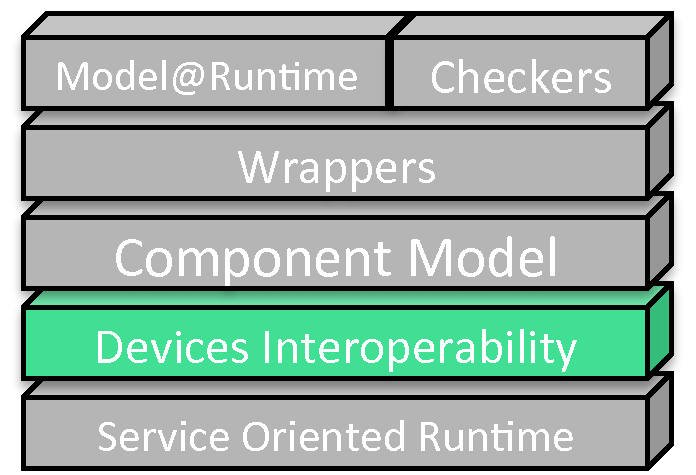
**Chapitre** **6**

# Details on strata

Like a geologist, this chapter dissects the contribution layer by layer. For each stratum composing the proposition, a section gives details on the roles taken on by the layer, its achievements, and its interactions with other strata.

## 6.1 Devices Interoperability



The Device Interoperability layer is probably the most important layer of the approach, since it answers the first requirement. Variability management, adaptations, or evolutions, would be compromised if only two devices are not able to communicate. Interoperability of devices is a central concern. It brings a foundation on which other layers can be built. This section presents how this interoperability has been realized.

In the domain of home automation, communication protocols used by manufacturers, and their devices, are not compatible. This incompatibility makes impossible any direct interaction of devices coming from different brands. To overcome this barrier, some manufacturers worked in a consortium to define a unique communication protocol, for their respective products to be compatible. However, some of their products are coding boolean values on a single bit, while others are coding it on a byte. Again, two products may not be operable one to each other.

In [**Erreur ! Source du renvoi introuvable.**] Bromberg et al. propose to automatically generate gateways between protocols, to address this issue. But building protocol-to-protocol translators solves the problem only partially, because the number of translators exponentially explodes with the number of protocols. Nevertheless, this proposition seems very interesting for an automatic generation of translators, from specific protocols to an abstraction model of higher level.

### 6.1.1 Use of drivers

To realise this abstraction, drivers has been developed. A driver makes the link between, the real world devices, and their virtual representation in a software system. Thus, they take on two responsibilities. In one way, they convert from vendor specific communication messages to actions on their virtual representative. Naturally, they translate and transmit orders sent to a virtual device, to the physical device on the other way.

Secondly, drivers provide the virtual structures for each product they are able to interact with. All implementations specific to a given manufacturer are thus contained in drivers, or separate libraries. This makes the core system completely independent from devices’ implementations specificities.

This independence implies the creation of a common structure, for the system to be able to properly handle devices in a good abstraction level.

### 6.1.2 Functional interfaces

This common structure may take the form of a set of programming interfaces. Each interface could specify a set of methods for a specific functionality. Then, drivers just have to provide objects, decorated with some interfaces, selected according to the abilities of each device. This set of programming interfaces was created. A survey of devices’ functions allowed us to extract the minimum set of common methods for each functions, as presented in figure  for instance. Once the set defined, a library containing all interfaces was compiled and included in the framework.

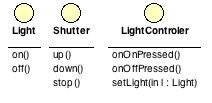


Fig. 6.1: Functional Interfaces

First experiments were promising. Interoperability was almost solved, but several drawbacks were rapidly identified while using this approach in real use cases. The set of interface is only extensible by augmentation of the framework. The development of a device driver could have been broken because the required function interface was not available in the library. Moreover, direct method calls are not appropriate if the system has to consider a dynamic environment, in which objects unpredictably appear and disappear. In this case, an object-oriented development using synchronous method calls becomes quite hazardous. Finally, if for any reason a component implementing the *LightControler* interface has to be plugged to a shutter, the operation is simply not feasible without an ad-hoc adapter (illustrated in figure ). Interoperability was not solved.

### 6.1.3 Event-based approach

Since real-life events can not be predicted, we made use of event-base mechanisms. mom offers a simple, and efficient means of communications, using the producer/consumer principle. Consumers subscribe to a topic they are interested in, and producers just have to publish on the right topic. Thus, producers do not care about the presence of consumers. It is a good point to get rid of apparition issues.

To be able to use this approach, physical devices have been considered through two perspectives. *Sensors* are sensing real life and human actions. Their role is to feed the system with events coming from real life. They are producers of events. Consuming these events, *Actuators* are acting on real life, using real-life equipments. They realize orders such as switching on the lights, or moving up the shutters. Components are not limited to a unique role, and can both consume and produce events.

**Actuators** propose two main methods. *getAvailableActions()* returns a lit of actions that can be realized on the device. If a light can answer [*on,off*], a shutter would answer[*up,down,stop*]. For each action, actuators are waiting for messages on a specific topic. For a sensor to ask for an action to be realized, it must know the matching topic. *getTopicFor(String action)* aims at providing the topic and the parameters that can be accepted for a given action in form of a *Message*.

**Sensors** are maintaining a list of messages for each event they sense. An On/Off switch maintains two lists of messages : one for each action. The messages stored also embed the topic on which they have to be published, for the action to be realized. When an action is sensed, each message stored for this action is sent on its topic.

### 6.1.4 Example

For instance, figure  shows the configuration phase for the connection of a switch and a light. The *Configurator* retrieves the message to be sent to switch on the light. This message is added to the list of the "on" sensed value of the switch. Later, when the "on" button is pressed, all messages stocked in the list are sent. When an actuator recognizes its "on action message", it forwards the order to the real light.

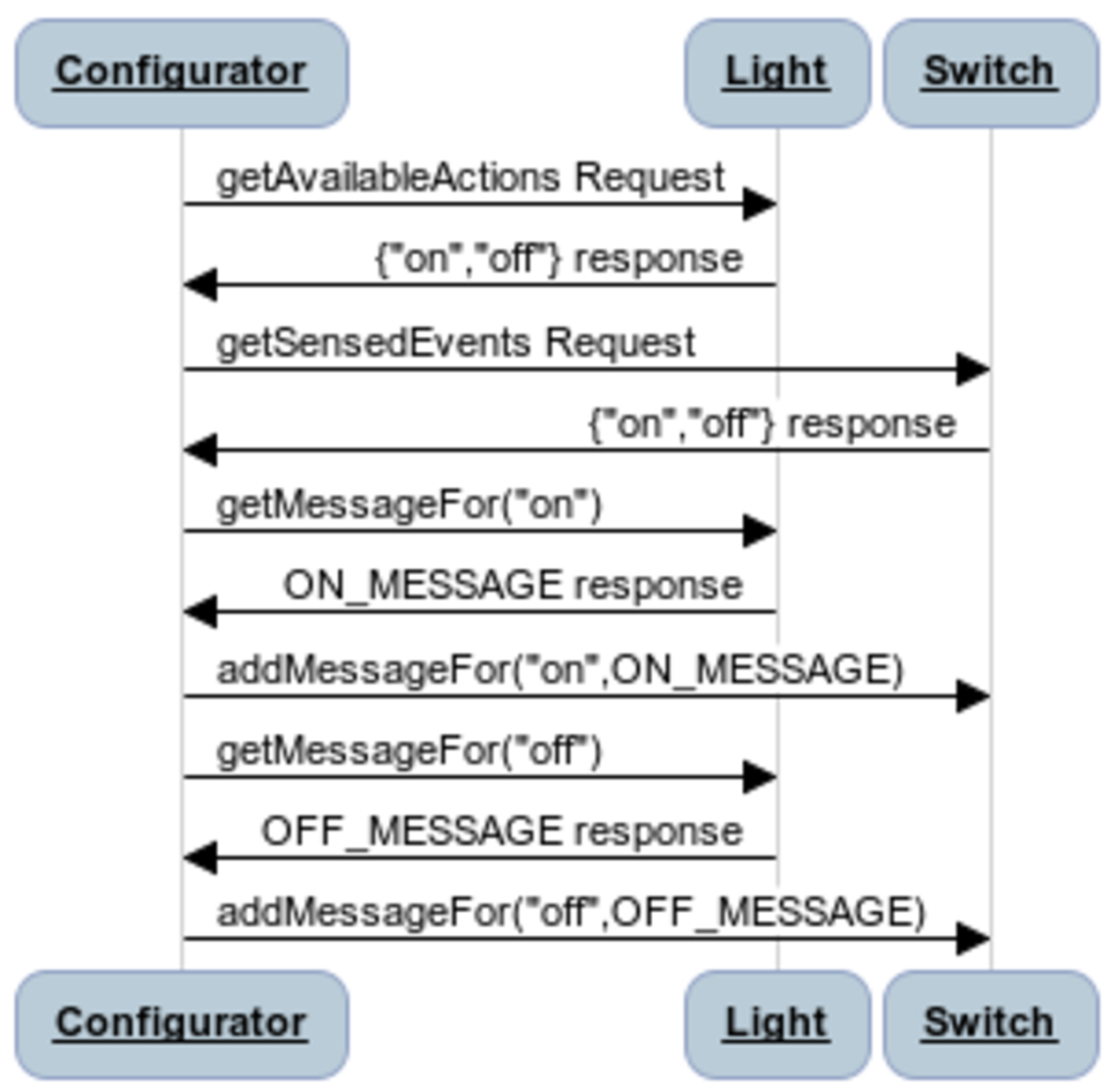


Fig. 6.2: Configuration Phase

This mechanism allowed us to get rid of asynchronous aspects. It also allowed any sensor to control any actuator, since they do not have to know each other to be able to work together. The action realized, and the value sensed, do not have to necessarily be the same. Thanks to the mechanism of messages, it is possible to send the "on" message to the light when "down" is sensed by a shutter command.

