Xtext / Sirius - Integration

The Main Use-Cases

White Paper

December 2017

SUMMARY

Introduction

You are going to create a domain-specific modeling tool and you wonder how users will edit and visualize the models: textually with a dedicated syntax and a rich textual editor ? or graphically with diagrams drawn with a palette and smart tools?

Both approaches are interesting and can be used complementary: While text is able to carry more detailed information, a diagram highlights the relationship between elements much better. In the end, a good tool should combine both, and use each notation where it suits best.

In this white paper, we will explain the benefits of each approach. Then we will present Eclipse Xtext and Eclipse Sirius, two open-source frameworks for the development of textual and graphical model editors. And finally, we will detailed two use-cases where these two technologies can be integrated in the same modeling workbench.

Let's start

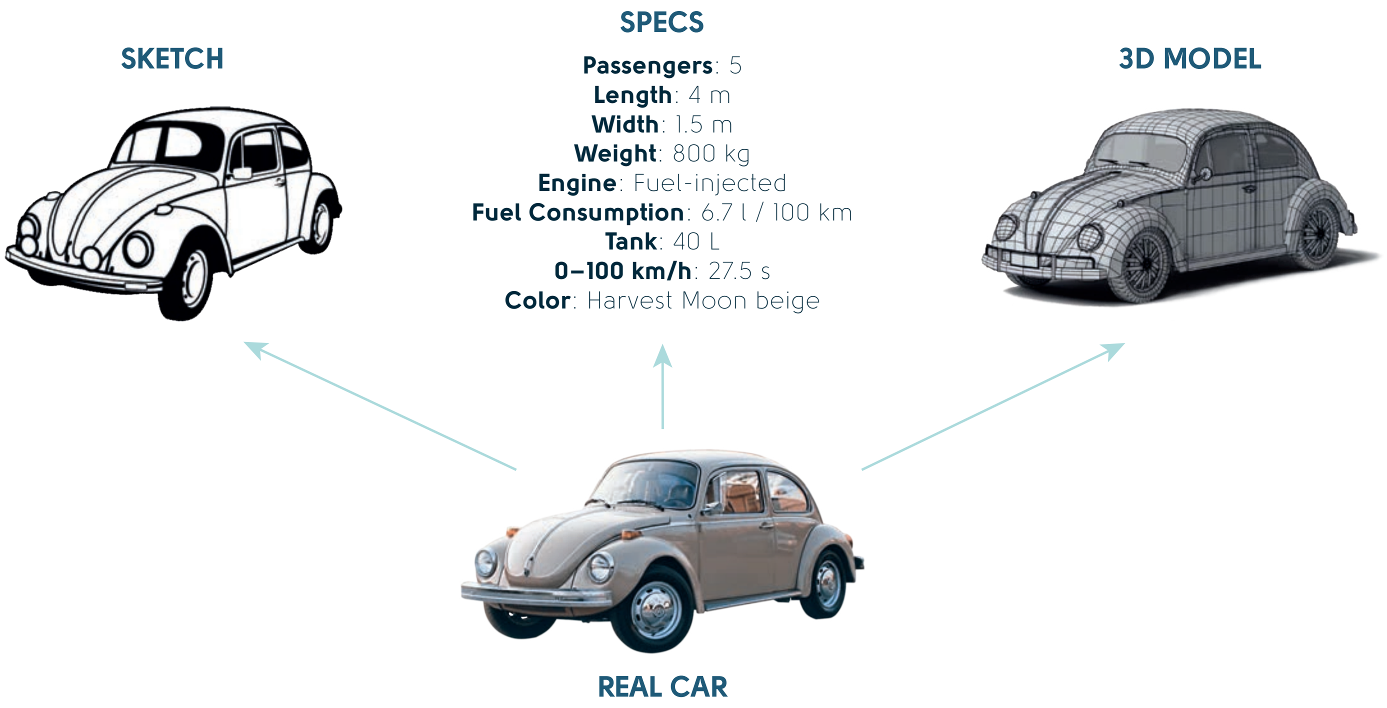
What modeling is about?

Before presenting the graphical and textual modeling approaches, it is important to briefly clarify what we mean by modeling.

Commonly, models are a simplified representation of a real system. But, they can be created to achieve different purposes:

- **Communication**: to help understanding a system by hiding its unnecessary details
- **Construction**: to specify what has to be concretely created
- **Simulation**: to virtually run the system and verify its behavior.

The model will not be necessary the same for each objective: in some cases it can remain high-level and imprecise, as long as the model conveys the right idea, while in other cases it requires to be very detailed and true to the (future) reality: if it serves to take important decisions or if it generates physical outputs.



Modeling is also a matter of domain expertise. Depending on their job, people building or analyzing a system are used to manipulate a vocabulary specific to their domain. In building industry, a plumber and an electrician do not use the same terms and notations to create their plans. In software industry, a database administrator and a web designer also work on two different domains.

In any case, Modeling is framed by a Domain Model (also named metamodel). This Domain Model precisely defines the elements that can be used to create the model. It consists in a list of concepts, their properties and the relationships with other concepts.

For example, to model a database, the Domain Model would define concepts such as *Table* and *Column*, with properties such as *name* and *size*, and relationships such as columns between a *Table* and its *Columns*. Thanks to these concepts, a database administrator can create a model with tables (*Person*, *Address*, …), columns (*firstname*, *lastname*, *city*, *country*, …), and link them via a columns relationship (*Person to firstname* and *lastname*, *Address to city* and *country*…).

The main benefits of modeling are:

- **Abstraction**: as a simplification of the real world, modeling forces to focus on the solution, independently from irrelevant details.
- **Communication**: expressing models with a vocabulary framed by a Domain Model makes the share of the information easier with people of the same domain.
- **Consistency**: modeling produces data that can be verified and leveraged to simulate the system or produce concrete parts of the system. Any change on the model can more easily be integrated during the system's life-cycle.
- **Agile tooling**: based on a meta-level (the Domain Model), modeling tools can benefit from powerful reflective mechanisms that facilitate its adaptation to changing contexts.

To implement modeling tools based on a given Domain Model, the Eclipse platform offers the Eclipse Modeling Framework (EMF). Using this framework, tool makers can build modeling workbenches offering editors for creating or visualizing models, and tools to transform, store, compare these models.

To facilitate the creation of such domain-specific modeling workbenches, EMF is completed by a rich eco-system of compliant Model-Driven technologies:

- **Xtext**: creation of textual editors
- **Sirius**: creation of graphical editors
- **Acceleo and Xtend**: code generation
- **CDO**: storage in a database
- **EMF Compare**: model comparison
- Etc.

With Eclipse Modeling technologies, tool makers have the choice to create the workbench that best suits the needs of their users. That being said, selecting the best kind of editor for a specific need is not always easy in practice.

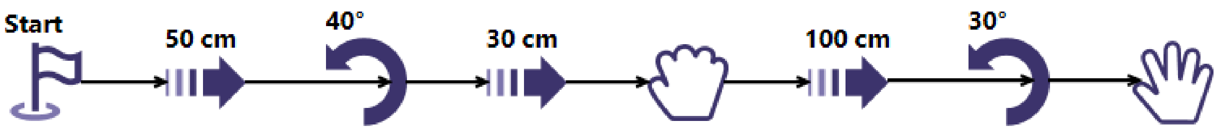
Benefits of graphical modeling

The well-known proverb "A picture is worth a thousand words" means that visualizing an object conveys its meaning more effectively than a long sentence does.

Indeed, we all know the efficiency of a map to describe a geographical area or path. Also, since the dawn of times architects have been using plans to describe the construction they have imagined and to drive the many different specialists building it. Graphical representation is already everywhere, through common things such as an organization chart which illustrates the company's hierarchy and shows the links between the group's members.

From a technical point of view, graphical modeling consists in representing a model through diagrams that visually represent the model elements and their relationships.

Diagrams are frequently composed of boxes and edges: a box representing a model element and an edge representing the relationship between two model elements. Boxes can use different shapes and colors, eventually icons, depending on their nature and properties. They can also contain lists or other boxes to represent sub-elements.



Graphical representation of a robot's choreography

Tools for graphical modeling often provide facilities to graphically create or modify elements: for instance, a new shape on the diagram automatically creates a new element in the model.

To facilitate the understanding of a model is considered the main benefit of visual modeling:

- **Distinguish concepts**
  Specific color, shape, icon or font associated to each kind of concept help the reader distinguish the nature of model elements at a glance.
- **View relationships**
  The relationships between elements can be identified directly. Their nature can also be represented graphically with a specific color or arrow style (plain, dot, dashed, end's decorator…).
- **Reveal hidden information**
  Some interesting information of the model elements are not always formally expressed. They can be computed from the properties or the relationships of the element. On a graphical representation, this informal information can be displayed: it is possible to show an indirect link between two elements or to change the rendering of an object depending on formulas (for example the combination of several properties, or the number of links of a given type).
- **Focus on objects' eco-system**
  But, to take advantage of these benefits you must avoid falling in the trap of the Spaghetti Nightmare! Don't put too much information on the same diagram for keeping your diagrams readable. A best practice when graphically modeling is to limit the size of a diagram to a letter size format by focusing on a model subset. Partial representations can be created to focus on one object and only visualize the other objects linked to it (directly or not).

