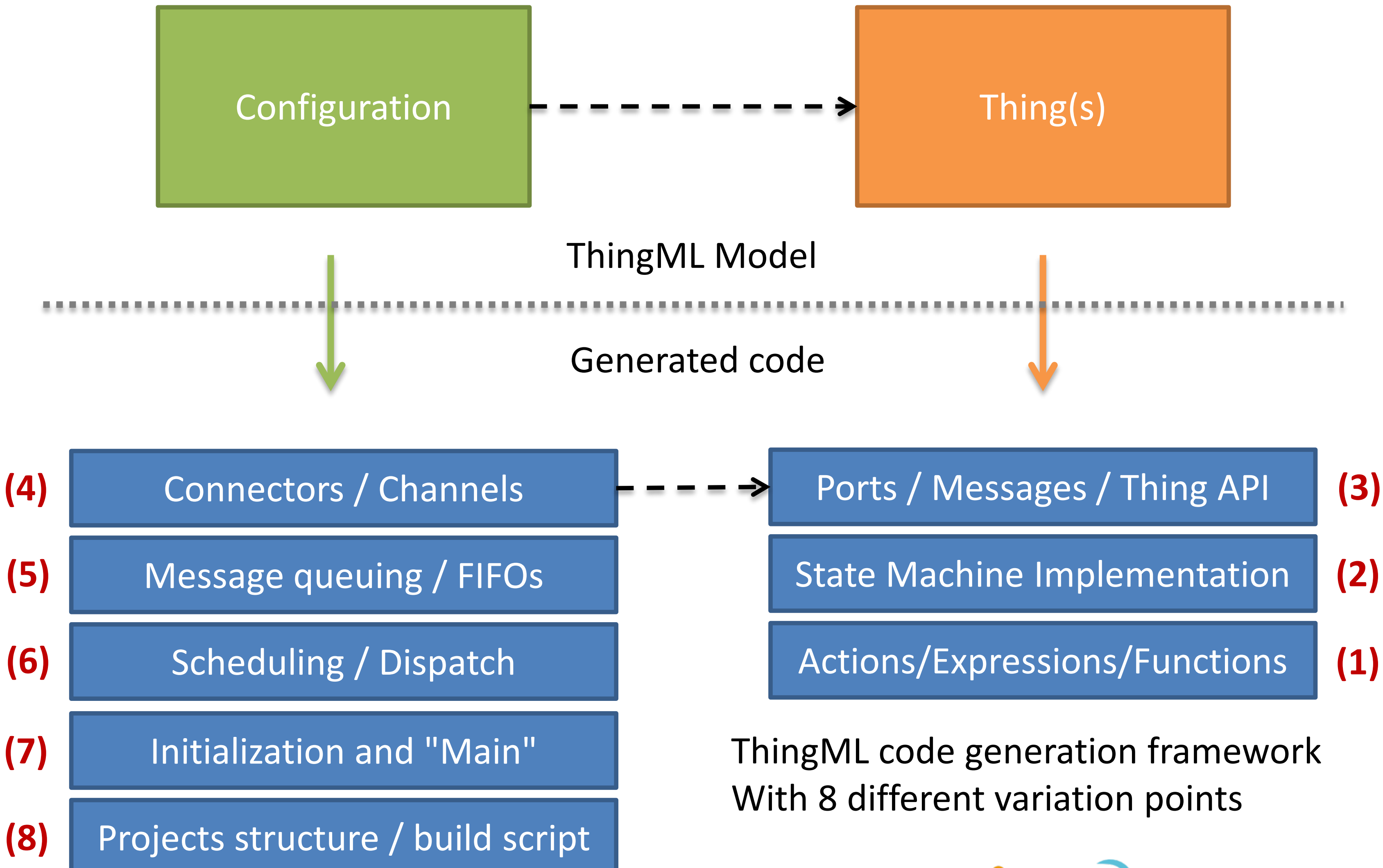
(ThingML)

- Based on the state of the art
  - Architecture (Components, Connectors, etc.)
  - Asynchronous messages / events
  - Composite state machines
  - Action Language
  - Deployment model
- Used as a commons model for integration
  - Not to replace individual modelling tools, programming frameworks or legacy components
  - Complete enough to fully implement the logic of the integration
  - All the way to deployment (and runtime management)

1. Domain Specific Modelling Language



2. Code Generation Framework



ThingML code generation framework
With 8 different variation points

3. Deployment and Runtime

(Kevoree)

- On the level of the architecture model
  - Nodes, Components, Connectors, Channels
- "Models@runtime" + Causal connection
  - Deployment
  - Monitoring
  - Adaptation
- Support heterogeneous components
  - Not "yet another middleware"
  - Easy to extend for supporting new execution platforms
  - Easy support for managing legacy/proprietary components

Conclusion

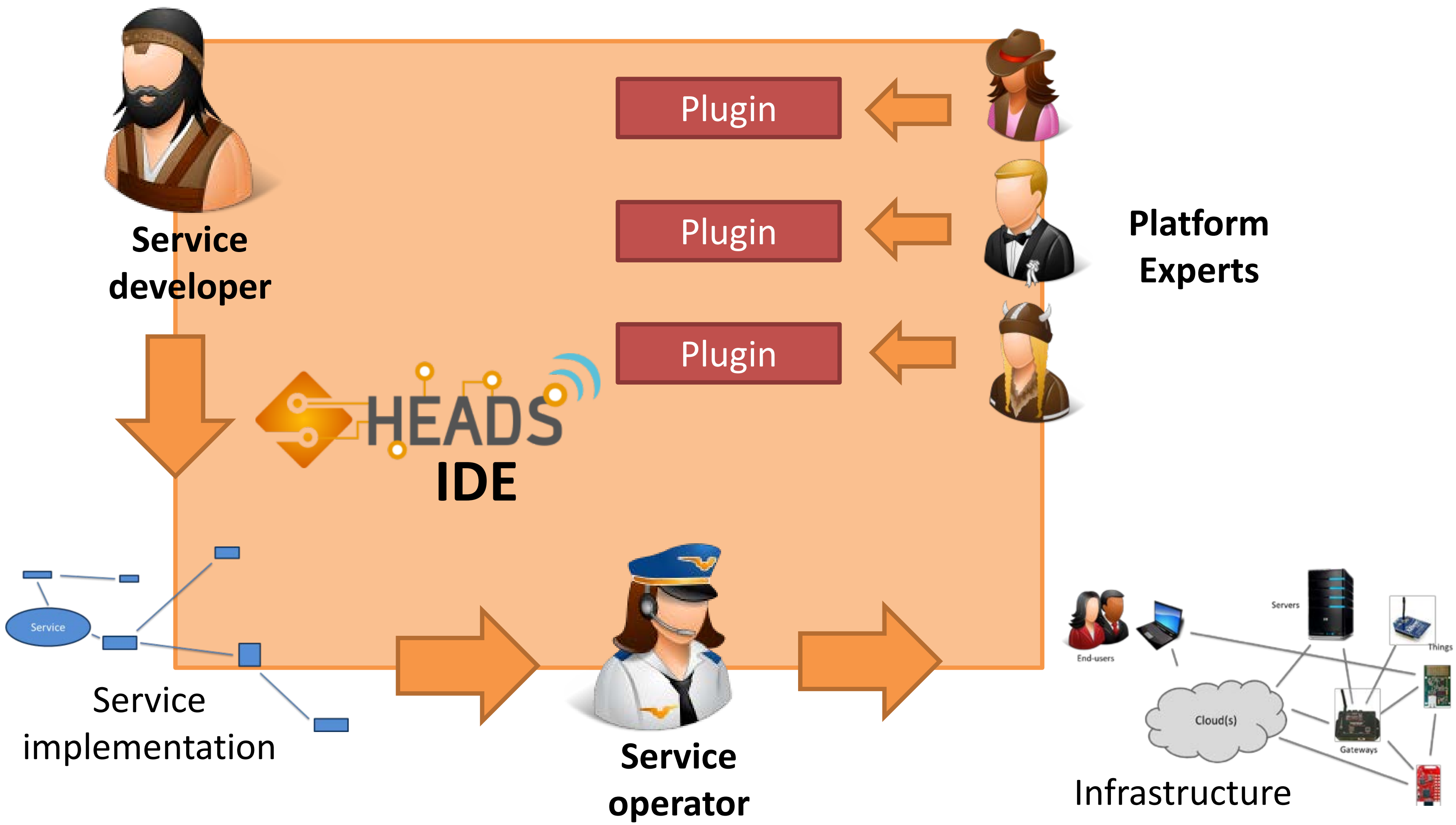
- Experiences using (part of) the HEADS approach
  - Medical Rehabilitation Robotic System
  - Unmanned vehicles (aerial and subsea)
  - Smart home and ambient assisted living
  - Media system
- Status of the implementation
  - Initial version is available, tutorial are available
  - Fully open-source
- Ongoing work and challenges
  - Modelling of complex-event processing
  - Modelling of different communication semantics
  - Code generation for resource constrained devices
  - Verification and Validation (Analysis, early testing, stub generation, etc.)
  - Evaluation of the code generation framework
- More info and apraoch implementation
  - ThingML: http://www.thingml.org
  - Kevoree: http://www.kevoree.org
  - HEADS: http://www.heads-project.eu

Thanks for your attention!

