A Web Specific Language for Content Management Systems

Viðar Svansson and Roberto E. Lopez-Herrejon

Computing Laboratory, University of Oxford, England

Abstract. Many web applications can be specified in terms of the content-types and security policies they provide. Some web frameworks aim to ease the implementation of this family of programs. We propose an extensible domain-specific language to specify these programs, and methods to synthesize the applications from their specification. This allows domain experts, notably web designers, to describe the business domain at higher level than code and automate the implementation process. We target a software product line architecture for content-management systems. This allows new features of the product line to be derived on demand rather than limiting the variability of new products to existing features. A case study is presented that demonstrates how the application design can be captured in our language and its implementation fully automated as features of a product line. The final product is then a composite of the derived features and selected existing features.

1 Introduction

A *Content Management System (CMS)* is a software system for authoring and controlling content of web applications. The CMS requirements are constantly increasing, resulting in very complex systems. This has lead to a wide adoption of general purpose open source CMS packages, which are applicable to any domain, providing flexible customisation via programming or configuration properties to adapt to the organisational profile.

Increasing complexity in web application development has resulted in a plethora of web-specific frameworks to aid the implementation of applications with rich APIs, reusable components, configuration files, and reasonable defaults (to name a few). The infrastructure and conventions employed in these framework can be hard to master. Propagating changes through various layers of the frameworks and keeping up with changes in the framework API, coding conventions, or configuration files, is even harder.

On the other hand, web frameworks and content management systems contribute to better understanding of an application domain. Well understood structure of web content and its workflow allows applications to be described at more abstract level than code and makes it feasible to generate the framework completion code.

Zope[1] is an example of a platform for the development and deployment of CMSs. It consists of a rich object-oriented web framework with strong emphasis on reusable components, an application server container for web-based applications, and an object-oriented database [16]. An example of a Zope based application is the Plone CMS[2]. Plone is commonly deployed both by large and small organisations to power both their public websites as well as their intranet.

Model Driven Development (MDD) is a software development paradigm where models capture one or more aspects of a program design and model transformations synthesize programs [11, 13]. The models can be written in a *Domain Specific Language (DSL)*, which is a formal language to describe concepts and concerns in a particular problem domain [7]. The concepts that form a DSL can be extracted from the supporting *Domain-Specific Frameworks (DSF)* [1]. These frameworks provide common abstractions for the programmer in order to implement a concrete application. *Ruby on Rails*[3] is an example of a DSF. Rails popularity is mainly due to its recognition of common programming patterns and providing abstractions for them, and by implementing many of the techniques promoted within the Agile Data community[4]. Many other similar frameworks have appeared in the last few

[1] See http://zope.org
[2] See http://plone.org
[3] See http://rubyonrails.org
[4] See http://www.agiledata.org

years, which lead us to investigate the commonalities within the frameworks that could be extracted to a DSL.

Software Product Lines (SPL) are families of related programs distinguished by the set of *features*, i.e. increments in program functionality, they provide [3, 5]. Extensive research has shown how SPL practices can improve factors such as asset reuse, time to market, or product customization and the important economical and competitive advantages they entail [6].

In this paper we present *Web Specific Language (WSL)*, a language to describe business domains for the Web and its MDD architecture to synthesize applications. Our focus is on providing a language to define business domains based on concepts commonly found in a CMS, but at the model level rather than at the metamodel level. Furthermore, using WSL allows us to synthesize features and thus develop a product line. We illustrate our approach by building a simple domain-specific application for academic research web sites.

2 Web Specific Language

When designing a DSL for content management, we focus on two aspects. First, we need to describe the datamodel of applications. This is essentially the structure of the web application, which maps to the concepts in the business domain. This follows closely to the *model* in many web frameworks. Second, the workflow of content management describes how we can interact with the content based on workflow engines.

Web Specific Language (WSL) is a domain-specific language for specifying web applications in terms of the content-types and the security policies it provides for content management. Content is structured hierarchically to reflect the hierarchical nature of most web applications; folders and pages in markup languages are examples of this. The main security policies are workflows.