But finally, all these benefits for the user, may be reduced to nothing if the creator of the diagrams spends too much time arranging the size and place of boxes and arrows. It is the reason why the graphical editors must also provide two essential kinds of tools:

- **Edition helpers**: these tools (drag & drop, contextual popups, direct edit, alignment guides, …) limit both the number of clicks required to graphically create or modify model elements and the number of back-and-forth movements between the canvas and the palette.
- **Automatic layouts**: graphical editors must also provide tools that automatically place elements on the canvas, depending on some visual strategies

Benefits of textual modeling

Since the early days programmers have used text to specify their programs. High-level programming languages raise the level of abstraction in a similar way models do, so you can use the well-established techniques from language engineering and compiler construction to specify models. This results in textual modeling languages.

```
Choregraphy Main {
    GoForward(50),
    Rotate(40),
    GoForward(30),
    Grab,
    GoForward(100),
    Rotate(50),
    Realease
}
```

Textual representation of a robot's choreography

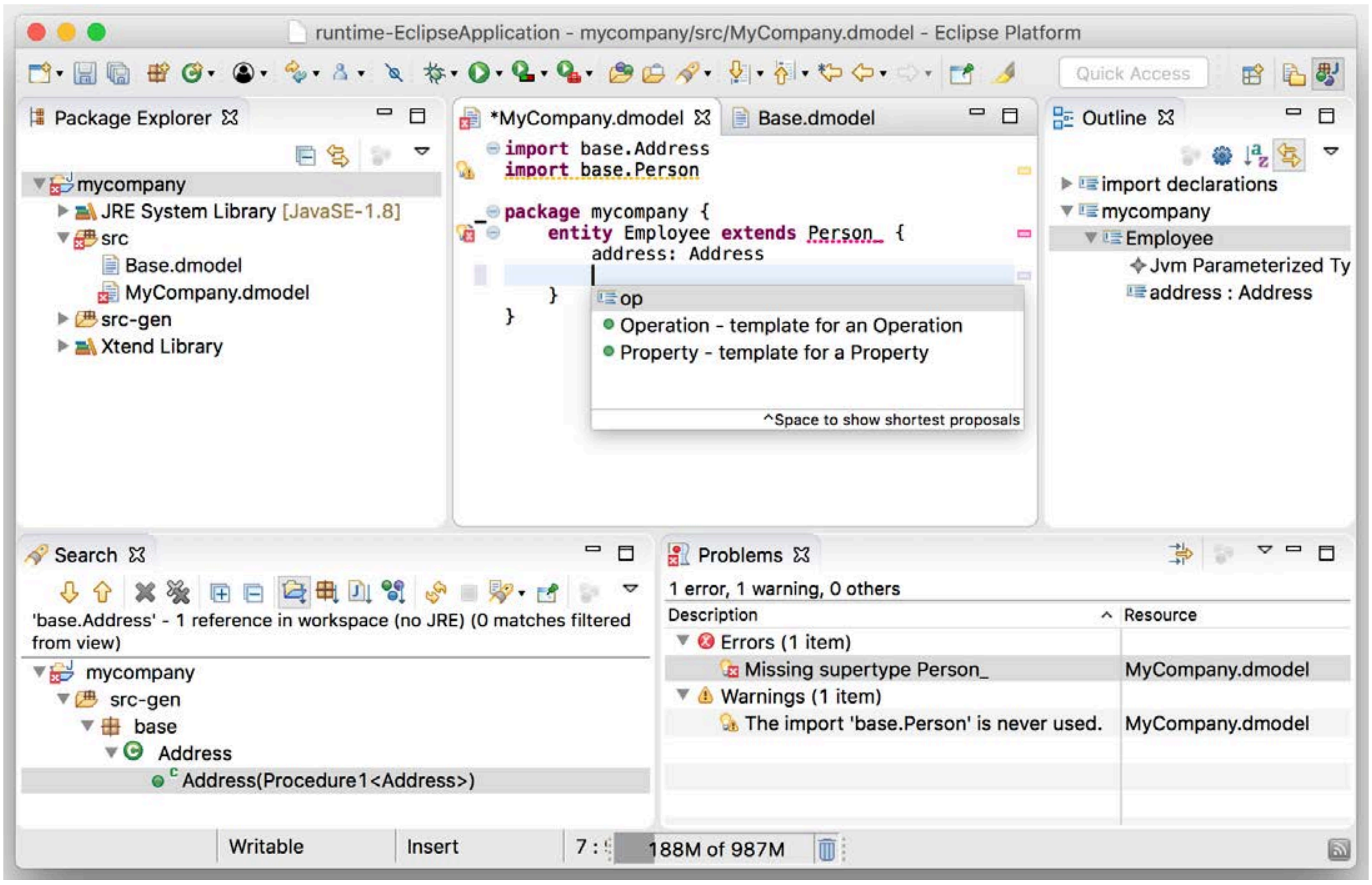
A textual modeling language is usually processed by a parser that transforms information expressed in textual format into a model. The parser bases its execution on the syntactic structure of a textual modeling language that is formalized in a grammar. A grammar defines keywords of a language, the nesting of its elements and the notation of their properties.

Textual models have a number of benefits as well

- **Convey lots of details**
  When it comes to elements with a lot of properties, text usually excels graphics. The same holds for structures that consist of a large number of very small parts, like arithmetic expressions or sequences of statements (code).
- **Completeness**
  A textual model usually specifies an element entirely in one place. While this may be bad for high-level views, this makes it easier to locate the definition of a certain low-level property.
- **Fast editing**
  When editing text, you don't need to switch between keyboard and mouse. Into the bargain, you will likely spend less time formatting your code that trimming the edges of your diagram.
- **Generic editors**
  You don't need a specific tool to create or modify textual models, neither to fix them when they are broken. For simple changes any generic text editor will do, and don't underestimate the value of a textual search and replace operation. For bigger tasks it is of course nicer to have some support for your modeling language in your editor.
- **Reuse of programming tools**
  Your textual models can be treated just like all other code artifacts in your development process. You can just reuse all tools of your programming stack, such as revision control, diff and merge, etc.

What is Xtext?

Xtext is an open-source framework for developing IDEs for textual programming languages. Just like in traditional parser generators, you define your language in a grammar, from which the code of the parser is generated. But in Xtext the grammar also describes how the parser should instantiate a model from the text. Where a traditional parser creates a generic syntax tree, an Xtext parser will create a strictly typed EMF model including cross-references between elements. This way, Xtext can be seen as a frontend to instantiate your Ecore model.



```
import base.Address
import base.Person

package mycompany {
    entity Employee extends Person_ {
        address: Address
        
    }
}
```

'base.Address' - 1 reference in workspace (no JRE) (0 matches filtered from view)

