POLITECNICO DI MILANO

DIPARTIMENTO DI
ELETTRONICA,
INFORMAZIONE
E BIOINGEGNERIA

How to cook an Agile Web Based Model Driven Environment in a night

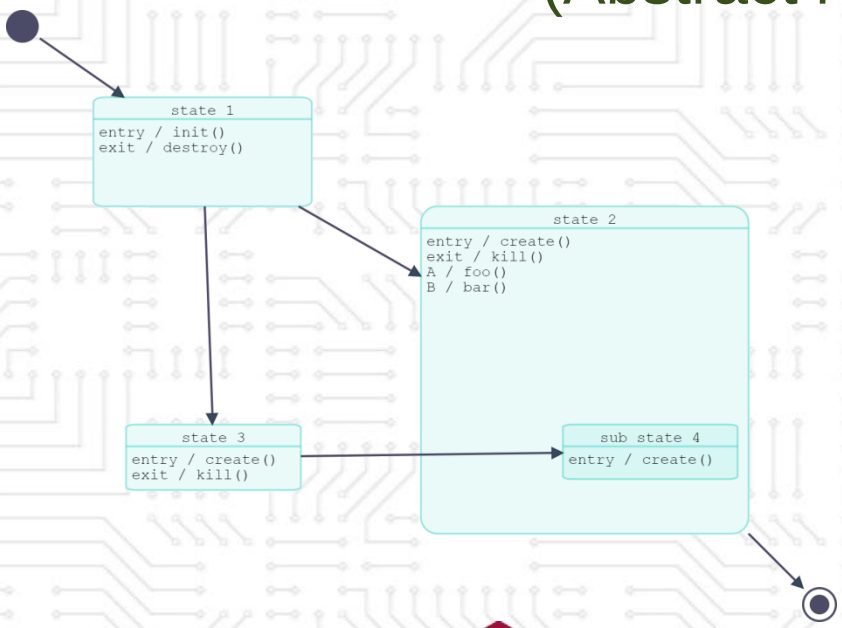
Carlo Bernaschina

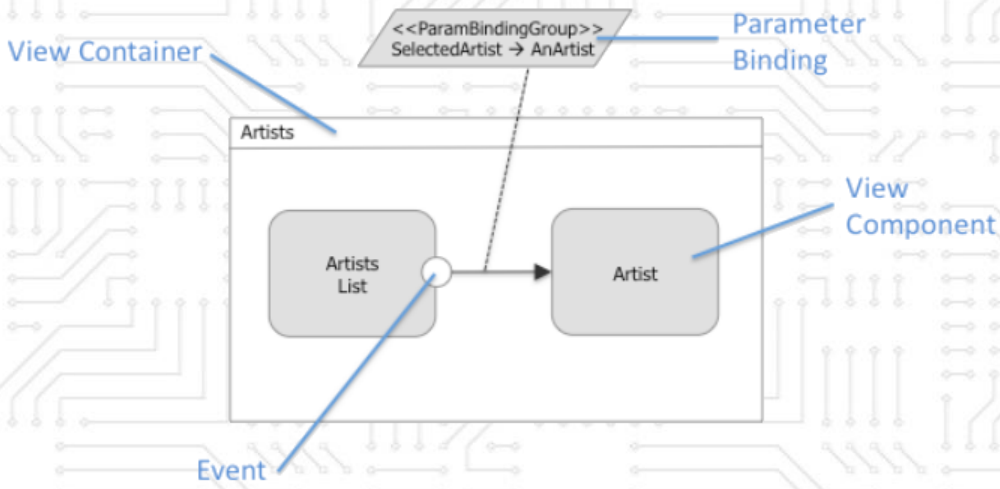
Cáceres, June 7, 2018

Model Driven Development

MDD is the branch of software engineering that advocates the use of models.

(Abstract representations of a System)