WSL is defined with a BNF based grammar, thus the programs are plain text files. This representation has advantages over XML based models. There is no need for a specific tool to create and edit WSL programs, there is better support for versioning control of text based artifacts, and it is more accessible to developers not familiar with modeling techniques.

The WSL syntax is written in *xText* format, a tool to build textual languages [10] that is part of the *openArchitectureWare* MDD framework[5]. A fragment of the WSL syntax is depicted in Figure 1.

A WSL program consists of entities. An *entity* is either a domain, a feature, or a metadata. A *domain* is a collection of entities that belong to the same business domain. A *feature* represents a concept or concern within a domain. Examples of features are content-types, workflows, and permissions (Line 9). A content-type is a composite of primitive types and references to other composites. WSL provides several built-in content-types (e.g. folder, event, and document defined in Figure 1, Lines 15-17) to specify more complex user-defined content-types by adding primitives or references to them. Stereotypes explicitly override default properties of types. Metadata adds information to features and domains. An example of common metadata is the title and the description of the domain or feature.

Figure 2 depicts a simplified metamodel for content-types. The relationships between content-types is denoted with associations. We can specify that certain type is restricted within particular `containers`, or that a folder restricts what it `contains` to types. The containment relationships are used to model the possible hierarchical structure of the application. The `relation` association denotes relations between types that are not in a direct hierarchical relationship. The actual content on each page is then composed of `fields`. The `FieldType` is presentation oriented, *i.e.*, it defines how the field is presented to the user of the system. Example types include primitive types such as integers, booleans, and floats; data structure such as strings, lists, and maps; and web-centric types such as URLs, images, and HTML.

Content management systems need to support workflows to enable content to change states, where each state has different access control. A workflow specifies the possible states a content can be in,

[5] See http://openarchitectureware.org

```
1  Application: 'application' name=ID '{'
2    (entities+=Entity)*
3  '}' ;
4
5  Abstract Entity: Metadata | Domain | Feature ;
6
7  Domain: 'domain' name=ID '{' (entities+=Entity)* '}' ;
8
9  Abstract Feature: Content | Workflow | Permission ;
10
11 Abstract Metadata:
12   Description | Title | Author |
13   Version | URL | Email | Licence ;
14
15 Abstract Content:
16   Base | Folder | Document | File |
17   Event | Image | Link ;
18
19 Base: 'content' name=ID '{'
20     (concepts+=Concept)*
21 '}' ;
22
23 Abstract Concept:
24   Metadata | Field | Containment ;
25
26 Field:
27 (stereotypes+=Stereotype)*
28 type=Type name=ID '{'
29     (attributes+=FieldAttribute)*
30 '}' ;
31
32 Abstract FieldAttribute:
33     Widget | Title | Description | Default | Stereotype ;
34
35 Abstract Stereotype: Required | Searchable | Ordered ;
36
37 Abstract Containment: Contains | Container ;
38 Contains:  'contains'  (containment=ID)* ';' ;
39 Container: 'container' (containment=ID)* ';' ;
```

Fig. 1. WSL Grammar Fragment

the transitions between states, the guards that have to be satisfied to trigger the transitions, and the permissions required to perform operations in a state.

Figure 3 shows the organisation of the main security concepts in our metamodel. `Security` maps roles to permissions and a `State` is composed of transitions and security mappings. The `guards` on a `Transition` allows us to select which `roles` are allowed to change the state.

In Section 4 we elaborate how the WSL grammar is used to build a MDD infrastructure for the case study described next.

3 Akademia Case Study

Our case study considers the basic concepts present in websites of academic research shown in Figure 4. A WSL program is an `application`, *Akademia* in our case (Line 1). As described in previous section, an application can contain many domains, in our case the `domain` is *Research* (Line 3). Lines 5 and 6 are examples of metadata for this domain.

Figure 4 shows three features: `ResearchProject` (Line 8), `ResearchGroup` (Line 28), and `Publication` (Line 32). A research project has: a title (Line 9), a set of publications (Line 10), associated research groups (Lines 12-15), references to other research projects possibly external to this site denoted with a *Link* type (Lines 17-20), and a `required text` field that contains a mandatory field with the aims of the project (Lines 22-25).