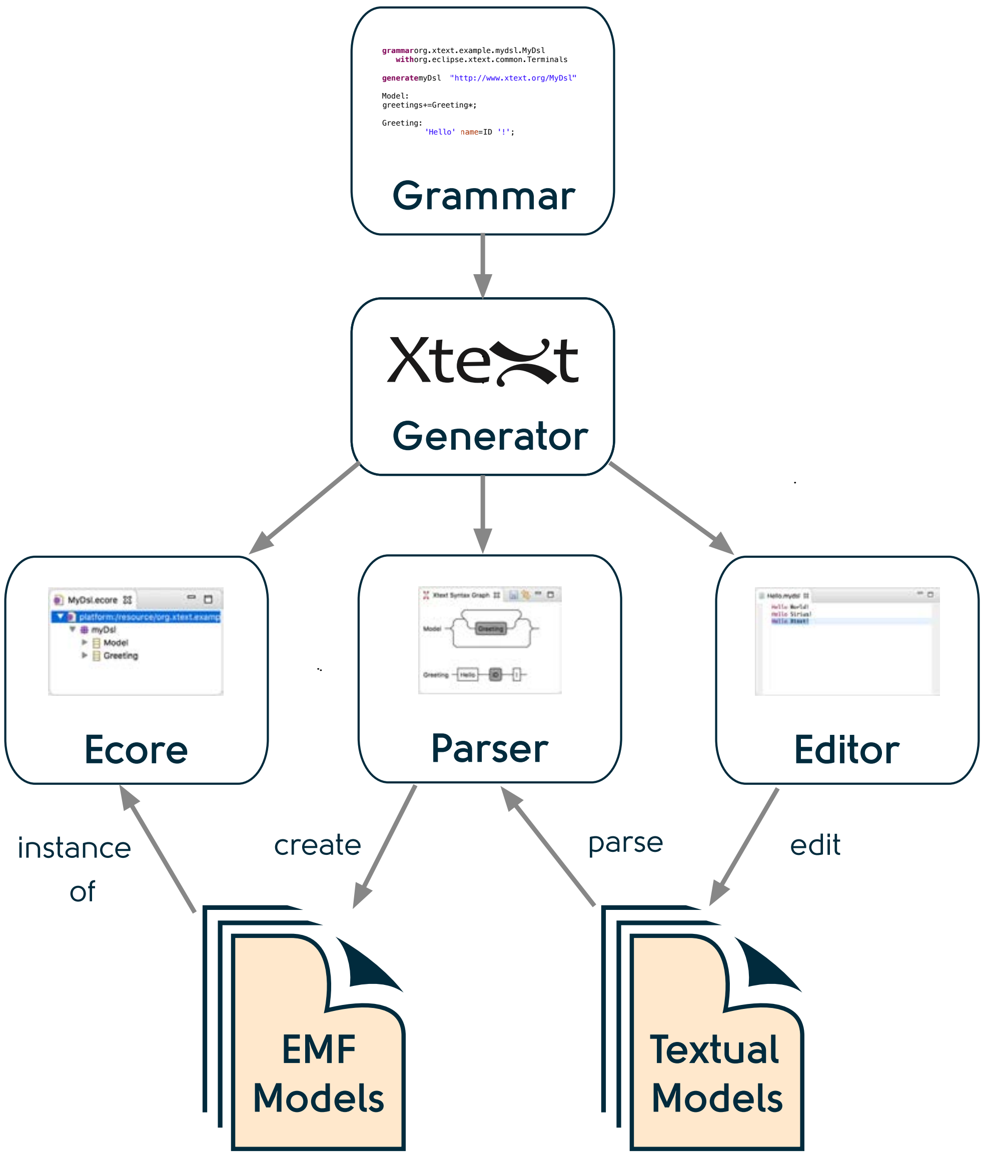
```
mycompany
  src-gen
    base
      Address
        Address(Procedure1<Address>)
```

1 error, 1 warning, 0 others

Description	Resource
Errors (1 item)	
Missing supertype Person_	MyCompany.dmodel
Warnings (1 item)	
The import 'base.Person' is never used.	MyCompany.dmodel

If you define your own language, there will not be too many people that are familiar with it. Having a language specific editor that guides users in creating valid models is of great value. Based on the grammar with the strict type information, Xtext generates the language specific smartness for various editors platforms, e.g. *Eclipse*, various *JavaScript* editor widgets or the *Language Server Protocol*. This includes live validation for syntactic and semantic errors, code completion, syntax coloring, cross-references and hyperlinks, hovers, code folding, find references, rename refactoring etc.

Defining your own language and editing models conforming to it is nice, but in the end you want something that is executable. For this purpose, Xtext allows you to easily hook in a code generator, that is automatically run when you save your models. The generator takes the EMF model as an input, traverses it and creates text on the fly. This text can be code in any other target programming language, e.g. Java or C. If the target language needs compilation, the Eclipse IDE can automatically compile it. This way, you can have your models, the generated code, and manually written code all within the same workbench. Code generators can be easily written in Xtend, a Java dialect that has also been implemented with Xtext.



```
grammarorg.xtext.example.mydsl.MyDsl
   withorg.eclipse.xtext.common.Terminals

generatemyDsl  "http://www.xtext.org/MyDsl"

Model:
greetings+=Greeting*;

Greeting:
          'Hello' name=ID '!';
```

You see that Xtext gives you a head start to create an IDE for your language. But what if you don't like the generic defaults? Xtext relies on dependency injection (DI) to wire up the infrastructure for your language. By changing the DI configuration, you can replace literally every component with your custom implementation. This will keep you satisfied in the long run, and allows for implementing real programming languages.

What is Sirius?

Eclipse Sirius main principles

The open source Eclipse Foundation project Sirius is a modeling tool which has been originally created in the context of complex systems modeling. The complexity of those systems makes necessary the collaboration of people with different concerns (different levels of analysis, roles...). Each of these corresponds to a different viewpoint on the same Domain Model that you can specify with Eclipse Sirius.

Sirius assumes that your Domain Model is defined using EMF. You can also easily define your own Domain-Specific Model (sometimes called Domain-Specific Language or metamodel) to more precisely suit your specific needs by defining concepts and their relations in the abstract.

Given such a Domain Model, Sirius allows to easily specify visual representations of these models: diagrams, tables, matrices (cross-tables) or hierarchies (trees). You can define as many of them as you need, some more detailed than others, some customized for certain tasks or roles (e.g. initial design, review, certification, etc.). These representations are not just static: with the appropriate tools (see below), compose a complete modeling environment where users can create, modify and validate their designs. When several stakeholders having particular set of concerns need to work on the same model, it is possible to group representations according to different viewpoints that end-users can enable or disable at will to work on certain aspects only (for example: cost, performance, etc).

Sirius allows to specify dedicated representations and associated tooling. As a specifier, you have to configure the generic Sirius platform to provide these representations to your users. This is done by creating and configuring a *Viewpoint Specification Model* which describes the structure, appearance and behavior of your modelers.

The main concepts you will work with when using Sirius are:

- **(a) Domain Model (Ecore)** An Ecore model is used to specify the model of a specific domain, in terms of classes, attributes and their relationships.
- **(b) Viewpoint** A viewpoint is one of the core elements that a specifier defines: it is a logical set of representation specifications and representation extension specifications.
- **(d) Representation** A representation is a projection representing model instances. It has a definition which describes the structure, appearance and behavior of your editors. By default, Sirius supports four kinds of representations (called dialects): diagrams, tables, matrices (aka cross-tables), and trees.
- **(c) Model Instances** The instances of domain specific concepts which are described by the Domain Model.

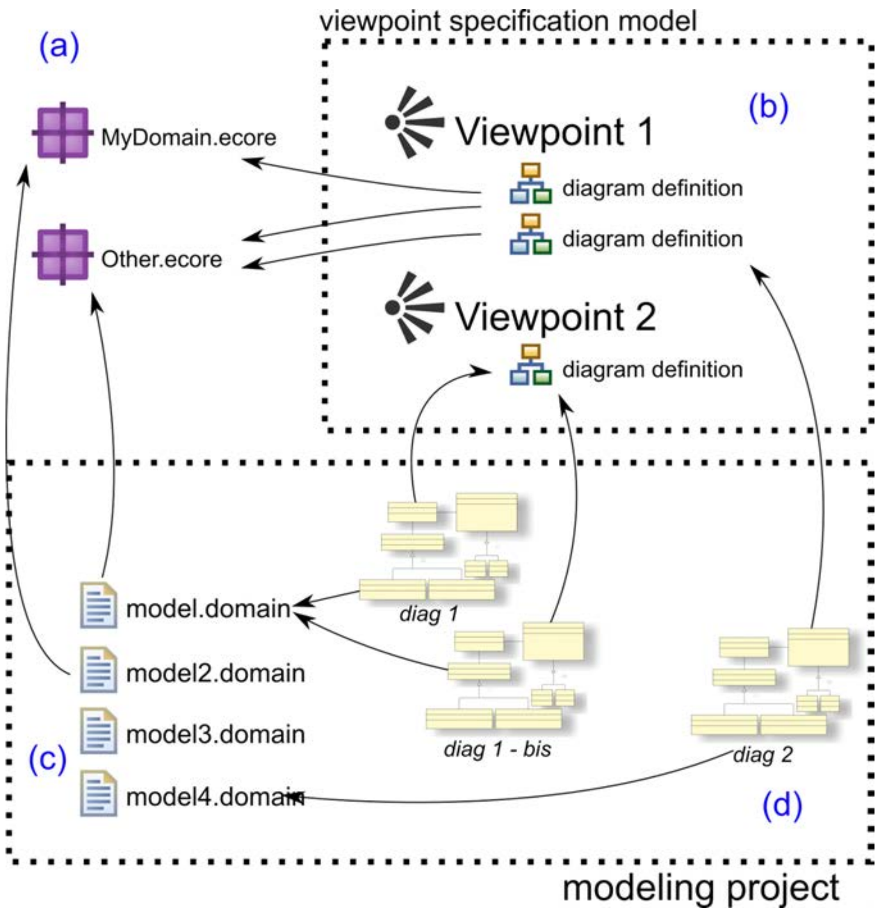


Diagram editors can be specified by configuring the Viewpoint Specification Model. Sirius is able to execute this specification **without relying on any code generation** and will instantiate the diagram editor: any change to the Viewpoint Specification Model is instantly reflected in the corresponding editor allowing for very fast testing and short feedback loops.