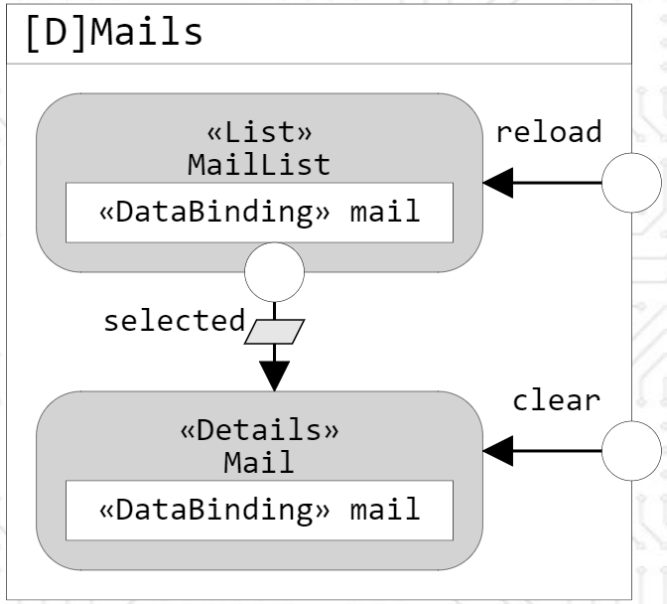


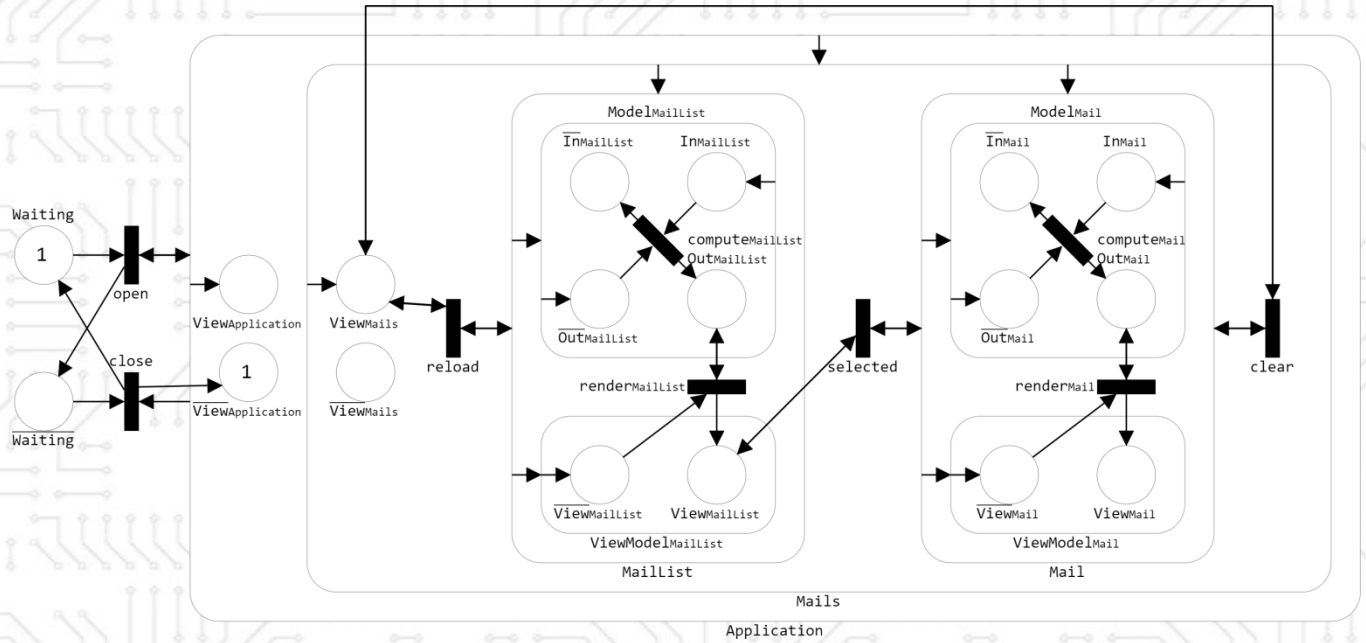
UNIFIED MODELING LANGUAGE ™

IFML ™

Model Transformations (M2M)

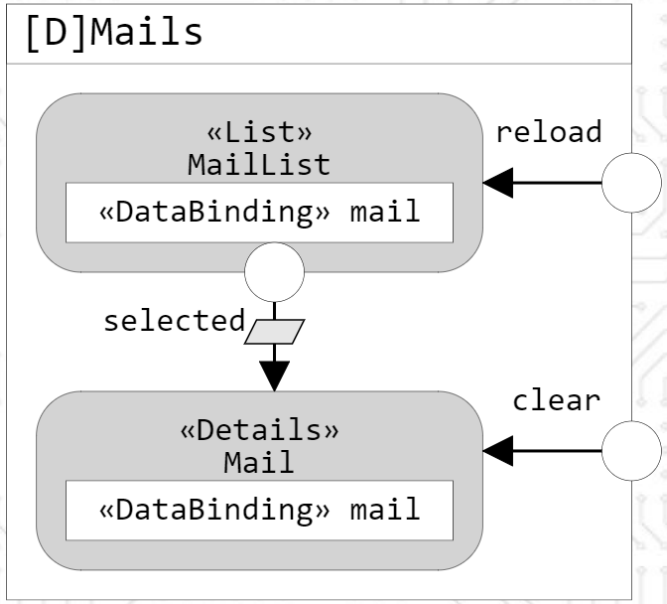
Model Transformations are a Key ingredient of MDD.
Iteratively adding details.





