DashThings
Project in Software Customization Ubiquitous Computing

Nikolaj Schademose Reibke, nirei12@student.sdu.dk
Emil Sebastian Rømer, emroe12@student.sdu.dk

May 24, 2016

1 Introduction

The first step towards automatization of buildings and homes with Internet of Things is to understand the data provided. Through this understanding it is then possible to decide actions to take based on one or more sensor input given. Making decisions and taking action for one building is not generalizable enough to apply to other buildings without customization. Further are actions rooted in a problem that needs to be solved. This means that customization is needed whenever one of the following is true:

- Problems that needs to be solved might differ from one building to another building
- The buildings infrastructure might not be the same between buildings meaning that the solution might differ
- Context of the environment have changed

In general there is a great need for customization or customizable application in building automatization through Internet of Things.

Using charts or graphs to visualize large amounts of complex data is easier than poring over the data itself. Data visualization is a quick, easy way to convey concepts in a universal manner. Data visualization can be used for different purposes. Some of these are:

- Decision making
- Identify areas that need attention or improvement
- Clarify which factors influence behaviuor
- Identify behaviour or patterns

The problem is that a visualisation is not universal in sense of the purposes it can be used for. Therefore, there are a great need for customized or customizable visualizations that supports the different purposes.

The purpose for the project is to have a website, with responsive design, which can be viewed on any browser like Mobile, Desktop, Laptop and tablet. The pages on the website will contain Links between the pages on the webpage. It will also be possible to link to external webpages. Additionally the pages will contain graphs and tables which shows data. The data will either be fetched from external sources or external sources will have the possibility to post data to a data source given a predefined data schema.

Data in data sources can be transformed using formula expressions and have to go through at least 1 formula expression in order to be displayed on a graph. This expression will be on the graph.

In order to enable the user to create the above in a easy customizable way three self made language will be used.

2 Tools

- Explain which tools and languages we have used (R, which R packages, Emils python/java code etc)

3 section1

4 The Domain Specific Language, DSL

4.1 Overview

The DSL is divided into three components, a arithmetic expressions language, a language for specifying a web site and a language for specifying input sources for the system. The expression language is a utility language which is included by the two others. This language defines how a user can create arithmetic expressions in either of the other two languages. The Web visualizer is a language to create dynamic web sites, like the sample page shown on the front page. This language lets the user create pages on a site, link different pages to each other

and visualize information through the use of graphs. The Datasources language, which is still to be developed, will be used to specify different data sources, like for instance a external databases or APIs. In addition this language will also be used to specify internal persistence and a API for other systems to use for posting data to the Dashk system.

4.2 The Formula Expression Language

The Formula Expressions DSL is responsible for understanding mathematical formulas, which will be applied to tables of data.

Meta model The formula metamodel consist of multiple elements for an overview seen in Figure 2. The types include in the elements included in the model are:

1. Formula: The overall Structure Element
2. Variable: The variable names included in the formula, appears both left and right side of the equation
3. Expression: The right side of the equal sign in a formula. The Expression collects the parts which is connected with plus(+) and minus(−).
4. Factor: A semi-structure of the expression, Factors collects the parts connected with multiplication(·) and division(/).
5. Op1: Identifies a addition or substraction operation
6. Op2: Identifies a multiplication or divison a operation
7. Primitiv: Identifies as either a number or a Variable

Given the following the example:

$$f(x, y) = 5 + x \cdot 3 - y$$

The rules for the grammar is explained as below also referring to Figure 1:
A Formula Contains a name which is defined as the letter/word in front of the paranthesis also underlined in the following formula: $\underline{f}(x, y) = 5+x\cdot 3-y$. In Figure 1 "f" is defined as "name" seen as an attribute for the overall Formula. It also contains a list of variables, defined within the paranthesis seperated by "," as underlined: $f\underline{(x, y)} = 5+x\cdot 3-y$. The variables is saved as a list under formula as "vars" seen at List of Variables. Finally the expression of the formula is everything on the right side of the equal-sign("="), here follows the rule that every variable used in the expression has to appear