- Questions?

- More questions: franck.fleurey@sintef.no

HEADS Aproach



What is ThingML ?

- A DSL to model distributed reactive systems
  - IoT systems, embedded systems, sensor networks, ...
- Components, State machines and action language
  - « Main stream » MDE
- Contribution of ThingML
  - « Complete » action language
  - Slots, Mixins and Aspects instead of Inheritance and Composites
  - Enforced encapsulation and actors semantics
- Target Platforms and Applications
  - MDE for resource constrained systems (microcontrollers, IoT)
  - Development of applications distributed across heterogeneous hardware
  - Other types of reactive systems?

Why ThingML ?

- Typical MDE benefits
  - Reduce development, maintenance and evolution costs
  - Perform verifications and analysis on the models
  - Model application at a platform independent level

- No existing approach can deal with microcontrollers
  - ThingML can run on hardware less than 1ko of RAM

- No existing approach is really platform independent
  - Since actions are written in the target language

ThingML Goals

- Provide tools and methods
  - For each actor to concentrate on his task
  - For decoupling the tasks of different actors
  - Using state of the art software engineering practices
    - Modularity, reusability, runtime deployment, continuous integration, validation, etc…
  - Cost efficient and practically usable
    - No large overhead, integrated with legacy systems, etc…

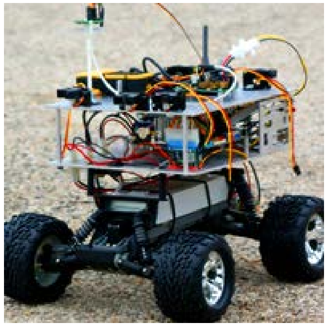
The ThingML tools

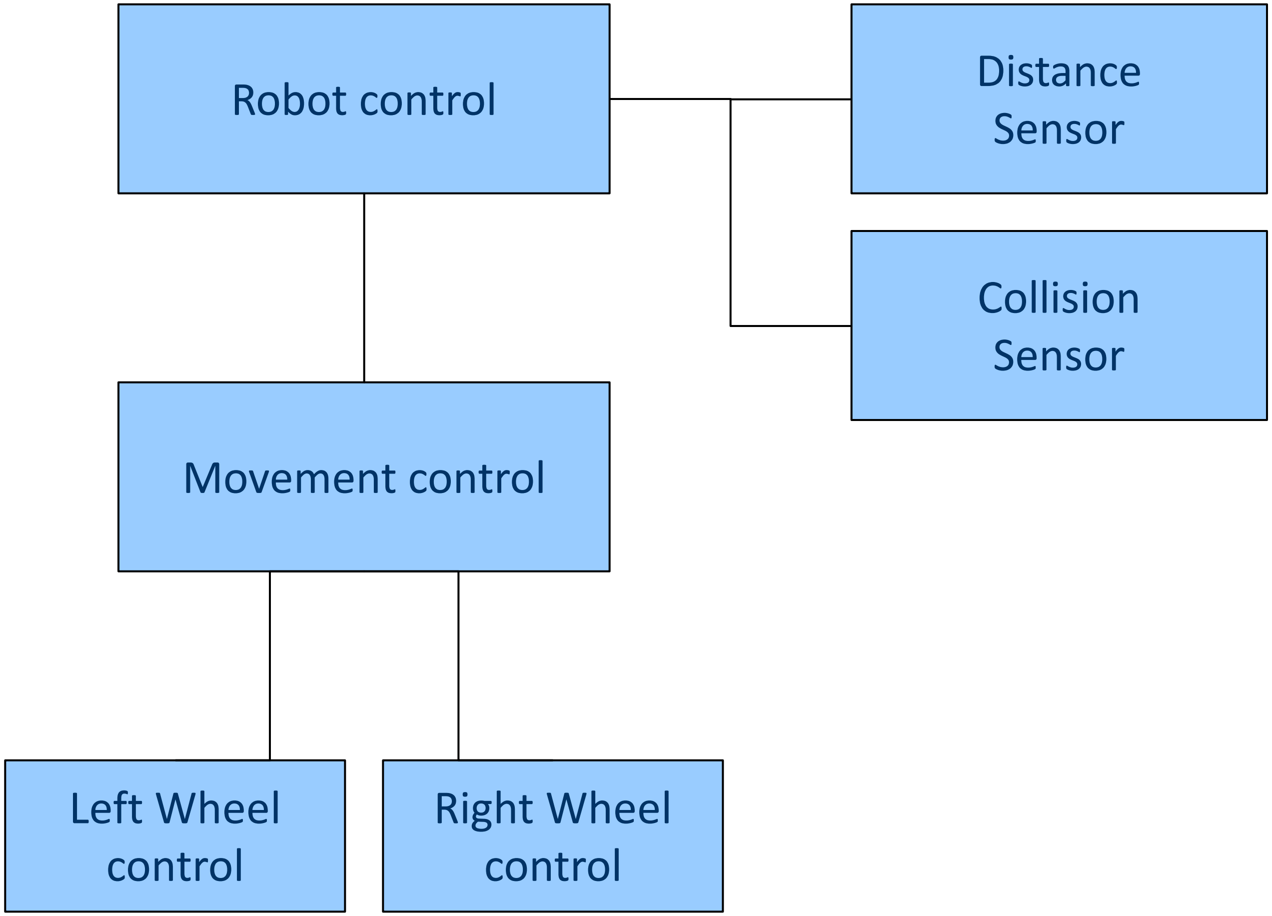
- Based on Eclipse / EMF Metamodel
- Textual Syntax with EMFText
  - For good usability and productivity
  - To keep the development cost of the editor(s) reasonable
- Graphical exports (graphML, graphviz, …)
- Static well formedness and type checker
- **Equivalent** compilers for a set of platforms
  - C/C++ for different microcontrollers, linux, embedded linux
  - Java for computers, smartphones, …
  - Javascript (NodeJS)
  - Maybe others if needed
- Generators for communication channels
- Easy to distribute ThingML IDE
  - Standalone and lightweight IDE
  - Eclipse plugins