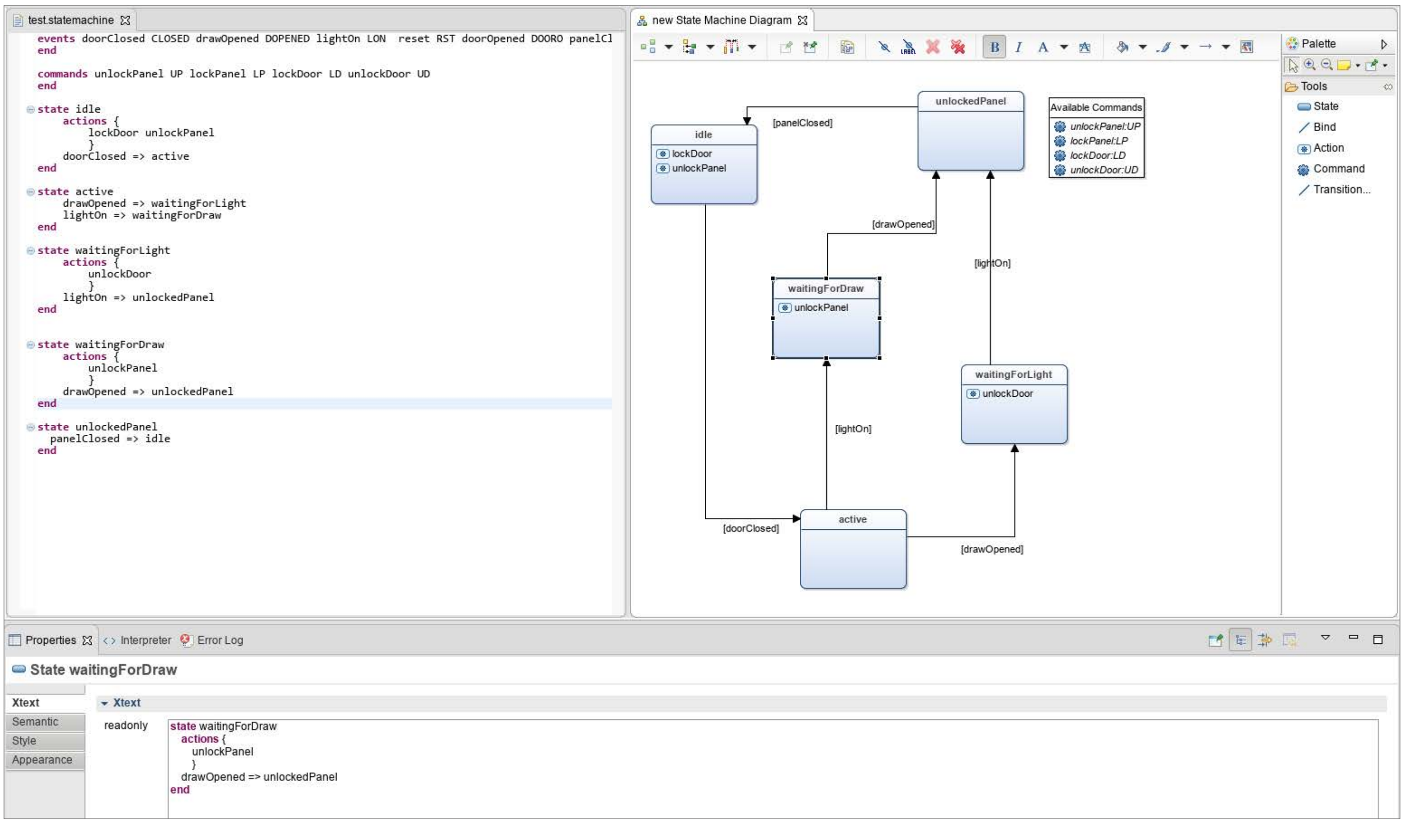
Xtext & Sirius in action

Rather than opting for only one approach, textual or graphical, it is possible to get both by integrating Xtext and Sirius.

There are two main possibilities. The first one consists in implementing two kinds of editors on the same model, while the second one consists in providing textual edition inside graphical editors.

Case 1: Editing the same models both graphically and textually

A diagram can be built on top of the data kept in text files by using Sirius and Xtext. In that situation you get a graphical representation of the very same information and you can edit it either using the diagram or the text editor. The synchronization between the two representations happens when an editor is saved.



```
events doorClosed CLOSED drawOpened DOPENED lightOn LON  reset RST doorOpened DOORO panelCl
end

commands unlockPanel UP lockPanel LP lockDoor LD unlockDoor UD
end

state idle
    actions {
        lockDoor unlockPanel
        }
    doorClosed => active
end

state active
    drawOpened => waitingForLight
    lightOn => waitingForDraw
end

state waitingForLight
    actions {
        unlockDoor
        }
    lightOn => unlockedPanel
end


state waitingForDraw
    actions {
        unlockPanel
        }
    drawOpened => unlockedPanel
end

state unlockedPanel
  panelClosed => idle
end
```

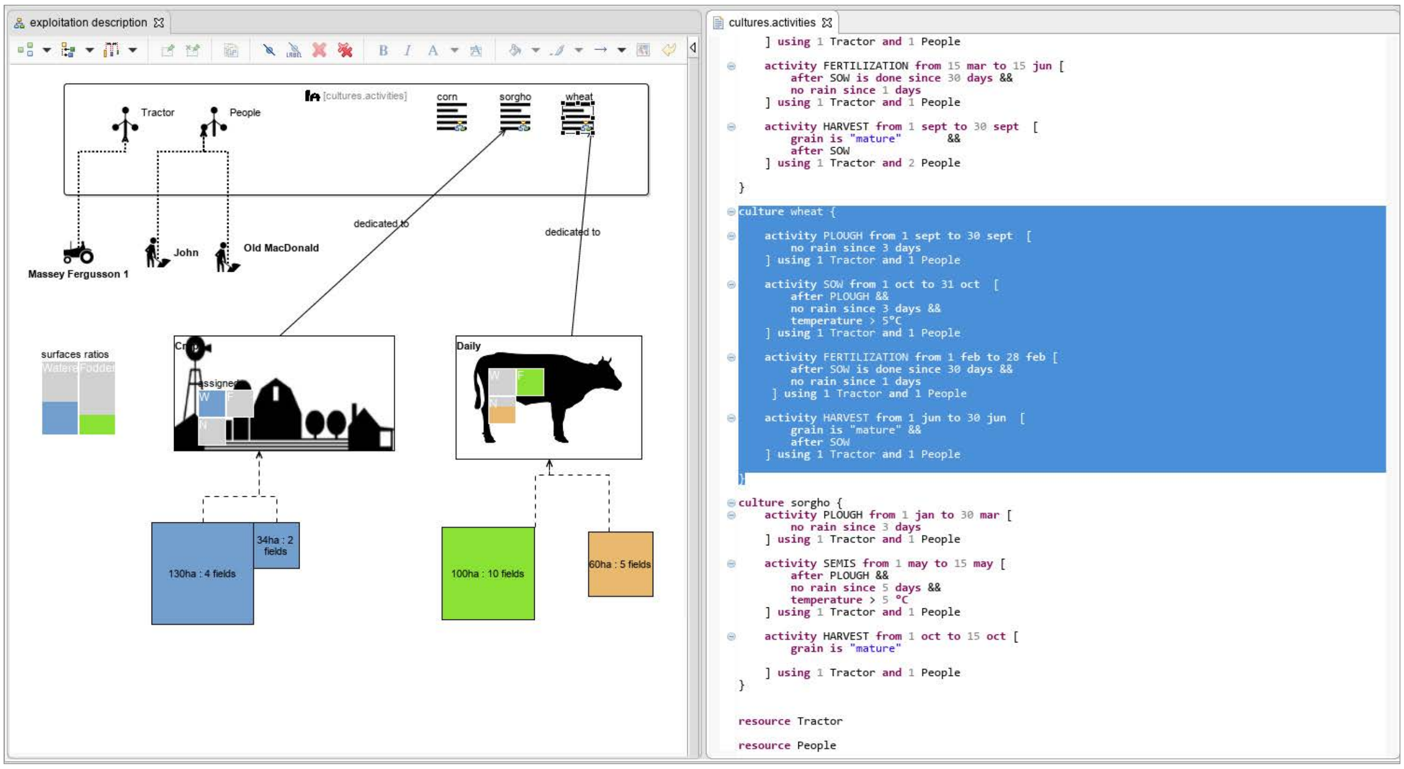
```
state waitingForDraw
  actions {
    unlockPanel
  }
  drawOpened => unlockedPanel
end
```

In the example above, you can connect two state boxes of the state machine diagram on the right with a transition connection. Hitting “Save”, updates the textual representation by adding the new transition.

The other direction also works: if a user adds a new state definition through the text editor and saves it, then the diagram is refreshed and it displays the corresponding new shape.