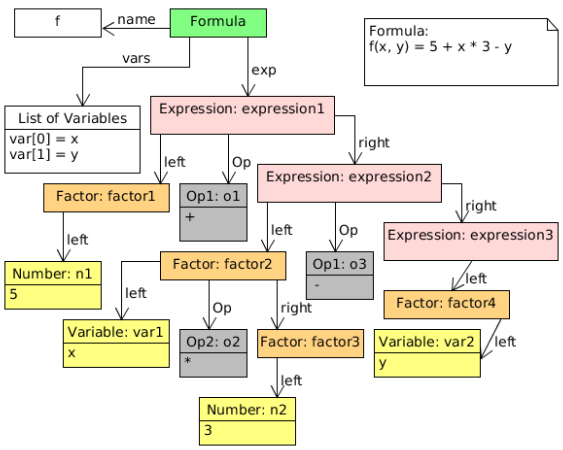

Figure 1: Instance of Formula

inside the paranthesis on the left side of the formula as well, as it works as a function, the expression part underlined: $f(x, y) = \underline{5 + x \cdot 3 - y}$ refered to as "exp" and seen as "expression1" in Figure 1.

Looking at the expression("expression1"): $5 + x \cdot 3 - y$ it is broken down by continouisly working from the left to the right. the left part of the expression is defined as a factor until it meets an addition or substraction sign. For the formula example this would put only the number "5" into the left side factor as "factor1" in Figure 1, futher containing "5" in the Number object "n1". Expression2 is then the rest of the equation after 5+. The new expression contains again one factor which is the left side($x \cdot 3$) seperated by the substraction sign The factor "factor2" contains on the left "var1" which is x, and on the right another factor is spawned seen as "factor3" in Figure 1, only containing a left side being the number seen in "n2" as 3. Coming back up to the expression. it contains the last part of y as "expression3" which only a left side being a factor(factor4) and that factor also only contains a left side which is the variable "var2" being the y.

For the variables "var1" and "var2", both is included in the List of Variables in as seen in Figure 1

Language Validation has been archieved partly by the BNF[1] using Xtext, but also using Xtend to write additional validation code, where the Xtext language was not sufficient to validate that the structure did indeed not violate any rules.

Xtext could only validate the general structure of the model. what xtext was not archieved was to have xtext validate that the only variables used on the right side og the equation sign was declared on the left side. In order to validate that the vari-

[1] ?

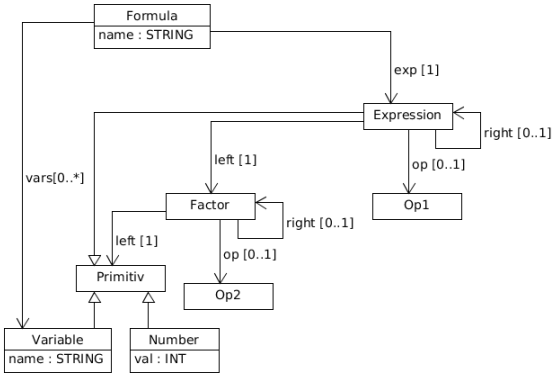

Figure 2: Model of the formula module

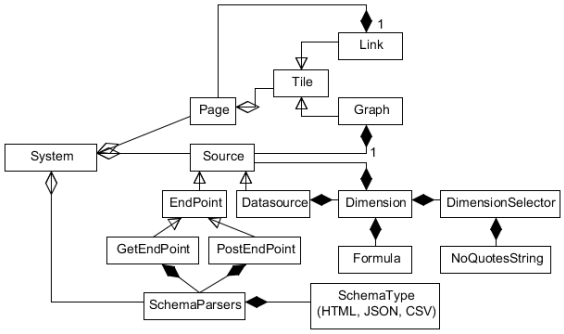

Figure 3: Model of the language

ables on the right was declared on the left a collection containing the declared variables on the left was matches against all the variables on the right fetched using recursion with depth first search.