For sake of simplicity, we omit details of this domain in the Figure. For instance, a research group contains text fields on the group, such as the overall goals and history of the group. A publication is

any printed material resulting from a research project. It has a title, authors, an abstract, and a year of publication as mandatory fields. It optionally contains the publisher, funding bodies, and a link to the original file. For further details on Akademia refer to [14].

4 WSL Implementation

WSL programs are first transformed to *Eclipse Modeling Framework (EMF)*[6] models from which text artifacts are derived. We follow Batory's view of MDD as metaprogramming paradigm [4], and denote transformations as functions that map values (programs or models) to other values. We denote transformation as functions T_f, where f is the name of the function. Artifacts or programs are denoted as values X_y, where X is the name and y is the type. Note that each WSL program maps to a single EMF model.

4.1 Step 1: Transforming WSL to EMF

As we mentioned in Section 2, we used xText [10] to build our language. This tool generates from a syntax definition: an EMF metamodel (Ecore) for the language, a transformation function from textual programs to EMF models, and a language-specific Eclipse textual editor. The following equation describes xText as a metaprogram:

$$\{WSL_{ecore}, T_{wsl2emf}, WSL_{editor}\} = T_{xtext}(WSL_{ebnf}) \quad (1)$$

WSL_{ebnf} corresponds to the grammar depicted in Figure 1. WSL_{ecore} is the metamodel generated for WSL. $T_{wsl2emf}$ is the transformation function to map WSL programs to EMF models. The WSL_{editor} allows the creation of WSL programs, such as $Akademia$ in Figure 4, which we denote as $Akademia_{wsl}$.

4.2 Step 2: Transforming EMF to Plone

Our code generation relies on *Xpand* [9] and *Extend* [8][7], which are also part of the openArchitectureWare framework. Xpand is a language for model-to-text transformation that we use to traverse the EMF model in a depth-first search manner. A limitation of this approach is that some pieces of information needed to generate artifacts may not be available when traversal reaches a node. We use Extend, a language to define model operations, to address this issue. We also used Extend to recover information lost in grammar-to-metamodel transformation (see Section 4.3), compute platform specific boilerplate code, map types from WSL to our target platform, and derive default values for variables.

[6] See http://eclipse.org/modeling

[7] In other words, our T_f functions are implemented using both languages.

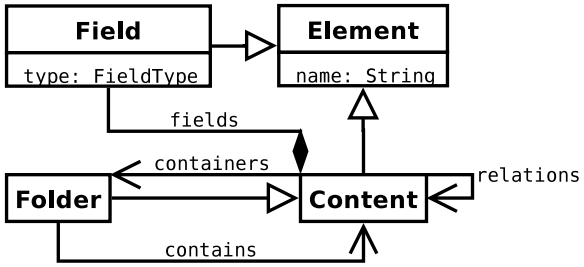

Fig. 2. Metamodel for Content-types.

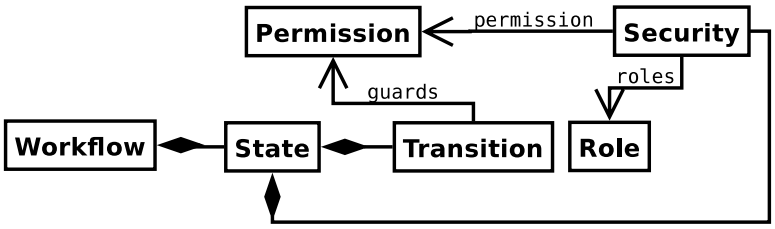

Fig. 3. Metamodel for Security Policies.

After evaluating both QVT technologies based on MDA and Xpand, we found the combination of model traversal and code generation enhanced with *Aspect-Oriented Programming (AOP)* techniques[8] in Xpand to be a more natural model-to-text process for our puposes. Model traversal support, including polymorphism and parametrisation, are essential to make tree walking a straightforward process and solves most code generation use cases. In the case where more context is needed, escaping to Extend proves valuable. AOP functionality allows fine grained customisation of the generated artifacts per project. To utilise the full potential of the AOP support in Xpand, we follow a concise traversal and transform pattern to be able to weave around the generation of: particular artifacts, groups of related artifacts, or particular pieces of code within artifacts.