Devolopping the ThingML tools

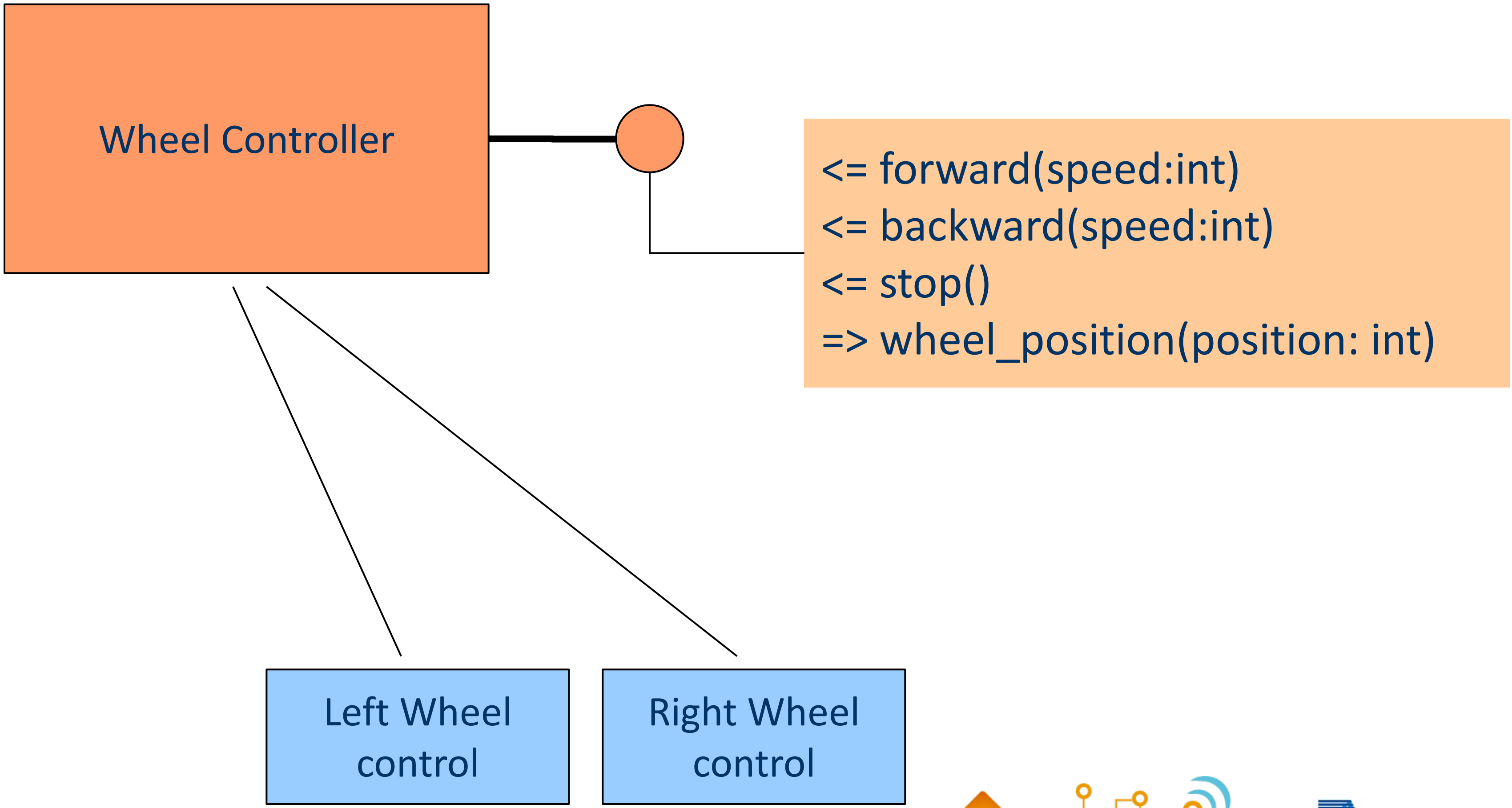
- Technologies
  - Eclipse / EMF and EMFText for metamodels and editors
  - Scala for constraints, transformation and code generation
  - Swing lightweight standalone editor
- Continuous integration process (using our thingml.org cloud server)
  - Maintain a code repository : Github open-source forge based on git
  - Automate the build : Maven build tool + Jenkins server
  - Make the build self-testing : Maven + JUnit
  - Everyone commits to the baseline every day : Github
  - Every commit (to baseline) should be built : Github triggers Jenkins
  - Keep the build fast : About 2 minutes at this point
  - Test in a clone of the production environment : Maven
  - Make it easy to get the latest deliverables : Archiva, Jenkins web interface
  - Everyone can see the results of the latest build : Jenkins web interface
  - Automate deployment : Java Web Start (JNLP)

ThingML: Architecture Model

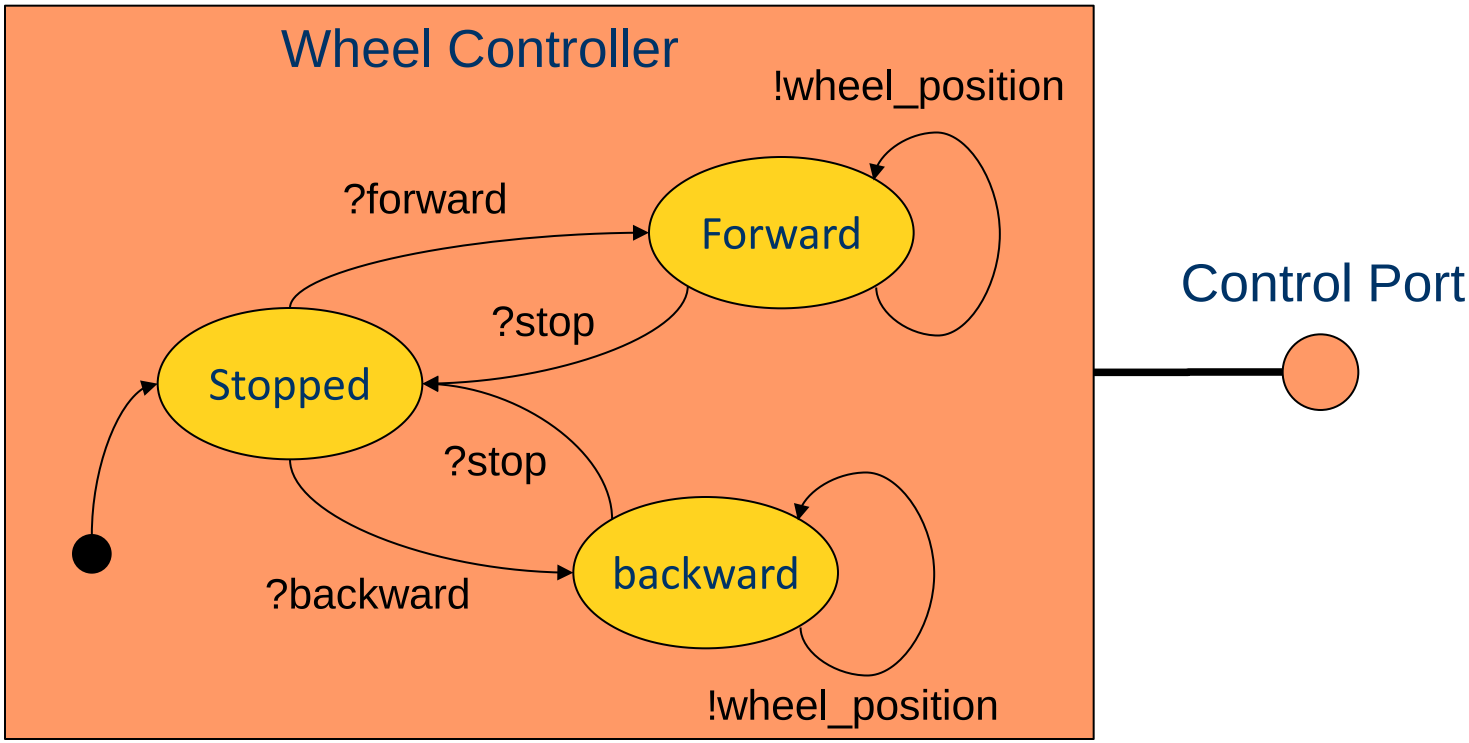




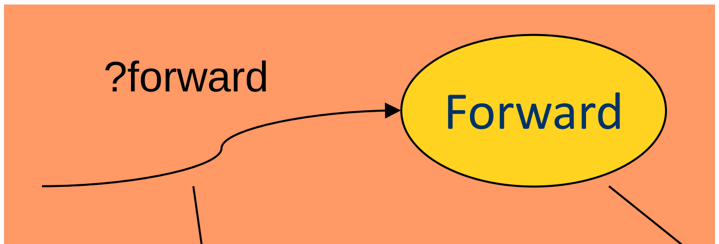
ThingML: Component



ThingML: State Machines



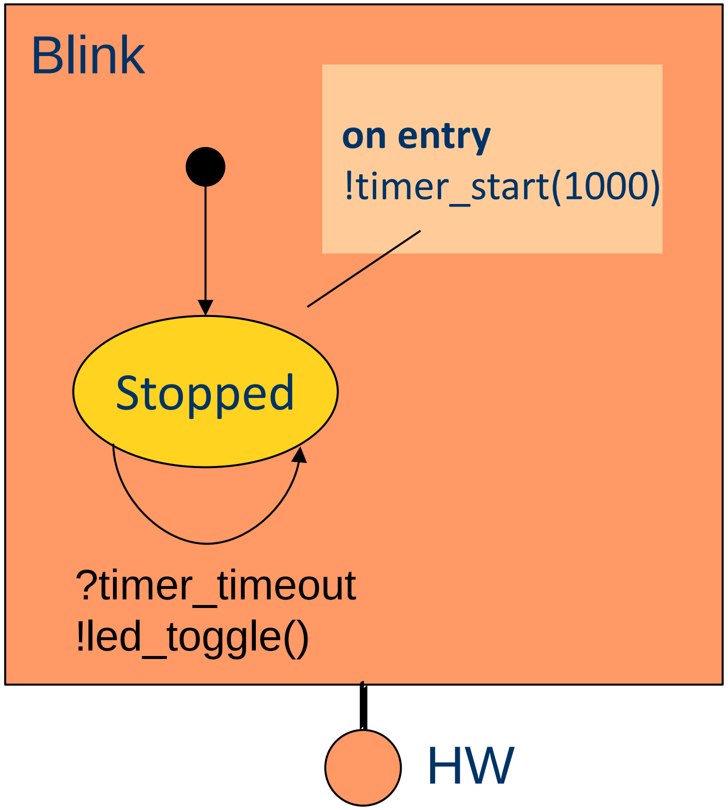
ThingML: Action Language




```
on entry do
  reset_wheel_position()
  motor_start()
end
...
```

```
action do
  motor_set_speed(speed)
  motor_set_direction(FW)
end
```