4.3 The Web visualizer Language

For generating dashboards custom language have been created. This language specify:

- Which pages the web-interface consist of
- The navigation between pages
- Which data to display and where
- Where and how to obtain the data

  Internal data streams

  External data streams

- How to manipulate the data streams

The following subsection will look into detail of how the language is structured and the syntax and grammar of the language.

```
System:
    (pages += Page | sources += Source
        | schemas += SchemaParser)*
;
Page:
    'Page' name=ID
    '{'
    (tiles += Tile)*
    '}'
;
Tile:
    Link | Graph
;
Link:
    'Link' name=ID 'to' page=[Page]
;
Graph:
    'Graph' name=ID    source=[
        Datasource] ('label=' label=
        STRING)?
;
```

Figure 4: Sample grammar definition, For a complete view of the grammar, see Appendix

The Grammar Figure 3 shows the structure of the language. This model shows that the language defines of a system containing three parts:

- Pages
- Sources
- Schemas

The Pages, is what the language in turns will use to create web pages in a system defined in the language. A page is a collection of tiles. Each tile is an extendible definition of a visual object on a page. In the current version, there are two types of Tiles, Links and Graphs. A Link definition contains a pointer to a page. A Graph however, has a reference data *source* from witch it needs to fetch data from. Figure 4 shows a sample code snippet from the grammar definition. The Figure shows how the grammar is defined for the Pages, Tiles, Links and Graphs. Similar does Figure 5 show and example use of this grammar. This example creates a system object with two pages inter connected by links and with a number of graphs.

The Sources, can be of two different types, an *EndPoint* or a *DataSource*. An EndPoint is intended to function as interface towards external systems or data sources. Whereas a Datasource is an internal definition intended for filtering, data manipulation or data grouping.
This Source, Endpoint, Datasource-Dimension

```
Page index
{
    Link toPageOne to pageOne
    Graph testGraph a
}

Page pageOne
{
    Link toHomePage to index
    Graph testGraph a
    Graph testGraph1 c

}
```

Figure 5: Example language use, defining pages

```
Datasource temperatureRaw {
    Dimensions:
        Formula x20508a1(x) = x using
            temperature[x20508a1] as x,
        Formula x205101(x) = x using
            temperature[x205101 ] as x,
        Formula x205111(x) = x using
            temperature[x205111 ] as x
}
```

Figure 6: Example language use, defining data sources

structure have been defined as a compositional pattern, with source as the component, the Endpoint as a leaf and the Datasource as the composite. This pattern makes the it possible to compose any number of different structures. In addition every defined dimension comes with a formula, which will be applied to the data. Since Source can have multiple dimensions (data streams) when specifying a how a Source an additional dimension selector can be specified to select a subset of the dimensions. Figure 6 shows and example of a Datasource created through the language. The first line creates a Datasource and the declares a list of dimensions. Each declaration of a dimension starts by declaring a formula. After the formula follows a declaration of Sources to use in the formula.

Figure 7 is an example of the use of Endpoints. In this, a GetEndPoint is created. This Endpoint type contains a url, a header and a schema parser. In addition this can contain a data field used when for POST request returning data, like for instance the query language of the SMap system.

The Schemas, or SchemaParsers are a global way of defining how to parse date from an external source. This is both needed when defining an source

```
GetEndPoint temperature {
    url "http://10.123.3.12:8079/api/
        query"
    data "select␣data␣in('{(datetime.
        datetime.fromtimestamp(time.
        time())␣-␣datetime.timedelta
        (1)).strftime('%m/%d/%Y')}',␣
        '{datetime.datetime.
        fromtimestamp(time.time()).
        strftime('%m/%d/%Y')}')␣where␣
        Path␣like␣'/1␣-␣Ground/%/
        Temperature'"
    Headers {
        "type" : "application/json"
    }
    use Schema schemaB
}
```

Figure 7: Example language use, defining endpoints

to request data from and when external sources posts data to the system.

Meta model The language compiles a dynamic website using the Django framework. This framework uses an MVC architektur which is extended with a template pattern with HTML documents. In order to make the output easy readable and customizable for a user the output needs to be mapped into a model that follows this pattern with more explicit information.