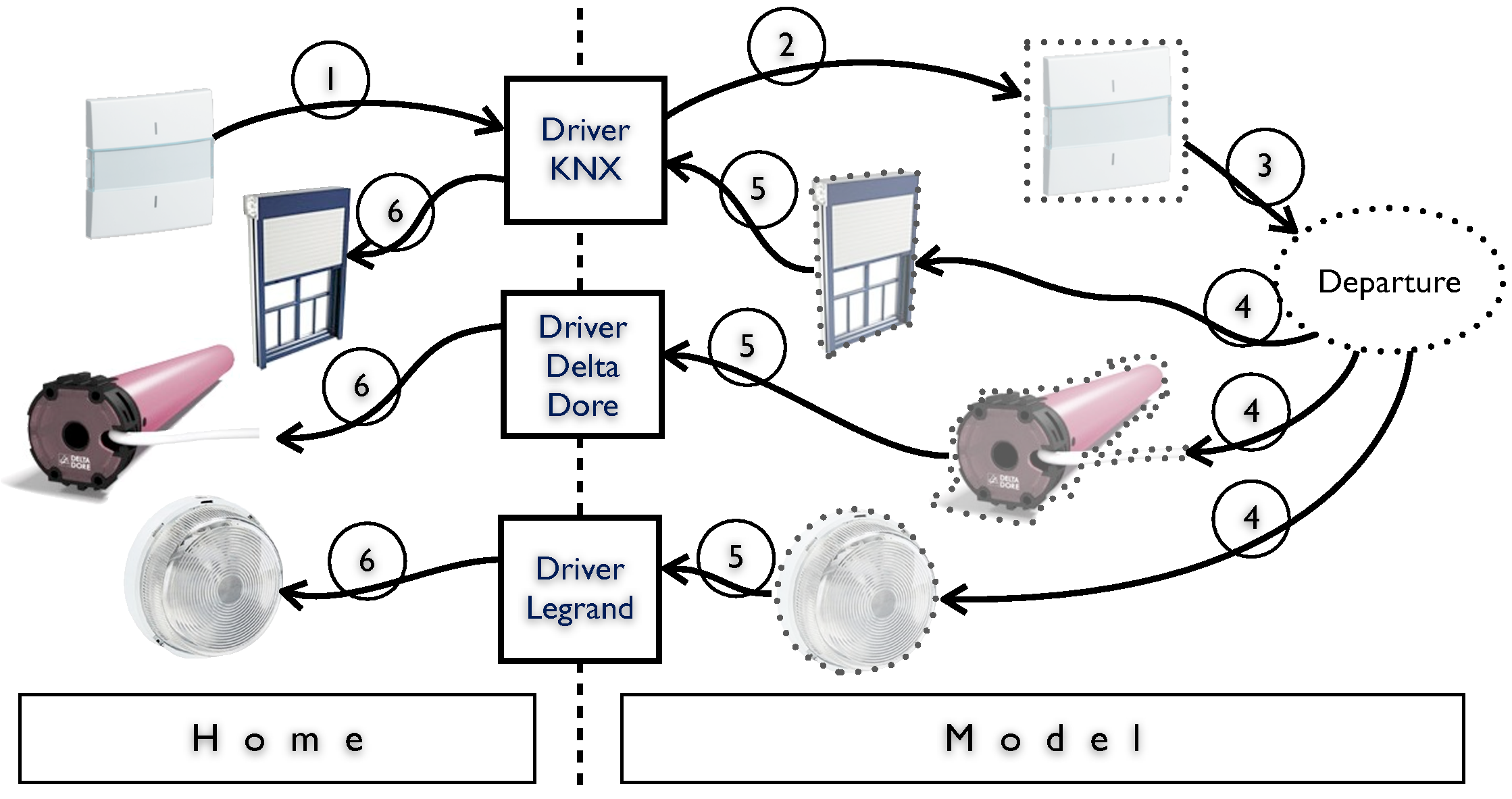


Fig. 6.3: Example Interoperability

Let us consider a home with a switch to trigger a departure scenario. This scenario switches off all lights, and pull down shutters. In the considered example, there are only two shutters, and one light. The first shutter has been motorized when the house was built, and works on a KNX network, just as the scenario switch. The second shutter has been added after, and as owners did not want to make holes in the walls, they chose a shutter engine communicating with the command by radio frequencies. Lastly, lights are controlled by Legrand equipments. All these elements are visible on the left of figure .

The interoperation of all these elements is described in the case of a departure scene execution. Numbers on the figure present the sequence of actions.

**1-** An inhabitant presses the button. This action is sensed, and generates a message on the KNX network.

**2-** This message is read by the driver, and translated into a message for the EnTiMid system.

**3-** The driver then selects the virtual representation of the device responsible for the message, and activates the sending of stored messages.

**4-** Its activation makes all connected elements to be activated in parallel.

**5-** On receiving the message, each model representative of a real product asks its driver to send an order to the real product.

**6-** The driver executes the query, and sends the order.

In this example, various devices with various actions are connected together. A switch that senses a *departure* is connected to two *down* actions on two different components and one *off* port. Interactions between components are possible thanks to the exchange of messages.

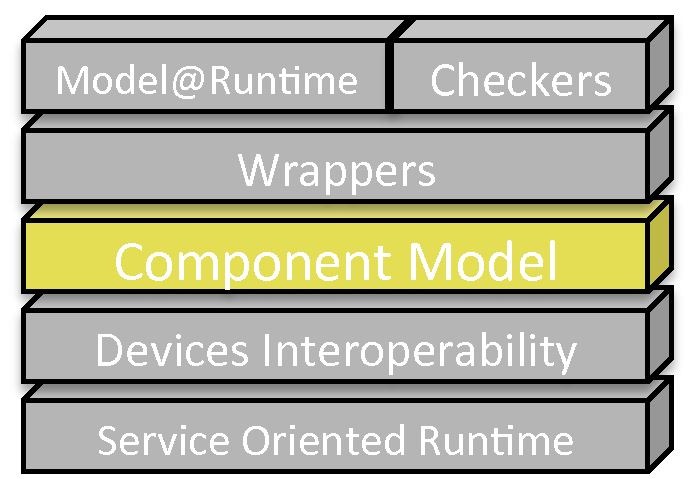
### 6.1.5 Summary

The *Device Interoperability* layer enables products from any manufacturer, to work with any other product, in any imaginable way, by just implementing a driver. The use of this method is however limited by the non-availability of actions lists at design time. Each device provides information about the actions it supports, but a method has to be called. Since method calls can not be realized at design time, the only way to get available actions is to go and seek them in the implementation code.

The sequence diagram, presented in figure , describes a part of the work that a piece of program has to execute to set up the application’s behavior. The sequence can not be implemented once for all, since the application may have to be adapted and to evolve while running. The configuration has to be expressed another way, to be able to change it, and to ease the reading and understanding of the behavior.

Both of these issues require a tool. It has to be able to provide information about the devices at design time, and it has to support and help the design of devices’ assemblies. This tool, a new component model, is made available by the *Component Model* layer.

## 6.2 Component Model



On top of the *Device Interoperability* layer, a tool is required to explicit devices abilities and their interactions. This tool has to take into account the sporadic apparition of devices. Component models have been identified as good candidates to take on this role. They are very good representatives for real life devices, since they can use or provide services. Their interfaces(as list of actions used or provided) are made explicit. Their life cycle is very helpful to catch the dynamicity of devices’ presence. Finally, the concept of components in software engineering is very close to electronic components. It makes it easy to understand and manipulate for anyone familiar with electronic components. Device manufacturers and software developers have here a common discussion base, a common language.