Model Transformations (M2T)

Model Transformations are a Key ingredient of MDD.
Generating final artifacts.




```
/* jslint node: true */
'use strict";

var Promise = require('bluebird');

exports.createInitializer = function (options) {
    var repository = options.repositories['mail'];

    function ViewModel(context, req) {
        if (!(this instanceof ViewModel)) { return new ViewModel(context, req); }
        this.context = context;
        this.filters = {};
        if (req.query[this.id]) {
            this.selected = req.query[this.id];
        }
        this.status = 'ready';
    };

    ViewModel.prototype.id = 'maillist';
    ViewModel.prototype.fields = {
        id: 1
        ,subject: 1
    };

    ViewModel.prototype.compute = function () {
        if (this.computing) { return this.computing; }
        var self = this;
        function compute() {
            return repository.find(self.filters, self.fields)
                .then(function (items) {
                    self.items = items;
```

How to build an MDD Environment

- Eclipse Modeling Framework (EMF)
  - ALMOsT.js
- Custom tools

Eclipse Modeling Framework (EMF)

- **Defining a Meta-Model**
  - Ecore www.eclipse.org/ecoretools/
  - MOF
  - …
- **Graphical Representation**
  - GMF www.eclipse.org/modeling/gmp/
  - EuGENia www.eclipse.org/epsilon/doc/eugenia/
  - …
- **Textual Representation**
  - Xtext https://www.eclipse.org/Xtext/
  - …
- **Model to Model**
  - ATL www.eclipse.org/atl/
  - QVT wiki.eclipse.org/M2M/QVTO
  - ETL www.eclipse.org/epsilon/
  - …
- **Model to Text**
  - Acceleo www.eclipse.org/acceleo/
  - …

https://www.eclipse.org/modeling/emf/

Model Transformation Languages

We have a plethora of transformation languages, that can be organized as follow:

- Declarative
  EMF Henshin
- Imperative
  Kermeta
- Hybrid
  ATLAS Transformation Language (ATL)

They all require specific tools and environment, making them not easy to integratee inside other tools.

So…

Easy, Right?

D.W.T.F.Y.W.

Do With The Framework You Want

Pros and Cons

EMF

Pros

- Standard Languages
- Documentation
- Interoperability

Cons

- Steep learning curve
- (personal) Is it actually future proof?
- (personal) Eclipse bounded

Custom Tools

Pros

- Tailored to your needs
- Full control
- Reuse preexisting knowledge

Cons

- High development costs (time)

Agile Software Development

Agile Software Development is an incremental and iterative approach based on principles that aim at increasing productivity and adherence to requirement, while keeping the process as lightweight as possible.

- Extreme Programming
  Test Driven Development
- SCRUM

Agile Model Driven Development (AMDD)

The majority of the attempts to use apply Agile techniques to Model Driven Development focus on the mapping of the development process.

- Incremental & Iterative Development
  support for incomplete models
- Test Driven Development
- SCRUM
  mapping MDD development steps to the SCRUM workflow

What about tools/meta-models?

We need to integrate tool into the loop.

Tools need to co-evolve iteratively with the models in order to support new functionalities/requirements that were not foreseen during the initial phases.

Requirements

1. No installation
2. No new language
3. Fast start-up
4. Parallel development
5. Customized output
6. Customized generation

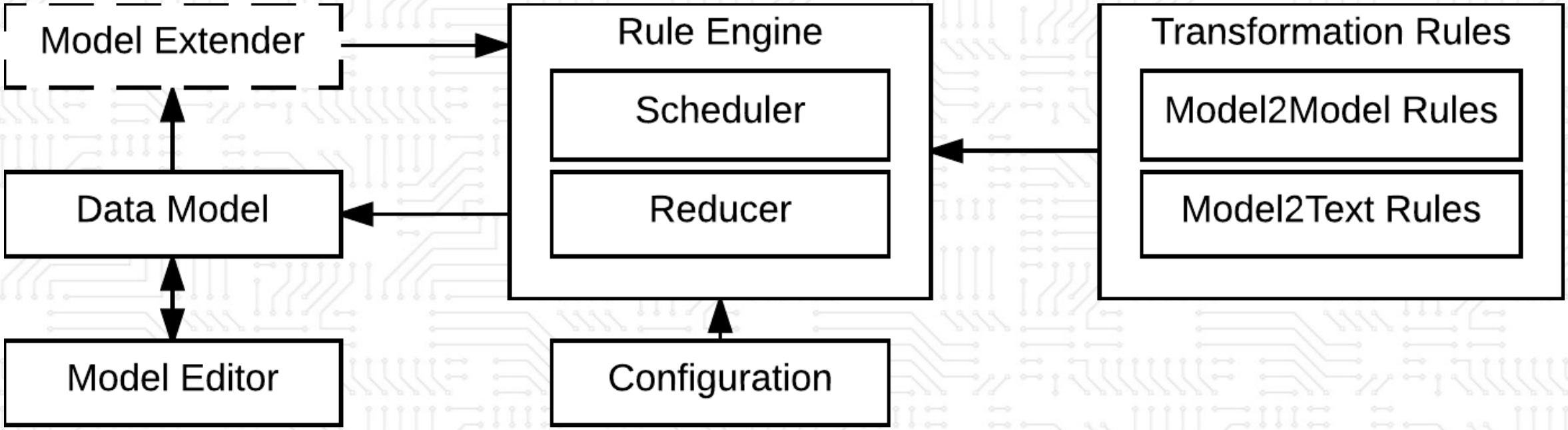
ALMOsT.js

AgiLe MOdel Transformations

https://www.npmjs.com/package/almost

What you need to know is ALMOsT JavaScript

Architecture



The Model

The model must be an object with two array properties (**elements**, **relations**)

```
{
    "elements": [],
    "relations": []
}
```

The Rule (Model)

Rules are pairs of plain functions.