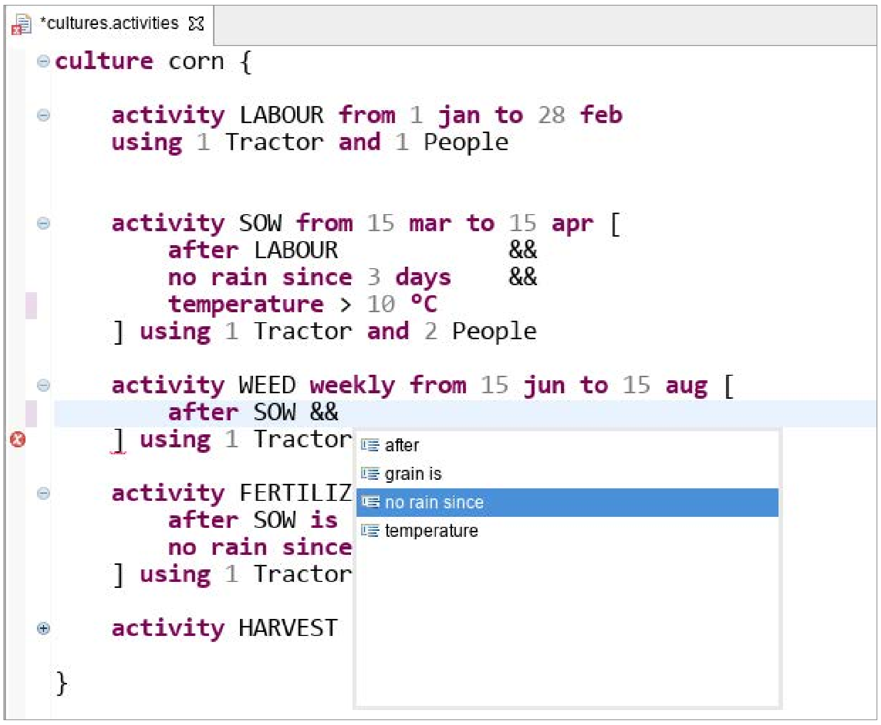
Another fairly common situation in using both the textual and the graphical syntax for what they are best at and from one to another.

In the next example a set of domain specific languages is built to analyze and assess farming activities. The goal is to optimize the water consumption used to irrigate a given exploitation by simulating what would happen if some fields are assigned to another type of culture: from the corn culture which tend to be water demanding to the sorgho culture which requires less water but might grow differently depending on the weather.



```
    ] using 1 Tractor and 1 People

    activity FERTILIZATION from 15 mar to 15 jun [
        after SOW is done since 30 days &&
        no rain since 1 days
    ] using 1 Tractor and 1 People

    activity HARVEST from 1 sept to 30 sept  [
        grain is "mature"       &&
        after SOW
    ] using 1 Tractor and 2 People

}
culture wheat {

    activity PLOUGH from 1 sept to 30 sept  [
        no rain since 3 days
    ] using 1 Tractor and 1 People

    activity SOW from 1 oct to 31 oct  [
        after PLOUGH &&
        no rain since 3 days &&
        temperature > 5°C
    ] using 1 Tractor and 1 People

    activity FERTILIZATION from 1 feb to 28 feb [
        after SOW is done since 30 days &&
        no rain since 1 days
     ] using 1 Tractor and 1 People

    activity HARVEST from 1 jun to 30 jun  [
        grain is "mature" &&
        after SOW
    ] using 1 Tractor and 1 People

}
culture sorgho {
    activity PLOUGH from 1 jan to 30 mar [
        no rain since 3 days
    ] using 1 Tractor and 1 People

    activity SEMIS from 1 may to 15 may [
        after PLOUGH &&
        no rain since 5 days &&
        temperature > 5 °C
    ] using 1 Tractor and 1 People

    activity HARVEST from 1 oct to 15 oct [
        grain is "mature"

    ] using 1 Tractor and 1 People
}

resource Tractor

resource People
```

A given culture is defined using a textual DSL stating all the activities which are required, to grow corn for instance. Each of those activities has specific constraints which have to be met to get started during the year: a

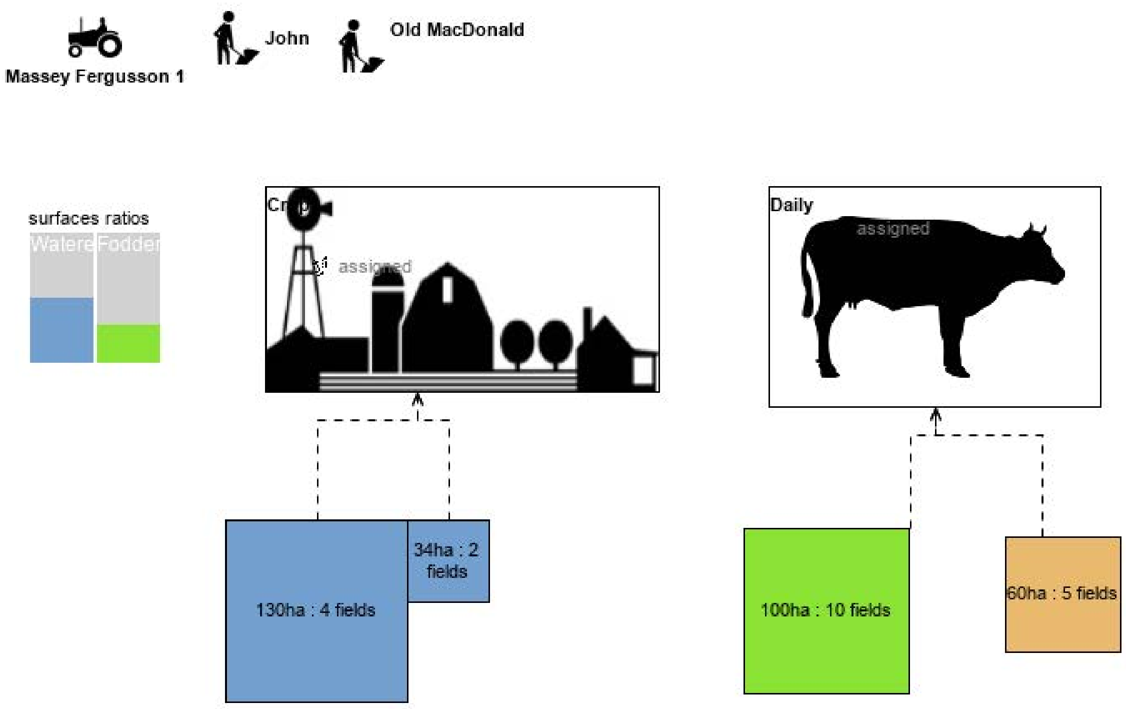

*cultures.activities

```
culture corn {

    activity LABOUR from 1 jan to 28 feb
    using 1 Tractor and 1 People


    activity SOW from 15 mar to 15 apr [
        after LABOUR            &&
        no rain since 3 days    &&
        temperature > 10 °C
    ] using 1 Tractor and 2 People

    activity WEED weekly from 15 jun to 15 aug [
        after SOW &&
    ] using 1 Tractor

    activity FERTILIZ
        after SOW is
        no rain since
    ] using 1 Tractor

    activity HARVEST

}
```

range of possible dates, a specific temperature or a rain requirement.

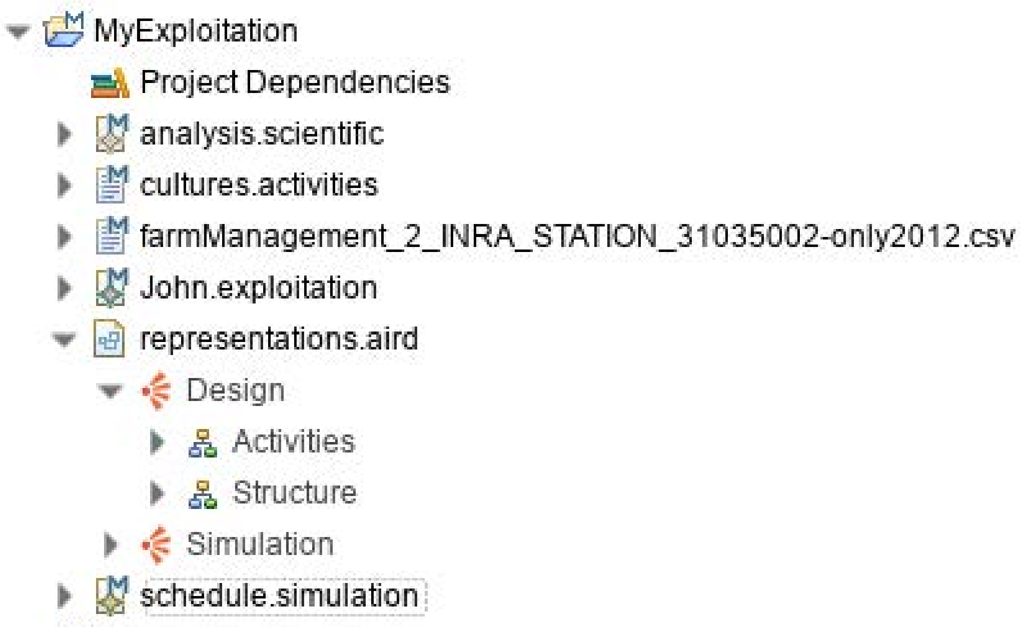
The textual syntax is especially powerful here: when composing the constraints, all the details are visible and can be reworked with great flexibility.