As presented in the state of art, component models are often too strict, and prevent from connecting components with non-identical interfaces. This restriction could compromise the interoperability gained by the *Device Interoperability* level, whereas this layer just aims at simplifying the configuration and the management.

This section organises as follow. Section  emphasis the relation between the proposed component model and electronic components, relation illustrated in section . The mechanisms responsible for the synchronization of model and code are presented in section . Lastly, the section  describes how the *Device Interoperability* integrates with this layer.

### 6.2.1 Make software components closer to electronic components

Talking about components, electronic ones are probably the first kind of component to come in mind of a lot of people. An electronic component, as shown in the bottom left part of figure , is a black-box surrounded by pins. The shape of the pins is standard and allows components to be connected to any board. Obviously, neither the pins nor the board have the ability to refuse the connection of two components. This absence of constraints allows electronic components to be used in a large variety of contexts. They can be connected to a multitude of other components to create appliances. This is the perfect description of the behavior required for a software component.

Nevertheless, software components’ ports are generally specialized by a programming interface(API). Thus, unlike electronic components’ pins, their shapes are not standard. The goal of this specialization is to ensure the alignment of services. However, this is a too strong limitation in our context.

In electronics, components admit only three kinds of ports(pins).

**Input Ports** collect all necessary information from the outside, for the component to realize its job. In the same time, they can trigger the execution of an associated task. Typical examples are the *A* and *B* input values of a comparator.

**Output Ports** release the data resulting of the execution of a task. The *C* result value of a comparator, or the tick of a timer, are two illustrations of this kind of port.

**Parameter Port** are used to set specific values for an instance. It specializes the behavior of the instance for a specific context. An example could be the *clock* port of a microchip, which can be set to several frequencies according to the application it is involved in.

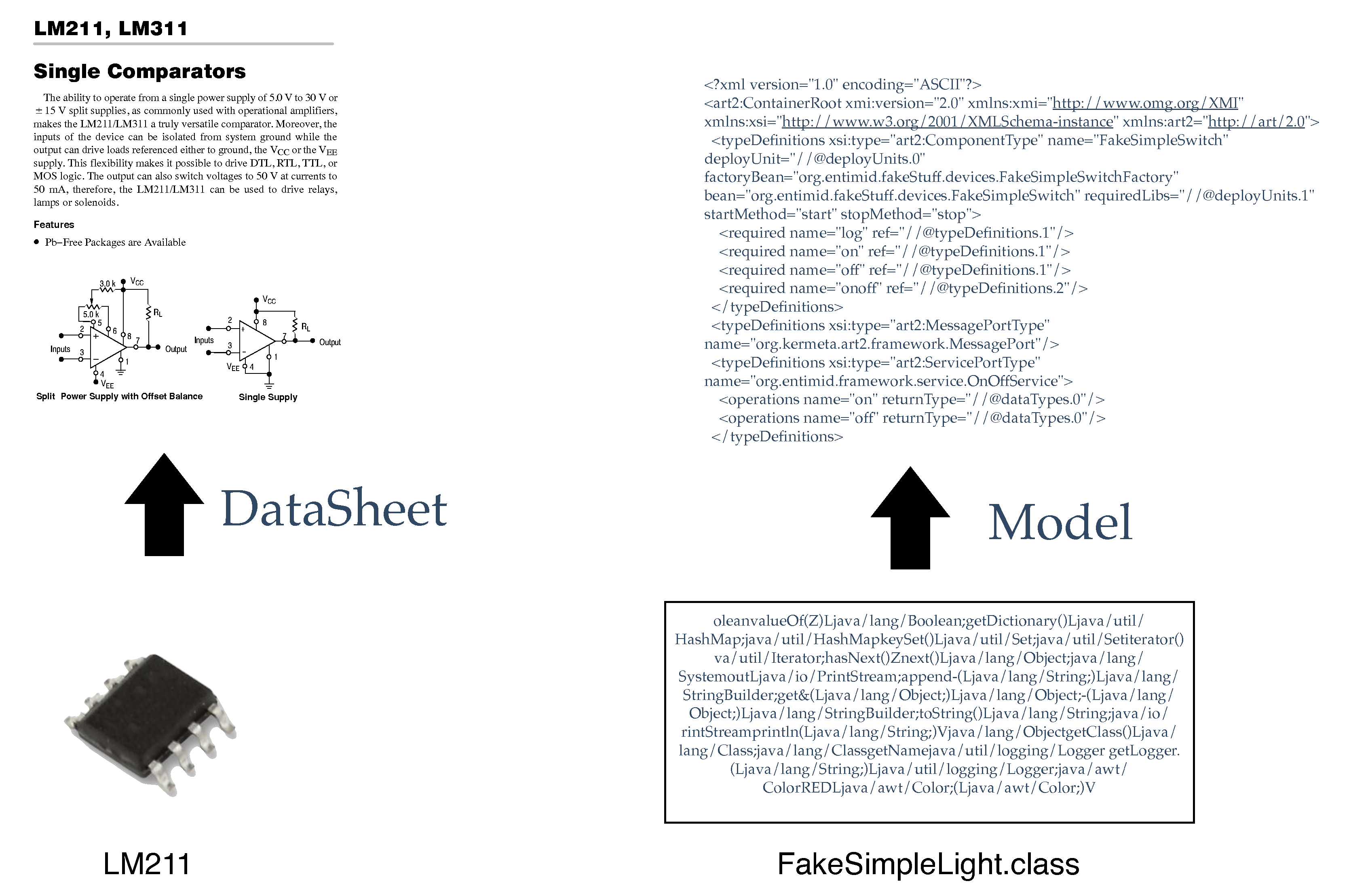


Fig. 6.4: Electronic Parallel : Datasheets

The component model created strongly inspired from electronic components. Indeed, *InputPorts* and *OutputPorts* have been implemented as presented in figure . They have been divided into *synchronous* and *asynchronous* kinds, to handle both object-based method calls handling, and message-based communications between components.

To keep close to existing component models, and promote compatibility, the component model makes use of classical terms in its implementation. As a consequence, *InputPorts* are implemented as *provided* ports and *OutputPorts* as *required* ports, as shown of figure **Erreur ! Source du renvoi introuvable.**. On their side *ParameterPorts* have been implemented as <key,value> dictionaries.

The following details each kind of port.

##### ParameterPorts

This kind of port is used to specialize a component behavior. For example, the *Timer* component uses a *delay* parameter that represents the amount of time to be spent before the time out occurs. A component can have multiple parameters. They are uniquely named in the component’s scope, and can be optional or mandatory. All parameters a component admits are listed in a dictionary at the model level. At runtime, each parameter port is instantiated as a setter method, which only admits a dictionary as parameter. Indeed, each parameter port has its own setter method. Types of both keys and values are Strings. This ensures the transmission of parameters in a unified way. Each component is responsible for the conversion from String to the real type of its parameters.

To keep the link with electronic components, the method is the pin, and the dictionary describes the shape of the signal to be sent(voltage, intensity, shape of the signal).

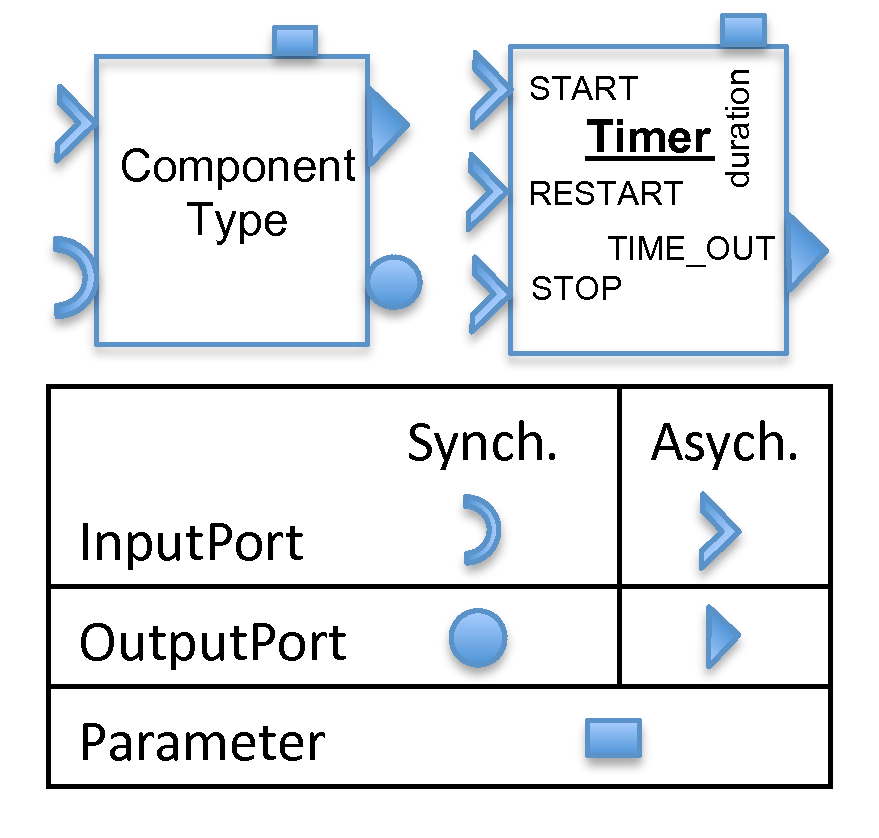
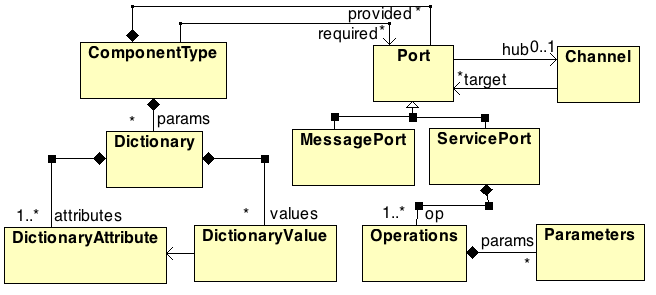
[Component Type] [Meta-Model Excerpt]