Component Generation. Zope components are objects with clear understanding of their functionality and responsibility [16]. A component is specified with a Python interface, a class that implements the interface, and an XML configuration that binds the two at run-time. Archetypes is an object-oriented DSF written in Python that we used to implement the components with support for persistency, web interaction capability and other content management functionality.

The following equations denote the generation of interfaces, implementation classes, and their configuration respectively[9]:

$$X_{interfaces} = T_{emf2interfaces}(X_{emf}) \tag{2}$$

$$X_{archetypes} = T_{emf2archetypes}(X_{emf}) \tag{3}$$

$$X_{config} = T_{emf2config}(X_{emf}) \tag{4}$$

Let us illustrate this process with the Akademia language. Recall from Equation 1 that xText produces $T_{wsl2emf}$ transformation function and WSL_{editor}. We write the Akademia language using WSL_{editor} and transform it with $T_{wsl2emf}$ to obtain its EMF representation as follows:

$$Akademia_{emf} = T_{wsl2emf}(Akademia_{wsl}) \tag{5}$$

For each `Content` in the EMF model of a WSL program, a Zope component is generated. This is denoted by plugging $Akademia_{emf}$ into Equations (2), (3), and (4). Figure 5 shows the interface generated for `ResearchProject` feature and Figure 6 depicts a snippet of its generated configuration. As recommended in [2], the implementation of a component is a class that uses the Archetype framework. The generated implementation is not shown for simplicity, for further details refer to [14].

Profile Generation. Profiles permit a selection of components and properties to create different configurations, effectively creating a product line. They also specify *non-functional features* of the

[8] See http://www.aosd.net

[9] X_{emf} is an instance of WSL_{ecore}.

```
application Akademia {

  domain Research {

    title "Research Content";
    description "Concepts for academic research";

    folder ResearchProject {
      title "Research Project";
      contains Publication;

      reference groups {
        title "Associated Groups";
        type ResearchGroup;
      }

      reference projects {
        title "Related projects";
        type ResearchProject, Link;
      }

      required text aims {
        title "Project Aims";
        description "A summary of what this project tries to achieve";
      }
    }

    folder ResearchGroup {
      /* details omitted */
    }

    content Publication {
      /* details omitted */
    }
  }
}
```

Fig. 4. Akademia Program Fragment

product line, such as `Workflow` and `Permission`. *GenericSetup*[10] is the tool that supports initialisation of applications based on a profile. It registers the components to the CMS, declares their properties, loads the workflows into the workflow engine, and carries out other management tasks declared in the profile.

Figure 7 shows state `pending` of a workflow that represents a content waiting for a reviewer to make an editorial decision on it. It can be published (triggering transition `publish`), rejected (transition `reject`), or terminated (transition `terminate`). There are two operations defined on this state, `Modify portal content` and `View`, and their associated roles, `Reviewer` and `Anonymous` respectively.

[10] See http://plone.org/products/genericsetup

```
class ResearchProject(Interface):
    """ Research Project """
    contains('akademia.research.interfaces.Publication')
    groups = schema.Field(title = _(u"Associated Groups"),
                          description = _(u""))
    projects = schema.Field(title = _(u"Related projects"),
                            description = _(u""))

    aims = schema.Text(title = _(u"Project Aims"),
                       description = _(u"A summary of what this project tries to achieve"),
                       required = True)
```

Fig. 5. ResearchProject Interface

```
<class class=".researchproject.ResearchProjectImpl">
    <require
         permission="zope2.View"
         interface="..interfaces.ResearchProject"
    />
    <require
         permission="cmf.ModifyPortalContent"
         set_schema="..interfaces.ResearchProject"
    />
</class>
```

Fig. 6. ResearchProject Configuration Fragment

A component configuration profile enables a finer degree of variability within a component, by customising properties such as constant string values in a non invasive way. For example, Figure 8 is the generated configuration for the `ResearchProject` component. It specifies properties, such as the `title` of the component (Lines 5-7), what icon it uses (Line 8-10), what type of content is allowed to add to it (Lines 17-19), and what view methods can be used on it (Lines 21-23).