(**condition** & **action**)

```
createRule(
    // Condition function
    function (model) { return model.elements.length > 0; },
    // Action function
    function (model) {
        return {
            project: { type: "folder", name: "myProject" }
        };
    }
);
```

The Rule (element)

Rules are pairs of plain functions.

(Condition & Action)

```
createRule(
    function (element, model) {
        return element.type === "ifml.ViewContainer";
    },
    function (element, model) {
        return {
            elements: [
                { id: element.id, type: "pcn.PlaceChart",
                    attributes: {name: element.name} }
            ],
            relations: []
        };
    }
);
```

The Rule (relation)

Rules are pairs of plain functions.

(Condition & Action)

```
createRule(
    function (relation, model) {
        return relation.type === "hierarchy";
    },
    function (relation, model) {
        var id = relation.child + "-init";
        return {
            elements: [
                { id: id, type: "pcn.Transition", attributes: {} }
            ],
            relations: [
                { type: "source",
                  transition: id, source: relation.parent },
                { type: "target",
                  transition: id, target: relation.child },
            ]
        };
    }
);
```

The Reducer

All the results of the rules are merged following a custom reduction policies.

ALMOsT.js has two predefined reduction policies:

- Model2Model
  The results of the rules must be partial models
- Model2Text
  The results of the rules must objects in which every attribute describes a file or a folder in the generated filesystem.

Reducer (Model2Model)

In a Model2Model transformation each rule must export a partial model. They will be reduced my concatenating **elements** and **relations**.

```
createRule(
    function (element, model) {
        return element.type === "ifml.ViewContainer";
    },
    function (element, model) {
        return {
            elements: [
                { id: element.id, type: "pcn.PlaceChart",
                    attributes: {name: element.name} }
            ],
            relations: []
        };
    }
);
```

Reducer (Model2Text)

In a Model2Text transformation each rule must export an object where each attribute is a descriptor for a file or a folder.

Mandatory properties are **type** and **name**. If an **isFolder** property is found it will be considered as a folder and the **children** properties will be concatenated.

```
createRule(
    // Condition function
    function (model) { return model.elements.length > 0; },
    // Action function
    function (model) {
        return {
            project: { type: "folder", name: "myProject" }
        };
    }
);
```

Usage (put everything together)

```
// Create a transformer
var transform = createTransformer(rules, 'm2m');

// Execute transformer;
var output_model = transform(input_model);
```

What About Meta-Models?

No explicit definition of Meta-Model is present in ALMOsT.js

There is though a suggested element and relations structure.

```
{
    "id": "mails",
    "type": "ifml.ViewContainer",
    "attributes": {
        "name": "Mails",
        "landmark": true,
    },
    "metadata": {
        "graphics": {
            "position": { "x": 100, "y": 50},
            "size": { "width": 160, "height": 90}
        }
    }
}
```

```
{
    "type": "hierarchy",
    "parent": "mails",
    "child": "mails-list"
}
```

What About Meta-Models? (2)

Using the ALMOsT-Extend plugin it is possible to extend the input model with helper functions that can be used to simplify the graph navigation:

- **Id ⇔ Element Lookup**
  toElement(), toId()
- **Type Checking**
  isType(), isOtherType()
- **Relation Navigation**
  getChildren(), getParent()
- **Custom Walks**
  getDescendants(), getAncestors()

Running Example (Model2Model)

```
createRule(
    function (element, model) { // custom type checking function
        return model.isNMRelation(element);
    },
    function (relation, model) {
        var role1 = model.getRole1(relation), // first entity
            role2 = model.getRole2(relation), // second entity
            id = relation.id,
            // generate ids for relational elements
            id1 = id + '-ref-' + role1.id, // role1 column id
            id2 = id + '-ref-' + role2.id; // role2 column id
            return {
                elements: [
                    // create bridge table
                    { id: id, type: 'ER.Table',
                      attributes: {name: relation.attributes.name } },
                    // create column referencing the 1st role table
                    { id: id1, type: 'ER.Column',
                      attributes: {name: role1.attributes.name } },
                        // create  column referencing the 2nd role table
                    { id: id2, type: 'ER.Column',
                      attributes: {name: role2.attributes.name } },
                ],
                relations: [
                    // relate columns with table
                    {type: 'ER.ColumnOfTable', table: id, column: id1 },
                    {type: 'ER.ColumnOfTable', table: id, column: id2 },
                ]
            };
        }
    )
```

Running Example (Model2Text)

```
createRule(
    function (element, model) {
        return model.isNMRelation(element);
    },
    function (relation, model) {
        var role1 = model.getRole1(relation), // first entity
            role2 = model.getRole2(relation), // second entity
            id = relation.id,
            name = relation.attributes.name, // name of the table
            results = {  // will contain the SQL source code
                project: { children: id }
            };
        results[id] = { type: 'file', name: name + '.sql',
        content:
            // CREATE TABLE statement composition
            'CREATE TABLE ' + name + ' ( ' +
            // add  column referencing the 1st table
            role1.attributes.name + ' int,' +
            // add  column referencing the 2nd table
            role1.attributes.name + ' int );'
        };
        return results;
    }
)
```

No installation

It must be possible for the team to use the framework instantly, with no installation.