In order to do so, a meta model of the a website using this framework have been created which incorporates the functionality of the desired dashboard system. Figure 8 shows this meta model. This meta model differs in a number of ways from the model created by the language. This is due to the high complexity of the Django framework. In order to keep the language simple the group decided to create a simpler model for the language (see Figure 3) and map this model to the created meta model of the system. The meta model however have adopted some aspects of the language model, like for instance the part concerning Datasources, Endpoints, parsers and selectors. The Django framework uses a MVC pattern. Which means that the constructed pages and content needs to be mapped into this form. The Django way, is to define a controller in form of a urls fils containing the mapping of urls to either views of the system or other urls files. The views can then either use a model and/or a HTML template file to build up the responses for a request with the use of a number of serializes.

Language Validation of the entire vizualizer project containing validating the following items:

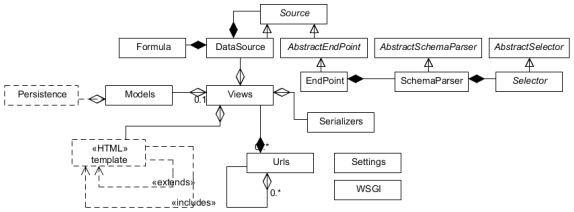

Figure 8: Model of the Website

1. **[XText]** The page linked to from one page to another existed
2. **[XText]** Graphs contains an existing reference to a Datasource
3. **[XText]** GetEndPoints contains a Schema
4. **[XText]** Schemas contain at least 1 Selector
5. **[XText]** Datasources contains at least one Dimension
6. **[XText]** Dimension contains at least one reference to a Source
7. **[XTend]** Datasources have no cyclic dependencies
8. **[XTend]** All the variables declared on the left side of a formula is also declared under one of the DimensionSelectors listed for that Dimension.
9. **[XTend]** All Datasources is directly or indirectly connected to an EndPoint

It was not archieved to validate all the rules in the grammar strictly using XText and a few of the rules had to be checked in the syntax tree. XText can make checks like that a reference is existing and that the overall grammar structure is correct. The things that XTend can do is that after the initial grammar, the instance of the grammar is built and that gives some additional possibilities to validate the program from XTend. Here only the full validation should be possible, given that the language is not turing complete[2]. As listed previously cyclic dependencies can be checked a long with more advanced type checking. Maybe a language would contain an odd rule that the number 5 would not be allowed. This could be put into the XText document, but would be much easier to validate using XTend. Here we could also extend the Formula projects language validation rules within the Vizualizer language futher using XTend by checking that declared variables on the left side of a formula was supported by a declared variable from a DimensionSelector.

[2]make ref to turing completeness

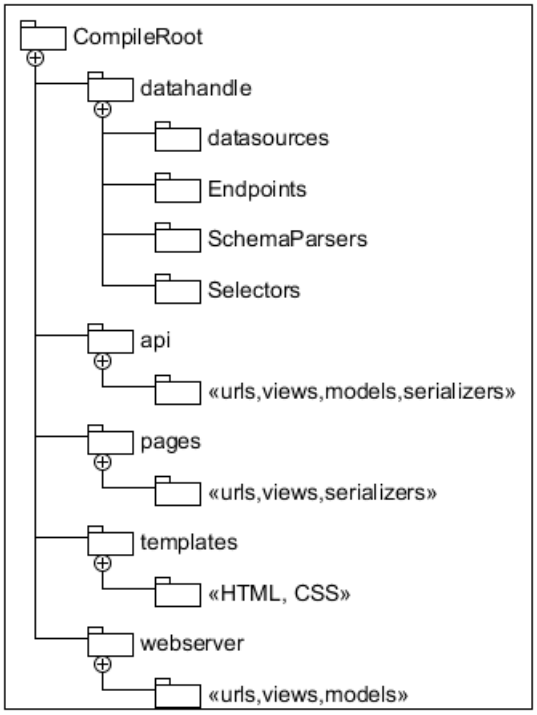

Figure 9: Package diagram of generated sources

Code generation To maintain a clean code structure all generated code are put into a number of different packages, see Figure 9. This diagram shows the structure of the generated code. The data handle package contains all code for handling obtaining data from external systems. Into here the data sources, endpoints selectors and schema parsers are generated. The api package contains the code generated for creating a API for external sources to post data. This package will contain models, views, serializes and a controller for the API. A page object from the language model is split into a number of fractions. A page needs to be divided into a model a view, a controller and a HTML template document. Better keep things apart the template documents have been gathered in a package by itself (the template package). The model views and controllers are generated into the pages package. Each page gets it own template file extending a base template file making them easy customizable. The individual template file contains only the structure of the specific links and graphs (without data) of that page.