The process of generating the profile from a WSL program is denoted:

$$X_{profile} = T_{emf2profile}(X_{emf}) \tag{6}$$

Summary. The entire generated Plone codebase for an application *X* is thus combined from its four component artifact types.

$$X_{plone} = \{X_{interfaces}, X_{archetypes}, X_{config}, X_{profile}\} \tag{7}$$

The Akademia application is generated from a 106 LOC model in a single source file. The generated artifacts span 1271 LOC, resulting in approximately 1:12 ratio and total of 46 files in 11 directories. The generated application includes a set of components that provide the required content functionality and do not require any modifications after they have been generated. The other main artifacts that we generate are configuration profiles that allow us to configure the generated applications in the context of a software product line.

4.3 Challenges

We faced several challenges using text templates for syntax-to-metamodel and model-to-text transformation. An important limitation of xText transformation is that association references are not preserved [10]. For instance, the metamodel does not capture that `ResearchGroup` in Line 14 of Figure 4 is a reference to `ResearchGroup` defined in Line 28. This can be solved with an additional pass on the abstract syntax tree before it is transformed to an instance of an EMF model.

```
<state state_id="pending" title="Pending review" i18n:attributes="title">
    <description>Waiting to be reviewed.</description>
    <exit-transition transition_id="publish"/>
    <exit-transition transition_id="reject"/>
    <exit-transition transition_id="terminate"/>
    <permission-map name="Modify portal content"
                    acquired="False">
        <permission-role>Reviewer</permission-role>
    </permission-map>
    <permission-map name="View" acquired="False">
        <permission-role>Anonymous</permission-role>
    </permission-map>
    <!-- remaining directives omitted -->
</state>
```

Fig. 7. Pending State Workflow Fragment

```
<object name="ResearchProject"
   meta_type="Factory-based Type Information with dynamic views"
   xmlns:i18n="http://xml.zope.org/namespaces/i18n"
   i18n:domain="akademia.research">
 <property name="title"
     i18n:translate="">Research Project
 </property>
 <property name="content_icon">
    ++resource++research_project_icon.gif
 </property>
 <property name="content_meta_type">ResearchProject</property>
 <property name="product">akademia.research</property>
 <property name="factory">addProduct</property>
 <property name="immediate_view">atct_edit</property>
 <property name="global_allow">True</property>
 <property name="filter_content_types">True</property>
 <property name="allowed_content_types">
     <element value="Publication" />
 </property>
 <property name="allow_discussion">False</property>
 <property name="view_methods">
     <element value="base_view"/>
 </property>
 <!-- remaining elements omitted -->
</object>
```

Fig. 8. ResearchProject Configuration Profile

Template languages are often not robust in whitespace handling and often delegate nice formating of the output to external tools. These tools are of limited use when the target language includes whitespace (mostly end-of-line and indentation characters) in its language definition, such as Python.

We found the choice of escaping characters in Xpand to be problematic. It practically forces template editing to be performed within Eclipse as other editors do not facilitate inserting escapes. It also limits the interoperability of template files because different development environments or tools can have different encoding for the escaping character.

These challenges indicate that our choice of tool support did not fully fit our needs. However, they did prove very valuable to prototype our MDD chain. For full control of the transformation process, we can directly manipulate the abstract synax tree by implementing visitors that traverse the tree and perform any required transformations.

5 Related Work

Using MDD to model and generate web applications is not new. Our work was inspired by ArchGenXML (AGX)[11], a tool that transforms platform-specific UML class diagrams to Plone content-types and state charts to workflows. AGX uses a Plone-specific UML profile to specify the architecture of the application, which is useful for people who are not familiar with the platform. Since AGX uses models Plone applications directly, it can not be used with other implementation technologies. Further, migrating to newer versions of Plone can require updates of code templates and the models.

We recently became aware of an ongoing work that also proposes a DSL to create web applications with rich domain models that can serve as content management systems [18]. A case study for academic research sites is also presented. WebDSL uses a textual DSL defined by SDF [17] and the abstract syntax tree is used for code generation. Contrary to our approach, WebDSL aims to model the page flow and presentation of the web applications, we delegate this work to the product line. This delegation solves many of the future work described in [18], such as better URLs, AJAX support, validation, and security. Both WebDSL and WSL support content-types and workflows but focuses on different implementation technologies.