ALMOsT.js is developed using pure JavaScript.

It can be integrated inside any web based platform, both on client-side and on server-side (Node.js)

No new language

It must be possible to start using the environment without learning languages that are not normally employed for application development.

ALMOsT.js is developed using pure JavaScript.

Both **data structures** and **rules** are plain JavaScript objects and code.

Fast start-up

It must be possible to create a minimum viable model editor and model transformation in a very short time.

ALMOsT.js is plug-in based, you use/learn just what you need.

- Graphical editors
  ALMOsT-Joint
- *Advanced graph analysis*
  ALMOsT-Extend
- Rule tracing
  ALMOsT-Trace

Parallel Development

It must be possible to work in a team on different aspects of the same sprint.

ALMOsT.js model format can be easily customized to introduce new concepts without the introducing breaking changes.

ALMOsT.js rules can be easily modularized.

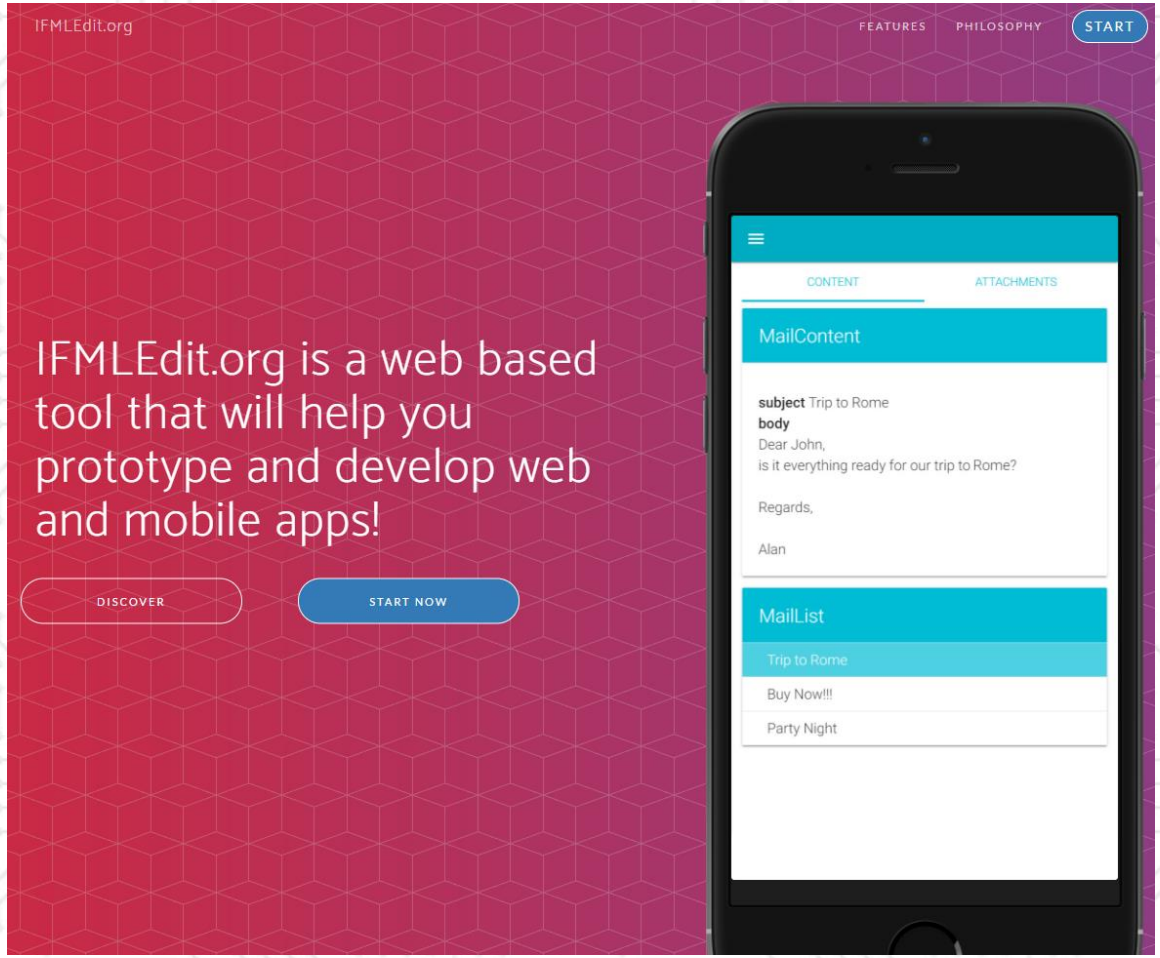
Customized output / Customized generation

It must easy to turn the generated code into a complete version by adding non functional aspects like graphics and sample data collections.