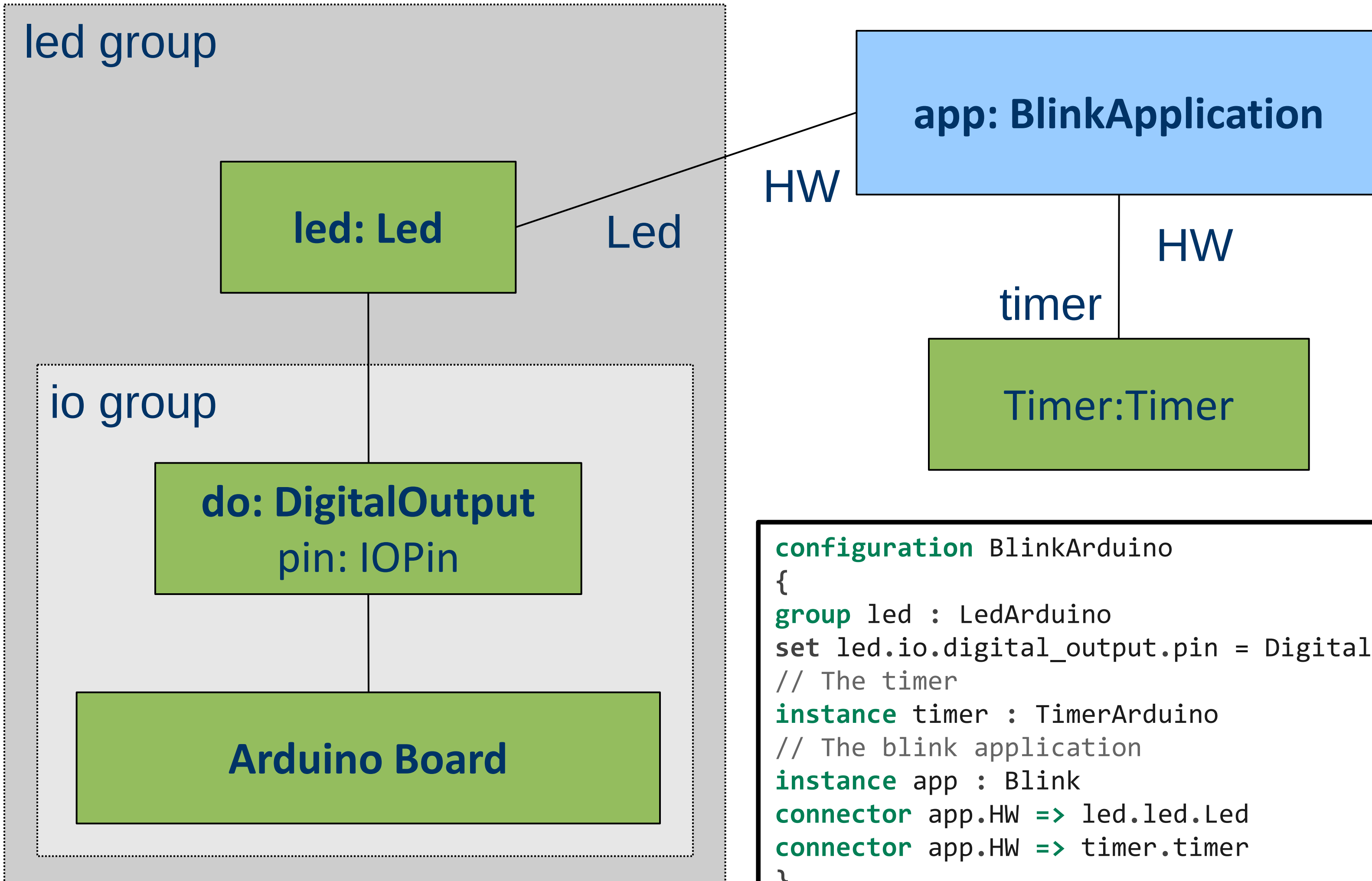
Blink example state machine



<= timer_timeout()
=> led_toggle()
=> timer_start(delay:int)

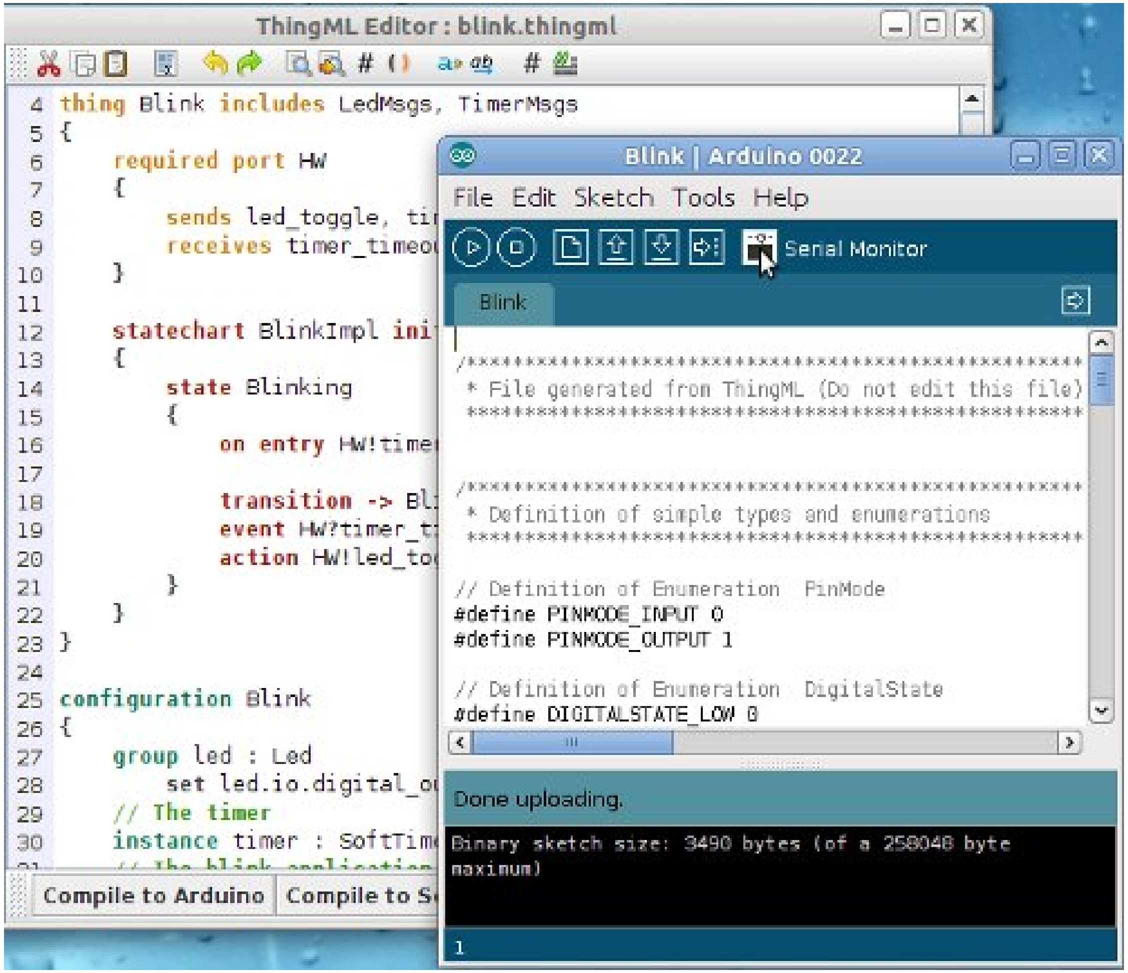
```
thing Blink includes LedMsgs, TimerMsgs
{
    required port HW
    {
        sends led_toggle, timer_start
        receives timer_timeout
    }

    statechart BlinkImpl init Blinking
    {
        state Blinking
        {
            on entry HW!timer_start (1000)

            transition -> Blinking
            event HW?timer_timeout
            action HW!led_toggle ()
        }
    }
}
```