[11] See http://plone.org/products/archgenxml

Feature Oriented Model Driven Development (FOMDD) [15] combines *Feature Oriented Programming (FOP)* [5] with MDD for the synthesis of Software Product Lines. FOP supports composition of features to create models from which MDD techniques can synthesize applications. A key concept of FOP is *refinement* that adds details to the different artifact types. There exists tool support for FOMDD that allows the refinement and composition of XML-based artifacts. This support can be easily used to realize the latent variability of the generated component configuration files. Refinements to these files can modify their properties to ultimately synthesize more applications. FOMDD constructs algebraic models of composition and synthesis from which useful tool validation and generation costs properties can be inferred.

WSL requires support of refinements for all of its artifact types for these properties to be derived. A crucial part missing is the composition and refinement of its models. Current research on feature-based composition of UML models can be leveraged to address this issue [12]. Supporting refinements and composition of our models is a topic of future work. Furthermore, the component architecture can be leveraged to support additional functionality of the synthesized components by using adapters [16].

6 Conclusions

This paper presented an extensible domain-specific language – WSL – to specify web applications in terms of the content-types and security policies they provide. The language allows domain experts, notably web designers, to describe the business domain free from implementation specifics.

We found textual DSLs to be an powerful alternative to more traditional graphical concrete syntax. Textual syntax is easy to learn for programmers, supports traditional versioning of text files, and is not tied to any particular model editors.

Designing a DSL is not an easy task and requires a great understanding of the problem domain. Implementing a transformation to a target architecture requires exhaustive analysis of the target. We found domain-specific frameworks helpful to conceptualise the domain in the language and to simplify the target code. In general, it is a good practice to put the code into the framework rather than into the generator, and only generate framework completion code.

We argue that by having horizontal languages, such as WSL, has three major advantages. First, they can be used to model vertical domains, providing greater audience for the tools and MDD in general as it does not target a particular company needs or domain. Second, they are not as technical as architecture- or framework-centric models and can thus be written by a non-technical domain expert. An example of a domain expert in our case is a web designer. Finally, this approach allows a better integration of MDD and SPLs by exposing product-line features in the language and generation of configuration profiles.

Although our work is focused on the Web, we believe that similar efforts on other horizontal domains can potentially bring MDD closer to the specification level as domain-specific modeling promises.

Acknowledgement. We thank Krzysztof Czarnecki for his feedback on drafts of this paper.

References

1. M. Antkiewicz and K. Czarnecki. Framework-specific modeling languages with round-trip engineering. In *MoDELS*, pages 692–706, 2006.
2. M. Aspeli. *Professional Plone Development*. Packt Publishing, 2007.
3. D. Batory. AHEAD Tool Suite. http://www.cs.utexas.edu/users/schwartz/ATS.html.
4. D. Batory. Multilevel models in model-driven engineering, product lines, and metaprogramming. *IBM Syst. J.*, 45(3):527–539, 2006.
5. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.
6. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

7. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
8. S. Efftinge. *OpenArchitectureWare 4.1 Extend Language Reference*.
9. S. Efftinge and C. Kadura. *Xpand Language Reference*, 2006.
10. S. Efftinge and M. Völter. *oAW xText: A framework for textual DSLs*, 2006.
11. A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
12. R. E. Lopez-Herrejon, A. Cavarra, S. Umapathy, and S. Trujillo. Feature modularity and composition of UML models. Submitted for publication.
13. T. Stahl and M. Völter. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.
14. V. Svansson. Model driven web development. Master's thesis, Oxford University, September 2007.
15. S. Trujillo, D. Batory, and O. Díaz. Feature oriented model driven development: A case study for portlets. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota, USA, May 20-26*, 2007.
16. P. v. Weitershausen and P. J. Eby. *Web Component Development with Zope 3*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
17. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, Amsterdam, 1997.
18. E. Visser. Domain-specific language engineering. a case study in agile DSL development (Mark I). Technical Report TUD-SERG-2007-017, Software Engineering Research Group, Delft University of Technology, 2007.