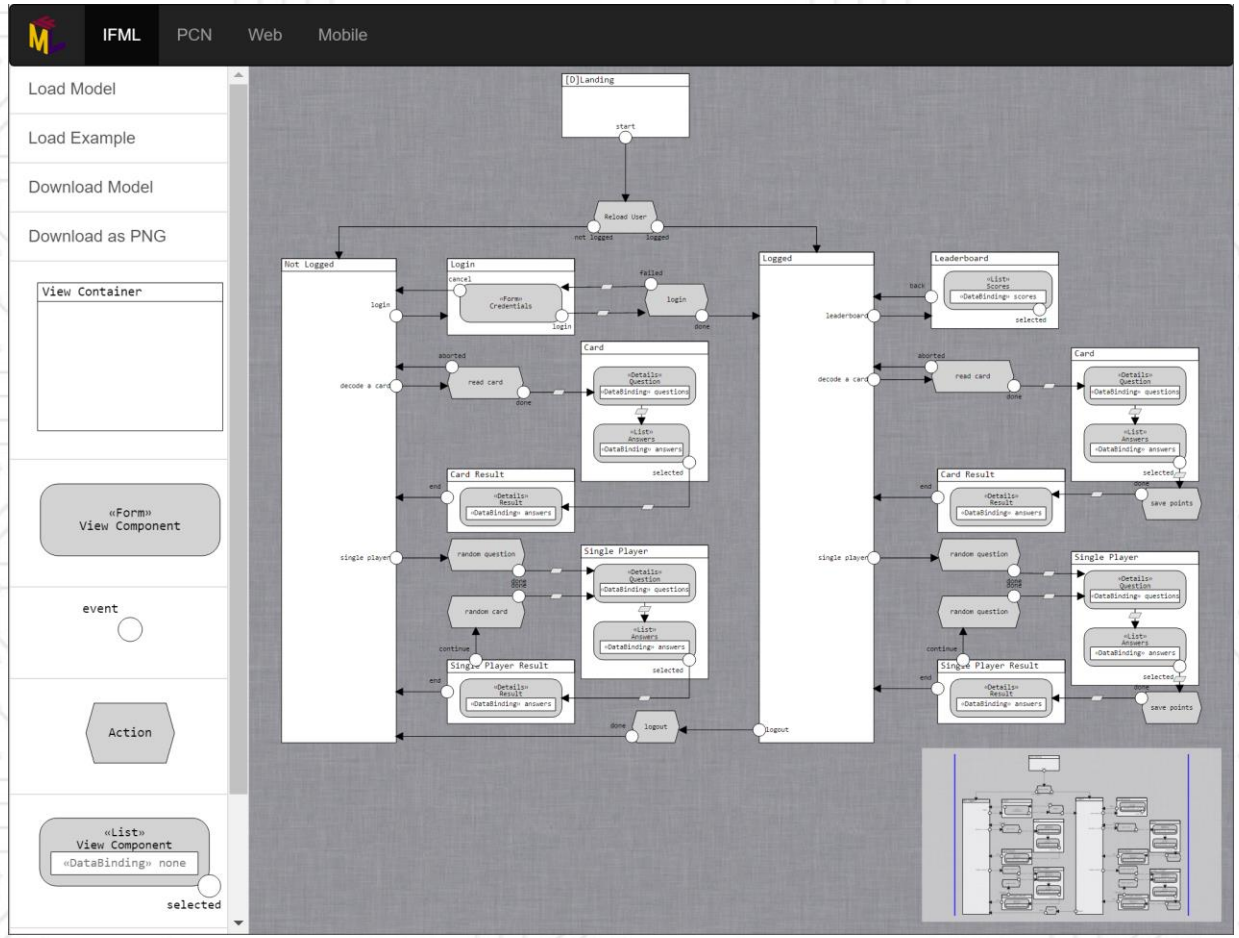
ALMOsT.js generation rules can be easily extended using state of the art template engines like **PUG** and **EJS**.