Generation The code generation have been done through the [XTend] language. This language provides some additional coding features for the plain Java language. One of the highly used features is the multi dispatching. Figure 10 how formulas are generated using multi dispatching in a

```
def dispatch String
    leftResursiveTraversal(Expression
     expression, String string) {
    var String result = expression.
        left.leftResursiveTraversal(
        string)
    result = expression.right == null
        ? result : expression.right.
        leftResursiveTraversal(result
        + factor.op)
    return result
}
def dispatch String
    leftResursiveTraversal(Factor
    factor, String string) {
    var String result = factor.left.
        leftResursiveTraversal(string)
    result = factor.right == null ?
        result : factor.right.
        leftResursiveTraversal(result
        + factor.op)
    return result
}
def dispatch String
    leftResursiveTraversal(Variable
    variable, String string) {
    return string + "input_" +
        variable.name + "[::, 1]"
}
def dispatch String
    leftResursiveTraversal(Number
    number, String string) {
    return string + number.^val
}
```

Figure 10: Code generation with multi dispatching

nested loop.

5 section3

6 Conclusion

- Concussion

A Author Contributions

All participants of the project has contributed equally to the entire report, in addition to that all participants has additionally .

B The Grammar

```
grammar org.xtext.sdu.
    iotvizualizerlanguage.Vizualizer
    with org.xtext.sdu.
        formularzlanguage.Formular

generate vizualizer
    "http://www.xtext.org/sdu/
        iotvizualizerlanguage/
        Vizualizer"

System:
    (pages += Page | sources += Source
         | schemas += SchemaParser)*
;
Page:
    'Page' name=ID
    '{'
    (tiles += Tile)*
    '}'
;
Tile:
    Link | Graph
;
Link:
    'Link' name=ID 'to' page=[Page]
;
Graph:
    'Graph' name=ID    source=[
        Datasource] ('label=' lael=
        STRING)?
;

Datasource:
    'Datasource' name=ID
    '{'
    'Dimensions' ':'
    dimensions+=Dimension (','
        dimensions+=Dimension)*
    '}'
;

Dimension:
    'Formula' name=Formula
        sourceSelectors+=
        DimensionSelector ('and'
        sourceSelectors+=
        DimensionSelector)*
;

DimensionSelector:
    'using' source=[Source]'['
        selectVar=NoQuotesString ']' '
        as' name=ID
;

NoQuotesString:
    name=ID
;
```

```
Source:
    EndPoint | Datasource
;

EndPoint:
    GetEndPoint | PostEndPoint
;

PostEndPoint:
    'PostPoint' name=ID
    '{'
    'url' url=STRING
    'use Schema' parser=[SchemaParser]
    '}'
;

GetEndPoint:
    'GetPoint' name=ID
    '{'
    'url' url=STRING
    ('data' json = STRING)?
    'Headers' '{'
        headers+=Header (',' headers+=
            Header)*
    '}'
    'use Schema' parser=[SchemaParser]
    '}'
;

Header:
    keyword=STRING ':' value=STRING
;

// A SchemaParser is used to parse a
    schema(data structure) into a
    time series
SchemaParser:
    'Schema' name=ID
    '{'
    'SchemaType' '=' schemaType=
        SchemaType
    selectors+=Selector+
    '}'
;

enum SchemaType:
    XML='XML' | CSV='CSV' | JSON='JSON
        '
;

// Select the path to a specific
    dimension of the data
Selector:
    'Selector as ' name=ID '{'
    steps+=STRING ('->' steps+=
        STRING)+
    '}'
;
```