On the other hand the exploitation definition is done through a graphical syntax. For maximum suggestiveness

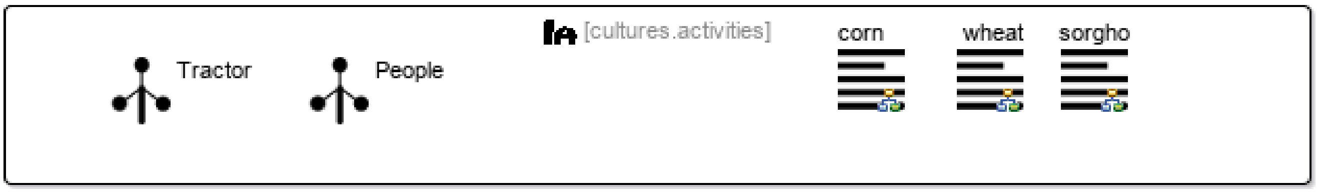


the size and colors of the shapes are bound to the actual field size and the fact that those fields are irrigated or not.

One can see at a glance that the exploitation is split in two groups, the crops culture and the cattle farming and how the fields are affected by those groups.

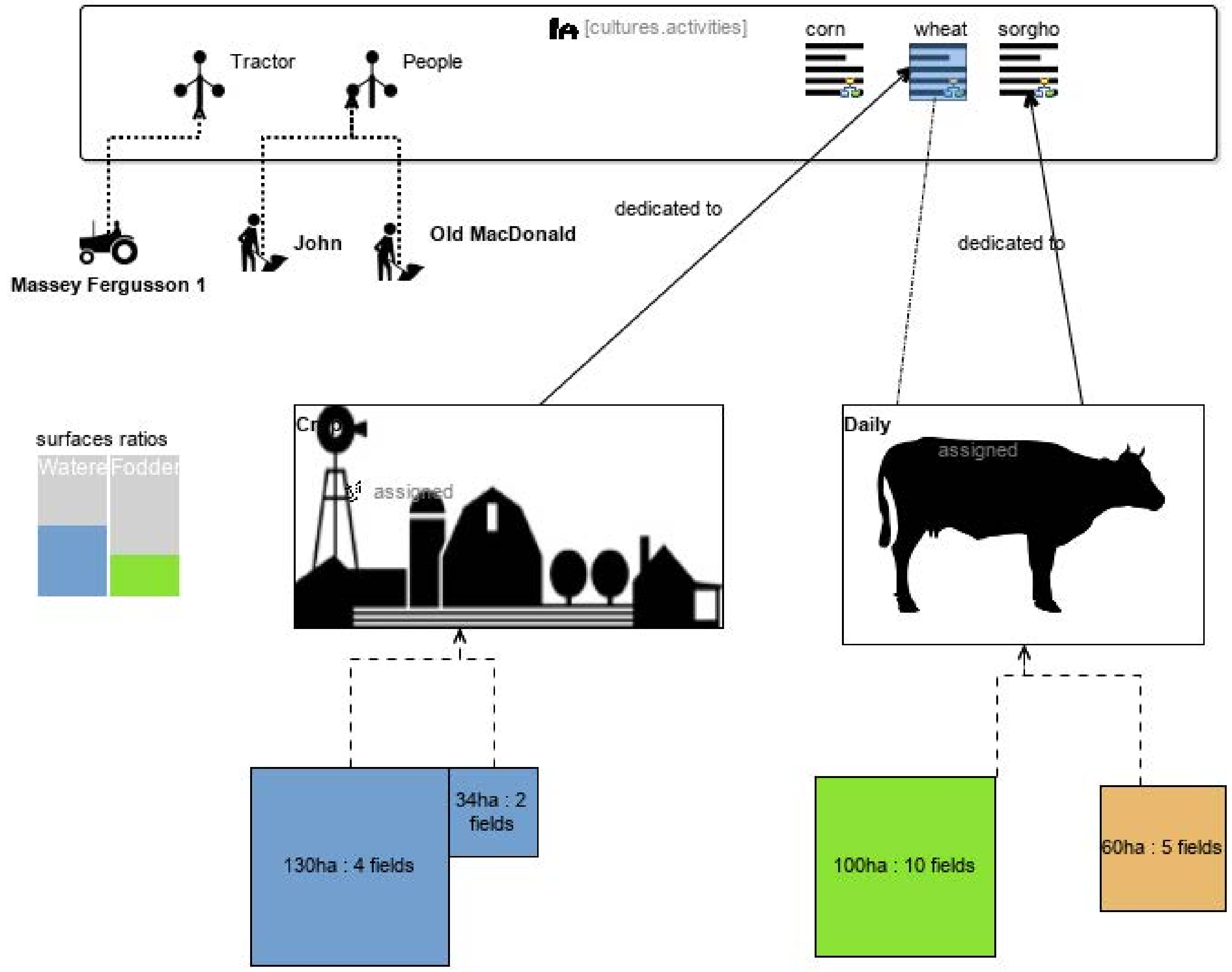

MyExploitation
Project Dependencies
analysis.scientific
cultures.activities
farmManagement_2_INRA_STATION_31035002-only2012.csv
John.exploitation
representations.aird
Design
Activities
Structure
Simulation
schedule.simulation

Sirius gives a lot of flexibility to structure your diagrams with respect to the model structure. In this case the model actually consists of five different files: three of those are written in the standard XMI format, The culture



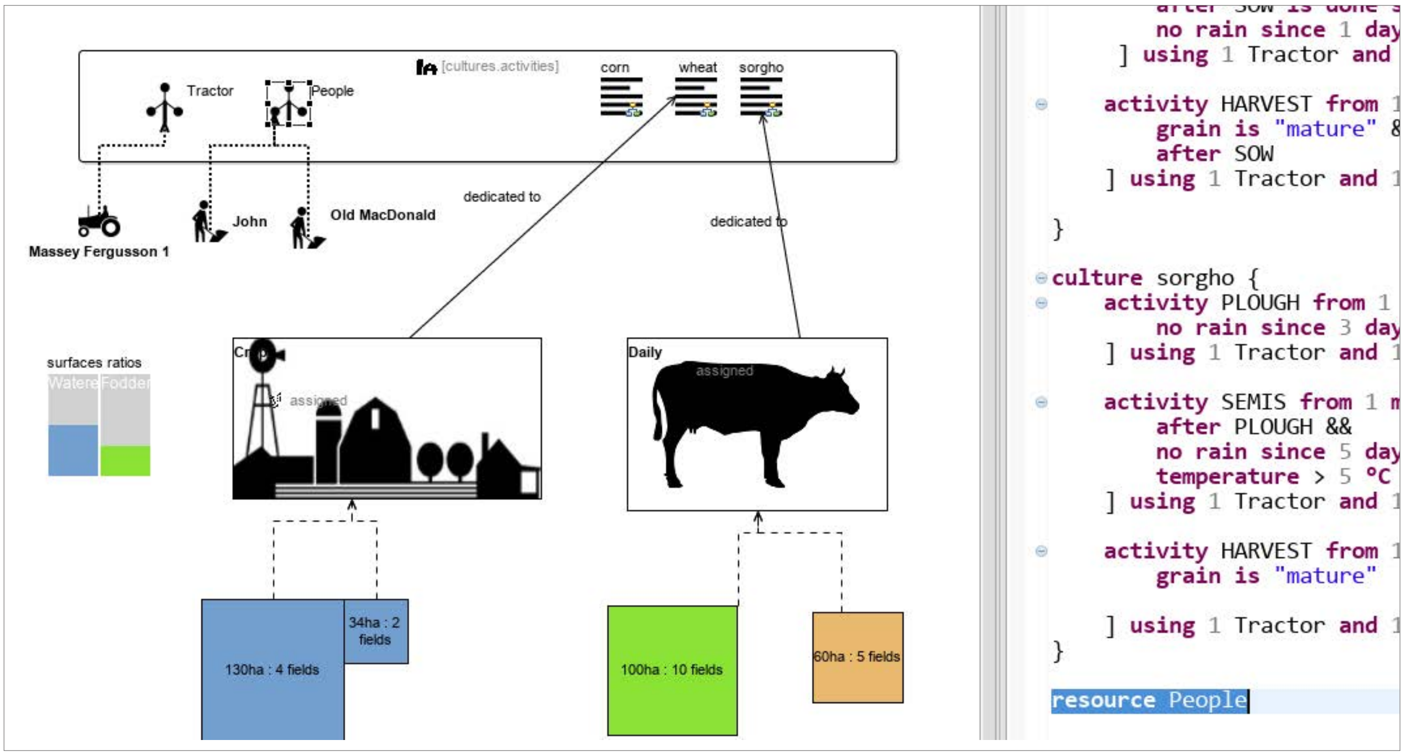
definition (culture.activities) is using Xtext and the weather data is a plain CSV file.