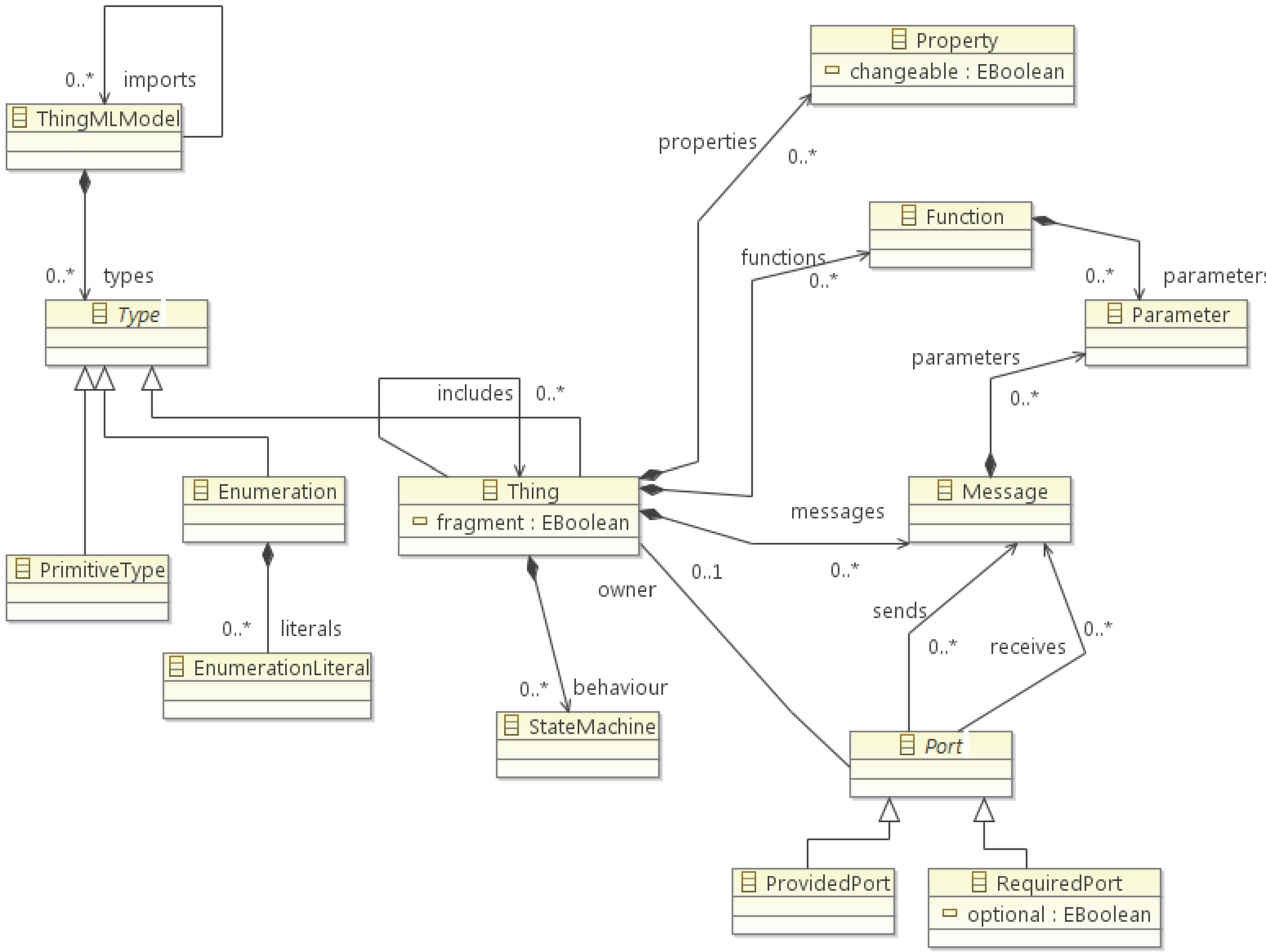
Blink example and instance groups

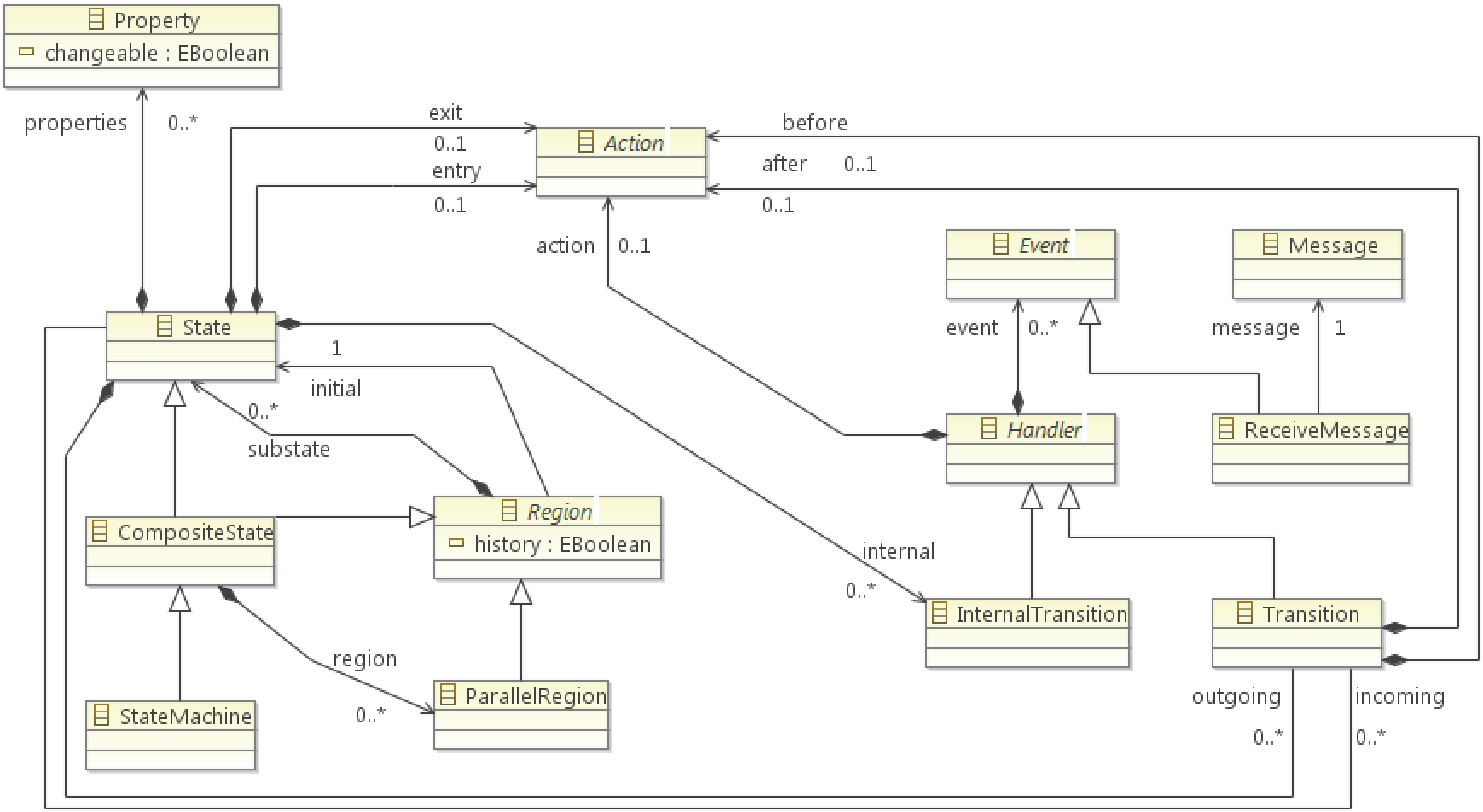



```
configuration BlinkArduino
{
group led : LedArduino
set led.io.digital_output.pin = DigitalPin:PIN_13
// The timer
instance timer : TimerArduino
// The blink application
instance app : Blink
connector app.HW => led.led.Led
connector app.HW => timer.timer
}
```