Fig. 6.5: Extraction of a part of the component model architecture

##### InputPorts

A component can provide several facilities to other components. This is illustrated by the *Timer* on figure  which offers start, stop, and restart actions. In classical software component models, the timer component would have provided a single synchronous port with the three start, stop, and restart methods. These methods would have been defined in the StartStopRestart API.

Synchronous ports (also called ServicePorts in the model) are acting the usual way : they are typed by an API, and are based on method calls. Thus, our component model is able to support the common software components behavior. However, this is not the way of designing components we encourage. Indeed, the API is typed by the programming language type system, and this typing may prevent components from being connected because of a mismatch. We want to get rid of the implementation’s typing, and deal with the typing at a higher level of abstraction.

Asynchronous ports, handled as *MessagePorts* in the component model(fig **Erreur ! Source du renvoi introuvable.**), are much more interesting for the promotion of component connectivity. Each method/action a component offers is accessible through a dedicated port. Each port is uniquely named in the scope of the component. In the same way, electronic components have one pin for each action, and actions are triggered when the value changes from 0 to 1 for instance, on the corresponding pin.

To mimic this behavior, all asynchronous *InputPorts* are implemented as a *Command* design pattern. They have a unique method *public void process(Dictionary<String, String>)*. The uniqueness, and standardization of the method, are mandatory to ensure the connectivity.

Just as an electronic component, actions in our component model can have parameters. Coded in the shape of the input signal passed through an input pin in electronics, our *InputPorts* admit a dictionary of <key, value> parameters. Like ParameterPorts, this dictionary only allows pairs of Strings. These values are specific to each execution, and may change form a call to another. On figure **Erreur ! Source du renvoi introuvable.**, *START*, *STOP* and *RESTART* are all InputPorts. The parameters used on the start or restart activation are transferred through the *TIME\_OUT* OutputPort to the connected component.

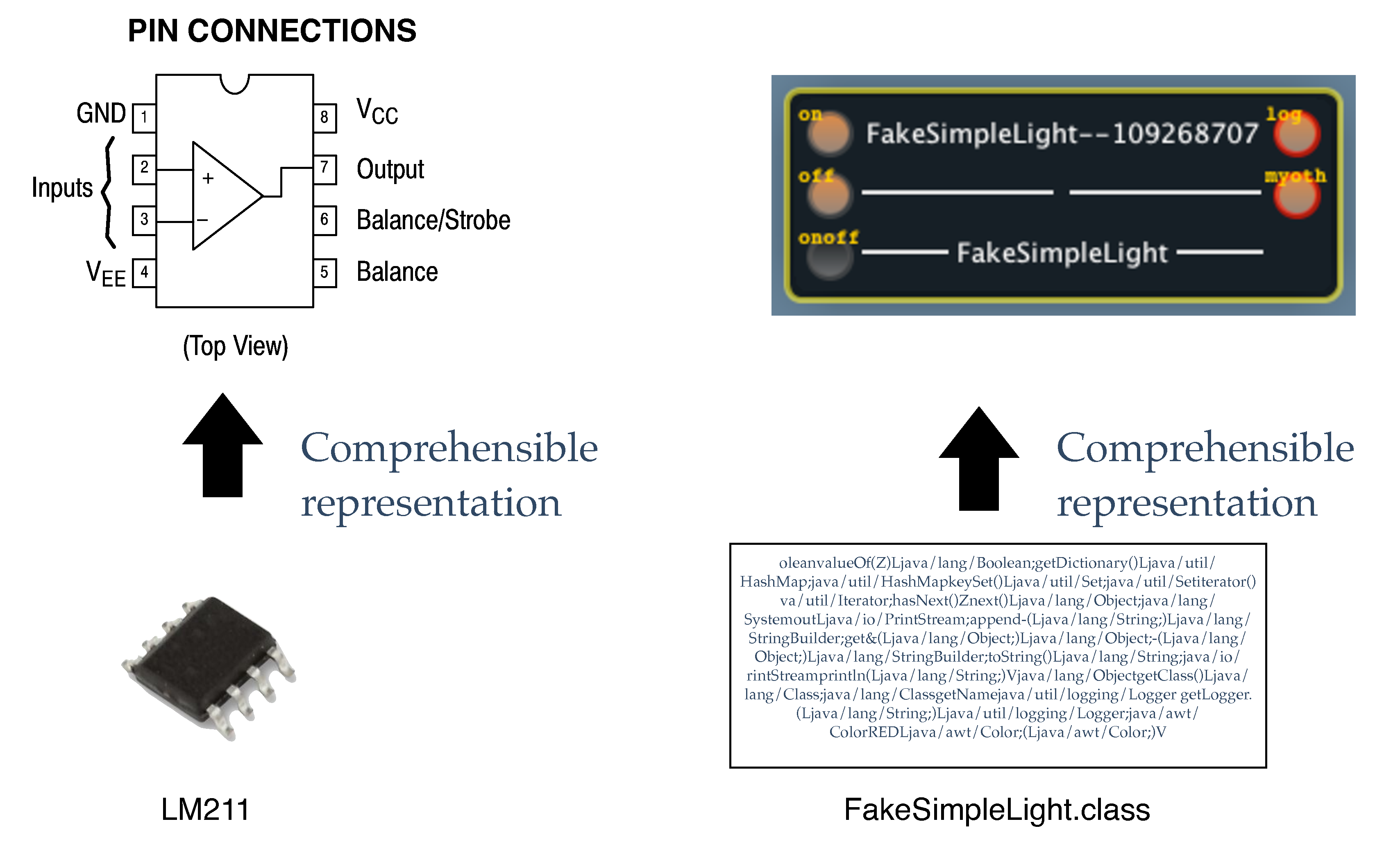


Fig. 6.6: Electronic Parallel : Components

##### OutputPorts

The main role of an OutputPort is to forward or release information. For instance, on figure  when the timer delay is over, the *TIME\_OUT* port is activated and thus, the connected InputPort (if any) also is. In case the activated *InputPort* is synchronous, the result of its activation is returned by the called method. This is a blocking behavior, and may not be adapted to events coming from real life. If the activated port is a MessagePort, the result of its execution (if any) is given though a dedicated *OutputPort*.

All these, results in a parallel between electronic and software components as shown by figure .

### 6.2.2 Concrete example

This example shows how a real product is implemented in this component model. The RMG4S, on the left side of figure , is a KNX product by Theben[[1]](#footnote--1). This product can control up to four 230V lights or sockets. Its virtual representative has 8 message input ports, two(ie : on, off) for each controllable element. When a physical event changes the state of an output of the product (somebody switches on the light using the dedicated switch), the state is propagated to any connected device, through the corresponding port of the component.

In addition, a *KnxEnv* input port allows the driver to circulate real-life events. On the other side, an output port *KnxNetwork* is used to send events from the model element to the real product through the driver. The last output port is a logging port.

Thus, to switch on the light physically connected to the first module of this product, one just has to activate the *m1\_on* port. The component then asks its driver to send a message to the real device to make it power up its first module.

On the other way, when the state of a module is physically changed, a message is sent from the driver to the component. The component then activates the *m1\_state* (for instance), to inform any connected component about the change.

If the application proposes a graphical user interface, the *on* (resp. *off*) port of the component is activated when the user presses the graphical button. On activation, the driver sends the order to the real product, which reacts and sends information about its state change. The driver catches the information, and sends it to the graphical interface for update, through the dedicated output port of the component.