This flexibility is used in this case to display in the same diagram elements from distinct model files, here all the



culture definitions are displayed alongside the exploitation structure.

Thanks to this the end user can affect another culture to a given group and simulate how this will impact the water consumption.



```
            after SOW is done s
            no rain since 1 day
         ] using 1 Tractor and

        activity HARVEST from 1
            grain is "mature" &
            after SOW
        ] using 1 Tractor and 1

    }

    culture sorgho {
        activity PLOUGH from 1
            no rain since 3 day
        ] using 1 Tractor and 1

        activity SEMIS from 1 m
            after PLOUGH &&
            no rain since 5 day
            temperature > 5 °C
        ] using 1 Tractor and 1

        activity HARVEST from 1
            grain is "mature"

        ] using 1 Tractor and 1
    }

    resource People
```

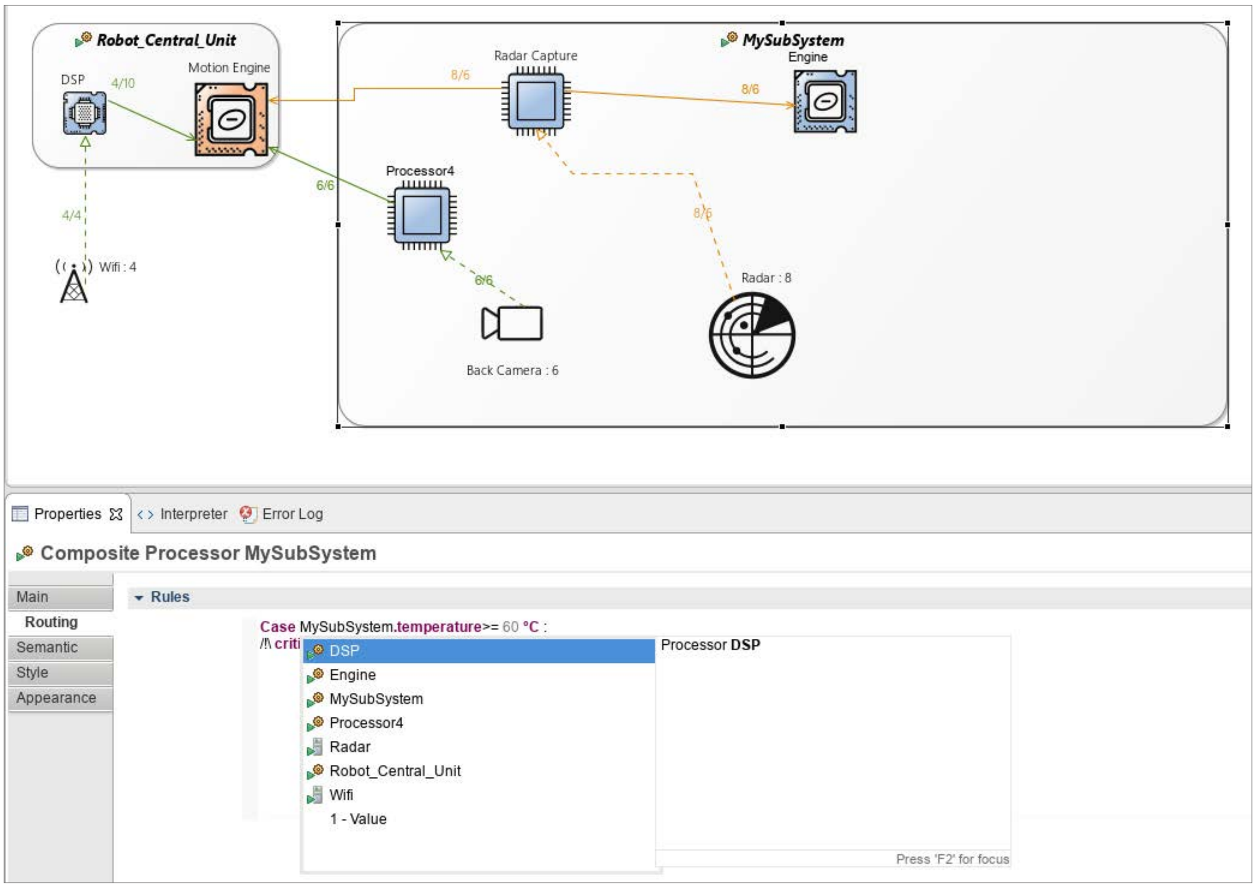
In cases like this, it is interesting to navigate from one editor to the other: double-clicking in the diagram on one of the elements originating in the textual model automatically opens the textual editor and highlights the corresponding passage. The other direction, to navigate from the text of an element to its representation in the diagram might be implemented.

The way model references are persisted plays a key role here. We must guarantee that references and model to shape correspondences are kept intact, e.g. when elements are reordered in the Xtext editor. By participating in refactoring operations we can make sure that all affected models are automatically updated when a culture definition is renamed.

Case 2: Embedding an Xtext Editor into Sirius

In this example we embed an Xtext editor in the property view of a Sirius based modeler.

The model describes systems which have to process input data from sensors and operate within some safety and performance constraints.



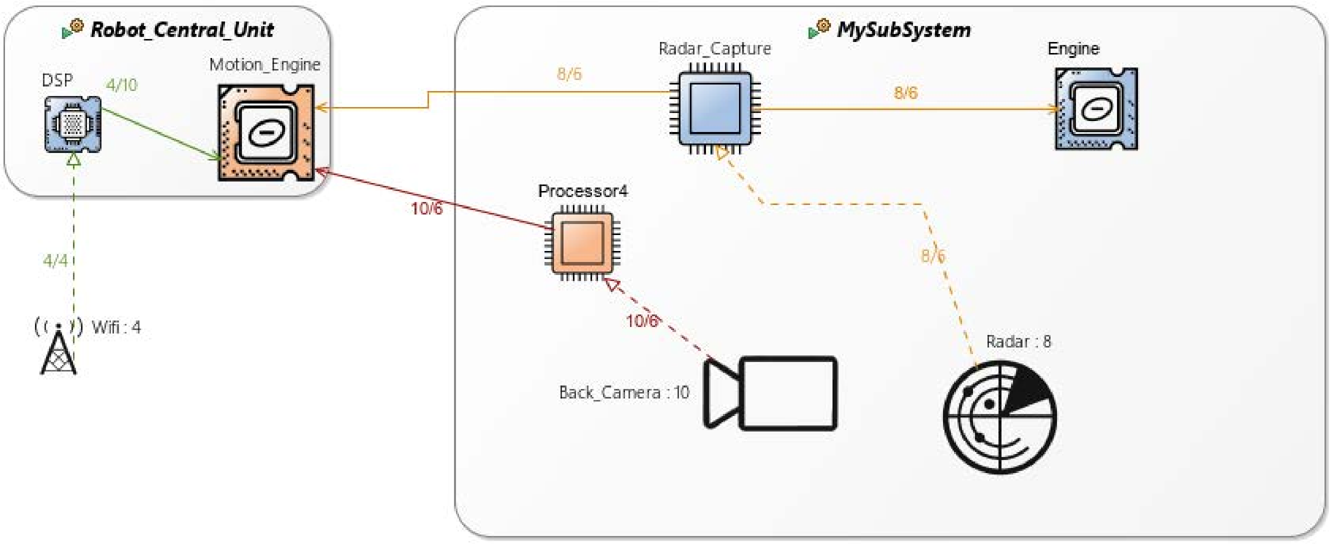
Composite Processor MySubSystem

```
Case MySubSystem.temperature>= 60 °C :
/!\ criti
```