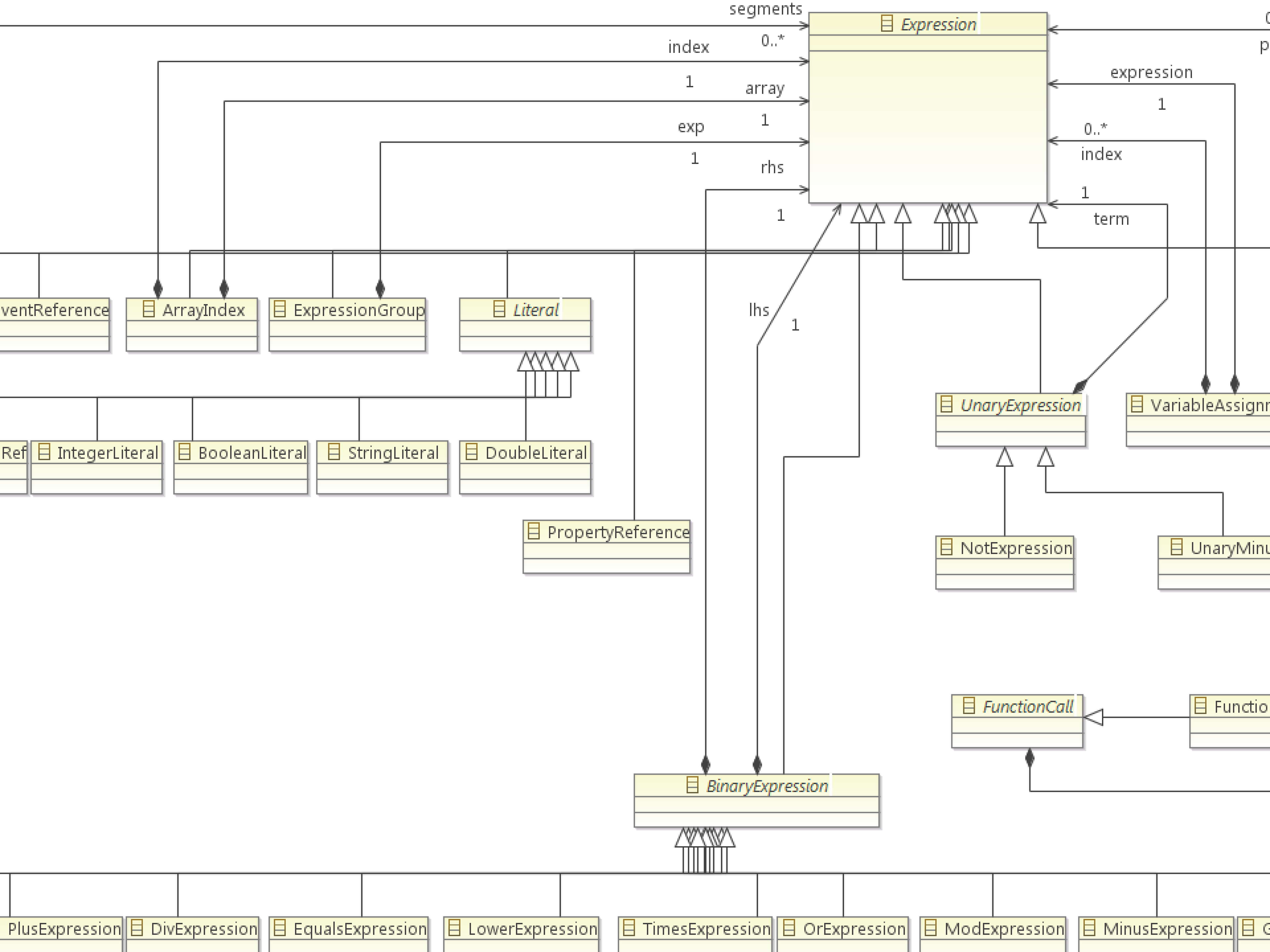
ThingML Editor

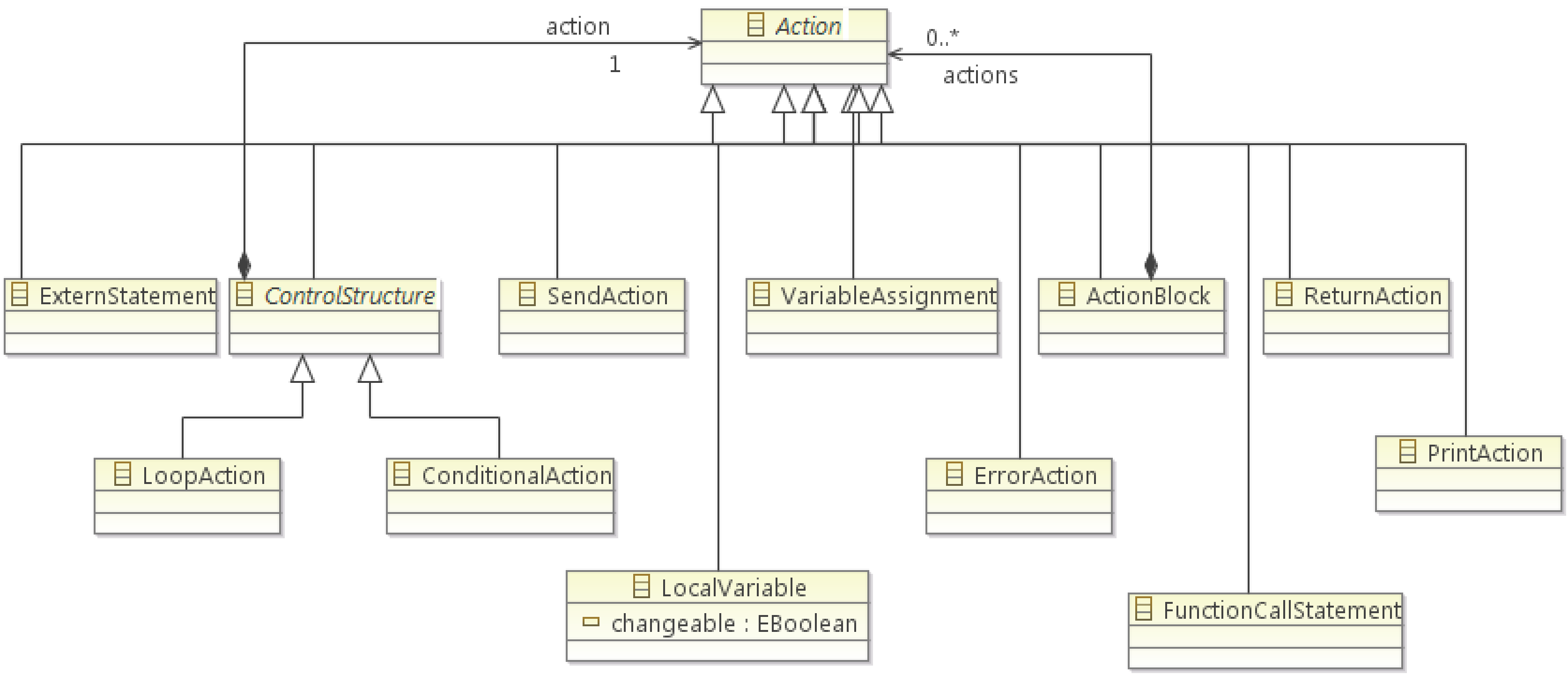


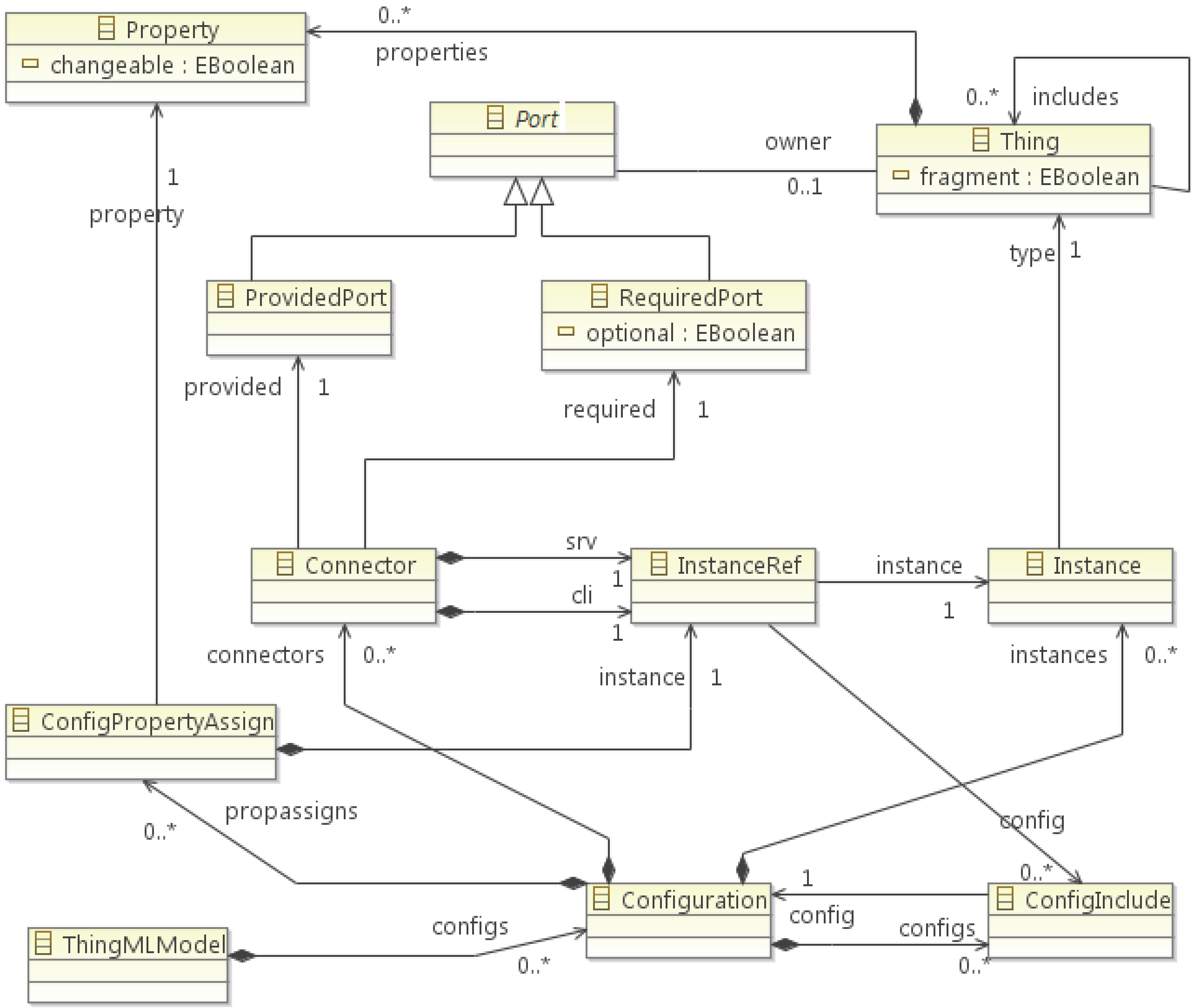

ThingMLModel
0..*
imports
0..*
types
Type
PrimitiveType
Enumeration
0..*
literals
EnumerationLiteral
includes
0..*
Thing
fragment : EBoolean
0..*
behaviour
StateMachine
properties
0..*
Property
changeable : EBoolean
functions
0..*
Function
0..*
parameters
Parameter
parameters
0..*
messages
0..*
Message
owner
0..1
sends
0..*
receives
0..*
Port
ProvidedPort
RequiredPort
optional : EBoolean