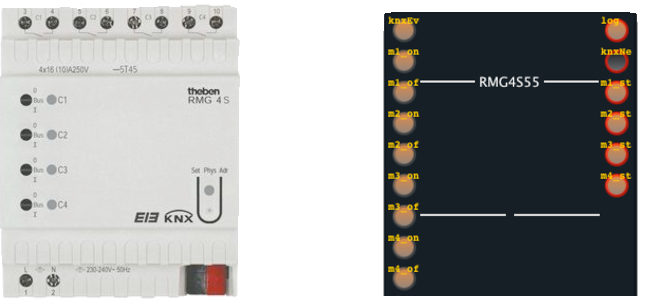


Fig. 6.7: Example Model

The component model makes our software equivalent to electronic components’ *DataSheets*. Assembly constraints, mandatory parameters on ports, component behaviour, and many other information on components can be expressed in the model. This abstract description of the component has no effect on the runtime implementation (just as *DataSheets* has no effects on black-box components by the way).

### 6.2.3 Binaries and Model Relationship

[Sorry. Ignored \begin{lstlisting} ... \end{lstlisting}]

The development of a component (ie : a virtual representative of a physical device) can be achieved in two ways. According to its preferences, the developer can make the model of what he wants, and ask for code generation. This approach is called *Model First*. The model can also be extracted directly from the implementation code made by the developer. It is the *Code First*. These two approaches are not exclusive. The model of a component can evolve, and its implementation has to be impacted. Respectively, if a change is realized on the code, it has to be reproduced at the modeling level. In other words, the consistency between implementation and model has to be guaranteed.

To illustrate the description, the listing **Erreur ! Source du renvoi introuvable.** shows the complete implementation class of a *Timer* component. This listing organizes as follow. On the first lines are annotations on the class that describe the component shape. Just after, the class definition comes with private attributes, object builders, then getters and setters. Next methods are rendering the services offered by the component. Life-cycle management methods are at the end of the class.

**Component shape**

First annotations on the class inform about the *Input*, *Output* and *Parameter* ports. As explained in section , *InputPorts* are implemented as *provided ports*. Common actions that can be realized on a timer (start, stop, restart) are listed under these terms.

This *Timer* implementation offers two outputs, visible as *Required Ports*. A *log* port, which sends information about the internal behavior of the component, and a *time\_out* port activated when the countdown ends.

The Timer admits a parameter. This parameter sets the delay between the start and the activation of the time\_out port. This parameter appears in a dictionary.

The *@Library* indicates that the component is part of the virtual library of components called "EnTiMid - Framework". In edition tools, all components of the same library are presented under the same package of components. A library of components can be defined using several deploy units.

The last annotation just tags the class as a component type implementation. This annotation is mandatory for the compilation tools to consider the class as a component type.

**Port mappings**

Once described, input ports have to be linked to the method implementing the action. In the example, one may remark that a port can be bound to at most one method, but a method can be reached from several ports. Indeed, the behavior of a *start* and a *restart* of a timer are implemented the same way. Classical solutions could have been to remove one action or to copy-paste the method. From a user perspective(sect. **Erreur ! Source du renvoi introuvable.**), a Timer should be able to be started and restarted which implies not to remove the port. Thanks to this multiple mapping, the user will be satisfied with no redundancy of code.

Moreover, the transfer of the annotation from a method to another, changes the method called when a port is activated. This change is completely transparent for assemblies already using the component, since the annotation is not modified. This is a great ability that enables changes in the implementation and method names, with no change in the component interface.

Finally, a component can offer the same service through both *Service* and *Message* port. Using the same mechanism, the same method can be called in both cases.

**Life cycle**

*Start* and *Stop* life cycle methods are mandatory. They are called when an instance is started (resp. stopped). Stateless components may just ignore these methods, but statefull ones may use these to persist their state.

The *update* method is used to inform a component that one of its parameters has changed.

**Code first**

The meta-information, concerning the component model, are introduced in the code using annotations. This method for including meta-information in the code has already been used in tools like Fraclet[**Erreur ! Source du renvoi introuvable.**]. A developer familiar with the annotation set, or in charge of the migration of existing components, may directly define the model in the code.

As in a classical development process, the new implementation code has to be compiled to incorporate the changes in binaries. Our component model takes advantage of the compilation phase to extract the model from the annotations. A visitor goes all over compiled classes and selects the *@ComponentType* decorated classes. Then sub-visitors navigate into the code to create the model.

At the end of the compilation process, the newly computed model is added into the compilation result. *id est* the model is included as an XML file into the *.jar* that results from the compilation. The model consistency with the latest code version is guaranteed this way.

**Model First**

Writing component type code, plus the annotations, may become a complicated task. A non-familiarized person may experiment difficulties in placing all needed annotations to describe its component. The model first approach aims at providing tools to graphically (or textually) describe the component first, and ask for the generation of the implementation. This method is made available by the use of tools such as graphical dsl, textual dsl or generic meta-modeling languages like Kermeta [**Erreur ! Source du renvoi introuvable.**]. Models bring a more intuitive approach for the description of a component.

Once the developer is done with its component’s model, the generation tool is activated. If the implementation class of the component does not already exist, a new file is created. This file contains the skeleton of the component implementation. Obviously, the code generation reaches its limits when the body of methods has to be created. The behavior of methods is the only part to be completed by hand by the developer. Otherwise, the class is already decorated with all annotations, and ports are mapped by default on generated methods.

When an implementation class already exists, the generation process is a bit more complex. In fact, a temporary model is extracted from the existing code and an ast of the code is created. The ast describes the code using a structured tree of object. Each object stands for a method, an argument, an attribute, etc. A comparison is made between the model created by the user, and the model extracted from the existing code. Each difference is analysed, and modifications are made on the ast. The final code is generated from the modified ast.

The model first approach fairly helps in linking the model and the code. Anyway, the resulting code still needs a developer to complete the new methods created, to remove dead code and to optimize the mappings.

Thanks to these mechanisms, models of components are available while not running. Their conformance with the actual implementation is guaranteed by construction. This model abstraction makes it possible to create and exploit tools from mde.

### 6.2.4 Link with the interoperability layer

The actual implementation of the component model is a bit different from the view developers can have on components. The connection between two components is graphically as simple as drawing a line linking two ports (see the top of figure ). Since ports can be synchronous or asynchronous, the runtime can not handle port connections in the same way. The activation of an output port implies different behaviors according to its kind. A message output port will send messages on topics to activate the linked input ports, but a service port has to start a method call.

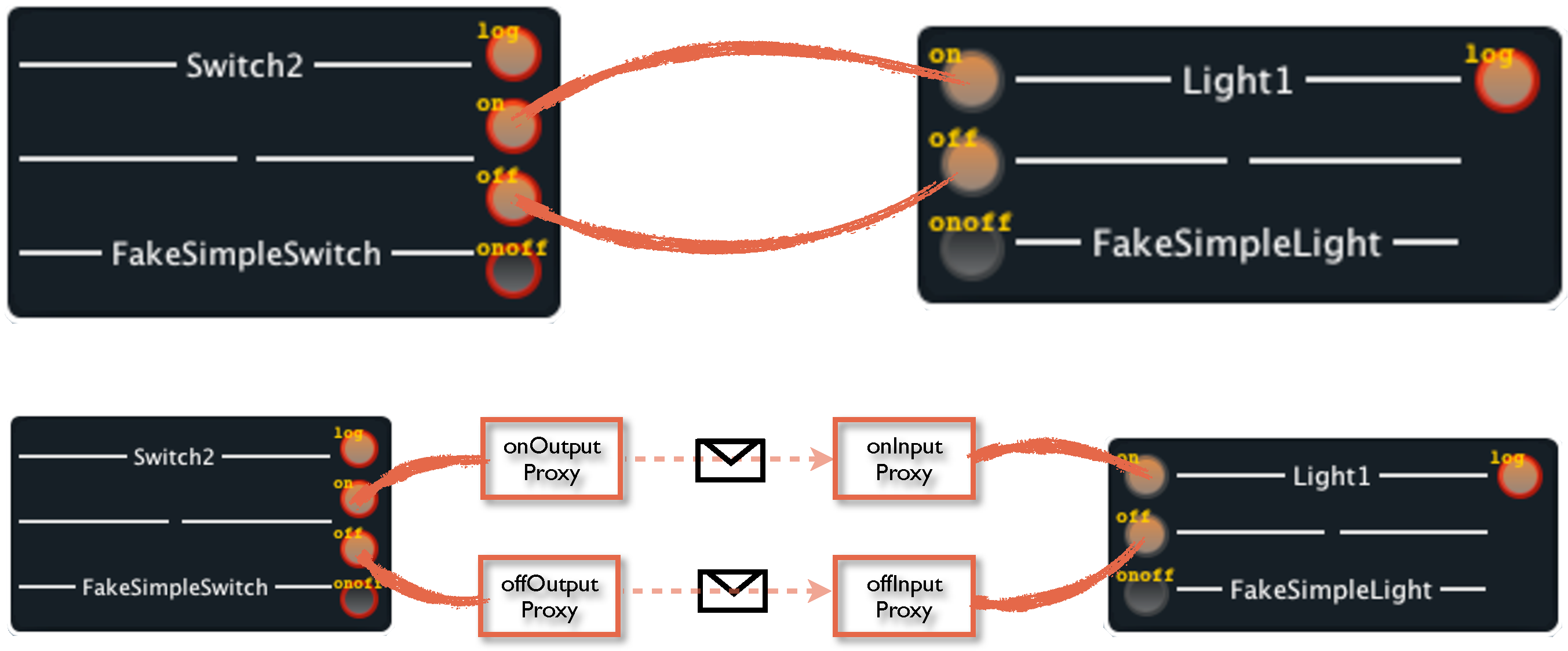


Fig. 6.8: Link between interoperability layer and component connections

This complexity is hidden from developers, in both the code, and the model view of the components, by the use of proxies. At runtime, a proxy is generated for each port connected to another component’s port, as illustrated in the bottom of figure .