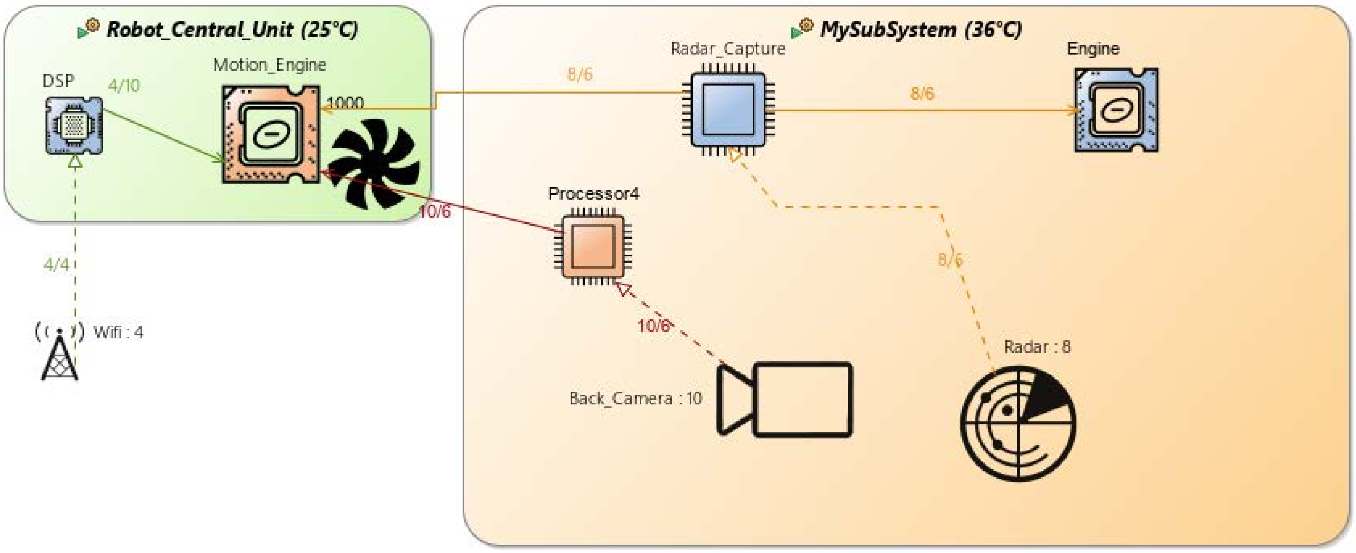
- DSP
- Engine
- MySubSystem
- Processor4
- Radar
- Robot_Central_Unit
- Wifi
- 1 - Value

Processor **DSP**

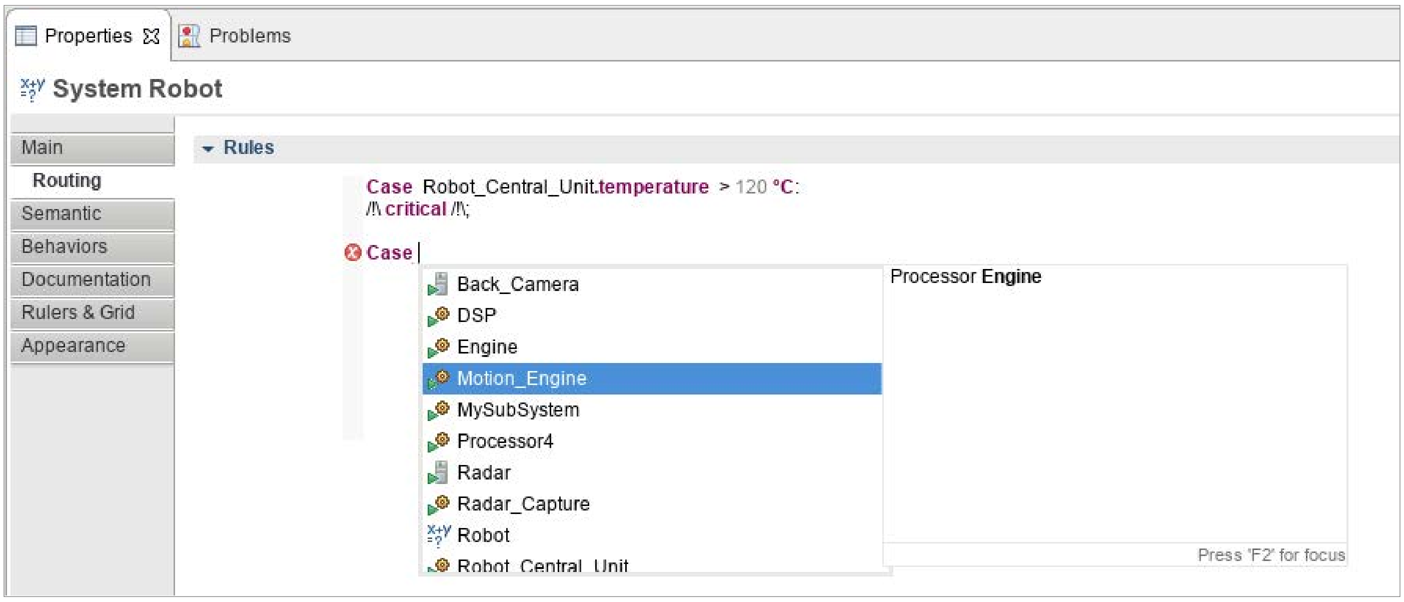
The graphical view highlights the usage of components and connections by means of dynamic colors.



By enabling the "Temperature" layer the analyst can check the corresponding constraints.



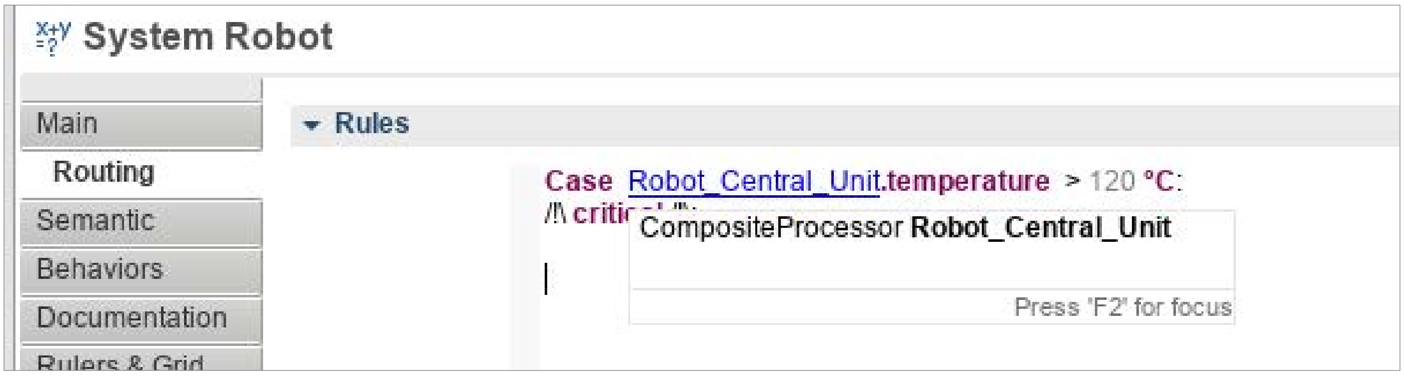
A textual DSL complements this tool. It describes the rules which might reconfigure the system under certain circumstances:



```
Case  Robot_Central_Unit.temperature  > 120 °C:
/!\ critical /!\;

Case |
```

The textual editor is completely embedded in the Sirius modeler and uses the in-memory state of the model to provide content assist. The system rules are just stored in a string valued attribute of the model, which itself is persisted in XMI.

In this use-case, Xtext is used to provide an advanced text widget with code completion and error validation directly within the context of the properties editor. In this setup it is also a good practice to provide navigation from the Properties editor to the corresponding element in a diagram. The end user can use *CTRL+click* to reveal the element in the diagram.



```
Case Robot_Central_Unit.temperature > 120 °C:
/!\ critic
```

CompositeProcessor **Robot_Central_Unit**

This second use-case requires a pretty deep integration of Sirius and Xtext. Even though the resulting code is quite small, deep knowledge of the core mechanisms of both technologies is necessary in order to synchronize the unsaved states of the Xtext and Sirius editors and their model change transactions. It is also worth noting that the refactoring capabilities in this case will be limited, as such renaming "Robot_Central_Unit" in the example will lead to validation errors in the textual parts to be fixed by the analyst.

How may we help you?

Now that you know how to integrate Sirius and Xtext and benefit from both graphical and textual approaches, it’s up to you to create the modeling workbench of your dreams.

We have explained the benefits of such integration, but also the main challenges you need to address to avoid some common pitfalls. Our main advice is that you should take time identifying which parts of your models needs which kind of editor (graphical, textual or both?), to serve which use-case. Then you will be able to identify the kind of integration which matches these needs best and anticipate the potential trade-offs you will have to balance.

In any case, you can count on Obeo and TypeFox to help you in this integration. We both have cutting-edge experts who are strongly involved in Sirius and Xtext development. They have all the software engineering skills required to implement efficient and maintainable domain-specific workbenches in Eclipse.

They can advise you from the definition of your Domain Model to the development of your editors and complementary add-ons, including the integration of all your plug-ins into an efficient and maintainable environment.

And don’t forget, Sirius and Xtext are open source technologies, so that you can easily influence the roadmap! Some features you need are not yet implemented? Our committers working on these projects can help you specify the missing features and directly contribute to the source code if applicable.

Unleash the power and openness of Sirius and Xtext and come on-board the Eclipse Modeling community!

obeo.fr/fr/contact
Twitter: @Obeo_corp

typefox.io/contact
Twitter: @typefox_io