User Case

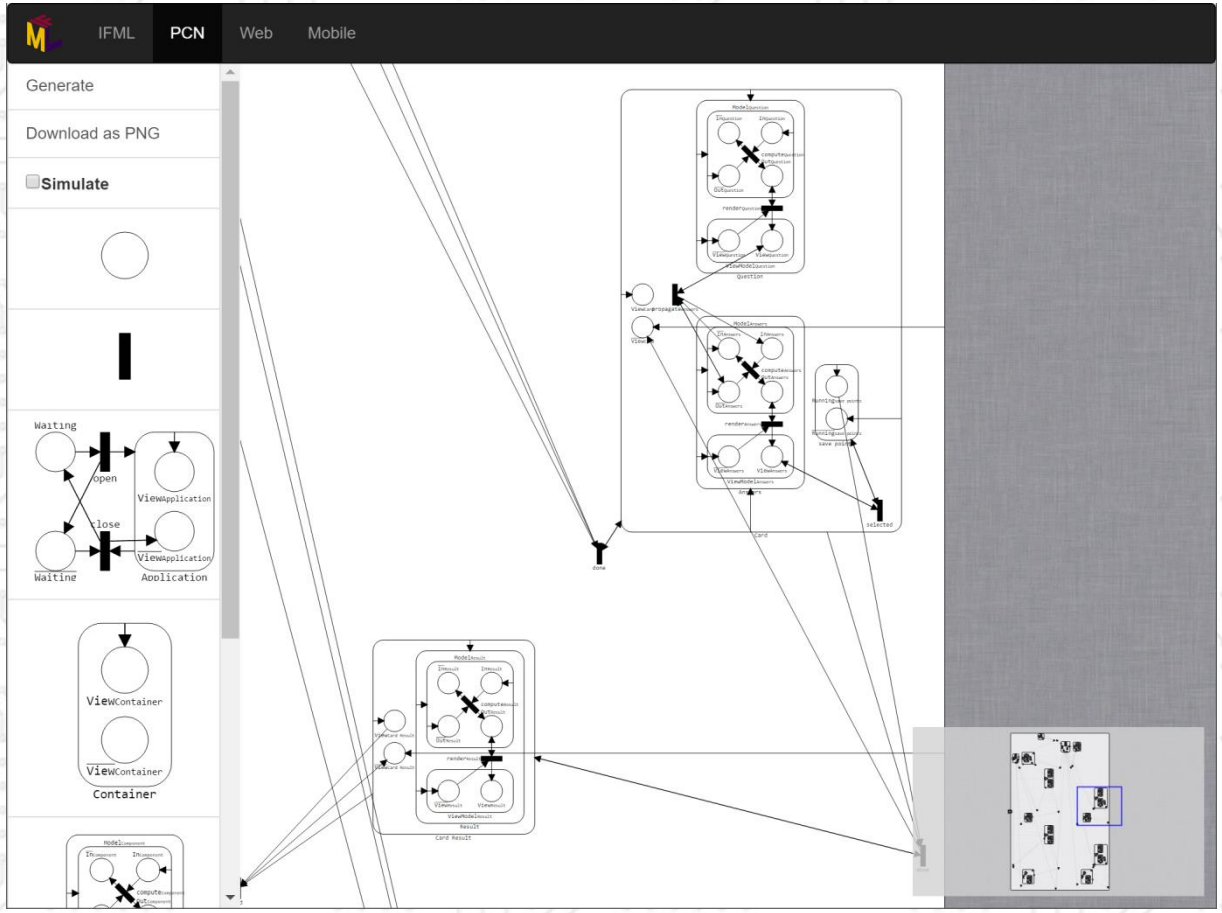
IFMLEdit.org



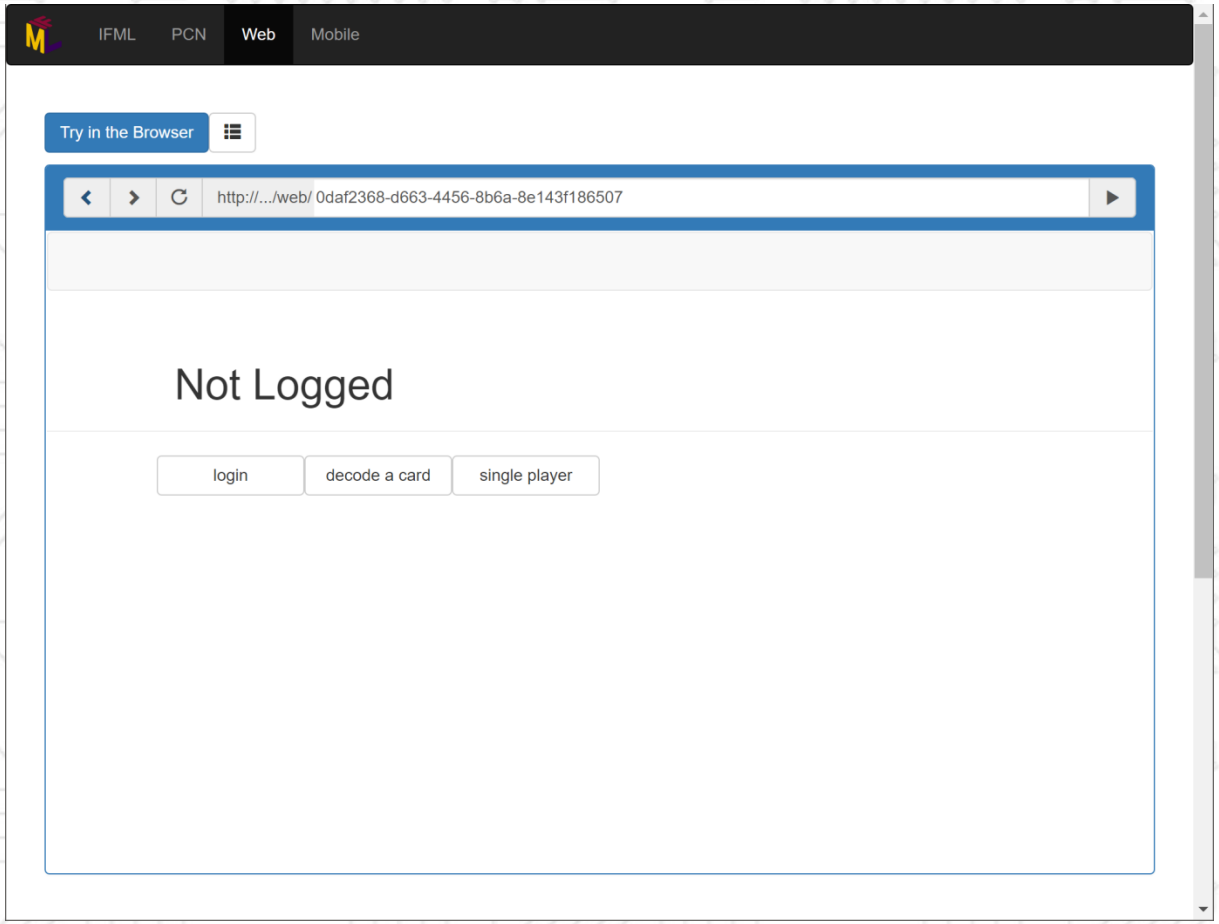
IFML Model Editing

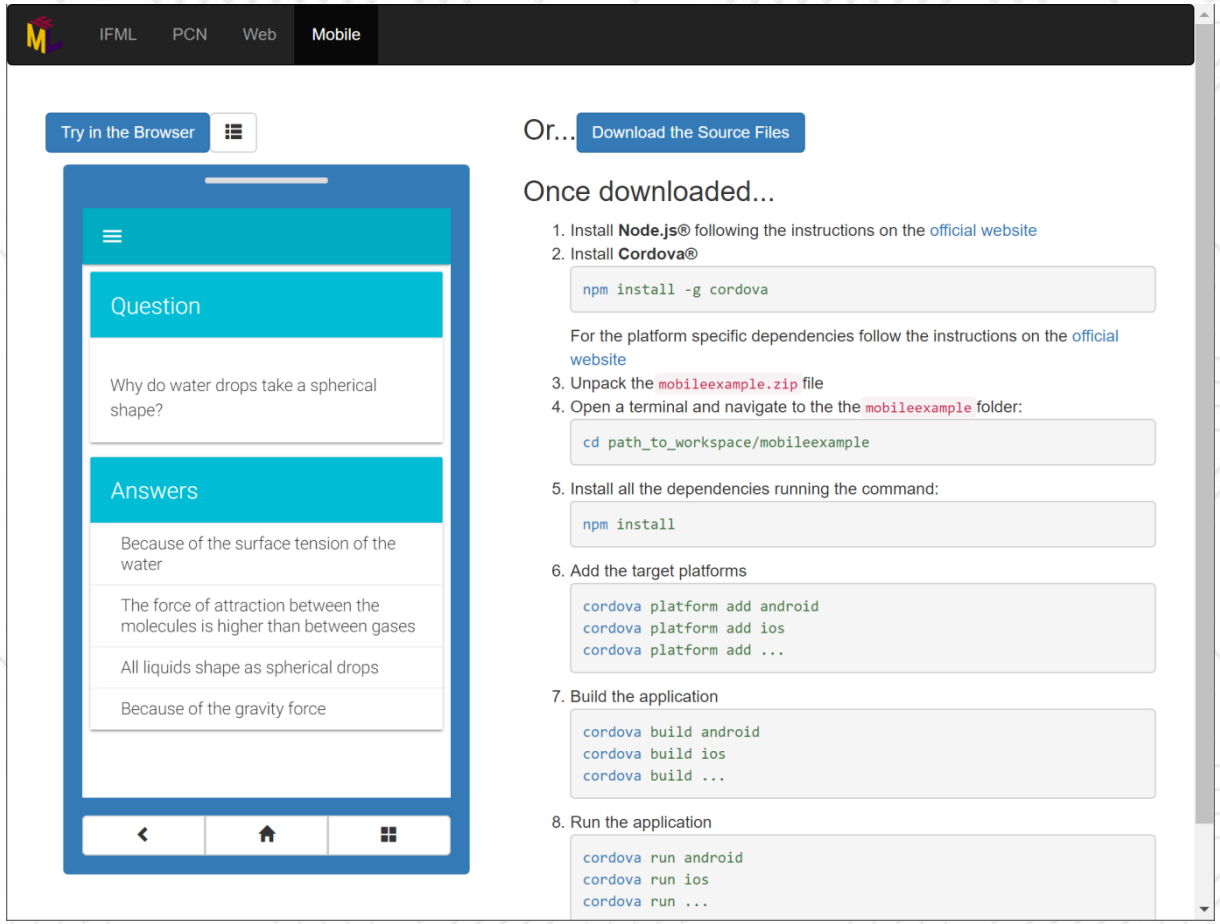


PCN Model Generation and Analysis



Code Generation & Simulation





Let’s get to work?

https://github.com/B3rn475/ICWE2018-Tutorial