Property
changeable : EBoolean
properties
0..*
exit
0..1
entry
0..1
Action
before
after
0..1
0..1
action
0..1
Event
Message
State
1
initial
0..*
substate
event
0..*
message
1
Handler
ReceiveMessage
Region
history : EBoolean
CompositeState
internal
0..*
InternalTransition
Transition
region
0..*
ParallelRegion
StateMachine
outgoing
0..*
incoming
0..*

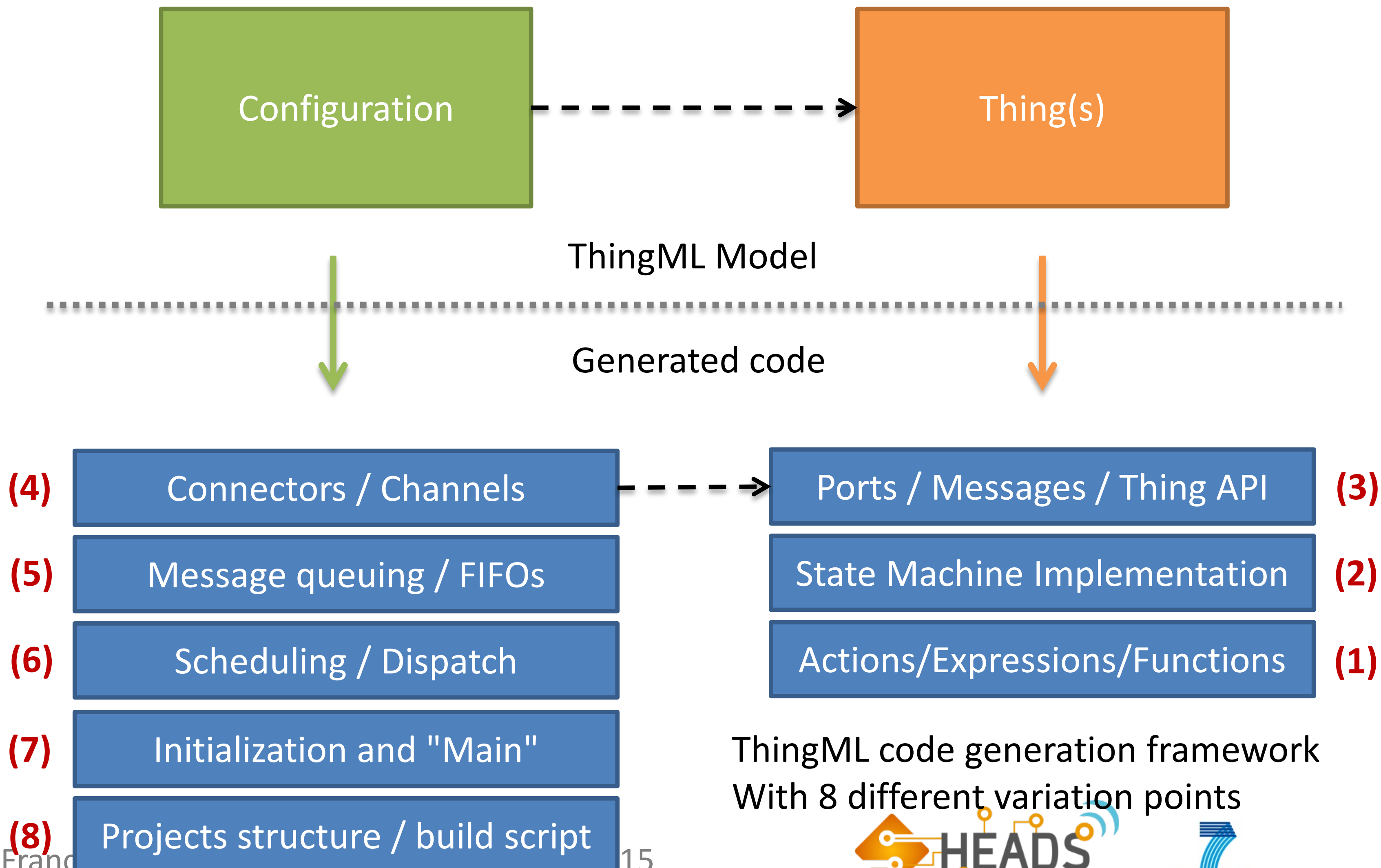

segments
Expression
index
0..*
1
array
1
exp
1
rhs
1
expression
1
0..*
index
1
term
lhs
1
ventReference
ArrayIndex
ExpressionGroup
Literal
UnaryExpression
VariableAssign
Ref
IntegerLiteral
BooleanLiteral
StringLiteral
DoubleLiteral
PropertyReference
NotExpression
UnaryMinu
FunctionCall
Functio
BinaryExpression
PlusExpression
DivExpression
EqualsExpression
LowerExpression
TimesExpression
OrExpression
ModExpression
MinusExpression


Action
action
1
0..*
actions
ExternStatement
ControlStructure
SendAction
VariableAssignment
ActionBlock
ReturnAction
LoopAction
ConditionalAction
ErrorAction
PrintAction
LocalVariable
changeable : EBoolean
FunctionCallStatement



http://heads-project.eu

ThingML code generation framework



ThingML code generation framework
With 8 different variation points

http://heads-project.eu

(1) Actions / Expressions / Functions

- Scope
  - Depends only on the target language
  - Can be reused for different platforms
- Implementation
  - Visitor on the ThingML meta-model
- Customizable by
  - Implementing a new visitor for a new language
  - Inheriting from an exiting visitor and overriding some of its methods

(2) State machine implementation

- Scope
  - Specific to a specific state machine implementation strategy.
  - Can generate either the complete state machine in the target language or leverage a state machine framework on the target platform
- Implementation
  - Abstract state machine code generator
  - A set of reusable helpers to calculate states, transitions and events according to the common ThingML semantics.
- Customization
  - Implement the abstract state machine generator

(3) Ports / Messages / Thing APIs

- Scope
  - Depends on the language best practices
  - Depends on how components should be "packaged" on the target platform
    - Can generate any custom API for the Things
    - Can generate towards exiting middleware / OS
  - Can/should produce "manually usable" APIs
  - Different generators can be used for different things
- Implementation
  - Visitor on the "Thing" part of the metamodel
  - Helpers to collapse fragments and gathers all the elements of a thing (messages, ports, functions, etc).
- Customization
  - Implement a new visitor for a new target language / platform
  - Inherit from an existing visitor for light customization

(4) Connectors / channels

- Scope
  - Depends on how messages are transported from on thing to the next using the Things APIs
  - Can be local and/or remote, includes the serialization, transport through networks and deserialization
  - Different generators can be used for different ports
- Implementation
  - Abstract generator for serialization, deserialization and transport
- Customization
  - Implement new concrete generators
  - Easy to reuse serialization and just override transport

(5) Message Queuing / FIFOs

- Scope
  - Asynchronous behaviour of messages
  - Can target existing message frameworks or middleware or use custom made FIFOs
  - Different generators can be used for different ports
- Implementation
  - Abstract generator which can be customized
  - Helpers to calculate the sets of messages to be handled (combines fragments and prunes unused messages).
- Customization
  - Inherit and implement the abstract generator

(6) Scheduling / Dispatch

- Scope
  - Implements the main loop of the program, schedules the activation of the components and dispatches the incoming messages
  - Relies on underlying OS and libraries of the target platform.
  - Can generate a custom scheduler for microcontroller applications.
- Implementation
  - Template + Helper
- Customization
  - Create of modify an existing template

(7) Initialization and "main"

- Scope
  - Generate the entry point and initialize the components and connectors
  - Depends on the target languages and traget frameworks
- Implementation
  - Template + Helper providing the set of components and connectors to instantiate
- Customization
  - Create or modify a template

(9) Project structure / build script

- Scope
  - Produce the right file structure, additional project files and/or build scripts
  - Can be customize to fit a specific target environment (makefiles, maven files, etc)
- Implementation
  - Abstract generator with access to buffers containing all the generated code.
- Customization
  - Create a concrete generator. Possibility to use templates.