If the port is a message port, the proxies are using messages and topics to communicate with each other. When an output port is activated, its proxy generates and sends a message on a pre-defined topic. On the other side, a proxy listens to this topic only, and activates the input port on which it is connected when a message arrives. Activations are realised with a *Command* design pattern, from the output port to activate the proxy, and from the proxy to the input port. The mechanism is thus transparent from the developer view : an input port must provide a command pattern, an output port activates a command pattern.

The mechanism of proxies has also been implemented to handle method calls of service ports. Links between components’ ports are thus handled in a uniform way. The introduction of proxies makes it possible to use other means of communication (in case of distribution issues for instance), and enables some adaptation mechanisms.

### 6.2.5 Summary

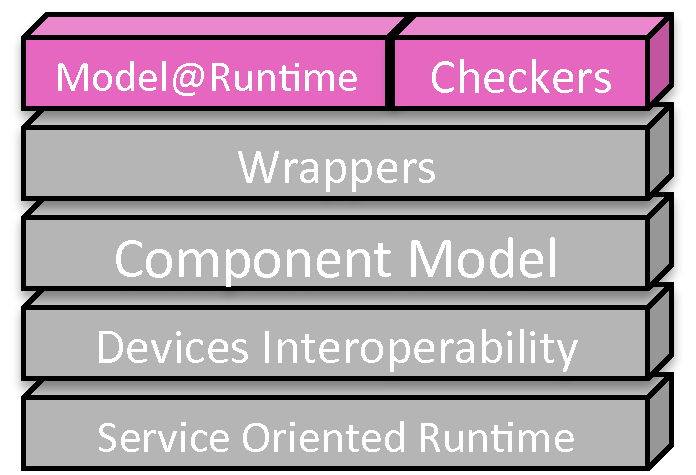
This new component model answers the need for a tool to explicit, at design time, the abilities of devices and their interactions. Annotations in the code simplify the integration of the component model into the implementation code, which ensure the synchronization between the model and the implementation. The component model also eases the reading and understanding of an application, since all links between components are made explicit.

The component model imposes the configuration to be completely defined, but it is not responsible for its deployment. A gap from the component assembly to the sequence of commands to set up the application at runtime still has to be filled.

Since the component model has been made flexible to allow any possible connection, any connection is possible, but some may not be desirable. Just as in electronics, assemblies are constrained by components’ specificities. If electronic boards allow all possible connections, components have constraints to be respected to assert their behavior. Assemblies have to be verified, and simulated, to prevent from any non desirable interactions.

To get rid of these two issues, the Model@Runtime approach and model checkers have been used in EnTiMid. Model checkers enable verifications of component assemblies at several steps of the development, while the model@runtime takes responsibility for bridging runtime elements and the component model. These two elements of the proposition are making the *Model@Runtime and Reasoning Engine* layer.

## 6.3 Model@Runtime and Reasoning Engine



The *Component Model* layer provides a great level of abstraction from the implementation specificities. It offers a unified model view of components and their constraints, and enables the creation of management tools. Reasoning engines, checkers, and models@runtime abilities can be used to ease the creation of component-based applications.

The first requirement targets the validation of assemblies, prior to their real deployments. This checking step is described in section . Section  presents an overview of the use made of Models@Runtime techniques in this approach.

### 6.3.1 Check to validate

In electronics, the components assemblies have to be approved. Their conformance with relation to components, and applications specific constraints has to be guaranteed. This validation prevents assemblies from any computable damage. This conformance check is often realized by simulations, based on components’ specifications described in their documentations. Figure  shows again, the parallel between the electronic approach and ours, where electronic simulations are replaced by model checking in our context.

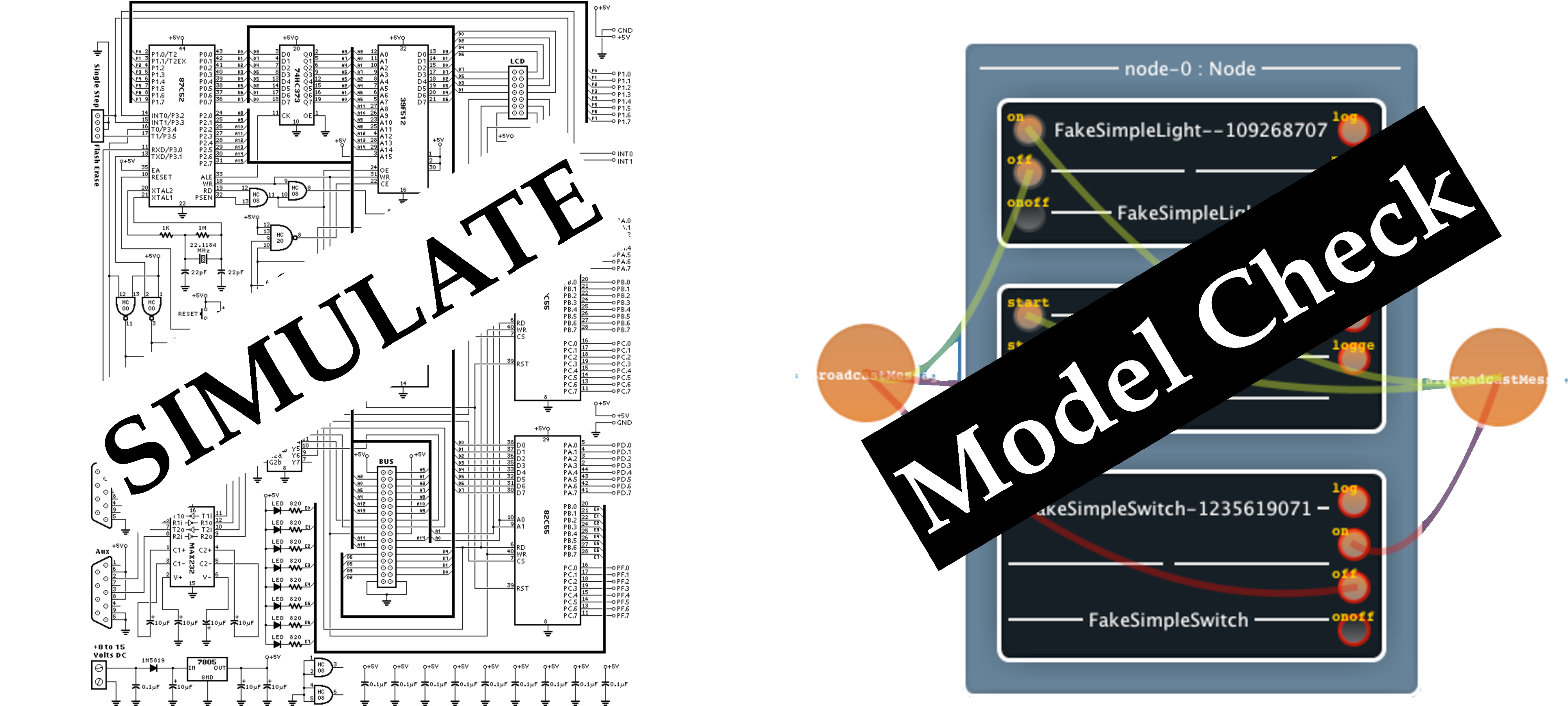


Fig. 6.9: Electronic Parallel : Simulation

In a classical software engineering process, checking of conformance are made at 1) design time by the developer, 2) compilation time automatically, and 3) by running tests on the application built. The modeling approach offers a way to perform more precise checks, targeting more specific concerns, at several moments in between design and deployment phases. Figure  displays the different moments where checks can be performed, and illustrates what can be check at each moment. These steps are detailed in the next paragraphs.