Consistency checking

- A suite of tests (27) written in ThingML
  - Takes characters as inputs (or nothing)
  - Generates characters as outputs
- A set of platform specific harness (also in ThingML)
  - For C/Linux, Java, Node.js
  - Write outputs into a file (or simply crash if severe bug)
- Discussion
  - Testing ThingML using ThingML: possible bugs that hide each others…
  - …less and less probable as the number of compilers augments

Current test results

- Java: 100%, C/Linux:96%, Node.js (started 10/14): 81%, now 100%

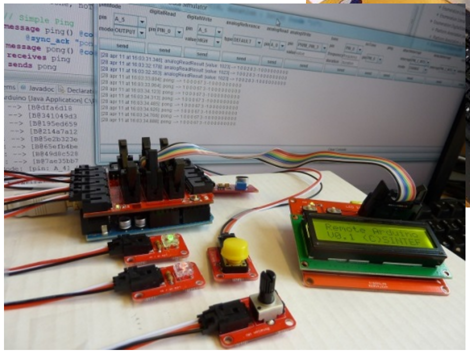
Test name	Compiler	Result
testArrays	Java	Succes
testFunction	Java	Succes
testCompEventCapture	Java	Succes
testInit	Java	Succes
testOnExit	Java	Succes
testCompositeStates	Java	Succes
testVariables	Java	Succes
testOnEntry	Java	Succes
testInternalTransition	Java	Succes
testArrays3	Java	Succes
testEnumeration	Java	Succes
testCompStatesExit	Java	Succes
testTransition	Java	Succes
testEmptyTransition	Java	Succes
testCompStatesEntry	Java	Succes
testMaskCompositeStates	Java	Succes
testHistoryStates	Java	Succes
testRegion	Java	Succes
testMaskProperty	Java	Succes
testHistory	Java	Succes
testSelfMessage	Java	Succes
testMultiClientPing	Java	Succes
testDeepCompositeStates	Java	Succes
testArrays2	Java	Succes
testHello	Java	Succes
testAutoTransition	Java	Succes
testNaming	Java	Succes

Test name	Compiler	
testArrays	Linux	
testFunction	Linux	
testCompEventCapture	Linux	
testInit	Linux	
testOnExit	Linux	
testCompositeStates	Linux	
testVariables	Linux	
testOnEntry	Linux	
testInternalTransition	Linux	
testArrays3	Linux	ErrorAtCompilation c
testEnumeration	Linux	
testCompStatesExit	Linux	
testTransition	Linux	
testEmptyTransition	Linux	
testCompStatesEntry	Linux	
testMaskCompositeStates	Linux	
testHistoryStates	Linux	
testRegion	Linux	
testMaskProperty	Linux	
testHistory	Linux	
testSelfMessage	Linux	
testMultiClientPing	Linux	
testDeepCompositeStates	Linux	
testArrays2	Linux	
testHello	Linux	
testAutoTransition	Linux	
testNaming	Linux	

Test name	Compiler	Result
testArrays	Javascript	Success
testFunction	Javascript	Success
testCompEventCapture	Javascript	12a5e12 does not match 12b3c4 for input 00 (00)
testInit	Javascript	Success
testOnExit	Javascript	Success
testCompositeStates	Javascript	Success
testVariables	Javascript	Success
testOnEntry	Javascript	Success
testInternalTransition	Javascript	Success
testArrays3	Javascript	Success
testEnumeration	Javascript	Success
testCompStatesExit	Javascript	Success
testTransition	Javascript	Success
testEmptyTransition	Javascript	Success
testCompStatesEntry	Javascript	Success
testMaskCompositeStates	Javascript	012320 does not match 012321 for input nnp (nnp
testHistoryStates	Javascript	Success
testRegion	Javascript	Success
testMaskProperty	Javascript	Success
testHistory	Javascript	Success
testSelfMessage	Javascript	IJJJJJ does not match IJJJ for input ttt (ttt)
testMultiClientPing	Javascript	Success
testDeepCompositeStates	Javascript	012abc65gf does not match (012abc|abc012)(63gd|gd63) f na (na)
testArrays2	Javascript	Success
testHello	Javascript	Success
testAutoTransition	Javascript	Success
testNaming	Javascript	ErrorAtCompilation does not match (AE|EA)BCD for inpu

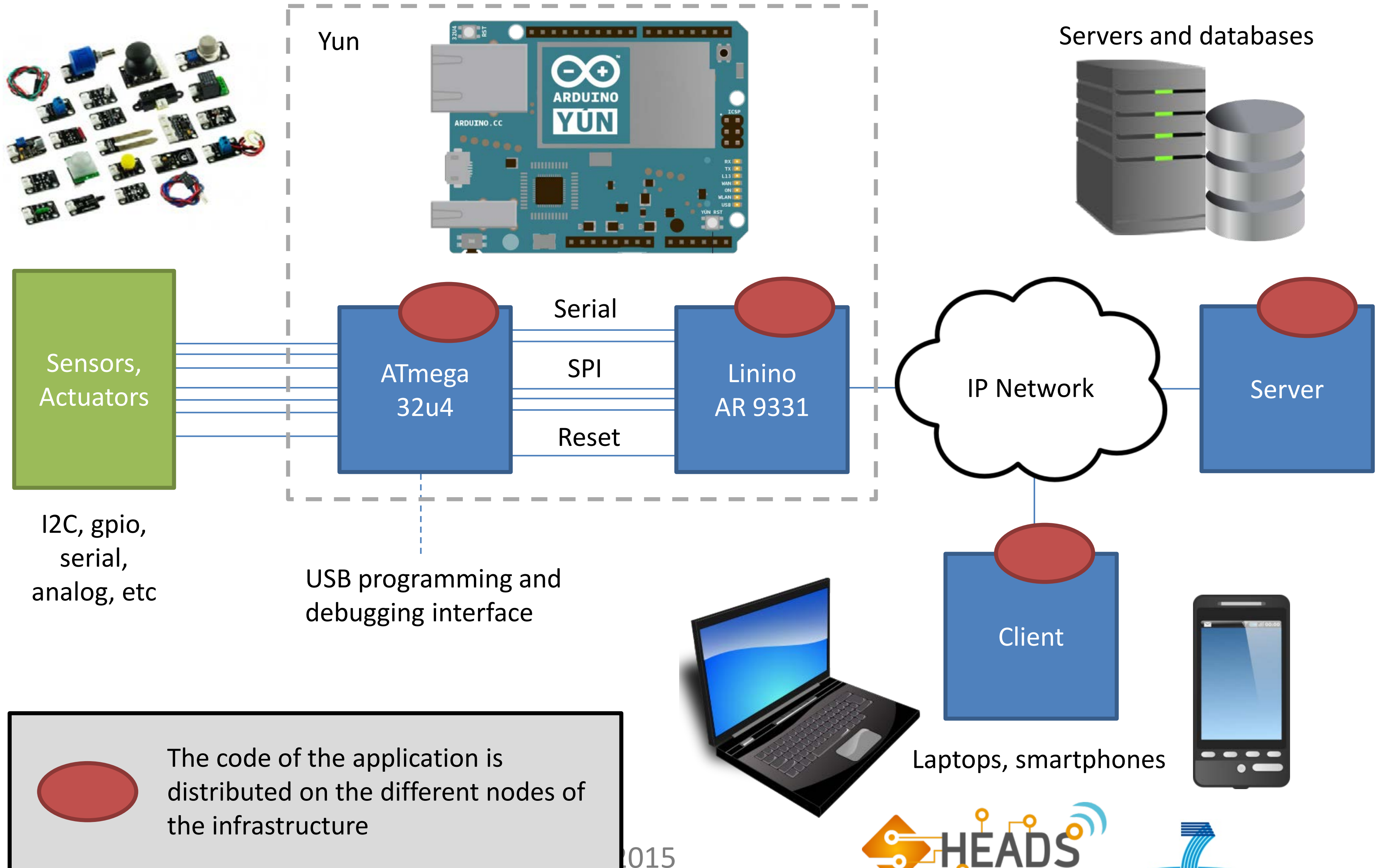
Experimental platforms and "lab"

- Cloud (Amazon, Flexiant, Rackspace, etc)
- Mini-Cloud (Openstack + Docker)
- Android (Java + Android)
- Cubietruck "cloud" (Linux + Docker)
- Raspberry Pi (Linux)
- Arduino Yun (dd-wrt linux + AVR µC)
- Arduino (AVR µC)
- TI ARM/MSP µC



- Home automation and wearable devices

Simple IoT Infrastructure Example



http://heads-project.eu