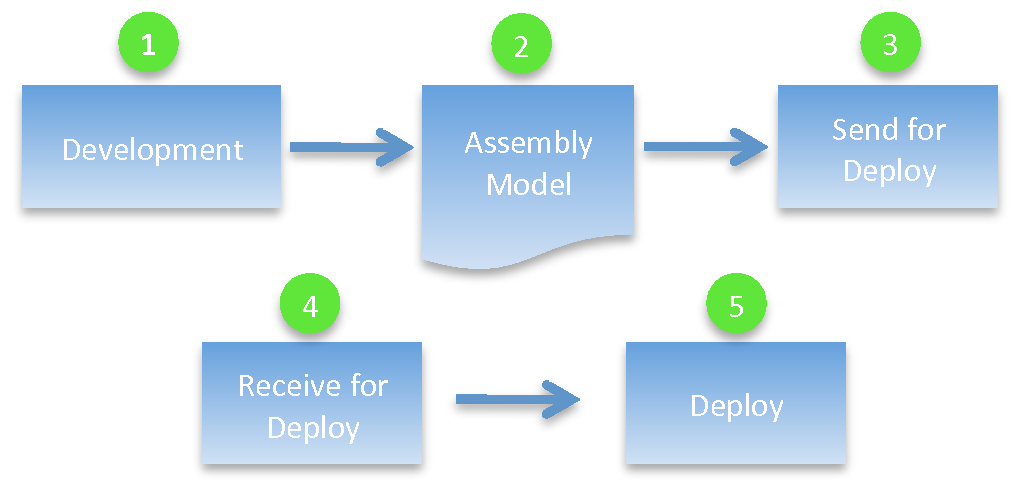


Fig. 6.10: Checkpoints positions in the assembly deployment chain

1. **Developer actions**

The assembly tool can monitor developers’ actions. During the conception, checkers can verify that only authorized operations are executed by the developer. When an inappropriate action is realized, the triggering of warnings and errors can improve the development process. Thanks to these information, developers can immediately correct their code, and learn from their mistakes. The earlier errors are detected, and corrections made, the more the impact on the global solution is reduced. Also, developers’ profiles could be created, and associated with different checking policies according to the developers’ expertise. This provides a fine grained checking process for the design of applications.

2. **Assembly constraints**

The structure of an assembly can be constrained by rules, due to runtime constraints, or due to the framework used. These rules are neither specific to the developer, nor to the targeted business. A general policy could impose assemblies to be composed with at least two communication components. This constraint aims at keeping a continuity in the communication service in case of failure. Valuable for all applications created by the company, this rule is shared by all software development projects.

3. **Business Rules**

An application created to control a plane has different constrains compared to a watering management system. Each application domain can require special rules to be considered. This checkpoint is placed just before the model deployment. The validation of conformance at this moment avoids the sending of corrupted models to the runtime.

4. **Platform Rules**

A platform is a system composed of both software and hardware. The composition of the execution platform may impact the development, or deployment of a component-based application. The role of this check is to verify that all constraints, inherent to the platform choice are respected. Checks being performed at the model level, they can be realized by the runtime platform itself, with no consequence on the running application. For instance, a model can be rejected if one of the components requires a serial connection, and the runtime hardware of the platform has none.

Modeling the platform resources could enable to move these checks before the deployment, and gain time.

5. **Check deployment commands**

The last step of checking sits in the first step of Models@Runtime mechanisms. As explained in section , the deployment of a component assembly is split into several commands. This last verification ensures all commands to be executable before running the sequence.

If the model of the assembly successfully passes all checkpoints, it is ready for deployment. This phase is handled by the Models@Runtime engine. Its job consists in (1) defining the best way to go from the current system assembly to the new assembly received, and (2) supervise the migration. This is explained in section .

### 6.3.2 The Model@Runtime engine work

Dr Morin has presented the Model@Runtime engine in his PhD Thesis[**Erreur ! Source du renvoi introuvable.**]. The model@runtime engine is responsible for several actions. First of all, it has to constantly maintain a model view of the running system. Secondly, when a new model is asked to be deployed, the model@runtime engine plans the migration (ie : identifies and sequence the necessary primitive commands). Lastly, it supervises the run of the migration commands sequence, in order to roll back to the previous stable state in case of failure.

The next paragraphs provide an overview of the engine work.

**Identify and validate the changes**

After validation, the first task is to identify the differences between the model representing the running system (source model), and the target model the system must switch to, as illustrated in figure . During the comparison, the next 7 types of primitive commands can be found. *1.* **start** and **stop** components. *2.* **add** and **remove** components. *3.* **add** and **remove** bindings. *4.* **update** components.

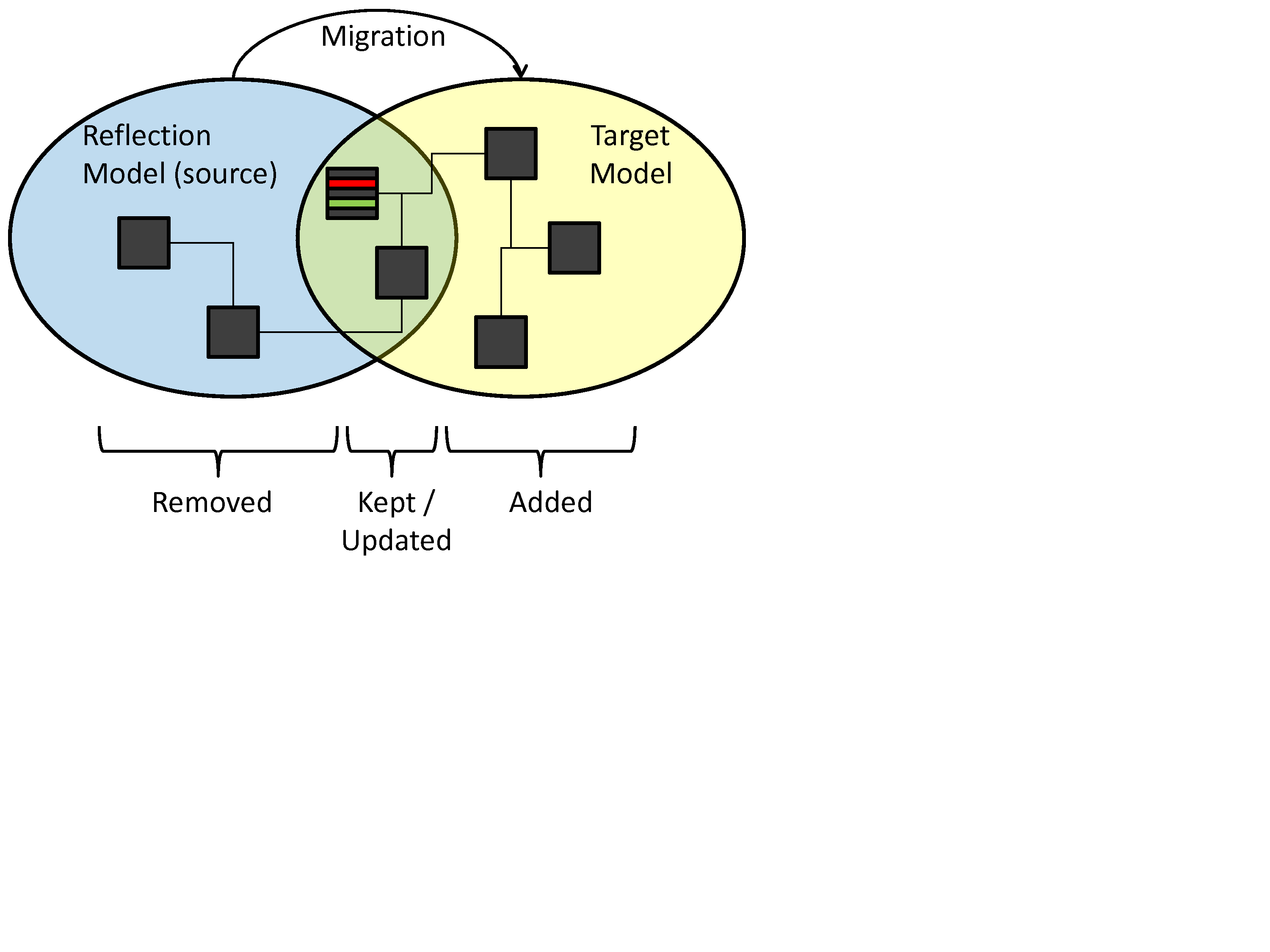


Fig. 6.11: Identifying Differences between the source and the target configurations.

The steps to go from the current configuration to the required one are specified by primitive commands that represent atomic differences between the two configuration models. The comparison system only deals with abstract commands, to allow a change of the component management policy. The real commands are instantiated (not yet executed) according to the actual policy, during the model comparison.

**Plan the execution sequence**

These commands are stored in a collection and ordered according to a heuristic[**Erreur ! Source du renvoi introuvable.**] that ensures a safe migration from the current to the target configuration. Before actually executing the commands, the list is parsed to verify that all the commands can be executed. For example, for all *AddComponent* commands, the presence of the specific component factory is checked, to ensure all components can actually be added without problem. Doing this kind of verification for all commands ensures that the command execution will properly execute. If a command is detected as non executable, a report clearly describes the problem, and no command at all is executed. This way, the system is always kept consistent.

**Roll-back abilities** In case the migration fails, each command is decorated with a roll-back equivalent command. Thus, each command executed before the failure can be cancelled. Moreover, a second protection in place consists in keeping the old model in memory. If everything goes wrong, it is always possible to restart from scratch, and migrate back to the old model.

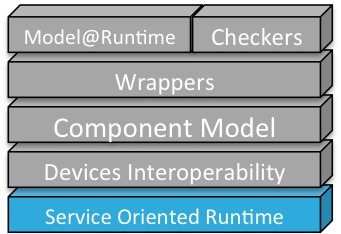
**Specificities of components and services**

Because of an adaptation, some links (bindings) between components may appear or disappear, for the system to act differently. In case of classic components, adding or removing bindings is as simple as setting or unsetting a variable. Generally, a component missing one mandatory binding is stopped, because it can not run any longer. However, in the case of service-based systems such as EnTiMid, the component may still offer its services to third-party applications, and thus, should not always be stopped. In other words, a "light component", virtual representation of a real light, may not be bound to any other component, but might still serve another application for the control of this light.

Other behavioural constraints can require more complex actions that just a set or an unset. For instance, if an alarm has been triggered, and if the user does not process this alarm, the system must be able to propagate the information anywhere else for the alarm to be treated. The removal of a communication link is structurally correct, but the link may take part in an operation being treated, and so, it has to be kept until the end of the action.

As explained, real commands for the migration are instantiated according to the current policy of the running system. Real commands can also be specialized for each runtime they have to be applied on. In our context, atomic commands have been instantiated to address a service-oriented runtime. Indeed, this runtime offered all facilities required by our approach. This is detailed in section .

## 6.4 Service-Oriented Runtime Architecture



For the proposed approach to efficiently cope with dynamic evolutions, the underlying runtime environment is required to offer dynamic abilities. As explained in[**Erreur ! Source du renvoi introuvable.**], the concept of service has emerged as good candidate to cope with dynamicity of adaptive systems. The adoption of this concept led to the development of technologies, standards, and methods to build service-based applications. Since the Service Oriented paradigm insists on the pervasiveness of services, it naturally imposes service-based applications to properly handle this requirement. Indeed, services can appear and disappear at any time, and applications built upon these principles obviously have to take these constraints into account.

The OSGi Alliance [**Erreur ! Source du renvoi introuvable.**], ’consortium of technology innovators’, has released a set of specifications that define a service-oriented platform [**Erreur ! Source du renvoi introuvable.**], and its common services. This Service-Oriented runtime has been selected to support commands that require to add or remove component instances and types(binaries), during the execution.

**Dynamicity in OSGi**

The OSGi kernel is a standard container-provider to build service-oriented software systems. It implements a cooperative model where applications can dynamically discover and use services, provided by other applications running inside the same kernel. It provides a continuous computing environment. Applications can be installed, started, stopped, updated, and uninstalled, without a system restart. It offers a remote management model for applications that can operate unattended or under control of a platform operator. Finally it embeds an extensive security model, so that applications can run in a shielded environment. According to these specifications, an application is then divided on several bundles. A bundle is a library component in OSGi terms. It packages services that are logically related. It imports and exports Java packages, and offers or requires services. Services are implementations of Java interfaces.

**Modularity**

Each OSGi bundle is designed to reach the highest level of independence, giving the software enough modularity to allow partial services updates, adds or removes. This programming style allows software-builders, to deploy the same pieces of software for all of their clients, either professionals or private individuals, and then simply adapt the services installed. Moreover, the services running on the system can be changed during execution.

**Component Types, Instances and Bundles**

Described in section , the model@runtime engine creates an ordered sequence of commands when it receives a new model to deploy. Each command of the list is then translated into an OSGi command.

In EnTiMid, component types are contained in OSGi bundles. These bundles are only used as deploy units and do not provide any service. They just embed components. When a **addComponent** command is parsed, the runtime checks if the component type is available in the environment. If not, the bundle containing the type is downloaded and installed. Once the component type available, a new instance can be created.

Component instances are mapped on bundles too, because of their independent life-cycle management. Indeed, **start** or **stop component** commands are directly translated to start and stop bundle OSGi commands.

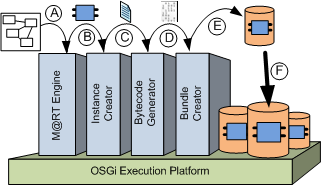
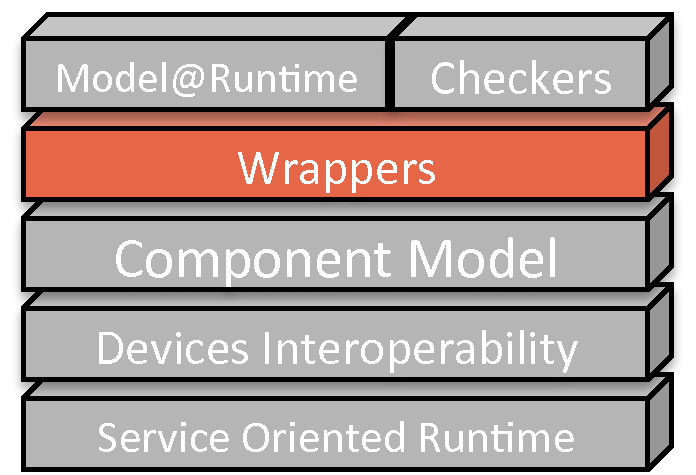


Fig. 6.12: Instance creation tool chain

The creation of an instance is realized as presented on figure . From the model(step A), a new instance is queried by the model@runtime engine(step B) to an instance creator. This instance creator can be a Java code generator, an XML generator, or whatever. The instance creator then asks for the bytecode generator to compile the instance. This compilation can be realized with ASM[[2]](#footnote-0) for Java code, handled by Spring for an XML file, etc. Step D consists in packaging the bytecode in a bundle. Step E makes it available for the runtime platform. The last step installs the instance bundle in the system.

Mapped on OSGi bundles, component instances can offer services to other components in the component model, or to other bundles on the OSGi platform. This facility makes it possible to dynamically expose instances on application-level protocols. This role is supported by the *Wrappers* layer described in section .

## 6.5 Wrappers



The wrappers layer is taking advantage of the component model layer, which makes it possible to dynamically, and automatically wrap devices into several current and future application level protocols. For instance, upnp, dpws or dlna are such kinds of protocols. Their implementations are often too heavy to be implanted into devices themselves. The role of this layer is thus to export all devices for free, on several protocols. Rather than offering an automatic publication mechanism to a selected protocol, the wrappers layer offers a means to publish any component to any protocol. Each component can thus be accessible using as many protocols as there are wrappers. This approach has been presented in [**Erreur ! Source du renvoi introuvable.**].

**Component Model** *versus* **Third party**

Wrappers need to get information about the devices present in a given deployment. This information can be retrieved in two ways.

(1) The wrapper is designed as a component. As a consequence, it can monitor the current model of the running system by asking the *Model@Runtime* layer. Any change in the model implies the wrapper to check new devices and removed ones. Another benefit of this approach is that a wrapper is considered as a classical component. The model can thus manage it as any other component.

(2) The wrapper is built as a third party application, running on the same platform. In this case, the wrapper monitors registrations of services in the OSGi context. Each time a device registers a service, this service is made accessible through the protocol handled by the wrapper. This approach has two drawbacks. First, the export and uses of services are not visible in the model. A device can thus be removed while in use through an application protocol. Secondly, the life cycles of the services exported on the application protocol, depend on the registration and unregistrations of the device’s services. Since no dependency is expressed in the model, the life-cycle management of exported services has to be handled ’by hand’ by the wrapper.

**Reversed drivers**

A wrapper is created for each application level protocol. Just as devices’ drivers, a wrapper is required to be deployed for each application protocol to address. Each wrapper monitors the current application to detect addition or removal of devices. For each device, the wrapper takes on the role of a proxy, and handle communications to, and from the application level protocol.

## 6.6 Summary

The service-oriented architecture of the runtime makes it possible to cope with evolutions and adaptations, since bundles can be installed or updates with no need to restart the system. These facilities are exploited by the Model@Runtime layer, which comes with tools and methods to address evolutions, variability, adaptations and safety of the application. However, these properties could not have been used without the creation of a new component model inspired by electronic components. This component model improves the flexibility, and enables connection of heterogeneous components, while keeping a high level of reliability thanks to checkers. Device Interoperability and Wrappers layers are providing respectively abstractions of manufacturers specificities, and free publications on application level protocols to promote interoperability and openness.

1. http ://www.theben.de/en [↑](#footnote-ref--1)
2. http ://asm.ow2.org [↑](#footnote-ref-0)